Single-Dimension Software Pipelining for Multi-Dimensional Loops

HONGBO RONG
Microsoft Corporation
ZHIZHONG TANG
Tsinghua University
R.GOVINDARAJAN
Indian Institute of Science
ALBAN DOUILLET
Hewlett-Packard Company
and
GUANG R. GAO
University of Delaware

Traditionally, software pipelining is applied either to the innermost loop of a given loop nest or from the innermost loop to outer loops. This paper proposes a 3-step approach, called *Single-dimension Software Pipelining (SSP)*, to software pipeline a loop nest at an arbitrary loop level that has a rectangular iteration space and contains no sibling inner loops in it.

The first step identifies the most profitable loop level for software pipelining in terms of initiation rate, data reuse potential, or any other optimization criteria. The second step simplifies the multi-dimensional data-dependence graph (DDG) of the selected loop level into a 1-dimensional DDG and constructs a 1-dimensional schedule. Based on it, the third step derives a simple mapping function that specifies the schedule time for the operation instances in the multi-dimensional loop.

The classical modulo scheduling is subsumed by SSP as a special case. SSP is also closely related to hyperplane scheduling, and, in fact, extends it to be resource-constrained. We prove that SSP schedules are correct, and at least as efficient as those schedules generated by traditional modulo scheduling methods.

We extend SSP to schedule imperfect loop nests, which are most common at instruction-level. Multiple initiation intervals are naturally allowed to improve execution efficiency.

Feasibility and correctness of our approach are verified by a prototype implementation in the ORC compiler for the IA-64 architecture, tested with loop nests from Livermore and SPEC2000 floating-point benchmarks. Preliminary experimental results reveal that, compared to modulo scheduling, software pipelining at an appropriate loop level results in significant performance improvement. Software pipelining is beneficial even with loop transformations beforehand.

Categories and Subject Descriptors: D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Compilers, Opti-*

Extension of Conference Paper of [Rong and Tang et al. 2004]. This version extends scheduling to imperfect loop nests, introduces scheduling with multiple initiation intervals, connects the method with hyperplane scheduling, implements it in a production-quality open source compiler, and reports additional performance numbers. There is major addition in Section 5.1 and 5.2. Section 5.4, Section 6 and 7 are new.
Authors' email addresses: hongbor@microsoft.com, tzz-dcs@tsinghua.edu.cn, govind@serc.iisc.ernet.in, alban.douillet@hp.com, and ggao@capsl.udel.edu.

mization

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Software pipelining, modulo scheduling, loop transformation

1. INTRODUCTION

Loop nests are rich in coarse-grain and fine-grain parallelism and substantial progress has been made in exploiting the former [Banerjee 1993; Darte and Robert 1994; Feautrier 1996; Lamport 1974]. With the advent of ILP (Instruction-Level Parallelism) architectures like Very-Long Instruction Word and superscalar processors, and the fast growth in hardware resources, it has been another important challenge to exploit fine-grain parallelism in the loop nests as well.

Software pipelining 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 [Allan et al. 1995; Huff 1993; Intel 2001; Moon and Ebcioğlu 1997; Rau 1994; Rau and Fisher 1993], only a few address software pipelining of loop nests [Lam 1988; Muthukumar and Doshi 2001; Wang and Gao 1996; Gao et al. 1993; Ramanujam 1994].

In [Lam 1988], a loop is modulo scheduled and is considered as an *atomic operation* of its outer loop. The outer loop can then be modulo scheduled. 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) of the software pipeline is addressed in [Muthukumar and Doshi 2001; Wang and Gao 1996].

We refer to the above approach as *innermost-loop-centric modulo scheduling*. This 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 might result in higher parallelism. (2) It cannot exploit the data reuse potential in the outer loops.

There are other software pipelining approaches, developed from hyperplane scheduling. They exploit parallelism from the multi-dimensional iteration space, based on dependences, but cannot handle resource constraints [Gao et al. 1993; Ramanujam 1994].

There has also been other interesting work that combines loop transformations with software pipelining [Wolf et al. 1996; Carr et al. 1996]. However, in these methods, software pipelining is still limited to the innermost loop of the transformed loop nest.

This paper presents a framework for resource-constrained software pipelining for a class of loop nests. Software pipelining is applied to the most "beneficial" level in a loop nest, in order to better exploit parallelism and data reuse potential, and match the hardware resources.

The problem addressed in this paper can be formally stated as follows: *Given a loop nest composed of n loops $L_1, L_2, \ldots, L_n$, from the outermost to the innermost level, with one loop at each level, identify the most profitable loop L_x ($1 \leq x \leq n$) that has a rectangular iteration space and software pipeline it.* Software pipelining L_x means that the consecutive iterations of L_x will be overlapped at run-time. In this paper, we only discuss how to parallelize the selected loop L_x. Its outer loops, if any, remain intact in our

approach. Since there is only one loop at each level, in this paper, the terms "*loop*" and "*loop level*" can be used interchangeably.

The above problem can be broken down into two sub-problems: how to predict the benefits of software pipelining a loop level, and how to software pipeline the most "beneficial" one predicted.

Our solution consists of three steps:

(1) *Loop selection*: This step searches for the most profitable loop level in the loop nest. Profitability can be measured in terms of initiation rate, data reuse potential, or any other optimization criteria. The selected loop may be a loop nest itself, i.e., it may have its own inner loops.

(2) *1-D schedule construction*: The multi-dimensional DDG of the selected loop is reduced to a 1-dimensional (1-D) DDG. Based on the 1-D DDG and the resource constraints, a modulo schedule, referred to as a *1-D schedule*, is constructed for the operations in the selected loop. No matter how many inner loops the selected loop 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 that specifies the schedule time of the operation instances in the selected 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). This approach 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. Any criterion can be used to judge the "profitability" in this step. This flexibility opens a new prospect to combine software pipelining with any other optimal criterion beyond the ILP degree, which is often the major objective of 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.

—*Simplicity*: The method retains the simplicity of the classical modulo scheduling of single loops. The scheduling is based on a simplified 1-dimensional DDG, no matter how many inner loops the selected loop has. This is an essential difference from previous approaches. The traditional modulo scheduling of single loops is subsumed as a special case.

—*Efficiency*: Our schedule divides the iterations of the chosen loop into groups, and executes them group by group. Each group is pipelined. The draining of a group is naturally overlapped with the filling of the next group. For this reason, the schedule is proved to have the shortest length, comparing with any schedule produced by the innermost-centric modulo scheduling under identical conditions for a perfect loop nest. In addition, we search the entire loop nest and choose the most profitable loop level, and consider data reuse in the cache, which also improves the actual execution time of the schedule.

—*Resource sensitivity*: Although our method has been developed independently, we can relate our approach as an extension of hyperplane scheduling [Darte and Robert 1994; Lamport 1974] in the context of software pipelining for uniprocessors. Our approach extends the traditional hyperplane scheduling to be resource-constrained. The major

differences between the two approaches are: (i) Software pipelining is used for uniprocessors, which has limited resources, and it is imperative to consider such constraints. Hyperplane scheduling, however, is usually used for large array-like hardware structures such as systolic and SIMD arrays, and does not consider resource constraints. (ii) Hyperplane scheduling often exploits parallelism from more than one loop level, whereas software pipelining focuses on a single loop level only.

—*Reducing overhead*: Being able to schedule an arbitrary loop level provides freedom for choosing good schedules with less overhead.

To understand this, let us ask a question: why is it necessary to develop a technique that can pipeline a multi-dimensional loop directly? Can't we achieve the same effect through loop transformations followed by innermost-loop-centric software pipelining?

A key insight is that when software pipelining is applied to a loop, the schedule has associated overhead, including initialization, prolog, epilog and finalization. Take the IA-64 architecture as an example. Initialization sets up the initial values of the control registers, including the loop counter register `LC`, epilog counter register `EC`, and predicate rotating register (`pr.rot`). It also transfers live-in values of the loop variables to rotating registers. Finalization transfers live-out values out of rotating registers. Such overhead is incurred every time the loop is executed. The overhead is unavoidable, no matter whatever loop transformations have been performed previously. Some loop transformations, like tiling, would magnify the overhead of an inner loop level, by making the loop nest deeper and the inner loops having smaller trip counts.

Intuitively, the outer the loop is, the less overhead it has. In a 3-deep loop nest, where each loop level is executed 1000 times, the overhead is incurred 1,000,000 times if the innermost loop is software pipelined, 1,000 times if the middle loop is pipelined, and only 1 time if the outermost loop is pipelined. Unless software pipelining the innermost loop results in significant benefit that outweighs such overhead, innermost-loop-centric software pipelining may not be advantageous.

In terms of loop transformations, they are orthogonal to our approach. In this paper, we assume that a loop nest to be software pipelined has already been optimized by loop transformations, if any.

SSP can be applied to both perfect and imperfect loop nests. We will restrict it to perfect loop nests first, and then extend it to imperfect ones, where we introduce multiple initiation intervals into the schedule to improve execution efficiency.

In this paper, we focus only on the fundamental theory of SSP scheduling. The other equally important problems are how to design efficient scheduling algorithms, and how to allocate registers and generate compact code for the constructed SSP schedule. The scheduling algorithm used in our experiments is described in the appendix of a technical memo [Rong et al. 2007]. Register allocation and code generation have posed interesting challenges and are addressed elsewhere [Rong et al. 2005; Rong and Douillet et al. 2004].

We target ILP uniprocessors with support for predication [Intel 2001; Allen et al. 1983]. Our approach does not impose any constraint on the function units; they may be homogeneous or heterogeneous, pipelined or non-pipelined, and can have unit or multi-cycle latencies.

We have implemented a prototype of our method in the ORC compiler for the IA-64 architecture and tested it with loop nests from Livermore and SPEC2000 floating-point benchmarks. The resulting code is run on an IA-64 Itanium machine and the actual ex-

ecution time is measured. Preliminary experimental results on these loop nests reveal considerable performance differences in software pipelining different loop levels of a loop nest, and often, software pipelining an outer loop shows significant performance improvements over modulo scheduling that software pipelines the innermost loop. Furthermore, we observe that SSP is beneficial in the presence of loop transformations, such as loop interchange, loop tiling, and unroll-and-jam.

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 its correctness and efficiency in Section 5. We then extend SSP to imperfect loop nests and introduce multiple initiation intervals into the schedule in Section 6. Experimental results are reported in Section 7. A discussion on related work and concluding remarks are presented in Sections 8 and 9.

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, with each level having exactly one loop, and all operations are within the innermost loop. Each loop $L_x (1 \leq x \leq n)$ has an index variable i_x and an index bound $N_x > 1$. The index is normalized to change from 0 to $N_x - 1$ with unit step. The body of the loop nest consists of all the operations. It is assumed to have no branches; branches, if any, have been converted to linear code by if-conversion [Allen et al. 1983].

Since a loop, except the innermost loop, has its own inner loops, it is a loop nest itself. To emphasize this fact, we also refer to a loop as an *x-dimensional loop*, where x is the depth of the loop nest. For example, L_1 is an n-dimensional loop, L_n a 1-dimensional loop, etc. A 1-dimensional loop is also called a *single loop*.

The loop nest has an *iteration space*, which contains one point for each execution of the body of the loop nest. Such a point is called an *iteration point* in this paper, 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 $o(\boldsymbol{I})$. The iteration space is *rectangular* if its bounds, N_1, N_2, $\cdots$, and N_n, do not change during the execution of the loop nest, although they can change before and after it. In this paper, the loop level to be software pipelined must have a rectangular iteration space.

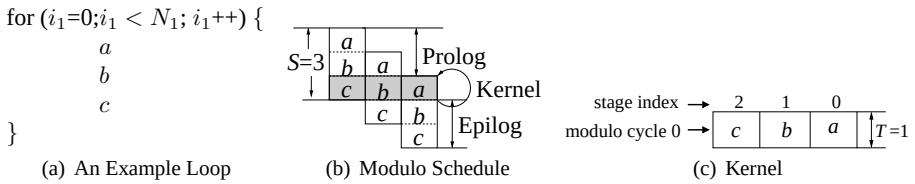
An L_x *iteration* is one execution of the L_x loop. Thus the L_x loop has a total of N_x number of iterations. One such iteration is also an iteration point if L_x is the innermost loop, i.e., $x = n$.

We use $(o_1 \rightarrow o_2, \delta, \boldsymbol{d})$ to represent a data dependence from operation o_1 to operation 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 *zero vector*.

Software pipelining exposes instruction-level parallelism by overlapping successive iterations of a loop. *Modulo scheduling (MS)* is an important and probably the most commonly used approach of software pipelining [Huff 1993; Lam 1988; Rau 1994; Rau and Fisher 1993]. A detailed introduction can be found in [Allan et al. 1995].

Modulo scheduling is usually applied to a single loop. Instances of an operation from



(a) An Example Loop (b) Modulo Schedule (c) Kernel

Fig. 1. Modulo Scheduling of a Single Loop

successive iterations of the loop are scheduled with an *Initiation Interval* (II) of T cycles. This is referred to as *modulo property*. A valid modulo schedule respects modulo property, the *dependence constraints*, and the *(hardware) resource constraints*.

The schedule length l is defined as the execution time of a single iteration. 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: the *prolog* to fill the pipeline, the *kernel* to be executed multiple times, and the *epilog* to drain the pipeline.

Let the schedule time for any operation instance $o(i_1)$ be $\sigma(o, i_1)$. When $i_1 = 0$, it can be expressed as $\sigma(o, i_1) = p * T + q$, where $0 \leq q < T$. We say that operation o is scheduled to *modulo cycle* q within stage p.

Example: Fig. 1(a) shows an example loop. Assume 3 function units and two dependences $(a \rightarrow b, 1, \langle 0 \rangle)$ and $(b \rightarrow c, 1, \langle 0 \rangle)$. Fig. 1(b) shows a modulo schedule for the loop with $T = 1$, and $S = 3$. Fig. 1(c) specifically shows the kernel, where the stages are numbered from 0 to 2 from right to left, and all operations are scheduled to modulo cycle 0.

3. MOTIVATION AND OVERVIEW

In this section, we motivate our method with the help of a simple 2-deep perfect loop nest. We bring out the practical limitations of the innermost-centric approach and motivate the necessity of our work. Subsequently, we illustrate our approach using this example. Then we summarize the intuitions we get from the example, and briefly describe our theoretical solution to the general problem.

3.1 A Motivating Example

Fig. 2 shows a perfect loop nest in C language and its data dependence graph. This loop nest certainly could be parallelized in a number of other ways, too. We use it only for illustration purposes.

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 inner loop cannot find any parallelism for this example. Innermost-loop-centric software pipelining approach exposes extra parallelism by overlapping the filling and draining of the pipeline between successive outer loop iterations. Since modulo scheduling failed to find any parallelism, there is no filling or draining and therefore no overlapping. Thus, innermost-loop-centric software pipelining cannot find any parallelism, either.

One may argue that loop interchange before software pipelining will solve this problem. Unfortunately, that will destroy the data reuse in the original loop nest: for large arrays

```
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];
      }
    }
```

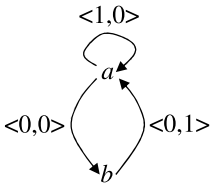


Fig. 2. A Loop Nest and its DDG

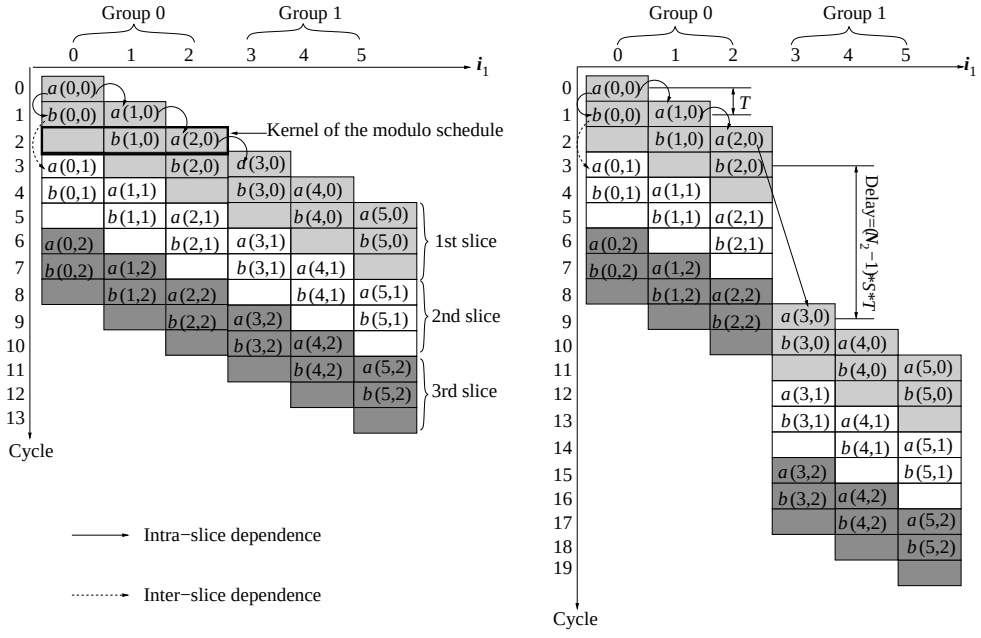


Fig. 3. A Conceptual Illustration of Our Software Pipelining Approach (Assume $N_1 = 6$ and $N_2 = 3$)

each iteration point will introduce 2 cache misses, as the array elements are now accessed column-wise rather than row-wise.

3.2 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 potential of the outer loop(s). 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? Before we expand discussion on these challenges, let us once again look at our motivating example shown in Fig. 2.

Example: Let us assume operations a and b have latencies of 1 and 2 cycles, respectively. Assume that we have two functional units, and both are pipelined and can perform any of

the operations.

Suppose that the outer loop L_1 is selected for software pipelining. We remember software pipelining a loop is to overlap its iterations. Fig. 3(a) shows such an overlapping, where the initiation interval between two adjacent iterations of the loop is $T = 1$ cycle, and we assume $N_1 = 6$ and $N_2 = 3$ for simplicity.

We consider the operations belonging to iteration points $(i_1, 0)$, for all i_1, constitute the first *slice*, and operations belonging to points $(i_1, 1)$ the second slice, etc. Then the overlapping can be reinterpreted in this way: each slice is modulo scheduled so that successive iteration points within this slice initiate at an interval of $T = 1$ cycle. For the first slice, the kernel of the modulo schedule is highlighted in a box. There are $S = 3$ stages, with one stage being empty.

Although the resource constraints are respected within each modulo scheduled slice, they are violated between slices because a slice is issued greedily without waiting for the resources to be released by the previous slice. To remove the conflicts, we cut the slices into *groups*, with each group having $S = 3$ iterations of the outer 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. A dependence is still respected after the pushing-down because that action either does not affect, or only increases, the time distance between the source and the sink of the dependence, as illustrated by the dependences in Fig. 3(a) before the pushing-down and in Fig. 3(b) after that.

Repeating patterns can be found in the final schedule. In Fig. 4, we add to the final schedule some ineffective operation instances, as shown in the shaded part. They are ineffective because their first indexes are beyond the legal range of i_1, the outer loop index variable. The range is assumed to be [0,6) in our illustration. For target architectures with predication support like IA-64, predicate registers can be used to make them ineffective during execution of the final schedule [Rong and Douillet et al. 2004]. With the added ineffective operation instances, it is clear to see that the final schedule is composed of two repeating patterns, referred to as *Outermost Loop Pattern (OLP)* and *Inner Loop Execution Segment (ILES)*. An OLP drains the pipeline of a group, and fills the pipeline with the next group simultaneously. When the pipeline is filled with the next group, an ILES starts. It runs all the inner loops of the next group until the group is going to drain. Then another OLP starts. Note that an ILES itself is composed of $N_2 - 1 = 2$ number of a smaller pattern, as shown in the figure. Apart from the OLPs and ILESes, the final schedule also contains a prolog and an epilog. The prolog is part of the first OLP in the perfect loop nest case. The last three cycles form the epilog.

Based on the above observation, it is straightforward to rewrite the final schedule in a more compact form, as shown in Fig. 5. An OLP (including the prolog), an ILES, and the epilog, are all composed of multiple copies of the kernel. The kernel copies in the prolog and the epilog are partial, with the left and right parts being masked from execution, respectively. The stages in a kernel copy in the ILES is permuted, to maintain the sequential execution of each iteration of the selected loop.

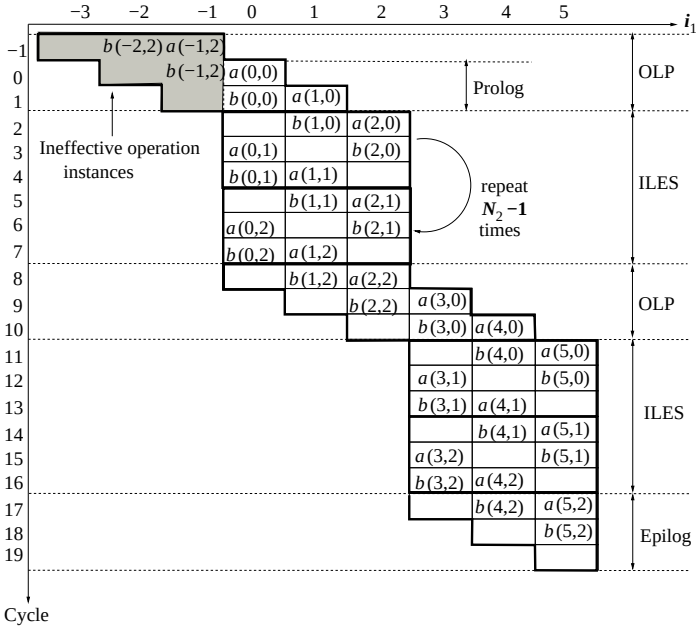


Fig. 4. Repeating Patterns in the Final Schedule

L_1': for $(i_1 = 0; i_1 < N_1; i_1 += 3)\{$

OLP
$b(i_1-2, N_2-1)$ $a(i_1-1, N_2-1)$
$b(i_1-1, N_2-1)$ $a(i_1, 0)$
$b(i_1, 0)$ $a(i_1+1, 0)$

ILES
L_2': for $(i_2 = 1; i_2 < N_2; i_2++)$ {
$b(i_1+1, i_2-1)$ $a(i_1+2, i_2-1)$
$a(i_1, i_2)$ $b(i_1+2, i_2-1)$
$b(i_1, i_2)$ $a(i_1+1, i_2)$
}

}

Epilog
$b(i_1-2, N_2-1)$ $a(i_1-1, N_2-1)$
$b(i_1-1, N_2-1)$

Fig. 5. Rewritten Loops

3.3 Overview of Our Approach

Based on the illustration, in this section, we briefly describe the steps and challenges in solving our general problem. The first challenge is how to select a loop level for software pipelining. Once the loop level is identified, the second challenge is how we software pipeline it, taking into account resource and dependence constraints. The principles are discussed below, while details are left to Section 4.

3.3.1 *Which Loop to Software Pipeline?.* Parallelism is surely one of the major concerns. On the other hand, cache effects are also important and govern the actual execution time of the schedule. However, it is 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 data reuse, or both. Any other objective can also be used as a criterion.

The selected loop, which is a loop nest itself, needs to have a rectangular iteration space. How to handle a loop with a non-rectangular iteration space is beyond the scope of this paper.

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

The iteration points are allocated in this way: 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.

To modulo schedule a slice, we reduce the DDG to have only the dependences at this L_1 loop level and simplify their distance vectors to be 1-dimensional. Based on the resource constraints and this 1-D DDG, we construct a modulo schedule, referred to as a *1-D schedule*.

This 1-D schedule is applied to every slice. So all slices have the same shape, and they can be packed together seamlessly. This leads to a schedule like that in Fig. 3(a). We can see that the iteration points are allocated to the slices in such a way that all iterations of the L_1 loop run in parallel, while each of them runs sequentially.

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. Modulo scheduling of a slice meets the resource constraints within the slice. However, by packing the successive slices together, two slices are partially overlapped. The collective resources required at an overlapping cycle may exceed the resources available. To solve this problem, we cut the slices into *groups*, with each group containing S iterations of the L_1 loop. Then we push down, i.e. delay the execution of, a group until resources are available. This results in a *final schedule*, which respects the resource constraints at each cycle, as illustrated in Fig. 3(b).

How to Handle Dependences? A major obstacle to software pipelining of a loop nest is how to handle the n-dimensional dependence distance vectors. As mentioned earlier, in modulo scheduling a slice, we consider only the simplified dependences with 1-dimensional distance vectors. Doing so is sufficient to satisfy all dependence constraints within a slice. Dependences across slices (in the forward direction) are also satisfied, since the slices are executed sequentially. After the slices are cut into groups and the groups are pushed down, the dependences are still respected, as pushing down either does not affect, or can only increase, the time distance between the source and the sink of a dependence, as illustrated in Fig. 3(a) and Fig. 3(b).

3.3.3 *Constructing the Final Schedule.* In this paper, we express the final schedule abstractly in a function, which is *independent* of any specific architecture. The function describes the final schedule time of an operation instance, based on the 1-D schedule time

of that operation.

It is important to see that this function is determined by the 1-D schedule. We do *not* unroll any loop in constructing either the 1-D schedule or the final schedule. The example in Fig. 3(a) and 3(b) has illustrated the formation of the final schedule in a way that can be easily understood: fully unrolling the chosen loop and allocating all the iteration points in it to slices, applying the 1-D schedule to all slices, cutting them into groups, and pushing down the groups appropriately. In our formal solution, however, the same effect is simply captured by the final schedule function.

For a specific architecture, the final schedule is constructed by composing the prolog, OLP, ILES, and epilog with the 1-D schedule. This realizes the function equivalently by code. Depending on the target architecture, the 1-D schedule may need to be duplicated in this process. For example, for an architecture like IA-64 that supports rotating register files, the final schedule rotates registers in an OLP, but stops rotating in an ILES. Thus the ILES has to duplicate the 1-D schedule to achieve the same effect of register rotating. The details of this code generation process are beyond the scope of this paper and are presented elsewhere [Rong and Douillet et al. 2004].

Later, when we extend our approach to allow all the loops in the loop nest have their own distinct IIs, it is too complicated to express the final schedule in a function. In this case, we also resort to the constructive code generation process.

In summary, our approach to software pipeline a loop nest consists of three steps: loop selection, 1-D schedule construction, and final schedule computation. We will describe them in detail in the next section.

4. SOLUTION

In this section, we first define the concept of simplified DDG. With this concept, we formalize our approach into 3 steps: (1) loop selection, (2) 1-D schedule construction [1], and (3) final schedule computation.

4.1 Definition of Simplified DDG

As illustrated in Fig. 3, conceptually, the final schedule of a multi-dimensional loop consists of a series of modulo scheduled slices, which are cut into groups, and the groups are then pushed down to resolve inter-slice resource conflicts. If a dependence is respected before pushing down the groups, it will also be respected after that. Therefore we only need to consider the dependences necessary to obtain the modulo schedule before the pushing-down.

Fig.6 pictorially illustrates the dependences in an n-dimensional loop nest in two successive slices, where each parallelogram represents a slice, and each dot an iteration point. Although not shown on the picture, each slice is software pipelined. The outermost level L_1 is assumed to be the chosen loop. There are two kinds of dependences: one is across two slices, and the other one is within a slice.

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

[1] In our previous work [Rong and Tang et al. 2004], this step is referred to as "dependence simplification and 1-D schedule construction". We remove "dependence simplification" from the name, because dependence simplification is also needed by the loop selection step. Therefore, it is more appropriate to be taken as a basic concept shared by both steps.

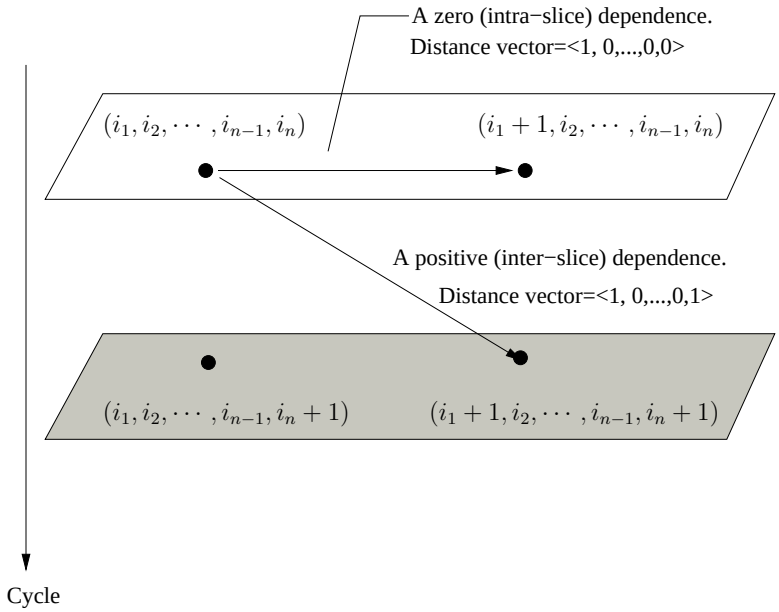


Fig. 6. Dependences

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 of the slice. Besides, only the distance d_1 is useful for software pipelining. That is, the dependence distance vector can be simplified as $\langle d_1 \rangle$ in pipelining.

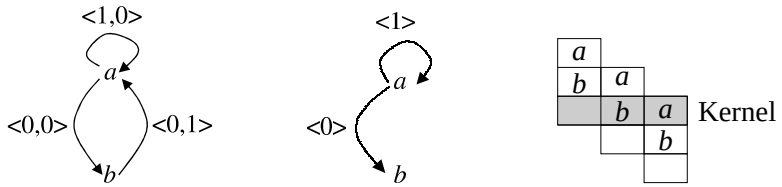
The two kinds of dependences are named *positive* and *zero dependences*, respectively. Note that a dependence from a slice to a previous slice is illegal. It is called a *negative dependence*. Negative dependences cannot be handled directly [2].

Below we formally classify the dependences, and define the simplified dependence graph.

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 zero vector with appropriate length. By *effective*, we mean that such a dependence must be respected by the final schedule if 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 $\langle d_x, d_{x+1}, \cdots, d_n \rangle \geq \mathbf{0}$. Of course the first element $d_x \geq 0$. We *classify the dependence by the sign of the sub-*

[2]The loop nest must be transformed to make negative dependences become zero or positive. It is always feasible to do so 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, predicate registers can be used to dynamically form a non-rectangular iteration space in runtime. Another way to handle the non-rectangular iteration space is to apply loop peeling such that the space is cut into a rectangle with a triangle before and after it, and SSP is applied only to the rectangle.



(a) Effective DDG at L_1 (b) Simplified DDG at L_1 (c) 1-D Schedule

Fig. 7. Dependence Simplification and 1-D Schedule Construction

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.

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.

In this paper, 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. As discussed above, a positive dependence is across slices in the forward direction and can be naturally honored in the final schedule. Only zero dependences are within a slice and need to be considered. 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.7(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.7(b).

4.2 Step 1: Loop Selection

In this paper, our objective is to generate the most efficient software-pipelined schedule 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. The specific decision is not made here, since that is implementation-dependent. This paper focus on presenting a general framework, not algorithm or implementation. In this section, we address the essential problem of evaluating these two criteria. For each criterion, we consider all the loop levels that have rectangular iteration spaces.

4.2.1 *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 minimum initiation interval at loop level L_x is $max(RecMII_x, ResMII)$, where ***RecMII***$_x$ and ***ResMII*** are the minimum initiation intervals determined, respectively, by recurrences in the simplified DDG at L_x, and by the available hardware resources [3].

$$RecMII_x = \max_{\forall C} \frac{\delta(C)}{d(C)}, \tag{1}$$

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 [Govindarajan et al. 1996].

$$ResMII = \max_{\forall \text{ resource type } r} ResMII_r \tag{2}$$

where $ResMII_r$ is the lower bound of MII determined by resource type r, which is

$$ResMII_r = \begin{cases} \dfrac{\text{total operations that use } r}{\text{total resources of type } r} & \text{if } r \text{ is pipelined.} \\ \dfrac{\text{total execution time of the operations that use } r}{\text{total resources of type } r} & \text{if } r \text{ is non-pipelined.} \end{cases} \tag{3}$$

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 1-D 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 loop L_x. 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.

4.2.2 *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: $\{(i_1, 0, \cdots, 0, i_n) | \forall i_1$ 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 indices of the other loops. 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* $\alpha = span\{(1, 0, \cdots, 0), (0, \cdots, 0, 1)\}$.

[3]The reader will find in section 4.3 that our approach has extra Sequential Constraints. They affect only the schedule length of the 1-D schedule, but not the initiation interval. In the worst case, we can always increase the schedule length to satisfy these constraints. Thus they do not influence the MII calculation here.

Now the problem is very similar to that discussed in [Wolf and Lam 1991]. Below we briefly describe the application of their method in this situation.

For a *uniformly generated set* of memory references in this localized space, let R_{ST} and R_{SS} be the self-temporal and self-spatial reuse vector spaces, respectively. And let *gT* and *gS* be the number of group temporal and group-spatial equivalent classes. Then for this uniformly generated set, the number of memory accesses per iteration point is [4]:

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

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 number of memory accesses per iteration point is the sum of accesses for each uniformly generated set.

The above data reuse model does not consider loop volume. Other models [Ghosh et al. 1999; Carr et al. 1994; Kennedy and McKinley 1992] may be used as well.

4.3 Step 2: 1-D Schedule Construction

Our method software pipelines only the selected loop. Enclosing outer loops, if any, are left as they are. Therefore, without loss of generality, we consider L_1 as the selected loop.

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.

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 if it were a single loop. Any modulo scheduling method can be applied to obtain the 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 $\sigma(o, i_1)$, where $0 \leq \sigma(o, i_1) < S * T$ when $i_1 = 0$.

The 1-D schedule must satisfy the following properties:

(1) Modulo property:

$$\sigma(o, i_1) + T = \sigma(o, i_1 + 1) \tag{5}$$

(2) Dependence constraints:

$$\sigma(o_1, i_1) + \delta \leq \sigma(o_2, i_1 + k) \tag{6}$$

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

(3) Resource constraints: At any modulo cycle of the kernel, no hardware resource is allocated to more than one operation.

[4] We use R_{ST} in the denominator here, instead of R_{SS} as in the original formula on page 39 of the literature[Wolf and Lam 1991], which we think was a typo.

(4) Sequential Constraints: if $n > 1$, then

$$S * T - \sigma(o, 0) \geq \delta \tag{7}$$

for every positive dependence with operation o as the source operation, and δ being the dependence latency.

The first three constraints are exactly the same as those of the classical modulo scheduling [Allan et al. 1995; Govindarajan et al. 1996]. 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. The sequential constraints have effect only for loop nests with more than 1 loop.

Example: For the loop nest in Fig.2 and its effective DDG in Fig.7(a), the simplified DDG at L_1 is shown in Fig.7(b). Based on this simplified DDG, a 1-D schedule can be constructed (Fig. 7(c)). 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 $\sigma(b, i_1) = 1 + i_1 * T$.

4.4 Step 3: Final Schedule Computation

As explained in Section 3.3.2, we first allocate iteration points in the loop nest to slices, 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, we obtain the schedule like that in Fig.3(a). 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).

With the above procedure in mind, a final schedule can be precisely defined by the following mapping function. For any operation o in the iteration point $\boldsymbol{I}$=($i_1, i_2, \ldots, i_n$), the schedule time $f(o, \boldsymbol{I})$ is given by

$$\begin{aligned} f(o, \boldsymbol{I}) \;=\; & \sigma(o, i_1) \\ & + \sum_{2 \leq x \leq n} (i_x * (\prod_{x < y \leq n+1} N_y) * S * T) \\ & + \left\lfloor \frac{i_1}{S} \right\rfloor * ((\prod_{2 \leq x \leq n+1} N_x) - 1) * S * T, \end{aligned} \tag{8}$$

where N_{n+1}=1.

Let us briefly explain how the above equation is derived. First, let us consider the ideal schedule before 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 1-D schedule time. This corresponds to the first term of the right-hand side of Equation (8).

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 $o(i_1, i_2, \ldots, i_n)$ equals

$$\sum_{2 \le x \le n} (i_x * (\prod_{x < y \le n+1} N_y) * S * T). \tag{9}$$

This corresponds to the second term of the right-hand side of Equation (8).

Next we discuss the effect of pushing down the groups. Iteration point $o(\boldsymbol{I})$ is located in group $\lfloor \frac{i_1}{S} \rfloor$. Each group 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 $- 1) * S * T$. Thus the group where $o(\boldsymbol{I})$ is located is pushed down by

$$\left\lfloor \frac{i_1}{S} \right\rfloor * ((\prod_{2 \le x \le n+1} N_x) - 1) * S * T \tag{10}$$

cycles. This is exactly the third term in Equation (8).

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(c), 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 can be seen from Fig.3(b).

5. ANALYSIS

In this section, we establish the correctness and efficiency of the SSP approach, and its relationship with MS. We also demonstrate the relationship between SSP and the traditional hyperplane scheduling.

5.1 Correctness

First, we show a simple fact in an SSP final schedule: the instances of any operation is initiated one by one every T cycles. For example, in Fig. 3(b), the instances of operation a are issued in a sequence every $T = 1$ cycle: $a(0,0)$, $a(1,0)$, $a(2,0)$, $a(0,1)$, $a(1,1)$, $a(2,1)$, $a(0,2)$, $a(1,2)$, $a(2,2)$, $a(3,0)$, $a(4,0)$, $a(5,0)$, $a(3,1)$, ... It is trivial to prove that such initiation pattern is true in general for any-deep loop nest with any number of operations.

A direct consequence of the fact is that no two instances of the same operation can be initiated at the same cycle. This result will be used in proving the following theorem:

THEOREM 5.1. *The final schedule defined in Equation (8) respects all the dependences in the effective DDG and the resource constraints.*

PROOF. Given a dependence $(a \rightarrow b, \delta, \langle d_1, d_2, \ldots, d_n \rangle)$ in the effective DDG, $\boldsymbol{I} = (i_1, i_2, \cdots, i_n)$, and $\boldsymbol{I'} = (i_1 + d_1, i_2 + d_2, \cdots, i_n + d_n)$, we show that $f(b, \boldsymbol{I'}) - f(a, \boldsymbol{I}) \ge \delta$. Consider

$$f(b, \boldsymbol{I'}) - f(a, \boldsymbol{I}) = \underbrace{\sigma(b, i_1 + d_1) - \sigma(a, i_1)}_{(A)} +$$

$$\underbrace{\sum_{2\le x\le n}(d_x * (\prod_{x<y\le n+1} N_y) * S * T)}_{(B)} +$$
$$\underbrace{(\left\lfloor \frac{i_1+d_1}{S} \right\rfloor - \left\lfloor \frac{i_1}{S} \right\rfloor) * ((\prod_{2\le x\le n+1} N_x) - 1) * S * T}_{(C)}. \quad (11)$$

First, if this is a zero dependence, then $(B) = 0$, and $(C) \geq 0$. Thus, using Inequality (6), we have

$$f(b, \boldsymbol{I}') - f(a, \boldsymbol{I}) \ \geq \ \sigma(b, i_1 + d_1) - \sigma(a, i_1) \geq \delta.$$

Therefore zero dependences are respected in the final schedule. On the other hand, if the dependence is positive, then it is easy to see that $(A) = \sigma(b, d_1) - \sigma(a, 0) \geq -\sigma(a, 0)$, $(B) \geq S * T$, and $(C) \geq 0$. So

$$\begin{aligned} f(b, \boldsymbol{I}') - f(a, \boldsymbol{I}) \ &\geq \ S * T - \sigma(a, 0) \\ &\geq \ \delta \text{(by sequential constraints in Inequality 7)} \end{aligned}$$

Therefore positive dependences are also respected in the final schedule.

Lastly, any two operation instances that have the same final schedule time must come from the same modulo cycle in the kernel, but they cannot be the instances of the same operation. Since the kernel contains exactly one instance for each operation, and is free of resource competition (by the resource constraints definition in Section 4.3), these operation instances have no resource contention either.

To show this, consider two distinct operation instances, $a(\boldsymbol{I})$ and $b(\boldsymbol{I}')$, scheduled at the same cycle. Then

$$f(b, \boldsymbol{I}') - f(a, \boldsymbol{I}) = 0.$$

By combining this with Equation (11), we get $(A) + (B) + (C) = 0$. Since $(A) = \sigma(b, i_1) + d_1 * T - \sigma(a, i_1)$, and $(B) + (C)$ is a multiple of T (say, $p * T$, where p is an integer), we have $\sigma(b, i_1) - \sigma(a, i_1) = (-p - d_1) * T$, which is also a multiple of T. This means operations a and b must be from the same modulo cycle in the kernel.

As discussed above, the instances of the same operation are issued in order. No two of them can have the same schedule time. Therefore, a and b must be different operations. □

5.2 Efficiency

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 of adjacent iterations of the outer loop, referred to as extended modulo scheduling (xMS) in this paper [Lam 1988; Muthukumar and Doshi 2001; Wang and Gao 1996] . Let us define the *computation time* as the (final) schedule time of the last operation instance+1.

THEOREM 5.2. *For an n-deep perfect loop nest, suppose that MS, xMS, and SSP, find the same initiation interval T and stage number S. Furthermore, suppose that SSP 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 SSP final schedule is not longer than that of the MS or xMS schedule.

PROOF. Modulo scheduling parallelizes the innermost loop, whose iterations issue once every T cycles. So the computation time is

$$Time_{MS} = N_1 * \ldots * N_{n-1} * (N_n + S - 1) * T. \tag{12}$$

In the best case of xMS schedule, the cost of filling and draining the pipeline is incurred only at the beginning and end of the execution of the entire loop nest, and an iteration point is issued every T cycles. The computation time is then

$$Time_{xMS} = (N_1 * \ldots * N_n + S - 1) * T. \tag{13}$$

Let o be any operation and ***I***=($i_1, i_2, \ldots, i_n$) be any index vector. In our approach, it is easy to see that $f(o,$ ***I***$)$ is maximum when ***I***=(N_1-1, N_2-1,$\ldots$, N_n-1) and $\sigma(o, 0) = S * T - 1$. The computation time of SSP is equal to the maximal $f(o,$ ***I***$)$+1, which is

$$\begin{aligned} Time_{SSP} \;=\; & (S + N_1 - 1) * T + \\ & \underbrace{\sum_{2 \le x \le n} ((N_x - 1) * (\prod_{x < y \le n+1} N_y) * S * T)}_{(D)} + \\ & \underbrace{\left\lfloor \frac{N_1 - 1}{S} \right\rfloor * ((\prod_{2 \le x \le n+1} N_x) - 1) * S * T}_{(E)}. \end{aligned} \tag{14}$$

It is easy to show that

$$(D) = ((\prod_{2 \le x \le n+1} N_x) - 1) * S * T. \tag{15}$$

Further, under the given condition that N_1 is divisible by S, we know that

$$N_1 - 1 = p * S + (S - 1),$$

where p is an integer. Thus

$$(E) = p * ((\prod_{2 \le x \le n+1} N_x) - 1) * S * T. \tag{16}$$

Combining Equations (16) and (15), we get

$$\begin{aligned} (D) + (E) \;&=\; (p + 1) * ((\prod_{2 \le x \le n+1} N_x) - 1) * S * T \\ &=\; N_1 * ((\prod_{2 \le x \le n+1} N_x) - 1) * T. \end{aligned}$$

Substituting it in Equation (14), we get

$$Time_{SSP} = ((\prod_{1 \le x \le n+1} N_x) + S - 1) * T. \tag{17}$$

From Equations (12), (13), and (17), we have:

$$Time_{SSP} = Time_{xMS} \leq Time_{MS}.$$

□

Intuitively, this theorem holds because the final schedule produced by SSP always issues one iteration point 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 $N_1 \geq S$ (according to the discussion in Section 4.2.1) and S is typically 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.

5.3 Relation to the Classical Modulo Scheduling of a Single Loop

If the loop nest is a single loop (n=1), the sequential constraints are trivially satisfied. 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 by SSP as a special case.

5.4 Relation to Hyperplane Scheduling

Next we establish the relation between our method and traditional hyperplane scheduling methods [Darte and Robert 1994; Lamport 1974]. We rewrite the mapping function for the final schedule in Equation (8) as follows.

$$f(o,\boldsymbol{I}) = \boldsymbol{I}.\pi + offset(o,\boldsymbol{I}), \tag{18}$$

where $\boldsymbol{I} = (i_1, i_2, \cdots, i_n)$, "." is the inner product operator,

$$\pi = (T, (\prod_{2<y\leq n+1} N_y) * S * T, \cdots, N_{n+1} * S * T)^{transpose},$$

and

$$offset(o,\boldsymbol{I}) = \sigma(o,0) + \underbrace{\left\lfloor \frac{i_1}{S} \right\rfloor * ((\prod_{2\leq x\leq n+1} N_x) - 1) * S * T}_{(F)}. \tag{19}$$

The mapping function for the final schedule consists of two parts. The first part $\boldsymbol{I}.\pi$ corresponds to hyperplane scheduling, which determines how to allocate the iteration points to slices. Unlike the traditional hyperplane scheduling that solves an integer programming problem to find out an optimal scheduling vector, here the scheduling vector π is predefined with S and T as parameters. The objective is thus not to find an optimal scheduling vector, but to find an optimal initiation interval.

This scheduling vector is unlikely to be found by the traditional hyperplane scheduling, because the vector expresses resource constraints through parameters S and T, while the traditional hyperplane scheduling does not consider resource constraints.

The second part, $offset(o,\boldsymbol{I})$, enforces dependences and resource constraints at the instruction level. In this offset, the first component is the 1-D schedule time, $\sigma(o,0)$, which enforces those constraints within a slice, while the second component, (F), enforces resource constraints across slices. This offset is not a constant determined solely by the

operation o; it is a function of the first loop index i_1. Thus, the form of Equation (18) is similar to, but not a special case of, the known extensions of hyperplane scheduling [Darte and Robert 1994; Ramanujam 1994; Gao et al. 1993] . The particular definition of this offset in Equation (19) is unlikely to be derived from the above methods.

5.5 Time Complexity

Our approach consists of loop selection, constructing a 1-D schedule, and computing the final schedule.

Loop selection is flexible and its complexity depends on the specific criteria. Let us consider the two criteria we presented in Section 4.2. Let *UG* be the number of uniformly generated sets, and u be the number of operations. In the worst case, for each loop level, we compute the lower bound of initiation interval that requires $O(u^3)$ time with Floyd's All-Points Shortest Path algorithm [Allan et al. 1995], and estimate data reuse by Gauss-Seidal which requires $O(UG * n^2)$ time. Therefore, the total time in this part is $O(u^3 * n + UG * n^3)$. In general, n is never greater than 6, and *UG* is typically small. Hence the dominant factor is still u. Thus the time complexity of the loop selection phase can be approximated as $O(u^3)$.

The construction of the 1-D schedule is traditional modulo scheduling applied to the simplified DDG, whose complexity is generally $O(u^3)$ or $O(u^4)$, depending on the algorithm used [Allan et al. 1995]. The sequential constraints do not increase complexity. Computing the final schedule does not increase time, either, as it is simple parameter substitution.

To summarize, the overall time complexity of SSP is bounded by $O(u^3)$ or $O(u^4)$, depending on the specific loop selection criteria and the modulo scheduling method used.

6. EXTENSION TO IMPERFECT LOOP NESTS

At instruction-level, it is common for a loop nest to be imperfect. Usually, a loop nest that is perfect in a high level representation becomes imperfect when lowered to instruction level. This is because operations for address calculation would be introduced between the loop levels.

Our study on scheduling perfect loop nests has set up a solid background, but cannot be applied directly to an imperfect loop nest: Not all operations appear with the same frequency now. An operation at an inner loop level runs more frequently than an operation at an outer loop level. Also, for efficiency, operations at different loop levels should be scheduled with different IIs, such that the instances of an operation at an inner loop level can be initiated at a faster rate, if possible.

In this section, we extend our scheduling approach to imperfect loop nests. First, we discuss how to schedule an imperfect loop nest with a single II. Subsequently, we discuss how the loop nest can be scheduled with multiple IIs to achieve higher execution efficiency.

6.1 Motivating Example

Fig. 8(a) shows an example imperfect loop nest. Compared with the example in Fig. 2, it differs only in the outer loop. Suppose we choose loop L_1 for software pipelining. Let there be two function units, both being able to execute any operation, and each statement be considered as an operation with unit latency, except operation e with 2 cycles.

The effective DDG in Fig. 8(b) can be simplified to the 1-D DDG in Fig. 8(c), using the same concepts in Section 4. Based on this DDG and the resource constraints, we schedule all the operations as if they were in a single loop.

A 1-D schedule is shown in Fig. 8(d). The 1-D schedule is a *kernel nest* now: it has two kernels, K_1 and K_2, corresponding to the two loops, and K_1 encloses K_2. They have the same II. Stages 2, 3 and 4 contain the innermost loop operations. In general, we denote the total number of stages for the innermost loop as S_n. For this example, $S_n = 3$.

According to the 1-D schedule, ideally, all L_1 iterations can be overlapped with the initiation interval of $T = 3$ cycles, and each of them is sequential, as shown in Fig. 9(a). For uniformity of representation, we assume the operations before the innermost loop, a, b, and c, have an index vector like $(i_1, 0)$, and the operation after the innermost loop, f, has an index vector like $(i_1, N_2 - 1)$.

In general, every S_n number of L_1 iterations compose a group. After pushing down, we achieve the final schedule as illustrated in Fig. 9(b). To understand it, one may think the process as follows: at the beginning, an L_1 iteration is issued every T cycles. After all the iterations in the first group have entered their innermost loop, they have filled the pipeline, and will hold the resources and continue running sequentially. All the other iterations stall in this period until the first group drains the pipeline and releases resources, when they get resources and continue to issue. Such a process repeats until all iterations finish. Note that the draining of a previous group and the issuing of the next group are overlapped.

Such a way of execution leads to repeating patterns in the final schedule. Thus it can be rewritten into a compact form shown in Fig. 10. Like the perfect loop nest case, it is composed of a prolog, the repetition of an OLP and an ILES, and an epilog, except that the prolog is no longer a part of the first OLP. Each of them still consists of multiple copies of the kernel.

6.2 Assumptions

We assume an imperfect loop nest model in Fig. 11, where each loop L_x has two sets of operations, $OPSETA_x$ and $OPSETB_x$. For the innermost loop L_n, $OPSETA_n = OPSETB_n$. For any loop L_x, its index bound $N_x > 1$.

If an operation o is in either $OPSETA_x$ or $OPSETB_x$, it is said to be *at loop level* L_x, denoted as $level(o) = x$.

In general, an operation in $OPSETA_x$ has an index of $(i_1, i_2, \cdots, i_x)$. We can expand it to be an n-D vector $(i_1, i_2, \cdots, i_x, 0, \cdots, 0)$ for convenience. Similarly, for an operation in $OPSETB_x$, we can expand its index to be an n-D vector $(i_1, i_2, \cdots, i_x, N_{x+1} - 1, \cdots, N_n - 1)$.

6.3 Solution with a Single Initiation Interval

Since any operation can be associated with an n-D index vector, a dependence distance vector is also an n-D vector; based on its value, the dependence can be classified as a zero, positive, or negative dependence. With this fact, the simplified DDG is defined in the same way as before.

Our approach remains to have the same three steps. Loop selection based on parallelism is the same as before. Loop selection based on data reuse can be the same as well if we estimate only the data reuse of the innermost loop, which is most frequently executed, and forget the operations at the outer loop levels. Hence we focus on the next two steps, namely, 1-D schedule construction and final schedule computation. We assume the outermost loop L_1 is chosen for software pipelining.

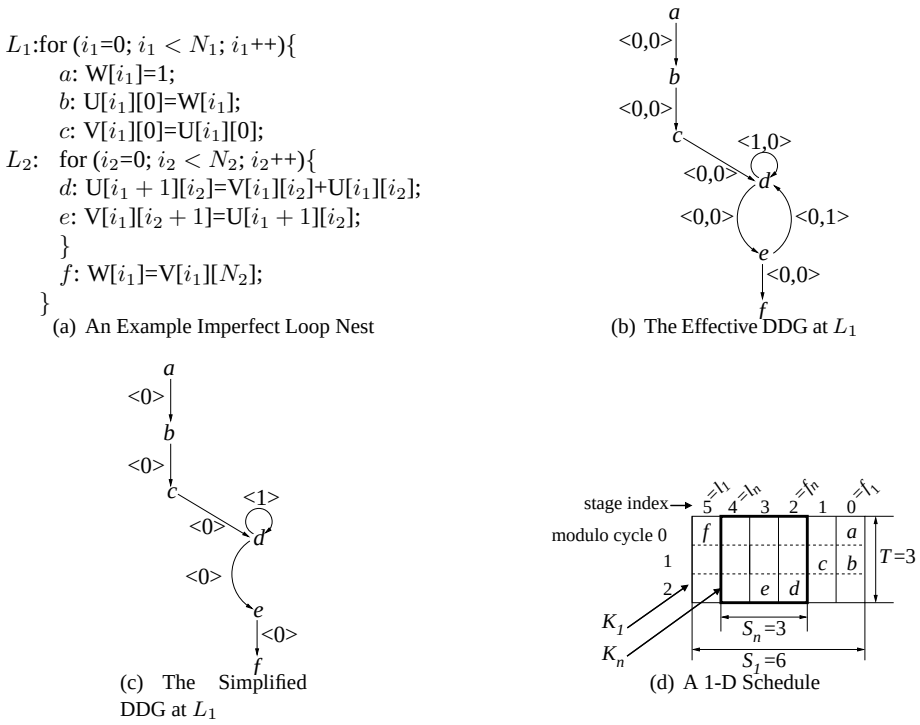


Fig. 8. An Imperfect Loop Nest Example

6.3.1 *1-D Schedule Construction.* The 1-D schedule is now composed of n kernels, each corresponding to a loop (See Fig. 12). Kernel K_x is the kernel for loop L_x. Let f_x and l_x be the first and last stages of it, respectively. Then it has totally $S_x = l_x - f_x + 1$ number of stages, including those of its inner loops. In general, $S_n \leq S_{n-1} \leq \cdots \leq S_2 \leq S_1$. All kernels have the same initiation interval of T cycles.

Consistent with the nesting relationship of the loops, K_x contains K_{x+1}. Operations at an outer loop level are scheduled outside the stages of the inner loops.

As a convention, the 1-D schedule time is defined with the outermost loop kernel K_1 as a reference. That is, for any operation o, if it is scheduled into modulo cycle $q(0 \leq q < T)$ in stage p, then its 1-D schedule time is $\sigma(o, 0) = p * T + q$. We also represent the stage as $stage(o) = p$.

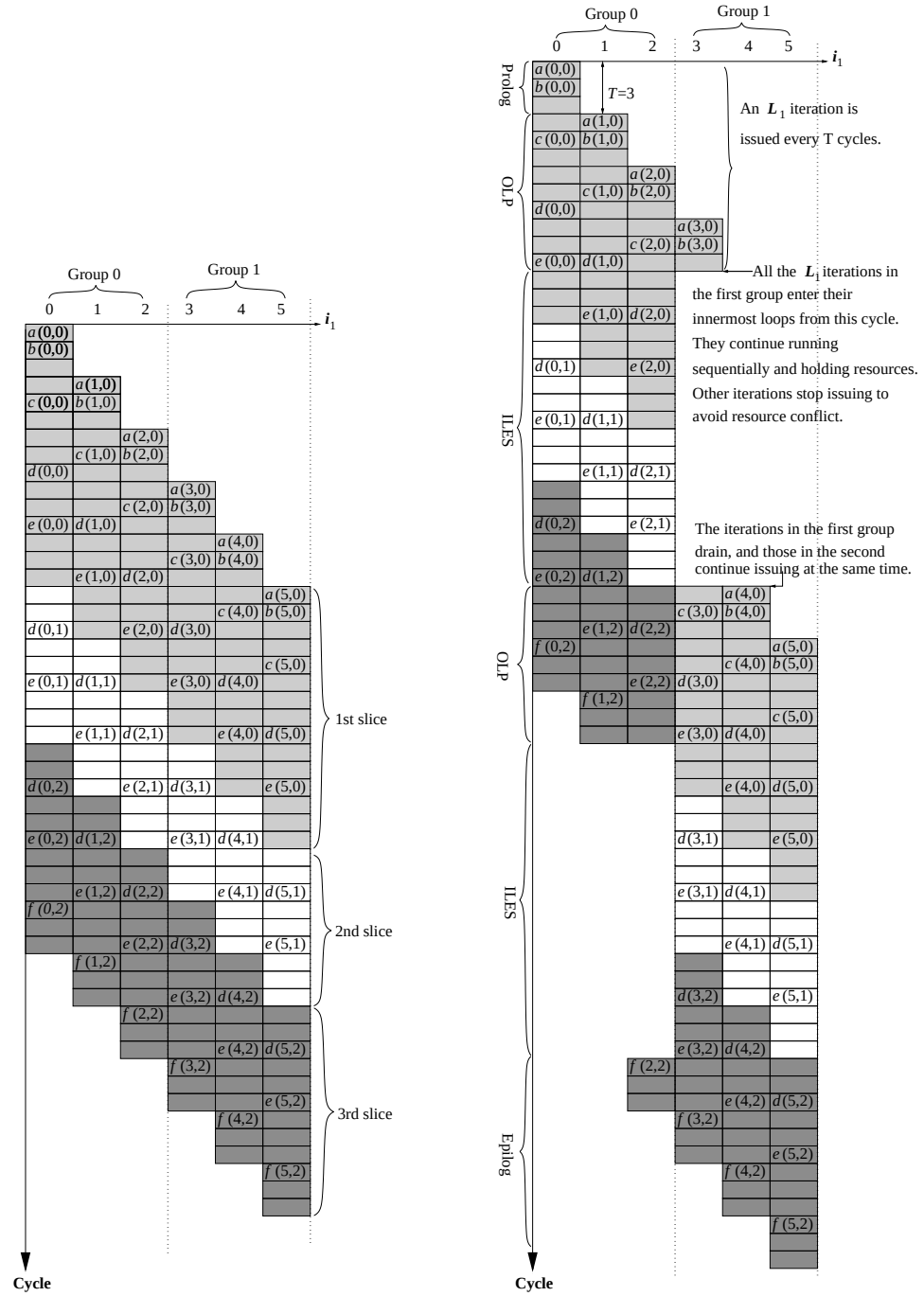
The 1-D schedule needs to respect the following constraints:

(1) Modulo property, dependence constraints, and resource constraints: they remain the same as those in Section 4.3.

(2) Sequential constraints: if $n > 1$, then for every positive dependence with o as the source operation, δ being the dependence latency, and $x = level(o)$,

$$\begin{cases} (l_x + 1) * T - \sigma(o, 0) \geq \delta & \text{if } x = n \text{ or } stage(o) > l_{x+1}, \\ f_{x+1} * T - \sigma(o, 0) \geq \delta & \text{otherwise.} \end{cases} \tag{20}$$

(3) Kernel nesting constraints: (i) $l_1 \geq l_2 \geq \cdots \geq l_n \geq f_n \cdots \geq f_2 \geq f_1$, and (ii) for



(a) The Schedule before Pushing down (b) The Final Schedule

Fig. 9. The Schedules before and after Pushing down (Assume $N_1 = 6$ and $N_2 = 3$)

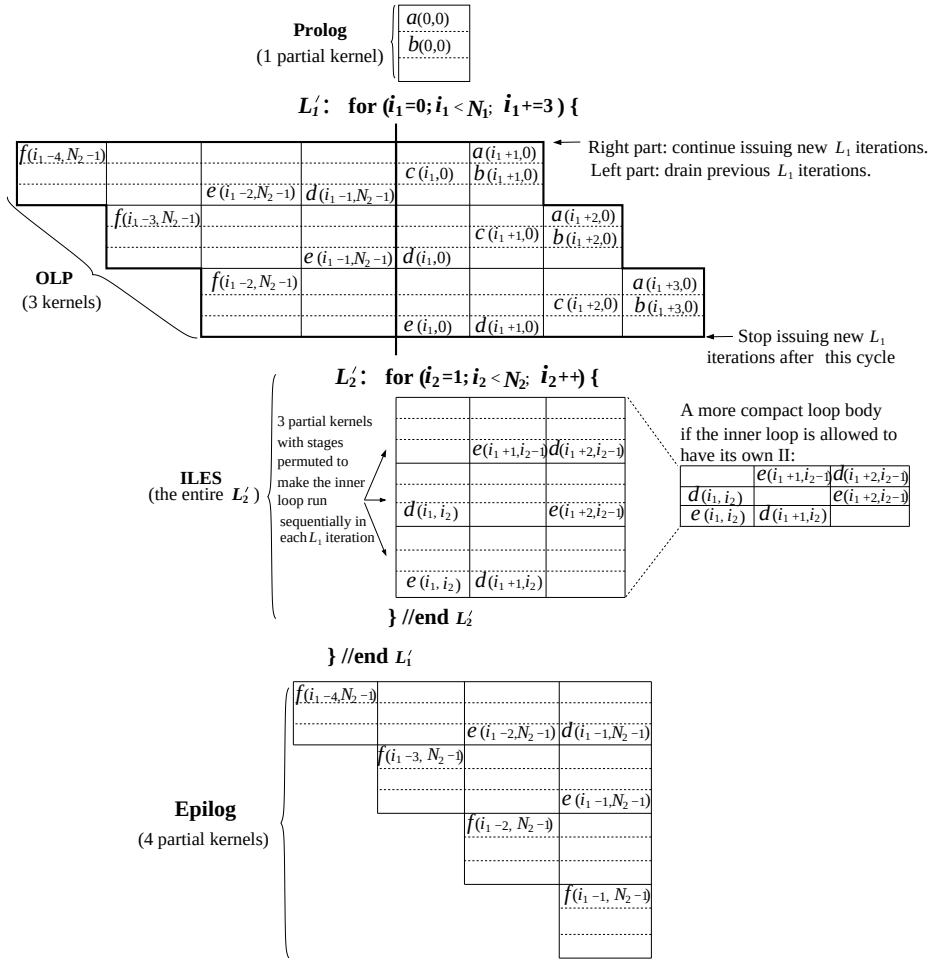


Fig. 10. The Rewritten Loop Nest Representing the Final Schedule

any operation o, if it is at loop level L_n, then $stage(o) \in [f_n, l_n]$. Otherwise, suppose it is at loop level $L_x (x < n)$, then $stage(o) \in [f_x, f_{x+1})$ if o is in $OPSETA_x$, or $stage(o) \in (l_{x+1}, l_x]$ if o is in $OPSETB_x$.

The sequential constraints conservatively require operation o to complete before any possible use of it is issued, and thus the positive dependence from it must be respected.

The kernel nesting constraints express the nesting relationship of the kernels, and conservatively restrict the operations in $OPSETA_x$ ($OPSETB_x$) to be scheduled before (after) the inner loop kernel K_{x+1}.

6.3.2 *Final Schedule Computation.* For any operation o in an iteration point $\boldsymbol{I}$=($i_1, i_2, \ldots, i_n$), the schedule time $f(o, \boldsymbol{I})$ in Equation(8) can be generalized as follows:

```
L1: for (i1=0; i1 < N1; i1++) {
        OPSETA1
L2:     for (i2=0; i2 < N2; i2++) {
            OPSETA2
            ...
Ln:             for (in=0; in < Nn; in++) {
                    OPSETAn
                } //end Ln
            ...
            OPSETB2
        } //end L2
        OPSETB1
    } //end L1
```

Fig. 11. The Imperfect Loop Nest Model

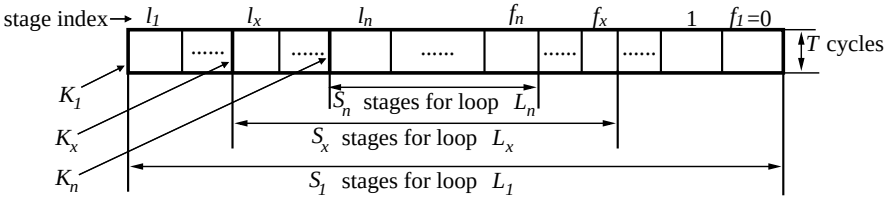


Fig. 12. The 1-D Schedule with a Single II

$$f(o,\boldsymbol{I}) \;=\; \sigma(o,i_1) + \sum_{2\le x\le n} (i_x * ctime_x) + push(o,\boldsymbol{I}) * (ctime_1 - S_1 * T) \quad (21)$$

where $ctime_x$ is the computation time of an L_x iteration in the ideal schedule where the outermost loop iterations are overlapped at the initiation interval of T cycles without delay, and $push(o,\boldsymbol{I}) * (ctime_1 - S_1 * T)$ is the total delay that $o(\boldsymbol{I})$ is pushed down to enforce resource constraints in the final schedule. In this delay, $push(o,\boldsymbol{I})$ is the total number of ILESes that appear before $o(\boldsymbol{I})$ due to the pushing-down, and $ctime_1 - S_1 * T$ is the length of an ILES [5].

$Ctime_x$ can be recursively defined as

$$ctime_x \;=\; \begin{cases} (S_x - S_{x+1}) * T + N_{x+1} * ctime_{x+1} & \text{if } x < n, \\ S_n * T & \text{otherwise.} \end{cases} \quad (22)$$

The total ILESes that appear before $o(\boldsymbol{I})$ due to the pushing down is found to be as

[5]It is easy to see from Fig. 9(b) or Fig. 10 that without the ILESes, the final schedule is nothing but a modulo schedule, where each iteration has $S_1 * T$ cycles. That implies that the length of an ILES equals $ctime_1 - S_1 * T$ cycles.

follows:

$$push(o,\boldsymbol{I}) = \begin{cases} max(0, \left\lfloor \frac{i_1+stage(o)-f_n+1}{S_n} \right\rfloor) & \text{if } (i_2,\cdots,i_n) = (0,\ldots,0) \\ & \text{and } stage(o) < f_n. \\ min(\left\lfloor \frac{N_1-1}{S_n} \right\rfloor, \left\lfloor \frac{i_1+stage(o)-l_n}{S_n} \right\rfloor) & \text{if } (i_2,\cdots,i_n) = (N_2-1,\ldots,N_n-1) \\ & \text{and } stage(o) > l_n. \\ \left\lfloor \frac{i_1}{S_n} \right\rfloor & \text{otherwise.} \end{cases} \tag{23}$$

Let us briefly explain the definition. The operation instance $o(\boldsymbol{I})$ is in group $\left\lfloor \frac{i_1}{S_n} \right\rfloor$. In general, the total number of ILESes appearing before it due to the pushing-down is $\left\lfloor \frac{i_1}{S_n} \right\rfloor$. However, there are exceptions when $o(\boldsymbol{I})$ is in the prolog, an OLP, or the epilog.

If $o(\boldsymbol{I})$ is in the prolog, it is not pushed down at all. If it is in the right part of an OLP that fills new iterations, the total number of ILESes equals $\left\lfloor \frac{i_1+stage(o)-f_n+1}{S_n} \right\rfloor$. Take $a(5,0)$ in Fig. 9(b) as an example. We have $i_1 = 5, stage(a) = 0, f_n = 2$, and $S_n = 3$ according to the kernel in Fig. 8(d). Therefore, there is a total of $\lfloor \frac{5+0-2+1}{3} \rfloor = 1$ ILES appearing before $a(5,0)$. To summarize, we can simply express $push(o,\boldsymbol{I})$ as $max(0, \left\lfloor \frac{i_1+stage(o)-f_n+1}{S_n} \right\rfloor)$, which is the first case in Equation (23).

Second, if $o(\boldsymbol{I})$ is in the epilog, the total number of ILESes is $\left\lfloor \frac{N_1-1}{S_n} \right\rfloor$. If it is in the left side of an OLP that drains previous iterations, the number is $\left\lfloor \frac{i_1+stage(o)-l_n}{S_n} \right\rfloor$. Take $f(2,2)$ in Fig. 9(b) as an example. We have $i_1 = 2$, $stage(f) = 5$, $l_n = 4$, and $S_n = 3$ according to the kernel in Fig. 8(d). Thus the total number is $\lfloor \frac{2+5-4}{3} \rfloor = 1$. That is, it is delayed by one ILES, as can be seen from Fig. 9(b). Note that the ILES that delays it is the second ILES, not the first one, which is always before it and not due to the pushing-down, and thus not accounted for. In short, the total number is $min(\left\lfloor \frac{N_1}{S_n} \right\rfloor, \left\lfloor \frac{i_1+stage(o)-l_n}{S_n} \right\rfloor)$, which is the second case in Equation (23).

THEOREM 6.1. *The final schedule defined in Equation (21) respects all the dependences in the effective DDG and the resource constraints.*

This theorem states the correctness of the final schedule. The proof is presented in the appendix of the technical memo [Rong et al. 2007].

6.3.3 *Relation to the Scheduling of Perfect Loop Nests.* When all the $OPSETA_x$ and $OPSETB_x$ ($x < n$) are empty, the loop nest in Fig. 11 is perfect. Then $l_1 = l_2 = \cdots = l_n \geq f_n = \cdots = f_2 = f_1$, $S_n = S_{n-1} = \cdots = S_2 = S_1$, and all the operations are in the innermost loop. Consequently, the sequential constraints in Inequality (20) are equivalent to Inequality (7). And the kernel nesting constraints are trivially satisfied.

For the final schedule, since any stage is within $[f_n, l_n]$, we get $push(o,\boldsymbol{I}) = \left\lfloor \frac{i_1}{S_n} \right\rfloor$. Then the schedule time function defined in Equation (21) is equivalent to that in Equation (8).

In short, when the loop nest is perfect, both the 1-D schedule and the final schedule constructed by the method in this section are completely the same as those by the method

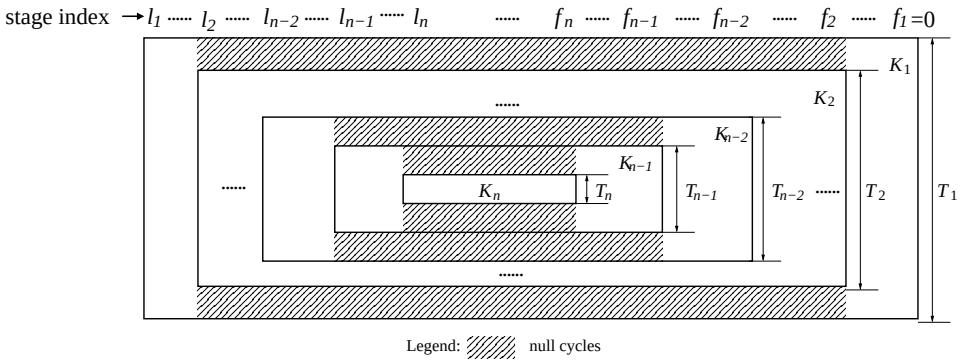


Fig. 13. The General Form of the 1-D Schedule with Multiple IIs

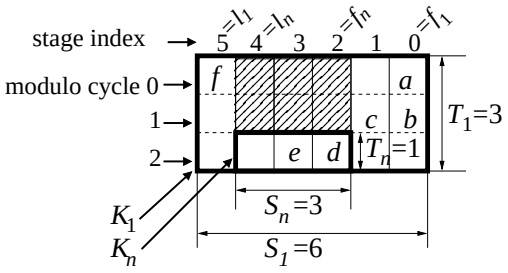


Fig. 14. A 1-D Schedule with 2 IIs for the Loop Nest in Fig. 8(a). There are 2 null cycles above K_n.

in Section 4. In this sense, scheduling of an imperfect loop nest subsumes that of a perfect loop nest as a special case, as expected.

6.4 Solution with Multiple IIs

So far, we have assumed a single initiation interval for all the loop levels. To achieve better performance, however, it is desirable to have multiple IIs. Intuitively, the operations at an inner loop level run more frequently than those at an outer loop level, and therefore should run in a smaller II to shorten the execution time, whenever possible. This leads to the interesting topic of multi-II scheduling, which is also useful in practice.

Fig. 13 shows the general form of a 1-D schedule. Now each kernel K_x has its own initiation interval of T_x cycles. In general, $T_1 \geq \cdots \geq T_{n-1} \geq T_n$. Although K_x takes only T_x cycles, the other cycles below and above it are empty, without any operation, as illustrated by the shaded places in the figure. We call them *null cycles*.

The 1-D schedule time is still defined with the outermost loop kernel K_1 as a reference. That is, for any operation o, if it is scheduled into modulo cycle $q(0 \leq q < T_1)$ in stage p, then its 1-D schedule time is $\sigma(o, 0) = p * T_1 + q$.

The 1-D schedule needs to respect the following constraints, which are an extension of the constraints for the single-II case:

(1) Modulo property:

$$\sigma(o, i_1) + T_1 = \sigma(o, i_1 + 1) \tag{24}$$

(2) Dependence constraints: for every dependence $(o_1 \rightarrow o_2, \delta, \langle k \rangle)$ in the simplified DDG, let $x = min(level(o_1), level(o_2))$, $y = max(level(o_1), level(o_2))$. Suppose $\sigma(o_1, 0) = p_1 * T_1 + q_1$, and $\sigma(o_2, 0) = p_2 * T_1 + q_2$, where $0 \leq q_1, q_2 < T_1$. Then

$$\begin{cases} k * T_y + (p_2 - p_1) * T_n + q_2 - q_1 \geq \delta & \text{if } \sigma(o_2, 0) \geq \sigma(o_1, 0), \\ k * T_y + (p_2 - p_1) * T_x + q_2 - q_1 \geq \delta & \text{otherwise.} \end{cases} \tag{25}$$

(3) Resource constraints: At any modulo cycle in K_1, no hardware resource is allocated to more than one operation.

(4) Sequential constraints: if $n > 1$, then for every positive dependence with o as the source operation, δ being the dependence latency,

$$\begin{cases} (l_x + 1) * T_x - p * T_x - q \geq \delta & \text{if } x = n \text{ or } p > l_{x+1}, \\ f_{x+1} * T_x - p * T_x - q \geq \delta & \text{otherwise.} \end{cases} \tag{26}$$

where $x = level(o)$, o is scheduled into modulo cycle q in stage p.

(5) Kernel nesting constraints: (i) $l_1 \geq l_2 \geq \cdots \geq l_n \geq f_n \cdots \geq f_2 \geq f_1$, and $T_1 \geq T_2 \geq \cdots \geq T_n$ (ii) for any operation o, if it is at loop level L_n, then $stage(o) \in [f_n, l_n]$. Otherwise, suppose it is at loop level $L_x (x < n)$, then $stage(o) \in [f_x, f_{x+1})$ if o is in $OPSETA_x$, or $stage(o) \in (l_{x+1}, l_x]$ if o is in $OPSETB_x$.

The final schedule is hard to be described by a mapping function. It can be considered to be constructed in this way: first, use K_1 to construct the final schedule as usual. That is, rewrite the loop nest into a parallel loop nest with K_1 as the kernel. It is composed of n loops, L'_1, L'_2, $\cdots$, L'_n. Each loop L'_x corresponds to the original loop L_x. Second, remove the null cycles. At the loop level of L'_x, only kernel K_x is involved. For example, in the final schedule in Fig. 10, L'_1 is composed of K_1, and L'_2 involves only K_2 (The stages of this kernel permute, though). Therefore, the null cycles in the final schedule can be removed such that in L'_x, the kernel is "shrunk" from K_1 into K_x, as illustrated by the inner loop in Fig. 10. In practice, the two steps can be combined: only the operations within the involved kernel is generated. The null cycles above it and below it are simply not produced. The dependences between the operations in this kernel and those outside it have been considered conservatively by the dependence and sequential constraints, such that without the null cycles, the dependences are still respected in the final schedule.

EXAMPLE: Fig. 14 shows an example kernel nest with two IIs for the loop nest in Fig. 8(a). With the outermost loop kernel K_1 only, we have constructed a final schedule as shown in Fig. 10. Clearly, in L'_2, 2/3 of the total cycles are null cycles and unnecessarily wasted. After shrinking K_1 to K_2, we reach a more compact schedule. See the annotation to the right of the figure. Now there are two initiation intervals: an outer loop iteration is issued at the II of 3 cycles, but after entering L'_2, an inner loop iteration is issued at the II of 1 cycle. The transition is natural without any special handling of the pipeline.

The correctness of the final schedule is shown in Appendix of the technical memo [Rong et al. 2007], which also contains an algorithm for constructing a 1-D schedule with multiple IIs, and a brief introduction to loop rewriting (code generation).

As expected, scheduling with multiple IIs subsumes scheduling with a single II as a special case. When all IIs are equal to a single value, say T, all the constraints for 1-D

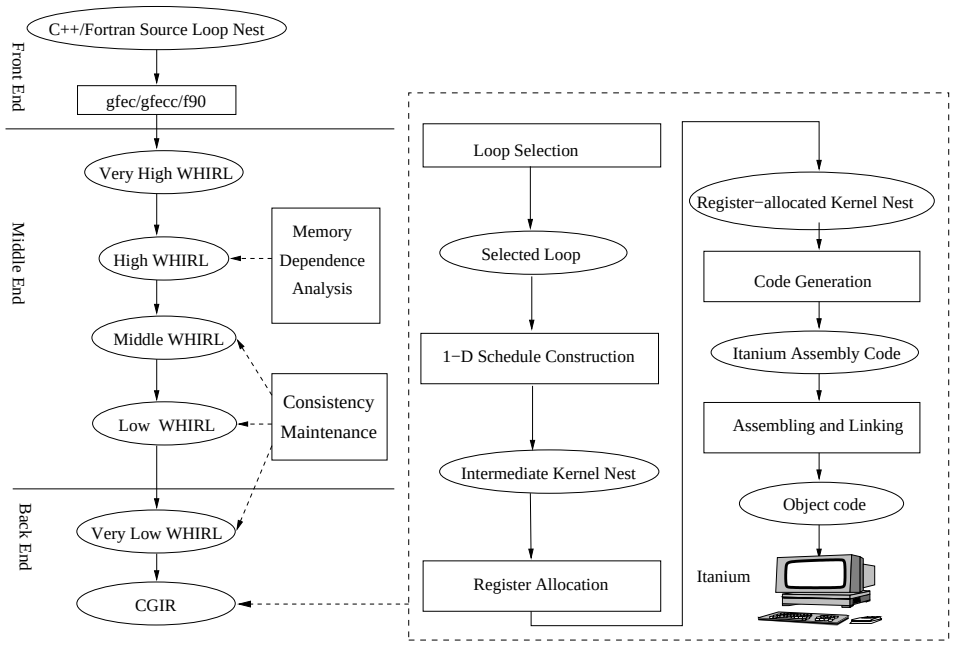


Fig. 15. Compile Flow

schedule construction become equivalent to those in Section 6.3. The two final schedules are of course the same, since their basic building blocks, the 1-D schedules, are the same and have no null cycles to remove.

7. EXPERIMENTS

The SSP framework, including loop selection (with parallelism as the criterion), scheduling, register allocation, and code generation, was implemented in the ORC 2.1 compiler for the IA-64 architecture. The resulting code is run on an IA-64 Itanium workstation with a 733MHZ processor, 2GB main memory, and 16KB/96KB/2MB L1/L2/L3 caches, and the actual execution time is measured. Parallelism serves as the first objective in loop selection, and ties are broken by data reuse, which is estimated manually by considering an abstract cache level with a line size of l according to Section 4.2.2.

Fig.15 shows the compile flow. The intermediate representation of the ORC compiler, termed as *WHIRL*, has 5 levels: Very High, High, Middle, Low, and Very Low levels, with increasing details and machine dependent information. The code generator translates the Very Low WHIRL to its own internal representation (CGIR) that matches the target machine instructions. Our implementation involves work from High WHIRL to CGIR.

At High level, it is relatively easy to get the multi-dimensional memory dependence information. This information is transmitted to the Very Low level throughout the process of Middle and Low levels, where many memory-related optimizations may happen. The information has to be consistently maintained during these optimizations.

At CGIR level, all instruction-level details have been exposed. The register dependences, and the memory dependences inherited from the High level, are combined together

to build a complete DDG. Based on this DDG, a loop can be chosen either manually or automatically by estimating parallelism. Then for the selected loop, its n-D DDG is simplified to be 1-D. Based on the 1-D DDG and the underlying resource constraints, a 1-D schedule is constructed with multiple IIs. It is represented as a multiple-II kernel nest, or more exactly, a multiple-II *intermediate kernel nest*. It is intermediate in that variables in it have not been assigned registers yet. The register allocator works on this kernel nest and outputs a *register-allocated kernel nest*. From it, the code generator generates the IA-64 assembly code. The assembly code is then assembled, linked, and run on the Itanium machine.

The scheduling algorithm is left to the appendix of the technical memo [Rong et al. 2007]. Register allocation and code generation are based on our previous work [Rong et al. 2005; Rong and Douillet et al. 2004], with minor extension to accommodate our generic loop nest model in Fig. 11.

In this work, we report performance results for two loop kernels from scientific applications as well as loop nests extracted from SPEC2000 floating point benchmarks. The loop kernels that we consider are matrix multiplication (MM) and 2-D hydrodynamics (HD) modified from the Livermore Loops. We also apply three loop transformations, viz., loop interchange (6 different versions), loop tiling and unroll-and-jam, for MM and test each independently. The cache misses are measured for performance analysis using the IA64 performance monitoring tool, Pfmon.

We extracted a number of loop nests from SPEC2000 floating point benchmarks. Each loop nest is wrapped as a function and called from the main routine with appropriate arguments. The function body, the loop nest, is compiled using our modified ORC compiler while the rest of the benchmark is compiled using gcc. This enables us to focus only on the implementation of SSP in the ORC compiler. The benchmarks are executed with the reference inputs of SPEC.

The loop level to be software pipelined by SSP can either be chosen by our compiler, or manually specified using a command line option. Let L_x be the loop level. We use $SSP - L_x^*$ to represents the first case where L_x is chosen by our compiler, and SSP-L_x the second case where it is specified.

To verify the accuracy of our loop selection methods, SSP is applied to every feasible loop level of a loop nest. For each case, we compare the performance of the SSP-compiled loop nest (referred to as SSP), with that of modulo scheduling (referred to as MS), and that without software pipelining at all (referred to as Serial). Specially, for the tiled (unroll-and-jammed) MM, Serial refers to the original loop nest without tiling (unroll-and-jam) or software pipelining being applied, while MS and SSP refer to the software pipelined schedules after the loop nest is tiled (unroll-and-jammed). MS has been implemented in the original ORC distribution based on slack scheduling [Huff 1993]. To test the effectiveness of SSP in the presence of other optimizations, the compiler optimization level is set to O3 (full optimization level) for all of Serial, MS, SSP.

Table I summarizes the average speedup for the loop nests tested. Speedup is defined as the execution time of a Serial loop nest divided by that of the optimized version (with MS, or with SSP applied to the selected loop level).

7.1 Performance of Kernel Loops

For MM, SSP always achieves the best speedup, with appropriate loop level being selected. Whether we apply loop interchange, tiling, or unroll-and-jam or no loop optimization at

	MS	SSP	SSP over MS
MM-ijk	0.91	3.04	3.33
MM-ikj	0.94	4.5	4.79
MM-jik	0.98	3.5	3.57
MM-jki	0.88	4.43	5.02
MM-kij	0.97	2.54	2.62
MM-kji	0.96	2.51	2.62
HD	1.14	1.22	1.07
MM-jki with loop tiling	2.14	4.29	2.01
MM-jki with unroll-and-jam	6.19	10.17	1.64
168.wupwise loop 1	0.91	0.88	0.97
171.swim loop 1	0.83	0.83	1.0
171.swim loop 2	1.37	1.04	0.76
171.swim loop 3	1.0	0.97	0.97
173.applu loop 1	1.15	1.77	1.54
173.applu loop 2	0.97	1.94	1.99
173.applu loop 3	1.16	1.66	1.43
173.applu loop 4	1.11	2.23	2.01
173.applu loop 5	1.11	2.49	2.24
173.applu loop 6	1.14	2.22	1.95
173.applu loop 7	1.26	1.70	1.35
173.applu loop 8	0.81	1.06	1.31
301.apsi loop 1	0.99	1.86	1.87
301.apsi loop 2	1.03	4.07	3.97

Table I. Average speedups

all, our method outperforms MS. Being able to work on a more profitable loop level, which is probably an outer loop level, allows the software-pipeliner to get around strong dependences or little data reuse opportunities of the innermost loop and to make use of the better properties of the other loops when possible. We discuss the performance in greater detail in the following subsections.

7.1.1 *Matrix Multiply.* We run SSP and MS on all the permutations of the matrix-multiply loop nest. The loop body is $A[i][j] += B[i][k] * C[k][j]$. The order of the loops in the nest are referred to as ijk, ikj, jik, jki, kij, and kji. Each loop order has different parallelism and data reuse potential. The performance results are depicted in Figure 16. We show the speedups achieved by MS and SSP for different matrix sizes.

For ijk and jik, the innermost loop is constrained by a recurrence cycle which limits the efficiency of MS. Consequently, applying SSP to other loop levels clearly achieves better performance. The upward tendency of the performance curves by software pipelining loop L_1 suggests that with the increase in the matrix size, the advantage of SSP's ability to retain data reuse becomes more important. For ikj and jki, the limited data reuse potential of one of the matrix operand of the innermost loop prevents MS to run efficiently when the size of the matrix increases. However, by software pipelining the outermost loop, such restriction is avoided. For kij and kji, all the loop levels show less data reuse potential or parallelism, limiting the speedup of all methods. However, software pipelining of the middle level still exhibits the best performance.

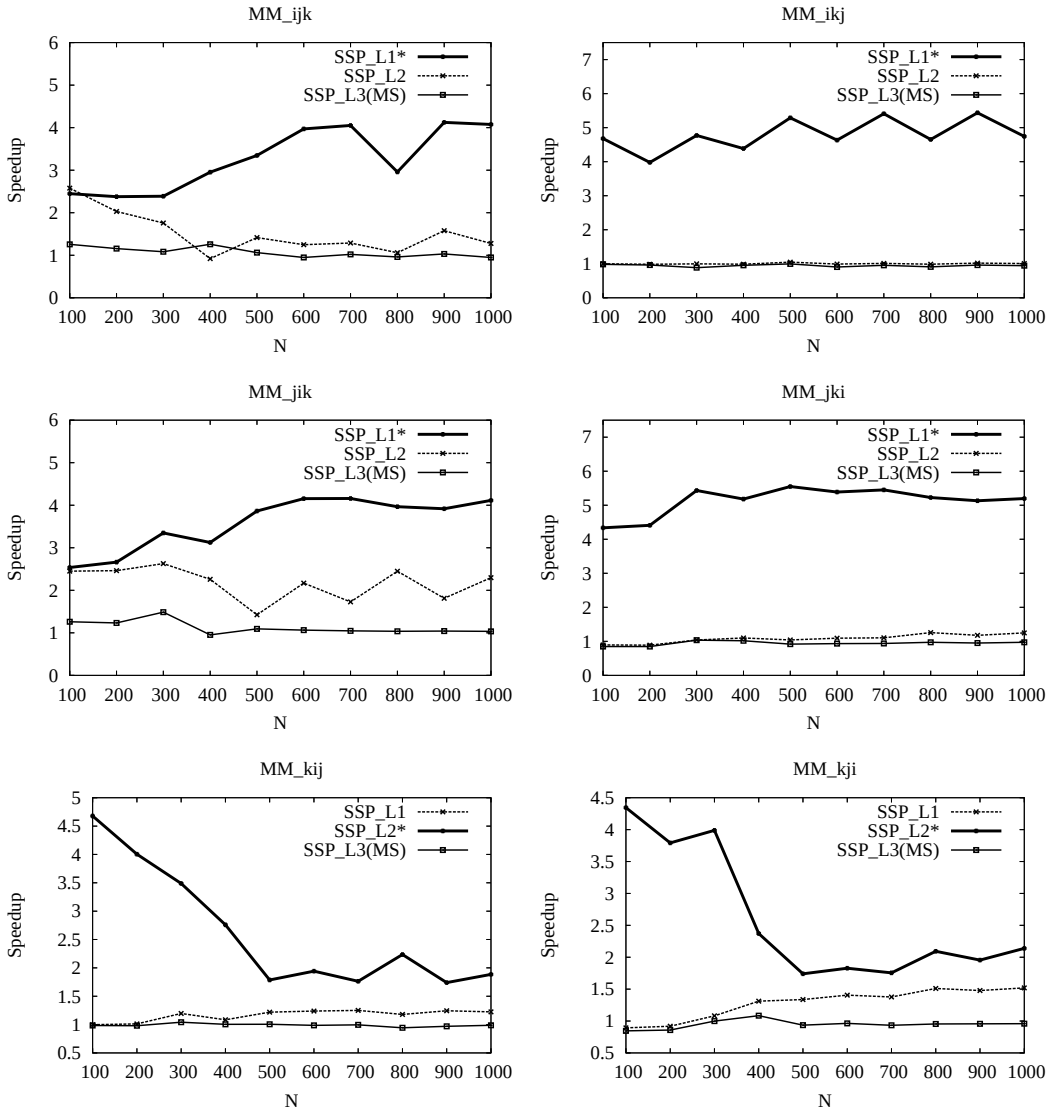


Fig. 16. Performance of Matrix Multiply, where $N * N$ is the Size of an Array

7.1.2 *Tiled and Unroll-and-Jammed Matrix Multiply.* Next we consider a classic tiled matrix multiply code [Wolf and Lam 1991] in which loop order is jki, and loops i and k are tiled [6]. The order of the loops is determined during tiling for best data locality. Hence only this order is considered in this experiment.

Figure 17 shows the performance for tiled matrix multiply with a constant tile size of 16 and the array sizes being multiples of it. After tiling, there is drastic improvement in

[6]This is the equivalent row-major code. The column-major order is ikj, and loops k and j are tiled.

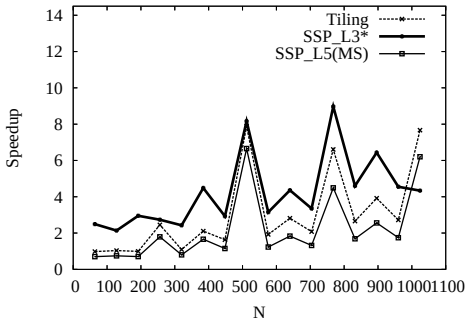


Fig. 17. Tiled MM

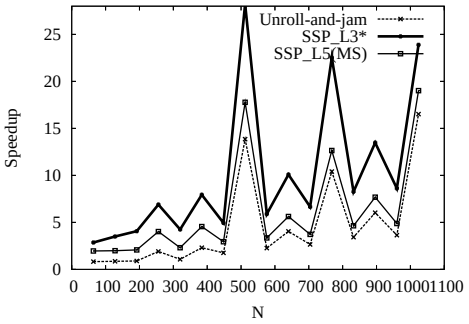


Fig. 18. Unroll-And-Jammed MM

speedup due to better data locality. The loop nest becomes 5-deep now. With software pipelining applied to the innermost loop, the performance is reduced by 38% on average. This is due to the overhead associated with the modulo schedule. Instead, when scheduling the third loop level, this overhead is amortized by the benefits from the longer execution time of the groups, and the natural overlapping of the draining and filling of adjacent groups.

Lastly, we consider an unroll-and-jammed version of MM (Figure 18). *Unroll-and-jam* [Carr and Kennedy 1994], also known as *register tiling*, attempts to match the available parallelism in the application with the hardware resources. It is usually performed upon tiled code to further explore register level data reuse. Like in tiling, after unroll-and-jam, software pipelining of the middle loop level results in significant improvement.

The advantage of SSP scheduling shows clearly in these two experiments. Although the loops tested are perfect in high-level language, they become imperfect in assembly level. After tiling and unroll-and-jam, the depth of the whole loop nest becomes deeper (from 3 to 5), and the inner loops have small loop counts. And thus it becomes important to efficiently schedule the operations that are not in the innermost loop. It is also important to offset the overhead of initialization, finalization, and filling and draining the pipeline. Due to the small loop count of the innermost loop, such overheads have significant impact on the performance. By scheduling a middle loop level, software pipelining can offset these overheads effectively. The relatively longer execution time of a group outweighs the overhead. On the other hand, the operations at every loop level have been considered during 1-D schedule construction such that each loop level has the smallest possible initiation interval. In contrast, MS mainly cares about the efficiency of running the innermost loop operations and its software pipelined kernel includes only such operations.

7.1.3 *Modified 2-D Explicit Hydrodynamics.* The benchmark kernel considered is a 2-D explicit hydrodynamic code modified from Livermore loops. In this experiment, we varied the upper bounds kn and jn, respectively, of the outer and the inner loops.

Figure 19 shows the performance for the hydrodynamics benchmark when $kn = jn$. Since there is no recurrence in either loop level, data reuse will play a more important role in the performance. When the loop trip counts are smaller than 400, the outer loop is more beneficial. However, with an increase in the matrix size, the inner loop is better.

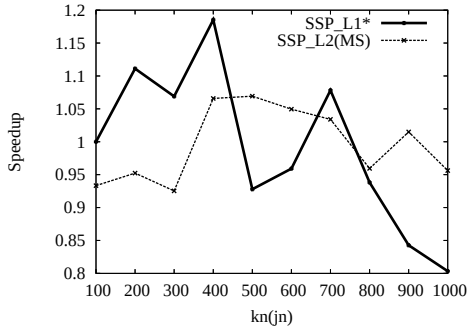


Fig. 19. Hydrodynamics

7.2 Performance of SPEC Loops

There are many loop nests in the SPEC2000 floating point benchmarks. However, many of them could not be software pipelined at an outer loop level due to either sibling inner loops, or function calls inside. Other loops have non-rectangular iteration spaces. We do not consider these loops.

In the loop nests extracted for experimentation, most of them have too short execution time and small loop counts (typically, less than 50 for a loop) to show meaningful performance improvement. However, they are perfectly fine for testing the correctness and effectiveness of our register allocation approach and heuristics, which we have reported in [Rong et al. 2005]. Here we report the performance of 14 loop nests from `168.wupwise`, `171.swim`, `173.applu`, and `301.apsi`, which have loop depths varying from 2 to 4 and have reasonable trip counts.

The results have been summarized in Table I for the appropriate chosen loop level. Here we study the results in more detail.

For 168.wupwise, there is a 3-deep critical loop nest. Scheduling the middle loop level results in a speedup of 0.97 over MS(i.e., a slowdown from 86.127 to 88.668 seconds). Software pipelining the outermost loop is possible, but register allocation method fails due to excessive integer register pressure.

For 171.swim, the table has shown the speedups for several loops. The least execution time within them is 32 seconds. Software pipelining the outer loop levels results in performance slowdown. However, in 173.applu, it improves performance by 30%-120%.

The two loop nests in 301.apsi have significant speedups when scheduling the outer loop levels: the first loop nest has good locality available in the outer loop, while the second loop nest has strong dependence cycles in the inner loop level.

The results suggest that any loop level, including the innermost one, may be the best or worst choice for software pipelining. Again, software pipelining should not be applied blindly to any loop level and loop selection is necessary, and the ability of SSP to exploit ILP from an arbitrary loop level is important.

7.3 Performance Analysis from Cache Misses

In this section, we investigate the cache effects of the different methods under comparison, with MM as an example. To correctly link the cache behavior to performance, we need to evaluate the relative weight of the cache misses at each cache level. Fig. 20 shows the

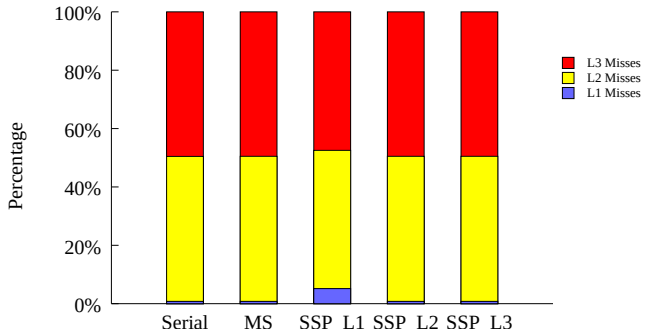


Fig. 20. Percentages of the Three Level Cache Misses in MM-ikj with Matrix Sizes 1024*1024

relative weight of the cache misses at the 3 cache levels in the Itanium processor for ikj matrix multiply. Almost all cache misses happen at L2 and L3 caches for every method. This is the common trend in all the benchmarks tested.

This is a paradox as the first sight, since a cache miss at a low level must be caused by a cache miss at a high level, and thus L1 cache misses should be higher than L2. However, this does not hold for Itanium architecture. For this architecture: (1) There are separate L1 instruction cache and L1 data cache. And a floating point memory operation bypasses L1 data cache. Since our experiments are performed for scientific applications, and have only floating-point memory operations, L1 cache misses are mainly caused by instruction fetch. (2) The loop nests experimented are small in code size. Although SSP suffers from code expansion due to the lack of hardware support, the generated code is small enough to fit in the 16K bytes L1 instruction cache. Therefore, practically there are only a few instruction cache misses at the start of the loop nest.

From the above discussion, it is clear that we must focus on L2 and L3 cache misses in order to correctly explain the performance. For these experiments, we fix the matrix sizes as 1024×1024. The L2 and L3 cache misses (normalized with respect to the L2 and L3 cache misses of the serial code) are shown in Fig 21 and 22, where `jki+T` and `jki+UJ` refer to the tiled and unroll-and-jammed MM, respectively. They partly explain the performance improvement achieved by software pipelining a good loop level.

For all matrix multiply benchmarks, except the tiled and unroll-and-jammed versions, the L2 and L3 cache misses in SSP schedules at the outermost loop level are significantly lower than those for MS schedules. These numbers reflect that software pipelining exploits better data locality by selecting an appropriate loop level.

The increase in L2 and L3 cache misses in SSP schedules at the middle loop level L_3 for the tiled and unroll-and-jammed benchmarks, at first, seems counter-intuitive as the SSP schedules have better speedups than the MS schedules for these benchmarks. However, in these two cases, MS affords cost in frequent pipeline filling and draining, and fails to schedule outer loop operations more effectively. The aggressive parallel execution of several iterations in SSP causes more cache pressure, leading to larger cache misses. The increase in cache misses affects the performance. Such effect is overcome partly by the increased parallelism in SSP schedules: when there are enough independent instructions that can be executed, the latencies due to cache misses can be masked. On the other hand, the Itanium architecture stalls only on uses. This also helps to overcome the effects of

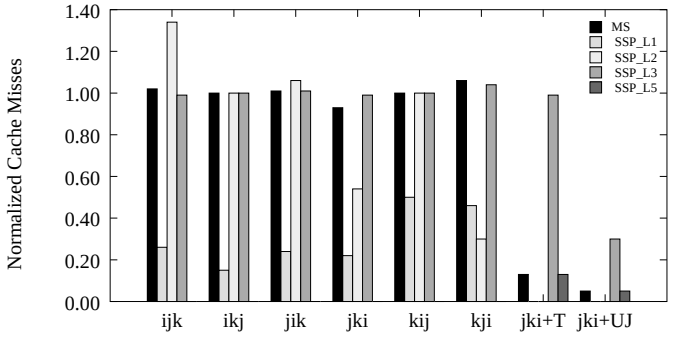


Fig. 21. L2 Cache Misses

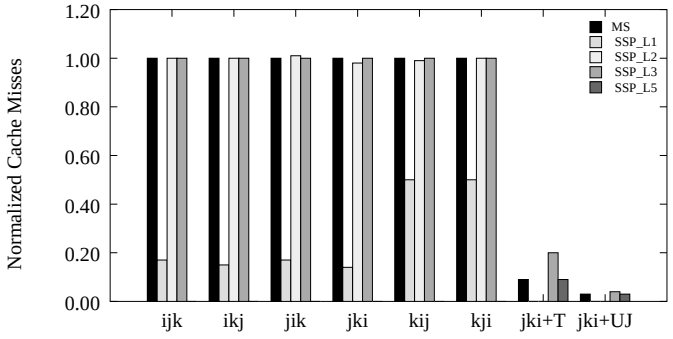


Fig. 22. L3 Cache Misses

increased cache misses. The results also suggest that software pipelining tiled loops should consider the tile size to avoid negative cache misses, which we leave for future study.

8. RELATED WORK

Most software pipelining algorithms [Intel 2001; Allan et al. 1995; Huff 1993; Rau 1994; Rau and Fisher 1993] focus on the innermost loop, and do not consider cache effects. The most commonly used method, modulo scheduling, is a special case of our approach.

A common extension of modulo scheduling from single loops to loop nests, including *hierarchical reduction* [Lam 1988], *Outer Loop Pipelining* [Muthukumar and Doshi 2001], and *pipelining-dovetailing* [Wang and Gao 1996], is to apply modulo scheduling hierarchically in order to exploit the parallelism between the draining and filling phases of adjacent outer loop iterations. In scheduling a loop, the DDG of its own is used.

In comparison, our method considers cache effects. The DDG is always the 1-D simplified DDG for the chosen loop, whatever loop level is currently under scheduling. The draining and filling phases are naturally overlapped without any special treatment.

Loop Tiling [Wolf and Lam 1991] maximizes data locality, instead of parallelism. Loop unrolling duplicates the loop body of the innermost loop to increase instruction-level parallelism. Both methods are complementary to SSP.

One question is: *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 performance due to different data reuse patterns. And for this interchanged loop nest, the choice for a good loop level might still be made by considering and comparing all the loop levels. 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.

Another question is: *What is the difference between our method and the one that tiles the selected loop, and then software pipelines the new innermost loop with MS?* In this case, the trip count of the new innermost loop is usually small as a result of tiling, and it is critical to hide the overhead of initialization, prolog, epilog, and finalization of the software pipelined innermost loop. Software pipelining an outer loop leads to less overhead, as discussed in Section 1 and confirmed in the experiments with tiled and unroll-and-jammed MM.

Unroll-and-jam [Carr and Kennedy 1994] has been applied to improve the performance of software pipelined loops [Carr et al. 1996]. The outer loop is unrolled but it is still the innermost loop that is software-pipelined. The $RecMII$ still strongly depends on the recurrences in the innermost loop, though reduced by the unroll factor. Unroll-and-squash first applies unroll-and-jam to a nested loop, and then reduce the code size of the jammed innermost loop by software pipelining and hardware support (rotating registers) [Petkov et al. 2002]. SSP is different from unroll-and-squash in the following ways: (1) the unroll-and-squash method presented in [Petkov et al. 2002] appears to be limited to 2-deep loop nest; (2) it does not overlap the epilog and prolog between successive outer loop iterations; and (3) it decides the unroll factor first, and then software pipelines the innermost loop.

In general, loop transformations, such as interchange, tiling, and unroll-and-jam, are orthogonal to our approach and can be applied independently. In Section 7, we have shown that our approach is beneficial with these loop transformations applied beforehand.

Hyperplane scheduling [Lamport 1974] 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 [Darte et al. 2002]. 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 [Darte and Robert 1994; Ramanujam 1994; Gao et al. 1993] 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 [Banerjee 1993]. Furthermore, they consider neither resource constraints nor cache effects.

Multi-dimensional retiming [Passos and Sha 1996] translates a loop nest to be fully parallel without resource constraints.

Unimodular and non-unimodular transformations [Banerjee 1993; Feautrier 1996] mainly care for coarse-grain parallelism or the communication cost between processors.

Fine-grain wavefront transformation [Aiken and Nicolau 1990] combines loop quanti-

zation and perfect pipelining to explore coarse and fine-grain parallelism simultaneously, based on outer loop unrolling and repetitive pattern recognition.

9. CONCLUSIONS

We have introduced the fundamental theory of software pipelining a loop nest at an arbitrary level that has a rectangular iteration space and has no sibling inner loops in it. This approach reduces the challenging problem of n-dimensional software pipelining into a simpler problem of 1-dimensional software pipelining. This approach provides the freedom to search for and schedule the most profitable loop level, where profitability can be measured in terms of parallelism, data reuse potential, or any other criteria.

This approach subsumes the classical modulo scheduling as a special case. It also extends the traditional hyperplane scheduling to handle resource constraints. We have extended this approach to schedule imperfect loop nests. Multiple initiation intervals can be naturally achieved to issue operations at different loop levels in their fastest initiation rates.

We have demonstrated the correctness and efficiency of our method. Future work needs to study the interaction of this approach with other loop nest transformations like tiling, unroll-and-jam, and loop interchange more extensively, and extend the loop nest model to allow sibling inner loops at one level. Loop selection is another area to explore. This paper presented the principles in loop selection. In future, other data reuse models and other objectives like power consumption need to be investigated.

ACKNOWLEDGMENTS

We are grateful to Bogong Su, Hongbo Yang, Chan Sun C., and the anonymous reviewers for their valuable advice.

REFERENCES

AIKEN, A. AND NICOLAU, A. 1990. Fine-grain parallelization and the wavefront method. In *Selected Papers of the 2nd Workshop on Languages and Compilers for Parallel Computing*. Pitman Publishing, London, UK, 1–16.

ALLAN, V. H., JONES, R. B., LEE, R. M., AND ALLAN, S. J. 1995. Software pipelining. *ACM Computing Surveys 27*, 3 (September), 367–432.

ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. 1983. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, New York, NY, USA, 177–189.

BANERJEE, U. K. 1993. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, USA.

CARR, S., DING, C., AND SWEANY, P. 1996. Improving software pipelining with unroll-and-jam. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*. IEEE Computer Society, Washington, DC, USA, 183.

CARR, S. AND KENNEDY, K. 1994. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. on Prog. Lang. and Systems 16*, 6 (Nov.), 1768–1810.

CARR, S., MCKINLEY, K. S., AND TSENG, C.-W. 1994. Compiler optimizations for improving data locality. In *ASPLOS-VI: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, USA, 252–262.

DARTE, A. AND ROBERT, Y. 1994. Constructive methods for scheduling uniform loop nests. *IEEE Transactions on Parallel and Distributed Systems 5*, 8 (Aug.), 814–822.

DARTE, A., SCHREIBER, R., RAU, B. R., AND VIVIEN, F. 2002. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Trans. Des. Autom. Electron. Syst. 7*, 1, 159–172.

FEAUTRIER, P. 1996. Automatic parallelization in the polytope model. *Lecture Notes in Computer Science 1132*, 79–103.

GAO, G. R., NING, Q., AND VAN DONGEN, V. 1993. Software pipelining for nested loops. ACAPS Tech Memo 53, School of Computer Science, McGill Univ., Montréal, Québec. May.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Prog. Lang. and Syst. 21,* 4, 703–746.

GOVINDARAJAN, R., ALTMAN, E. R., AND GAO, G. R. 1996. A framework for resource-constrained rate-optimal software pipelining. *IEEE Trans. on Parallel and Distributed Systems 7,* 11 (November), 1133–1149.

HUFF, R. A. 1993. Lifetime-sensitive modulo scheduling. In *PLDI'93: Proc. of the ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 258–267.

INTEL. 2001. *Intel IA-64 Architecture Software Developer's Manual, Vol. 1: IA-64 Application Architecture*. Intel Corporation, Santa Clara, CA, USA.

KENNEDY, K. AND MCKINLEY, K. S. 1992. Optimizing for parallelism and data locality. In *ICS '92: Proceedings of the 6th International Conference on Supercomputing*. ACM Press, New York, NY, USA, 323–334.

LAM, M. 1988. Software pipelining: an effective scheduling technique for vliw machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 318–328.

LAMPORT, L. 1974. The parallel execution of DO loops. *Communications of the ACM 17,* 2 (February), 83–93.

MOON, S.-M. AND EBCIOĞLU, K. 1997. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems 19,* 6 (Nov.), 853–898.

MUTHUKUMAR, K. AND DOSHI, G. 2001. Software pipelining of nested loops. *Lecture Notes in Computer Science 2027,* 165–181.

PASSOS, N. L. AND SHA, E. H.-M. 1996. Achieving full parallelism using multidimensional retiming. *IEEE Trans. Parallel Distrib. Syst. 7,* 11, 1150–1163.

PETKOV, D., HARR, R., AND AMARASINGHE, S. 2002. Efficient pipelining of nested loops: unroll-and-squash. In *16th Intl. Parallel and Distributed Processing Symposium (IPDPS '02)*. IEEE, Fort Lauderdale, FL.

RAMANUJAM, J. 1994. Optimal software pipelining of nested loops. In *Proceedings of the 8th International Symposium on Parallel Processing*. IEEE Computer Society, Washington, DC, USA, 335–342.

RAU, B. R. 1994. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*. ACM Press, New York, NY, USA, 63–74.

RAU, B. R. AND FISHER, J. A. 1993. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing 7,* 9–50.

RONG, H., DOUILLET, A., GOVINDARAJAN, R., AND GAO, G. R. 2004. Code generation for single-dimension software pipelining of multi-dimensional loops. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 175–186.

RONG, H., DOUILLET, A., AND R.GAO, G. 2005. Register allocation for software pipelined multi-dimensional loops. In *PLDI'05: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA.

RONG, H., TANG, Z., GOVINDARAJAN, R., DOUILLET, A., AND GAO, G. R. 2004. Single-dimension software pipelining for multi-dimensional loops. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 163–174.

RONG, H., TANG, Z., GOVINDARAJAN, R., DOUILLET, A., AND GAO, G. R. 2007. Single-dimension software pipelining for multi-dimensional loops. CAPSL technical memo, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware. January. In ftp://ftp.capsl.udel.edu/pub/doc/memos/memo049.ps.gz.

WANG, J. AND GAO, G. R. 1996. Pipelining-dovetailing: A transformation to enhance software pipelining for nested loops. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction*. Springer-Verlag, London, UK, 1–17.

WOLF, M. E. AND LAM, M. S. 1991. A data locality optimizing algorithm. In *PLDI'91: Proc. of the ACM SIGPLAN 1991 Conf. on Prog. Lang. Design and Implementation*. ACM Press, New York, NY, USA, 30–44.

WOLF, M. E., MAYDAN, D. E., AND CHEN, D.-K. 1996. Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 274–286.

Received March 2006; May 2006; July 2006; accepted September 2006