University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Code Generation for Single-Dimension Software Pipelining of Multi-Dimensional Loops

Hongbo Rong†
Alban Douillet††
R.Govindarajan†
Guang R.Gao†

CAPSL Technical Memo 051
September 26, 2003

†Dept. of Electrical and Computer Engineering
University of Delaware
rong,govind,ggao@capsl.udel.edu
††Dept. of Computer Sciences
University of Delaware
douillet@capsl.udel.edu

Abstract

Traditionally, software pipelining is applied either to the innermost loop of a given loop nest or from the innermost loop to outer loops. In a companion paper [1], we have proposed a scheduling method, called *Single-dimension Software Pipelining (SSP)* for multi-dimensional loops, to software pipeline at an arbitrary loop level.

In this paper, we describe several research and engineering issues encountered in code generation for the SSP method, which are interesting in their own rights, and our solutions to them. In contrast to traditional software pipelining, code generation for the SSP method requires to handle two distinct repetitive patterns. This, in turn, complicates register assignment to overlapping live ranges and requires two levels of register assignment. As available hardware support in the form of rotating registers can only support one level of register renaming, our solution method to the register assignment problem is based on a combination of dynamic register renaming (using rotating registers) and static register renaming (using code replication). Last, code size increase, another potential issue and more intense for SSP than for traditional software pipelining, has also been addressed. The proposed code-size-optimized code generation limits the code size increase without sacrificing the performance significantly.

We present our code generation scheme in general, and subsequently target it for the IA-64 architecture, making effective use of rotating registers and predicated execution. We have implemented the code generation scheme for Itanium, henceforth called the *SSPCodeGen*. We present some initial experimental results which not only demonstrate the feasibility and correctness of the code generation scheme but also its code quality.

[1]Also submitted to this conference, and available online as a technical memo [26].

Contents

List of Figures

1 Introduction

Software pipelining of loop nests has been a challenging research topic. While numerous algorithms have been proposed for single (innermost) loops [9, 3, 12, 13, 16, 22, 20], only a few address software pipelining of loop nests [11, 13, 17, 19, 27]. They share the essential idea of scheduling the loop nests hierarchically, starting from the innermost loop to the outermost one. This approach, henceforth referred to as *innermost-loop-centric scheduling*, naturally extends the single loop scheduling method to the multi-dimensional domain. This approach has two major shortcomings: (1) It commits itself to the innermost loop first without considering how much parallelism the other loop levels have to offer. (2) It cannot exploit the data reuse potential in the outer loops [6, 7].

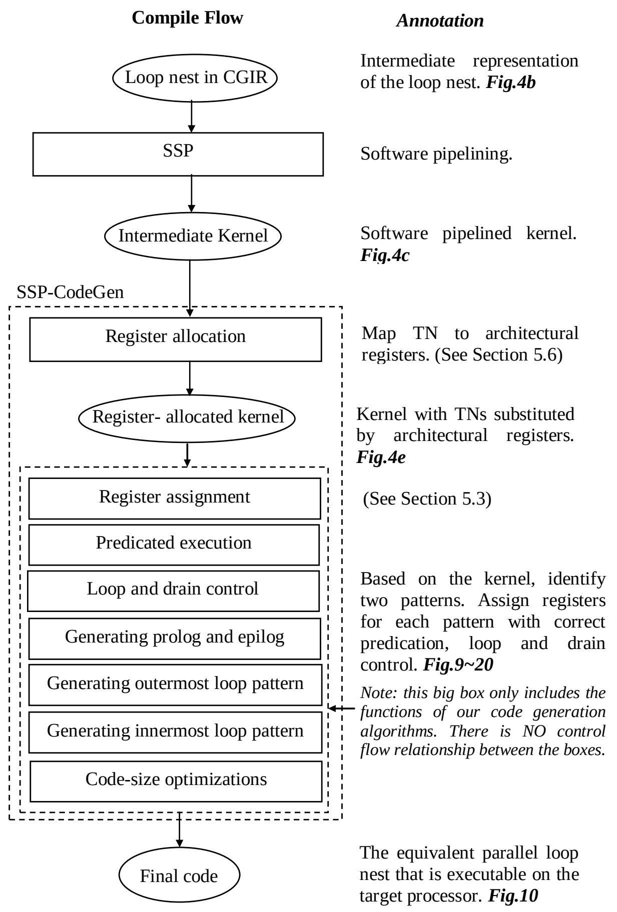
In a companion paper (also submitted to this conference) [26], we have introduced a *resource-constrained scheduling* method for software pipelining of loop nests – called *Single-dimension Software Pipelining for multi-dimensional loops (SSP)*. In contrast to the traditional innermost-loop-centric approach, SSP searches the entire loop nest and chooses the most profitable loop level to software pipeline, considering both the instruction-level parallelism and data reuse that can be exploited in order to reduce the actual execution time of the loop nest. SSP retains the simplicity of the classical modulo scheduling of single loops, yet achieves significantly higher performance than the traditional innermost-loop-centric approach.

This paper presents a code generation scheme for the SSP method which has not been discussed in the companion paper [26]. The code generation for the SSP method in the context of a modern compiler framework is shown in Fig.1, where each compile step is accompanied with a short description and an illustrative example. SSP takes the loop nest at CGIR (Intermediate Representation for Code Generation) level, and output a kernel – called *intermediate kernel* in the rest of this paper – for the selected (outer) loop level. The code generation scheme (the dotted box in the figure) presented in this paper is to translate the intermediate kernel into target machine code.

Code generation for the SSP method presents several interesting issues and this paper addresses them in an effective way. More specifically:

- The intermediate kernel generated by the SSP method leads to two repetitive patterns, independent of the number of loop levels in the loop nest. These patterns are referred to as the *outermost* and the *innermost repetitive patterns.* The presence of two repetitive patterns makes code generation for SSP a different and more challenging research problem.

- To ensure the correct execution of the program, overlapping live ranges corresponding to different instances of the same variable should be assigned different register names. Since the SSP method overlaps different iterations of outer loops which, in turn, may cause different instances of inner loop iterations to be overlapped, the live range overlap problem in SSP schedules is more involved than in software pipelining of single or innermost loops. Overlapping occurring at multiple loop levels necessitates a combination of



Compile Flow
Annotation
Loop nest in CGIR
Intermediate representation of the loop nest. Fig.4b
SSP
Software pipelining.
Intermediate Kernel
Software pipelined kernel. Fig.4c
SSP-CodeGen
Register allocation
Map TN to architectural registers. (See Section 5.6)
Register- allocated kernel
Kernel with TNs substituted by architectural registers. Fig.4e
Register assignment
(See Section 5.3)
Predicated execution
Loop and drain control
Generating prolog and epilog
Generating outermost loop pattern
Generating innermost loop pattern
Code-size optimizations
Based on the kernel, identify two patterns. Assign registers for each pattern with correct predication, loop and drain control. Fig.9~20
Note: this big box only includes the functions of our code generation algorithms. There is NO control flow relationship between the boxes.
Final code
The equivalent parallel loop nest that is executable on the target processor. Fig.10

Figure 1: Compile Flow

dynamic register renaming (using rotating registers) and static register renaming (using code replication).

- Traditional hardware support for software pipelining, such as rotating registers and predicated execution, are aimed at software pipelining a single loop level. While they are useful, as mentioned earlier, the requirements for SSP code generation method go beyond these.

- Code size increase in SSP schedules is much higher than that in traditional software pipelining. The challenge is how to reduce the code size increase while retaining the performance benefits of SSP methods.

We address the above issues in this paper and outline our solutions to the code generation problem. We discuss the code generation scheme is detail and target it for the IA-64 architecture, taking advantage of the available hardware support for software pipelining, such as rotating registers and predicated execution. . We call our implementation $SSPCodeGen$.

We present initial experimental results which demonstrate the feasibility and the correctness of the code generation scheme. It also reveals the code quality and the performance of the proposed scheme. We remark that the issues discussed in this paper are interesting in their own right, and the solutions presented are important as they make SSP a viable approach for modern high performance architectures.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts and briefly reviews modulo scheduling and SSP. Then we motivate our study by a simple example in Section 3. Section 4 outlines our code generation method, while Section 5 presents in details the algorithms for IA-64 architecture. Section 6 presents extensions and optimizations to the basic method. Experimental results are reported in Section 7. A discussion on future work, related work and concluding remarks are then presented in Sections 8, 9, and 10.

2 Background

In this section we introduce technical terms that will be used throughout the paper. Then we quickly introduce the notion of software-pipelining, modulo-scheduling and SSP.

2.1 Definitions

An *n-deep perfect loop nest* is composed of loop $L_1, L_2, \cdots, L_n$ from the outermost to the innermost level respectively. Each loop $L_x (1 \leq x \leq n)$ has an index variable i_x and an index bound $N_x \geq 1$. The index is normalized to increment from 0 to $N_x - 1$ by unit step. The loop body is assumed to have no branches.

The *iteration space* of the loop nest is a finite convex polyhedron [28]. A node in the iteration space is called an *iteration point*, or simply *point*, and is identified by the index vector $\boldsymbol{I} = (i_1, i_2, \cdots, i_n)$. The instance of any operation o of $\boldsymbol{I}$ is denoted by $o(\boldsymbol{I})$. An L_x *iteration* is one execution of the L_x loop. Thus the L_x loop has a total of N_x number of iterations. One such iteration is also an iteration point if L_x is the innermost loop.

2.2 Modulo-Scheduling

Software pipelining [9, 2, 6, 11, 12, 13, 17, 22, 20, 27] exposes instruction-level parallelism by overlapping successive iterations of a loop. Modulo scheduling (MS) [12, 13, 22, 20] for single loops is an important and probably the most commonly used approach of software pipelining. A detailed introduction to modulo scheduling can be found in [4].

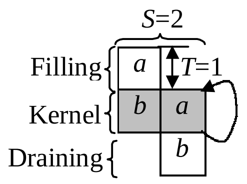
In modulo scheduling, instances of an operation from successive iterations are scheduled with an *Initiation Interval* of T cycles. Each iteration has a schedule length of l, and correspondingly divided into $S = \lceil \frac{l}{T} \rceil$ number of *stages*. Each stage takes T cycles.

The schedule consists of three phases: *filling the pipeline* (also known as *prolog*), *kernel*, which is executed repetitively, and *draining the pipeline* (also known as *epilog*). In the kernel, each stage is from a different iteration, and there is a total of S stages.

Example: Fig. 2(a) shows a perfect loop nest example. Fig. 2(b) shows a modulo-schedule for the innermost loop with $T = 1$, and $S = 2$.We assume that we have enough functional units available and that each operation takes one cycle to execute. Fig. 2(c) shows the corresponding parallelized loop nest. Note that only operations within the same innermost loop iteration overlap.

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

(a) Loop Nest Example



(b) Modulo Schedule for the Innermost Loop

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

(c) Modulo Schedule of the Loop Nest

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

(d) Extended Modulo Schedule of the Loop Nest

Figure 2: Example: MS and xMS

For better performance, it is sometimes possible to overlap the draining and filling part from two successive iterations of the innermost loop. The technique is an optimization on top of MS and is presented in [17]. It is an innermost-loop-centric approach for scheduling loop nests. We will refer to it as *eXtended Modulo-Scheduling* (xMS). When the required conditions for xMS scheduling are satisfied, the xMS schedule corresponding to the innermost loop schedule in Fig. 2(b) can be derived as shown in Fig. 2(d). The code should be correctly predicated to assure the correctness of the draining code (not shown in the figure). The cost of filling and draining the pipelines only occurs for the first and the last iteration of outer loop.

2.3 Single-dimension Software Pipelining of Multi-dimensional Loops

As opposed to MS and xMS, which always software pipeline from the innermost loop, SSP software pipelines a loop nest at an arbitrary loop level.

SSP schedules a loop nest in 3 steps:

1. It searches for the most profitable loop level in the loop nest. Profitability can be measured in terms of initiation rate, data reuse potential, or both, which have been addressed in the companion paper [26]. Any other objective, like power consumption, may also be feasible.

2. It reduces the multi-dimensional DDG (Data Dependence Graph) of the selected loop to a 1-dimensional (1-D) DDG. Based on the dependences and the resource constraints, it constructs a modulo schedule, referred to as the *1-D schedule*, for the operations in the loop nest. No matter how many inner loops the selected loop level has, it is scheduled as if it were a single loop. The 1-D schedule is represented by an *intermediate kernel* in this paper.

3. Based on the resulting 1-D schedule, it derives a final schedule, which specifies the schedule time of operations of the multi-dimensional loop. We refer to this final schedule as *SSP schedule* in this paper.

For details of scheduling perfect and imperfect loop nests, the reader is referred to [26] and [25], respectively. Without loss of generality, we assume that the outermost loop L_1 is selected and scheduled. Step 3 in the SSP method specifies the schedule time for any operation o and iteration point $\boldsymbol{I}=(i_1, i_2, \ldots, i_n)$ as:

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

where $\sigma(o, i_1)$ is the schedule time for any operation instance $o(i_1)$ in the 1-D schedule, S and T be the number of stages and initiation interval of the 1-D schedule, respectively, and $N_{n+1}=1$. The first two terms of the RHS of Equation 1 specify the schedule time of the operation within the same L_1 iteration. The last term *pushes* the operations in later iterations in the schedule in order to satisfy resource constraints. Note that the above schedule function is for perfect loop nests. The schedule function for imperfect loop nest is presented in [25].

Example: We consider the loop nest in Fig. 2(a) as an example and assume that the outermost loop is selected for software pipelining and that it has been scheduled. Furthermore we assume that the kernel found by SSP is the same as that in Fig.2(b) (the shaded part). Fig 3 shows the final schedule of SSP, with $N_1 = 6$ and $N_2 = 3$ for simplicity. We note that SSP overlaps the execution of different iterations of the outermost loop. For example, $b(0,0)$ and $a(1,0)$ at cycle 1 are from the first and second outermost loop iterations, respectively. The schedule exploits the same amount of instruction-level parallelism as xMS does in Fig. 2(d).

We must point out here that even if the ILP exploited is the same, the data reuse in our schedule is much different from MS or xMS. According to our loop selection criteria, the loop with favorable data reuse is selected. This may result in higher cache hits, which, in turn, may result in reducing the actual execution time.

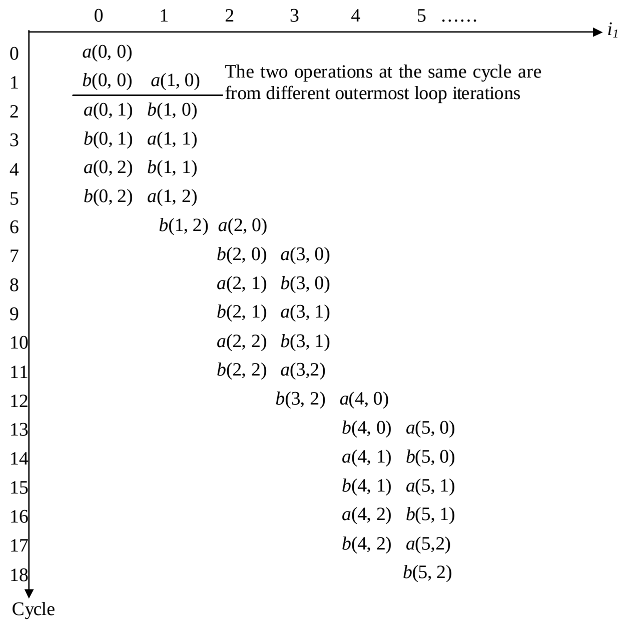


Figure 3: Example: SSP Schedule ($N_1 = 6$, $N_2 = 3$)

2.4 IA64 support for modulo scheduling of single loops

The IA-64 architecture provides support for modulo scheduling of *single* loops in terms of rotating register file, predication, and instruction set extension [9]. We briefly review the features that are the most relevant for this paper.

Register rotation provides each iteration with its own set of registers. A register is renamed by adding the name of the register to the value of *rotating register base (rrb)* modulo the size of the rotating register file [17]. The effect of register rotation is that the value in register x before rotation will appear to be located in register $x + 1$ after it. If x is the highest addressed rotating register, its value will wrap to the lowest addressed rotating register.

There are three sets of rotating register files in the IA-64 architecture: predicate, floating-point, and general purpose. Each has its own rrb and number of registers. One register rotation will increment (rotate) all rrbs simultaneously. The rotating predicate register file consists of predicate registers p16–p63, and are used to control the filling and draining of the pipelines. P16 is the predicate for the rightmost stage in the modulo scheduled kernel by default, and p17 for the second rightmost stage, etc.

The operation [2] used in this paper to rotate the register files is `br.ctop`. It branches to the target based on the values of loop count application register LC, and the epilog count application register EC. The behavior of this operation can be summarized as follows:

1. When $LC > 0$, the operation will decrement LC, rotate the registers, set the p16 register to 1 so that the next iteration is to be issued and executed, and the branch is taken.

[2]In this paper, we use *operation* and *instruction* interchangeably.

2. When $LC = 0$ and $EC > 1$, the operation will decrement EC, rotate the registers, make set the p16 register to 0 so that no new iterations will be issued, and the branch is taken.

3. When $LC = 0$ and $EC = 1$, the operation will rotate the registers, and the control will fall through the branch instruction.

4. When $LC = EC = 0$, the control simply falls through without rotating the registers.

The assembly code used in this paper follows IA-64 conventions. Independent instructions are grouped together and each group is delimited by a *stop bit* denoted by ';;'. Within a group of independent instructions, instructions are grouped in *bundles*, following some predefined templates [9]. For clarity reasons, the bundling will not be shown.

3 MOTIVATION AND PROBLEM STATEMENT

In this section, we motivate our code generation method with a simple example. Using this example, we illustrate the issues involved in code generation for SSP schedules and present problem statement.

3.1 A Motivating Example

Fig.4(a) shows a perfect loop nest. After transforming it into an equivalent internal representation for code generation (CGIR), however, it becomes imperfect. The CGIR code for the considered loop nest is shown in Fig. 4(b), where the `for` loops are shown in pseudo code rather than low-level intermediate representation for ease of understanding. Every register in the CGIR is a *logical register*, i.e., *Temporary Name (TN)*. TN{1} refers to the instance of the TN in the next outermost loop iteration.

The SSP method schedules this internal representation and outputs an intermediate kernel, in the form shown in Fig.4(c). The scheduling process of an imperfect loop nest is similar to that of a perfect loop nest [26]. Details are documented elsewhere [25].

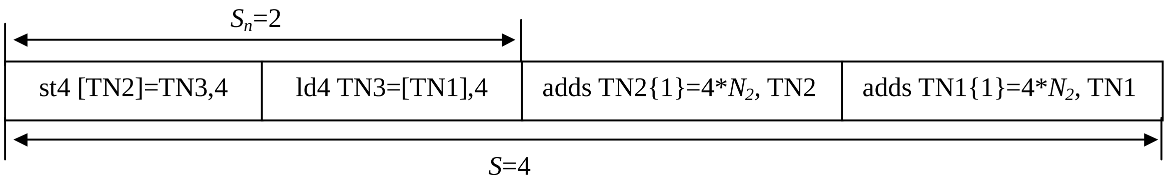
Similar to traditional software pipelining, a register allocator identifies the live ranges of each TN, and maps it to an architecture register r visible to the compiler. Fig. 4(d) shows one register allocation plan, and Fig.4(e) is the corresponding register-allocated kernel for the IA-64 architecture. For instance $TN2$ becomes $r45$ in stage 1, $r46$ in stage 2, and so on. $TN2\{1\}$ in stage 2 is the register that will contain the $TN2$ value at the next iteration and at the same stage. Therefore $TN2\{1\} = r45$ for stage 2. The main problem is then to generate the final executable code in a compact form from the register-allocated kernel.

```
int U[N1][N2], V[N1][N2];
for(i1=0;i1<N1;i1++)
    for(i2=0;i2<N2;i2++)
        V[i1][i2] = U[i1][i2];
```

(a) An Example Loop Nest

```
L1: for (i1=0;  i1<N1; i1++) {
        a: add TN1{1}=4*N2, TN1
        b: add TN2{1}=4*N2, TN2
L2:     for (i2=0;  i2<N2; i2++) {
            c: ld4 TN3=[TN1],4
            d: st4 [TN2]=TN3,4
        }
```

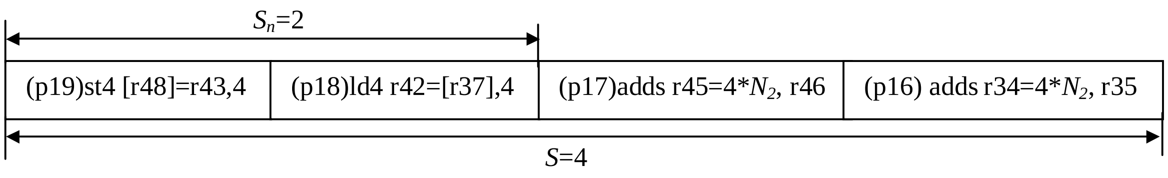
(b) CGIR



(c) Intermediate Kernel

TN	Register
TN1	r35
TN2	r45
TN3	r40

(d) Register Allocation



(e) Register-allocated Kernel

Figure 4: Example Compile Flow Before Code Generation

3.2 Assumption

3.2.1 Source Loop Nest

In this paper, the source loop nest for SSP is the loop nest shown in Fig. 5, where the loop depth $n > 1$ (When loop nest is 1, SSP is equivalent to the traditional modulo scheduling [26]).

Without loss of generality, we assume that the loop selected by SSP for scheduling is the outermost loop L_1.

In the loop nest, $OPSET_x$ represents a set of operations at CGIR level between the beginnings of two adjacent loops. We assume that there is no operation between the end of the two loops for simplicity reasons, although code generation for arbitrary loop nests is possible [25]. For the example in Fig. 4(b), $OPSET_1$ is composed of the two *add* operations and $OPSET_2$ is composed of *ld4* and *st4* operations.

The loop control, although presented in pseudo-code for clarity purposes, should be recorded in some data structure for storing the loop information.

In the following two sections, we assume that $OPSET_x$, $2 \le x \le n-1$, is empty to simplify our discussion. The code generation algorithms are then extended to the general cases when $OPSET_x$ are not necessarily empty in Section 6. Details are described in [25].

```
L1:  for (i1=0; i1 < N1; i1++){
         OPSET1
L2:      for (i2=0; i2 < N2; i2++){
             OPSET2
             ...
Ln:              for (in=0; in < Nn; in++){
                     OPSETn
                 } //end Ln
             ...
         } end L2
     } //end L1
```

Figure 5: Source Loop Nest

3.2.2 Intermediate Kernel

An SSP kernel consists of $S = \lceil \frac{l}{T} \rceil$ stages, where l is the length or execution time of a single iteration of the loop in the schedule and T is the initiation interval.Given the above loop nest, SSP will generate an intermediate kernel of S different stages, $A_{S-1}, \cdots, A_1, A0$ from left to right, including S_n leftmost stages that are executed by the innermost loop only (Fig. 6). Each stage takes T cycles. T is the *initiation interval* of the kernel. The reader is invited to read [25] for a more general case. At each cycle of a stage, there is a set of operations.

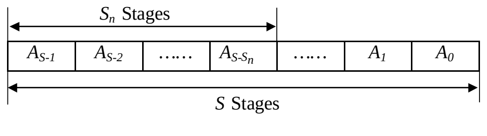


Figure 6: A Generic SSP Kernel

The S_n leftmost stages consist of operations from the innermost loop, i.e., from $OPSET_n$. Other stages consist of operations from the outermost loops, i.e. from $OPSET_1$ (the other $OPSETs$ are empty for the time being). With the intermediate kernel, a register allocator allocates registers to the kernel (cf. Section 5.6 and we get a register-allocated kernel (as illustrated in Fig. 4(e)).

3.3 Problem Statement

Now we state the code generation problem addressed in this paper as below:

> *Problem Statement:* Given an intermediate kernel generated by SSP and a target architecture, generate the final code such that the semantics of the original loop nest is preserved, and the code size and loop control overheads are reduced.

Several challenges arise in code generation. First, in the same way as is done in MS, we have to find repetitive patterns in the final schedule and use them to produce a compact code in a loop form, similar to the one shown in Fig. 2(c). As will be shown in Section 4.2 an SSP schedule exhibits two distinct patterns (independently of the number of loop levels), referred to as the outermost and the innermost loop patterns. Thus generating code for SSP schedule is a different and more challenging problem than code generation for software pipelined schedules for single loop levels. In addition to the repetitive patterns, the code generation scheme has to identify and generate code for the prolog and the epilog.

Second, multiple iterations of the outermost loop are executed in an overlapped manner in the SSP schedule. This, in turn, causes multiple iterations of the inner loops also to be executed in an overlapped manner. In order to ensure that the program executes correctly, and the overlapping live ranges of the same variable corresponding to different instances of the inner and outer loops use different registers, we require register renaming techniques similar to, but more complex than, the ones used in traditional modulo scheduling. Register renaming in modulo scheduling can be done by hardware, using rotating registers, or by software where multiple copies of the kernel are produced with static register renaming [13]. Register renaming for SSP schedules is more complex because to the following reasons: (1) Multiple repeating patterns and multiple iterations corresponding to different loops levels are executed in an overlapped manner in SSP. (2) Very often source-level perfect loop nests become imperfect at CGIR level due to address computation, initialization of reduction variables, etc. As will be discussed later, imperfect loop nests make it impossible to assign registers solely through hardware renaming like in traditional modulo scheduling. Hence we will have to use a combination of static register renaming through compiler and dynamic register renaming through rotating registers.

Third, in order to achieve high-performance code, we must efficiently map the schedule generated by SSP to the available hardware resources in the target architecture. However, some hardware supports, like specialized ISA and register rotation in the IA-64 architecture, are hard to consider in the scheduling phase. It is the code generation phase that should take advantage of them efficiently.

Last, we must reduce the cost of loop control overheads such as branch instructions, register copies, and filling/draining the pipelines. We also have to appropriately initialize the loop counters and bit masks in order to ensure correct program execution. Furthermore, like traditional software pipelining methods, SSP schedules also result in code size increases. Thus, the code generation scheme should limit the code size increase as far as possible without significant loss in performance.

In this paper, we propose to look at the code generation problem for an abstract architecture without assumption of any hardware support. We then direct the code generation scheme to the IA-64 architecture to make use of the available hardware support, i.e. rotating registers and predicated execution, which are originally for modulo scheduling of single loops, and use it for loop nests.

4 SSP CODE GENERATION OVERVIEW

In this section, we present a high-level overview of the code generation algorithm and explain the different phases of the algorithm based on the patterns appearing in the SSP schedule.

4.1 Input and Register Allocation Strategy

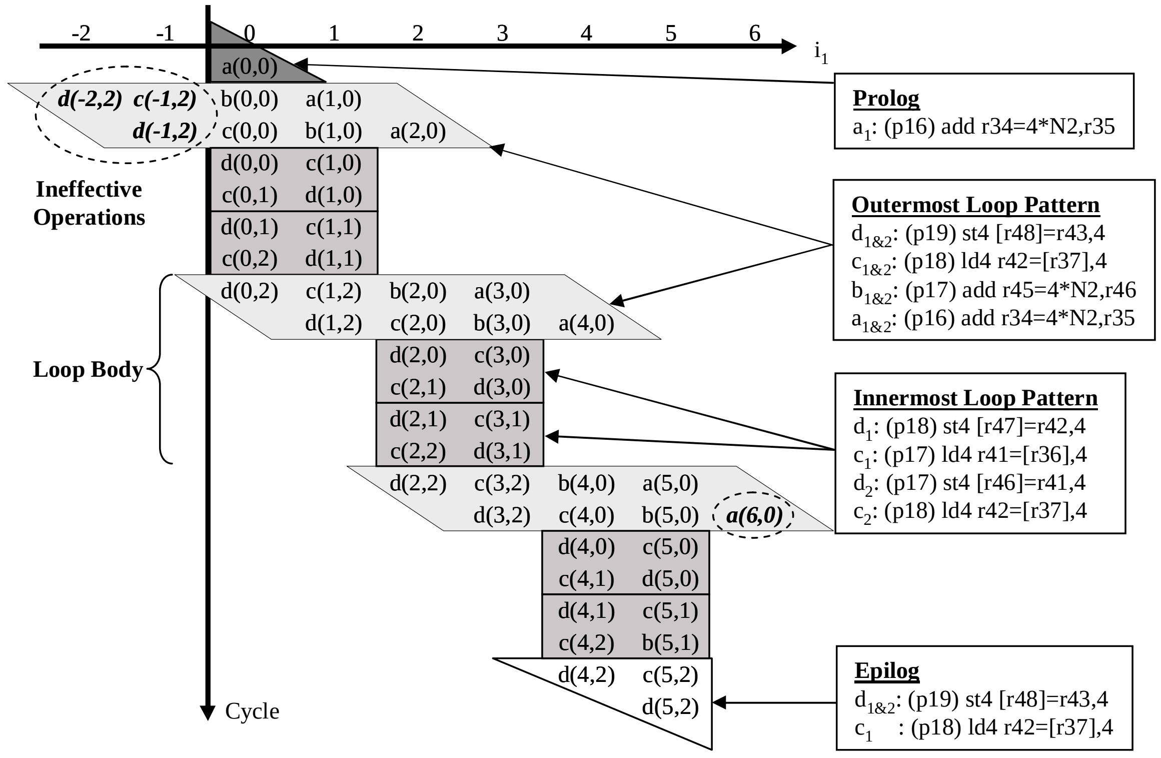
The code generation algorithm has only one input data: the intermediate kernel, which is produced by SSP as shown in Fig.1. TNs in this kernel are then assigned predicate registers and other registers (like the kernel code shown in Fig. 4(e)).

The register-allocated kernel will be used as a template throughout the entire subsequent code generation process. Thus in the following section, when we talk about kernel, we refer to the "register_allocated kernel" by default. The template might be used as-is or modified to fulfill register requirements. For instance it might be necessary to shift the indexes of the rotating registers, predicates and others. The full procedure is explained later in the paper.

4.2 Schedule Components

The first step to generate the code for the schedule is to identify the different phases involved. The SSP schedule can be partitioned into 4 separate components which we will refer to as the *prolog*, the *outermost loop pattern*, the *innermost loop pattern* and the *epilog*. These components for our example in Fig. 4 have been in the SSP schedule shown in Fig. 7. The corresponding IA-64 code for each component is shown in the right side of the figure, where o_1 means the operation o in the first cycle of the phase under consideration, etc.

There are only two loop patterns, independently of the number of loops. All operations in the outermost loop, i.e. $OPSET_1$ and $OPSET_n$, appear in the outermost loop pattern whereas



-2
-1
0
1
2
3
4
5
6
i_1
a(0,0)
d(-2,2) c(-1,2)
d(-1,2)
b(0,0) a(1,0)
c(0,0) b(1,0) a(2,0)
d(0,0) c(1,0)
c(0,1) d(1,0)
d(0,1) c(1,1)
c(0,2) d(1,1)
d(0,2) c(1,2) b(2,0) a(3,0)
d(1,2) c(2,0) b(3,0) a(4,0)
d(2,0) c(3,0)
c(2,1) d(3,0)
d(2,1) c(3,1)
c(2,2) d(3,1)
d(2,2) c(3,2) b(4,0) a(5,0)
d(3,2) c(4,0) b(5,0) a(6,0)
d(4,0) c(5,0)
c(4,1) d(5,0)
d(4,1) c(5,1)
c(4,2) b(5,1)
d(4,2) c(5,2)
d(5,2)
Ineffective Operations
Loop Body
Cycle
Prolog
a_1: (p16) add r34=4*N2,r35
Outermost Loop Pattern
$d_{1\&2}$: (p19) st4 [r48]=r43,4
$c_{1\&2}$: (p18) ld4 r42=[r37],4
$b_{1\&2}$: (p17) add r45=4*N2,r46
$a_{1\&2}$: (p16) add r34=4*N2,r35
Innermost Loop Pattern
d_1: (p18) st4 [r47]=r42,4
c_1: (p17) ld4 r41=[r36],4
d_2: (p17) st4 [r46]=r41,4
c_2: (p18) ld4 r42=[r37],4
Epilog
$d_{1\&2}$: (p19) st4 [r48]=r43,4
c_1 : (p18) ld4 r42=[r37],4

Figure 7: Phase-Partitioned SSP Schedule Example

only operations in the innermost loop, i.e. $OPSET_n$, appear in the innermost loop pattern. The outermost loop pattern has two purposes: it drains the previous group of outermost loop iterations while issuing the next group. The innermost loop pattern takes care of executing the innermost loop of the current group of iterations.

These two patterns together form another repetitive pattern that we refer to as the *loop body* of the outermost loop in the parallel loop nest after code generation.

Note that to make the outermost loop pattern appear repetitively, ineffective operations need to be added. The ineffective operations are circled in Fig. 7. They are ineffective because their first indexes are beyond the legal range of i_1, the outermost loop index variable (The range is assumed to be [0,6) in this example). In the IA-64 architecture, predicate registers will be used to make them ineffective when necessary.

A pattern is repeated periodically, as seen in Fig 7, and thus in code generation, we can use loop structure and avoid code duplication.

Although it is not directly linked to the code generation, it is interesting to note that the inner loops runs sequentially whereas the outermost loop iterations are overlapped. Also the end of a group of L_1 iterations overlap with the beginning of the next group of L_1 iterations without manual intervention or post-pass optimization, unlike xMS.

Identifying the phases in the SSP schedule leads to a more generic schedule form shown in

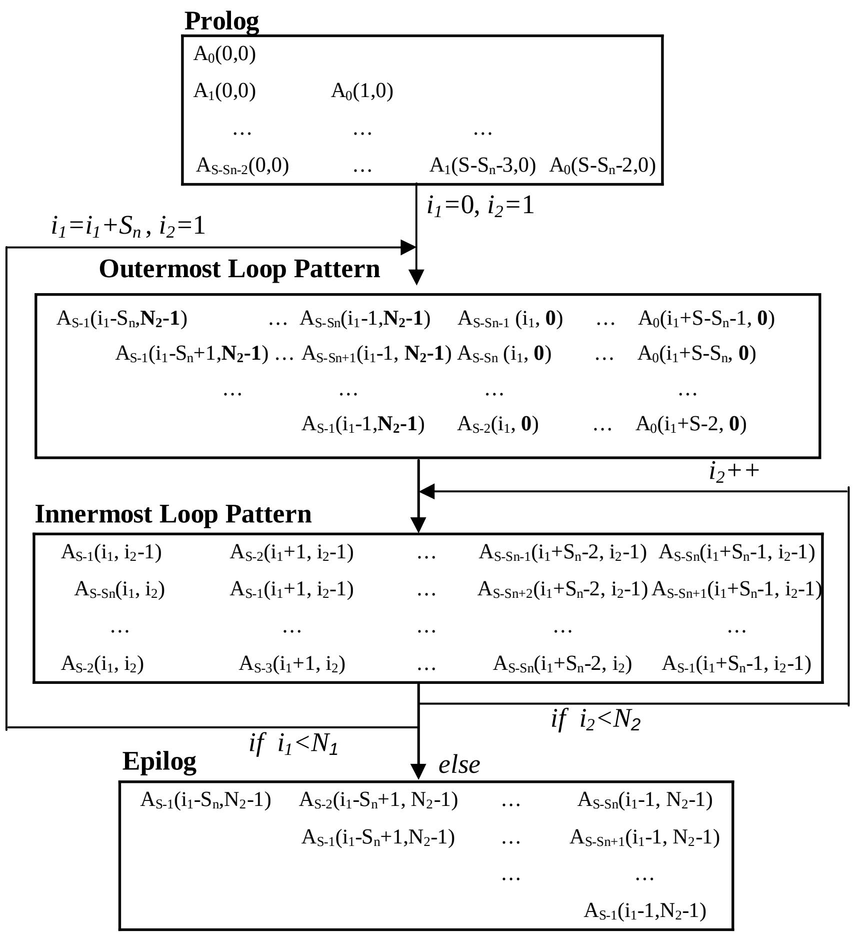

Prolog
$A_0(0,0)$
$A_1(0,0)$ $A_0(1,0)$
$A_{S-S_n-2}(0,0)$ … $A_1(S-S_n-3,0)$ $A_0(S-S_n-2,0)$
$i_1=0$, $i_2=1$
$i_1=i_1+S_n$, $i_2=1$
Outermost Loop Pattern
$A_{S-1}(i_1-S_n,\mathbf{N_2-1})$ … $A_{S-S_n}(i_1-1,\mathbf{N_2-1})$ $A_{S-S_n-1}(i_1,\mathbf{0})$ … $A_0(i_1+S-S_n-1,\mathbf{0})$
$A_{S-1}(i_1-S_n+1,\mathbf{N_2-1})$ … $A_{S-S_n+1}(i_1-1,\mathbf{N_2-1})$ $A_{S-S_n}(i_1,\mathbf{0})$ … $A_0(i_1+S-S_n,\mathbf{0})$
$A_{S-1}(i_1-1,\mathbf{N_2-1})$ $A_{S-2}(i_1,\mathbf{0})$ … $A_0(i_1+S-2,\mathbf{0})$
i_2++
Innermost Loop Pattern
$A_{S-1}(i_1, i_2-1)$ $A_{S-2}(i_1+1, i_2-1)$ … $A_{S-S_n-1}(i_1+S_n-2, i_2-1)$ $A_{S-S_n}(i_1+S_n-1, i_2-1)$
$A_{S-S_n}(i_1, i_2)$ $A_{S-1}(i_1+1, i_2-1)$ … $A_{S-S_n+2}(i_1+S_n-2, i_2-1)$ $A_{S-S_n+1}(i_1+S_n-1, i_2-1)$
$A_{S-2}(i_1, i_2)$ $A_{S-3}(i_1+1, i_2)$ … $A_{S-S_n}(i_1+S_n-2, i_2)$ $A_{S-1}(i_1+S_n-1, i_2-1)$
if $i_2<N_2$
if $i_1<N_1$
else
Epilog
$A_{S-1}(i_1-S_n,N_2-1)$ $A_{S-2}(i_1-S_n+1, N_2-1)$ … $A_{S-S_n}(i_1-1, N_2-1)$
$A_{S-1}(i_1-S_n+1,N_2-1)$ … $A_{S-S_n+1}(i_1-1, N_2-1)$
$A_{S-1}(i_1-1,N_2-1)$

Figure 8: A Generic SSP schedule

Fig. 8. Each A_i represents a stage of the kernel used as the input for the algorithm (Fig. 6). The generic schedule considers a two-level loop nest only for clarity reasons. How to generate the code for each phase will be explained in the code generation section.

4.3 Skeleton of Generated SSP Code

Knowing the different phases of the schedule, we can now show the skeleton of the generated code in Fig. 9, which happens to be the skeleton of the code generation algorithm.

The skeleton is written in pseudo-code and represents the generated code. Each *for* loop instruction is to be replaced by its equivalent in the target assembly language (Note that *br.ctop* is reserved for the outermost loop. Each L'_i corresponds to the L_i loop in the original loop nest. The code appearing in bold font is to be replaced by the code generated by the corresponding algorithm presented in the code generation section.

In this skeleton, variable $initial\_i_n$ is used to set the initial value of the innermost loop index i_n. When we reach the innermost loop pattern, it should be equal to 1 because the first iteration of the innermost loop has been issued in the outermost loop pattern.

```
        [initialization]
        [prolog]
$L'_1$:
        [outermost loop pattern]
        $initial\_in = 1$;
$L'_2$: for($i_2 = 0$;$i_2 < N_2$;$i_2$++) {
$L'_3$:     for($i_3 = 0$;$i_3 < N_3$;$i_3$++) {
                ...
$L'_n$:         for($i_n = initial\_in$;$i_n < N_n$;$i_n$++) {
                    [Innermost loop pattern]
                } //end $L'_n$
                $initial\_in = 0$;
                ...
            } //end $L'_3$
        } //end $L'_2$
        br.ctop $L'_1$:
        [epilog]
```

Figure 9: Generated Code Skeleton

The *br.ctop* instruction is used in the IA-64 ISA and represents a branch instruction that decrements the loop counter register and rotates registers automatically. Such an instruction will be called every time a new L_1 iteration is issued. Fig. 9 shows only one *br.ctop*, which will either branch back to L'_1, or fall through to the epilog. Other *br.ctop* instructions will appear in the prolog, the outermost loop pattern,and the epilog, as to be shown later.

Based on the above skeleton, the final code produced by our code generation method for our example (depicted in Fig. 4) is shown in Fig. 10. The generated code is shown in IA-64 assembly language notations and pseudo code. We can distinguish all the phases: the initialization (1-6), the prolog (7-9), the outermost loop pattern (10-20), the innermost loop pattern (22-25) and the epilog (30-36).

5 CODE GENERATION FOR THE IA64 ARCHITECTURE

The algorithms to generate the different components will now be described in detail in the context of the Itanium architecture. The code generation scheme, however, is general and can be easily adapted to any architecture with similar architectural support. Note that, there is more than one way to generate code for a given schedule and that we are only describing here one possible solution.

In the following explanations, we use the *emit_op* to emit an operation in assembly language and *emit_label* to create labels in the assembly file. When a stage A_i of the kernel is emitted, all operations of the stage are emitted as-is with all the registers assigned, predicate registers

```
          clrrrb;;
          r35=start address of array x
          r45=start address of array y
          LC=$N_1 - 1$
          EC=3 if $N_1$ is odd, =2 otherwise
          mov pr.rot=1<<16;;

          (p16) add   r34=4*$N_2$,r35;;
          br.ctop end_prolog_0;;
end_prolog_0:
$L'_1$:
          (p19) st4    [r48]=r43,4
          (p18) ld4    r42=[r37],4
          (p17) add    r45=4*$N_2$,r46
          (p16) add    r34=4*$N_2$,r35;;
          br.ctop end_outermost_pattern_0;;
end_outermost_pattern_0:
          (p19) st4    [r48]=r43,4
          (p18) ld4    r42=[r37],4
          (p17) add    r45=4*$N_2$,r46
          (p16) add    r34=4*$N_2$,r35;;
end_outermost_pattern_1:
$L'_2$:       for($i_2$=1;$i_2 < N_2$;$i_2$++) {
          (p18) st4    [r47]=r42,4
          (p17) ld4    r41=[r36],4;;
          (p17) st4    [r46]=r41,4
          (p18) ld4    r42=[r37],4;;
          }

          br.ctop $L'_1$;;

          LC=0
          EC=2
          (p19) st4    [r48]=r43,4
          (p18) ld4    r42=[r37],4;;
          br.ctop end_epilog_0;;
end_epilog_0:
          (p19) st4    [r48]=r43,4;;
          br.ctop end_epilog_1;;
end_epilog_1:
```

Figure 10: Final Code of our Example

included. Note that the code generation scheme uses the register allocated kernel (shown in Fig. 4(e)), which includes instructions guarded by predicate registers, as input in generating the schedule. We will keep using the example from Fig. 4 to illustrate each algorithm.

5.1 Prolog

First, let us discuss the code generation for prolog. It occurs only once per schedule. Afterwards, instances of the same operations are being taken care of by the outermost loop pattern. In our example, the prolog is only constituted of $a(0,0)$ (See Fig. 7). The other $a(i_1,0)$ operations appear only in the outermost loop pattern.

```
Generate_Prolog():
emit_label("prolog:");
for ($i = 0$;$i < S - S_n - 1$;$i$++){
    emit_stages($A_i, A_{i-1}, \cdots, A_0$);
    emit_op("br.ctop endprolog_$i$");
    emit_label("endprolog_$i$:");
}
```

Figure 11: Prolog Code Generation Algorithm

```
emit_stages(stages):
for ($t = 0$;$t < T$;$t$++){
    for each stage {
        for each operation $o$ at cycle $t$ in the stage {
            emit_op($o$);
        }
    }
    emit_stopbit();
}
```

Figure 12: Emitting Stages

The *br.ctop* instruction in the prolog code ensures that the (rotating) registers are rotated and the LC counter is decremented (as new iterations of the outermost loop are started). However, since the branch label (*end_prolog_i*) is to the next set of instructions in the prolog, control simply falls through. The prolog accounts for $S - S_n - 1$ copies of the kernel, with some stages peeled off in each copy. If only one stage of the kernel is not executed in the innermost loop, i.e., $S - S_n = 1$, then there is no prolog. The prolog for the example appears on lines 7-9 in Fig. 10.

The prolog code generation algorithm is shown in Fig.11, where function *emit_stages*() emits operations cycle by cycle from a series of stages, as shown in Fig.12. Here we simply emit a stop bit ";;" when all operations in a cycle are emitted.

Using ineffective operations, it is possible to merge the prolog with the first occurrence of the outermost loop pattern. However, this would force the outermost loop pattern to be different from its other occurrences. This, in turn, would force us to create a bigger prolog, thereby increasing the code size of the overall code.

5.2 Outermost Loop Pattern

By adding ineffective operations, there appears the first repetitive pattern, the outermost loop pattern (See Fig. 7). The outermost loop pattern is composed of S_n identical copies of the entire kernel shifted by one iteration of the outermost loop between each copy. Therefore, to generate the code associated with the outermost loop pattern, we use rotating registers and rotating branches: we emit S_n copies of the kernel alternated with a *br.ctop* instruction to force the rotation of the registers. Once again the *br.ctop* instruction is used not for effecting a control flow transfer, but for register rotation, and *LC* counter decrement.

Furthermore, the last kernel issued is not immediately followed by a *br.ctop* instruction. This is to *freeze* the hardware register renaming process until new iterations of the outer loops are initiated again, which is to happen in the next outermost loop pattern. Now that the hardware renaming is frozen to ensure that overlapping live ranges of different instances of the same TN in different outermost loop iterations do not use the same register, the code generator should explicitly generate code with different registers for the different instances of the same TN in the innermost loop pattern. We shall discuss this further in the following subsection.

```
Generate_Outermost_Loop_Pattern():
for($i = 0$;$i < S_n$;$i$++){
    emit_stages($A_{S-1}, A_{S-2}, \cdots, A_0$);
    if ($i$ != $S_n - 1$){
        emit_label("br_ctop end_outermost_pattern_$i$");
    }
    emit_label("end_outermost_pattern_$i$:");
}
```

Figure 13: Outermost Loop Pattern Code Generation Algorithm

Lines 10-20 in Fig. 10 shows the outermost loop pattern for our example. Note that we have exactly $S_n = 2$ copies of the kernel: one is within lines 10-13, and another within lines 16-19. After the first copy, there is one one *br.ctop* instruction (in line 14).

5.3 Innermost Loop Pattern

The innermost loop pattern contains S_n copies of the S_n stages in the kernel that are associated with the innermost loop.

Because we are executing the innermost loop, no new iteration of the outermost loop is issued and no register rotation occurs. To ensure live range of different instances (corresponding to different outermost loop iterations) of the same TN do not overlap, some kind of register renaming must be done. However, the available hardware register renaming is used for outer loop patterns, and the IA-64 architecture provides only one rotating register base. Hence, the register renaming in the innermost loop pattern must be handled by software. This is accomplished by replicating S_n copies of the kernel and statically renaming registers between each copy in a cyclic manner.

For our example, the $S_n = 2$ copies of the leftmost 2 stages of the kernel are shown in Fig. 10, lines 22-25. Note how the rotating registers have been frozen (no call to *br.ctop*) and how the registers in the original register-allocated kernel have been renamed to make sure each instruction uses the correct registers. Take the load instruction for instance, which is operation c in Fig. 4(b). It appears from the third to sixth cycle in Fig. 7 in this form:

```
3rd cycle ... c(0,0) ...  (in outermost loop pattern)
------------------------------------------------------
4th cycle      ...  c(1,0) (in innermost loop pattern)
5th cycle     c(0,1) ...
6th cycle      ...  c(1,1)
```

After mapping to real code, it becomes the following, which corresponds to line 17, 23, and 25 in Fig.10.

```
3rd cycle ... (p18)ld4 r42=[r37],4 ... (in outermost)
------------------------------------------------------
                                       (in innermost)
4th cycle     ... (p17)ld4 r41=[r36],4    (copy 0)
5th cycle     (p18)ld4 r42=[r37],4 ...    (copy 1)
6th cycle     ... (p17)ld4 r41=[r36],4    (copy 2)
```

It is clear that in the outermost loop pattern, the code is the same as that in the kernel (Fig.4(e)). After entering the innermost loop pattern, by simply decrementing the register indices of each operation in the leftmost S_n stages by 1, we get the first copy of the kernel in the innermost loop pattern. From that, the registers are renamed cyclically: First the registers used in the load operation are renamed from p17, r41 and r36 in 4th cycle to p18, r42 and r37 in 5th cycle, then they are renamed back to p17, r41 and r36 again in 6th cycle. (In our example, the kernel has an initiation interval of $T = 1$. Thus one copy of the kernel takes one cycle).

In general, the first copy of the kernel (copy 0) is formed by simply decrementing by 1 the register indexes of each operation in the leftmost S_n stages. From that on, indices of the non-static registers (integer, floating-point or predicate) must be rotated between copies of the kernel (copy 1 to copy $S_n - 1$). The value of the rotation can be deduced as follows. From their original values, the register indexes in the operations of stage A_j in the $i^{th}(1 \le i < S_n)$ kernel copy must be adjusted by:

$$offset(j, i) = (j - i - S)\%S_n - j + S - S_n - 1,$$

where "%" is the modulo division. It realizes the rotating effect of the register indexes.

The corresponding algorithm for generating innermost loop pattern is shown in Fig. 14, while the register index adjustment algorithm for a given stage is shown in Fig. 15, where $index(r)$ refers to the index of register r.

The loop control instructions around the innermost loop pattern are generated in an appropriate form. This is straightforward and we do not show the details here. In fact, we show the loop structure of the innermost loop in a high-level pseudo-code in line 21 in Fig. 10.

```
Generate_Innermost_Loop_Pattern():
1:  for($i = 0$;$i < S_n$;$i$++) {
2:      emit_stages(TS($A_{S-1}, offset(S-1, i)$),
3:                  TS($A_{S-2}, offset(S-2, i)$),
4:                  $\cdots$,
5:                  TS($A_{S-S_n}, offset(S-S_n, i)$));
6:  }
```

Figure 14: Innermost Loop Pattern Code Generation Algorithm

```
stage TS(stage S, int ofst):
1:  copy $S$ to $S'$;
2:  for each operation $o$ in $S'$ {
3:      for each rotating register $r$ in $o$ {
4:          $index(r) = index(r) + ofst$;;
5:      }
6:  }
7:  return $S'$;
```

Figure 15: Transform_Stage Algorithm

5.4 Epilog

The final phase of the SSP schedule is the epilog, which is basically constituted of S_n copies of the kernel where only a subset of the S_n leftmost stages of the kernel are executed. In a sense, it is similar to the prolog, and thus the code generation algorithm (shown in Fig. 16) is also similar.

```
Generate_Epilog():
1:  emit_op("LC=0");
2:  emit_op("EC=$S_n$");
3:  for($i = 0$;$i < S_n$;$i$++){
4:      emit_stages($A_{S-1}, A_{S-2}, \cdots, A_{S-S_n+i}$);
5:      emit_op("br.ctop end_epilog_$i$");
6:      emit_label("end_epilog_$i$:");
7:  }
```

Figure 16: Epilog Code Generation Algorithm

5.5 Initialization

The initialization part in Fig.9 uses the LC and EC registers provided by the IA-64 architecture. Their values will control all the generated code except the initialization itself and the epilog (Epilog has its own setting of LC and EC, as shown in Fig. 16). The setting of the values is crucial to the correctness of the generated code. The formulas found below assure that when (LC, EC) becomes $(0, 1)$,

1. We have issued all the outermost loop iterations, and have not issued any more iterations.
2. The next *br.ctop* to be executed must be the one shown in Fig. 9. According to the behavior of *br.ctop* introduced in section 2, the control flow definitely goes to the epilog.

There are totally N_1 number of outermost loop iterations. one *br.ctop* issues one iteration. Therefore LC is initialized to:

$$LC = N_1 - 1$$

To initialize EC we have to look at the value of N_1. If $N_1 - 1$ is not a multiple of S_n, some L_1 iterations in the last group will not be executed. The value of EC should be the total number of such iterations plus 1 to make sure that before executing the last br.ctop, LC=0 and EC=1. Therefore EC is initialized to:

$$EC = S - 1 - ((N_1 - 1)\%S_n)$$

In our example, $S = 4$ and $S_n = 2$. Therefore, $EC = 3 - ((N_1 - 1)\%2)$. That, $EC = 3$ when N_1 is odd, and 2 when N_1 is even.

The initialization phase should also prepare the live-in values for the rotating registers when needed and the bit mask for rotating register base. The final code for our example is shown in Fig. 10.

5.6 Register allocation

The target architecture provides the user with a set of rotating general-purpose/floating-point registers and predicate registers and a set of non-rotating general-purpose/floating-point registers and predicate registers.

Obviously enough registers must be allocated to make the code run and we assume that it is always the case. In our implementation, we made the simplistic choice of allocating S rotating registers per variable, excluding predicate registers. Our method is conservative and some allocated registers might never be used. For instance if variable x is allocated rotating register $r32$ whose value is referenced only in the first 2 stages of a 5-stage kernel, $r34$, $r35$ and $r36$ are not used by variable x and are not allocated to other variables either. An optimal/tight allocation of rotating registers remains open and is left for future work.

Rotating registers are only used for the outermost loop level. The other loops are scheduled traditionally. Because the generated code is based on the kernel and the kernel is allocated with registers under the assumption that overlapping lifetimes of the same variable can be address by hardware renaming of rotating registers in runtime , some adjustments must be made for the inner loops: the register indices of the instructions must be adjusted to cancel the effects of the register rotations that did not occur. The innermost loop pattern code generation algorithm does it automatically.

6 EXTENSIONS & OPTIMIZATIONS

Based on the basic algorithms introduced in the previous section, this section presents some skills on code-size reduction. We further generalize the algorithms to more general loop nests.

6.1 Code-Size Optimizations

To facilitate understanding, the version of the SSP code generation algorithms presented in the previous section are not optimized for code size. The prolog, outermost loop pattern, and epilog might contain several copies of the kernel that could be avoided.

If code size is an issue, the multiple copies of the kernel can be replaced by a single copy enclosed in a loop. The corresponding code generation algorithm for the outermost loop pattern is shown in Fig. 17. In this code, *pd* designates a non-rotating predicate register used for storing conditional, and *rc* a non-rotating register used as a loop counter. We are forced to use a general-purpose register *rc* because the *LC* and *EC* registers provided by the Itanium architecture are already used by the outermost loop.

```
Generate_CS-Optimized_Outermost_Loop_Pattern():
    emit_op("$r_c = S_n$");
    emit_label("outermost_begin:");
    emit_stages($A_{S-1}, A_{S-2}, \cdots, A_0$);
    emit_op("$rc = rc - 1$;;");
    emit_op("pd,p0 = cmp.eq $rc$,0;;");
    emit_op("(pd) br outermost_end;;");
    emit_op("br.ctop outermost_begin;;");
    emit_label("outermost_end:");
```

Figure 17: Code Generation Algorithm for Code-Size Optimized Outermost Loop Pattern

Note that the above code size optimized algorithm generates an outer loop with a single copy of the kernel. The same optimization can also be applied to the prolog and epilog. However, if S_n is small enough, the multiple copies version can still be smaller than the code-size optimized version due to the extra instructions needed to set *rc* and *pd* registers and loop control instructions (*cmp* and *br* instructions). Also, because of the loop control overheads, the code-size optimized version may run more slowly than the execution-speed optimized version.

To further reduce the code size, we can merge the epilog and the outermost loop pattern. As seen in Fig.7, the epilog and the outermost loop pattern contain the same instructions. The stages that are not used by the epilog in the outermost loop pattern can be peeled off by setting *LC* and *EC* correctly. Then predicate registers will turn off instructions that do not need to be executed. In order to achieve this, in the initialization phase in Fig. 9, we first initialize a predicate register *pe* to $false$ to indicate that we are not draining the pipeline yet. The register is used at the end of the outermost loop pattern to force the control flow to exit the loop nest at the end of the draining. This is done by adding an instruction *emit_op("(pe)br exit")* at the end of the outermost loop pattern code generation algorithm, where *exit* is a label. Correspondingly, we change the epilog generation algorithm to the one shown in Fig. 18, where *pe* is set to $true$, and the control branches back to reuse the outermost loop pattern.

6.2 Extension to Generic Source Loop Nest

In previous sections, we have considered the case when the $OPSET$s are empty for the loops between the outermost and the innermost loops. Let us now consider a more generic case when these $OPSET$s are not necessarily empty. Let the leftmost S_x stages in the kernel consist of operations executed by loop L_x and its inner loops.

Each time we finish an iteration of such an inner loop $L_x(1 < x < n)$, we should fill the pipeline with its next iteration, if any. For instance if the kernel is made of 5 stages e,d,c,b,a where e and d belong to L_3, c and b

```
Generate_CS-Optimized_Epilog():
1:  emit_op("$LC = 0$");
2:  emit_op("$EC = S_n - 1$");
3:  emit_op("$pe = 1$");
4:  emit_op("br outermost_pattern");
5:  emit_label("exit:");
```

Figure 18: Realizing Draining by Reusing the Outermost Loop Pattern

to L_2 and a to L1, then the filling pattern would be ed, be, cb and dc. So the generated code skeleton is a little different, as shown in Fig.19.

```
        [initialization]
        [prolog]
$L'_1$:
        [outermost loop pattern]
        $initial\_in = 1$;
$L'_2$: for($i_2 = 0$;$i_2 < N_2$;$i_2$++){
$L'_3$:     for($i_3 = 0$;$i_3 < N_3$;$i_3$++) {
                ...
$L'_n$:         for($i_n = initial\_in$;$i_n < N_n$;$i_n$++)
                    [Innermost loop pattern]
                } //end $L'_n$
                $initial\_in = 0$;
                if $i_{n-1} < N_{n-1} - 1$ {
                    [Fill $L_{n-1}$ Pipelines]
                }
                ...
            } //end $L'_3$
            if $i_2 < N_2 - 1$ {
                [Fill $L_2$ Pipelines]
            }
        } //end $L'_2$
        br.ctop $L'_1$:
        [epilog]
```

Figure 19: Generated Code Skeleton for the Generic Loop Nest

To fill the pipelines for an inner loop L_x, the stages from A_{S-1} to A_{S-S_x} need to be rotated using the algorithm shown in Fig. 20.

In the algorithm, $offsetx$ is defined as:

$$offsetx(j, i, x) = (j - i - S)\%S_x - j + S - S_x - 1,$$

which is an extension of function $offset(j, i)$ defined before.

```
Generate_Fill_Lx_Pipelines($x(1 < x < n)$):
for($i = 0$; $i \leq S_x - S_n + 1$;$i$++) {
    emit_stages(TSx($A_{S-1}, offsetx(S-1, i, x)$),
            TSx($A_{S-2}, offsetx(S-2, i, x)$),
            ...
            TSx($A_{S-S_x}, offsetx(S-S_x, i, x)$));
}
```

Figure 20: Fill Pipelines for an Inner Loop

TSx($A_j, ofst$) returns an empty stage if $ofst + j < S - S_n$. In this case, the stage A_j after rotation is not within the current group of the outermost loop iterations. So we simply ignore it. Otherwise, the algorithm is the same as TS() shown in Fig. 15.

6.3 Loop Skewing

In order to expand the set of loop nests that can be handled by SSP and increase the performance of the method, we looked at loop skewing [1] and how to apply it with SSP. The loop transformation is a strong ally of SSP for at least two reasons:

1. Loop skewing can expose more parallelism by breaking strong dependencies between iterations. And more parallelism means higher potential performance for SSP.
2. Loop skewing can transform negative dependences into zero or positive dependences [26] and therefore enlarge the set of loop nests that SSP can be applied to.

Note that any form of parallelism can be exploited by SSP, unlike MS which is limited to the parallelism at the innermost loop level. Therefore loop skewing and SSP form a stronger combination than loop skewing and MS.

We now describe how to solve two problems: for a given two-level loop nest with a rectangular iteration space (1) how to generate a final schedule and SSP kernel when loop skewing the innermost loop and (2) how to generate the corresponding code. We assume that the skewing factor is given and is applied to the innermost loop. The main challenge of SSP is how to handle the non-rectangular iteration space created by the loop skewing.

The first problem is easily solved. We apply the traditional loop skewing transformation to the original source code. Then the SSP method can applied as usual and a kernel is produced.

However the final schedule of the loop nest has changed. The iteration space is not rectangular anymore and the patterns are different. For instance, if we skew the innermost loop of the example shown on Fig. 4(a) by a factor of 1, the final schedule would be as shown on Fig. 21. We quickly notice that the innermost operations c and d were *pushed down* for outermost iterations 1, 3 and 5. This is a direct effect of the loop skewing with a factor of 1.

Having noticed the changes in the final schedule, we must generate the corresponding code. Within a group of outermost iterations, the first occurrence of the innermost loop pattern must now include the operations of the last innermost iteration of the previous group. On Fig. 21, $c(1,2)$ and $d(1,2)$ must be added to the first innermost loop pattern of the second group for instance. If those operations do not exist, we use ineffective operations as for $c(-1,2)$ and $d(-1,2)$. We refer to the first innermost loop pattern of each group as the *peeled innermost loop pattern.*

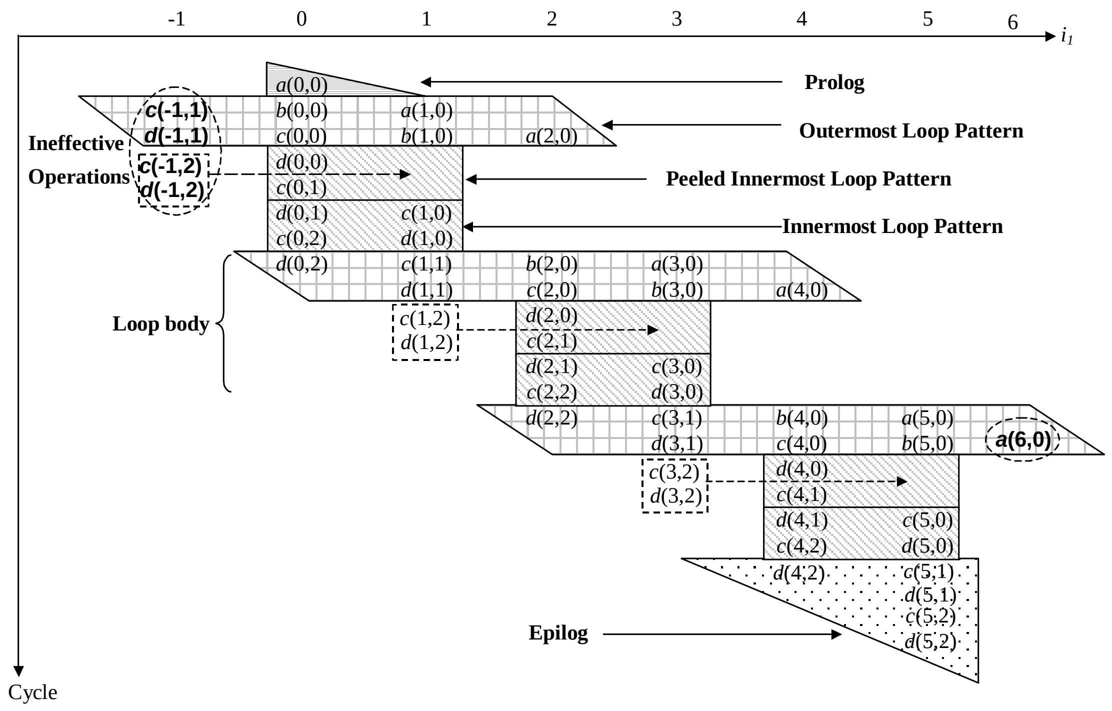


Figure 21: Phase-Partitioned SSP Schedule with Loop Skewing

The peeled innermost loop pattern is very similar to the innermost loop pattern. The operations are the same. However the indices are different. At the assembly level, the difference translates into different registers indexes. The predicate registers must match the previous innermost iteration and so must the other registers. For instance, $d(2,1)$ and $c(2,2)$ would be guarded with predicate $p18$ in the innermost loop pattern and so would $d(2,0)$ and $c(2,1)$ in the peeled innermost loop pattern. However $c(3,0)$ and $d(3,0)$ are guarded by $p17$ whereas $c(1,2)$ and $d(1,2)$ are guarded by $p19$. The registers appearing in c and d are adjusted in the same way. Because we are dealing with two-level loop nests only, the shift in register indexes is always 2.

```
   Generate_Innermost_Loop_Pattern():
1:   for($i = 0$;$i < S_n$;$i$++)
2:       emit_ops( TS($A_{S-1}, offset(i,1)$),
3:                 TS($A_{S-2}, offset(i,2)$),
4:                 $\cdots$,
5:                 TS($A_{S-S_n}, offset(i, S_n-1)$));
```

Figure 22: Peeled Innermost Loop Pattern Code Generation Algorithm

We can now present the code generation algorithm for the peeled innermost loop pattern shown on Fig. 22 and the new SSP code generation scheme on Fig. 23. Note that the innermost loop pattern is run one less time. That run is replaced by the peeled innermost loop pattern.

Note that the nature of skewing with predicates leads to code expansion. It should be used in cases when performance is favored over code size.

```
       [prolog]
$L'_1$:
       [outermost loop pattern]
       [peeled innermost loop pattern]
$L'_2$:  for($i_2 = (S_n - 1) * factor + 1$;$i_2 < N_2$;$i_2$++)
          [Innermost loop pattern]
       } //end $L'_2$
       br.ctop $L'_1$:
       [epilog]
```

Figure 23: Skewed Generated Code Skeleton

7 EXPERIMENTS

7.1 Experimental Setup

The code generation algorithms have been implemented as a tool set, called *SSP-CodeGen*, on an Itanium workstation. For simplicity reasons, our method was implemented as a stand-alone module working at the assembly level. The Gnu assembler is then used to assemble the resulting code. The 1-D scheduler (Step 2 of the SSP schedule method described in Section 2.3) was implemented using a standard modulo scheduling method [12]. We have implemented two versions of SSP, one with and another without code size optimization. We refer to these versions SSP and code size optimized SSP (CS-SSP) respectively.

We have compared SSP and CS-SSP method with two other methods: a traditional modulo scheduling method of the innermost loop *MS* [12], and extended modulo scheduling (*xMS*) which overlaps the draining and filling part of the outer loop [17]. We compare the different methods for their performance, code size, and instruction bundling capability.

For the experiments we chose important loop kernels extracted from scientific applications. Because SSP applied to the innermost loop of a loop nest is equivalent to MS, we considered only loop kernels where SSP would select a loop level other than the innermost one. The following benchmarks have been used: matrix multiply (MM), modified 2-D hydrodynamics (HD) [8], LU decomposition (LU) and Successive Over-Relaxation (SOR). Except for HD, the benchmarks were extracted from the Livermore Loops suite [15]. For matrix multiply, we have considered 6 different versions, corresponding to the 6 different ways in which the loops can be interchanged, in order to fully demonstrate the impact of data reuse and parallelism upon the final code quality. These version are referred to as: ijk, jik, ikj, jki, kij and kji, the order of the indices of the loop nest. We also applied *loop tiling* [28] to jki[3] with loops k and j tiled, for further comparisons. The chosen tile size was the one giving the best performance. The loop levels j and k are chosen for tiling as they maximize the data locality reuse. Upon tiling, we further applied *unroll-and-jam* [6], also known as *register tiling.* The tiled and register-tiled versions are named as $jki + T$ and $jki + UJ$ for short. Here we report the results for the matrix size 1000 $\times$ 1000, with double precision floating point values. Other matrix sizes were considered in [26].

7.2 Results & Analysis

In this section we report the performance results by running the code, generated by our code generation method, on an Itanium workstation equipped with a 733MHZ Itanium1 processor, 2GB of main memory,

[3]This is the equivalent row-major code.

16KB/96KB/2MB of L1/L2/L3 caches respectively, and running Red Hat Linux 7.2 operating system. In reporting the performance results our goal is three-fold. First, our performance results demonstrate the feasibility and correctness of the proposed code generation method. Second, we would like to know whether the code generation scheme retains the predicted performance benefits of SSP schedules. In particular we would like to answer whether the use of static register renaming or code size increase due to our code generation scheme for the SSP method hinder the performance? To address these questions we report the speedup of xMS, SSP, and CS-SSP schedules over the MS version, for each of the benchmarks, by directly measuring the execution time of the respective loop kernels on the Itanium workstation. Lastly, we report performance numbers relating to code size and bundle density for the code generated by our code generation scheme.

7.2.1 Correctness

To ensure that our code generation method produces correct code, we compared the outputs produced by SSP schedules with those generated by a serial version of the code (without any software pipelining). In certain cases we have also manually checked the generated code and compared the number of iterations executed by the SSP and serial versions. In all benchmarks, the outputs produced by MS, xMS, SSP, and CS-SSP schedule matched exactly with those generated by the serial version.

7.2.2 Performance

As reported in [26] and as shown in Fig 24, SSP schedules perform significantly better than xMS and MS schedules for every benchmark tested. The speedup achieved by SSP is between 1.1 and 4.24 times faster than MS or xMS with an average speedup of 2.1. This significant performance improvement of SSP is due to the fact that it is able to take advantage of available parallelism or data reuse in outer loop levels. The two SSP versions seem to perform equally well. SSP performed better in *ikj* and LU while the code-size optimized versions performs slightly better for *jki*, $jki + UJ$, $jki + T$ and *jik* benchmarks.

We note that neither the static register renaming method nor the code size increase has resulted in SSP schedules performing worse than MS or xMS schedules. Obtaining concrete performance numbers that indicate the actual impact of these two is left for future work.

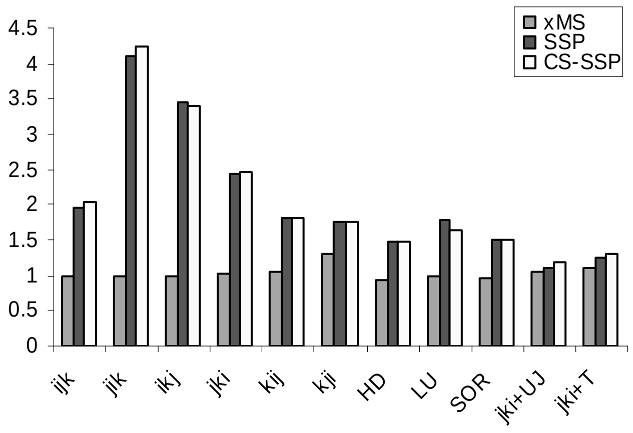


Figure 24: Speedup

7.2.3 Bundle Density

Bundle density is the average number of operations per bundle, excluding NOPs. Larger bundling density implies two aspects in code size and performance. It indicates more compact code, and probably more parallelism. We point out here that bundling density is a measure of parallelism in the *static* code, and does not necessarily equal to the instruction-level parallelism at exploited at run time. Bundle density also relates to instruction fetch bandwidth requirement. Higher the bundle density, lower is the requirement on instruction fetch bandwidth.

The bundle density of all schedule methods for the different benchmarks are shown in Fig. 25. While MS and xMS achieve a bundle density of 1.90, on an average, the average bundle densities of SSP and CS-SSP are, respectively, 1.91 and 2.1. The improvement in the bundle density of CS-SSP is especially better than those of MS and xMS.

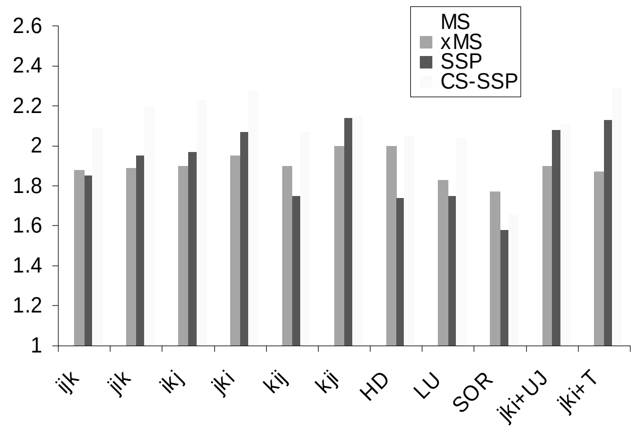


Figure 25: Bundle Density

7.2.4 Code Size

Last, we compare the code size of the different scheduling methods in Fig 26. Despite our precautions to avoid code duplication during code generation, the code size produced by SSP is between 3.6 and 9.0 times bigger than MS or xMS. The increase due to CS-SSP schedules is between 2 and 6.85 larger than MS or xMS.

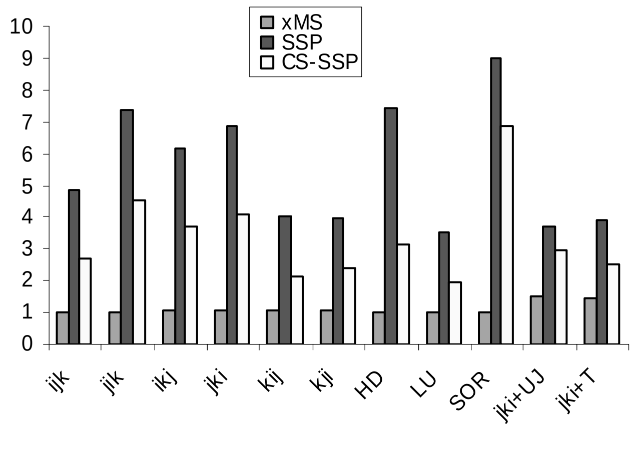


Figure 26: Code Size

Although the code size increase in SSP and CS-SSP schedules is high, it is not a surprise. There are several reasons for this code size increase. First, SSP method uses two patterns (outermost loop pattern and innermost loop pattern) instead of one like MS does. Second, and most important, SSP replicates the loop kernel several times to accomplish static register renaming. The S_n copies of the kernel in the innermost loop pattern accounts for about 60% or more of the final code size. In addition, prolog, epilog, and outer loop pattern also contain code duplication.

The code size increase, although noticeable, does not result in any performance degradation. In particular, the measured L1 instruction cache misses were still extremely low. Second, we observe that the maximum kernel size of the SSP schedules, over all the benchmark considered, is less than 4.2KB, less than a typical L1 I-cache size. Thus as long as the repeating kernel can be held in the I-cache, the code size increase does not affect the performance significantly. As we see in all our experiments, SSP and CS-SSP perform as well or significantly better than MS and xMS schedules. Thus we observe that the code size is largely outweighed by the improvement in execution time, a result quite acceptable in general purpose computing.

8 FUTURE WORK

Our experiments revealed that most of the code expansion is caused by the multiple copies of the kernel for the innermost loop pattern. This can be avoided if there is ISA support to rotate part of registers cyclically during the execution of the innermost loop pattern. Thus, one possible future direction is to investigate hardware support and ISA extensions for SSP schedules in order to limit code size increase.

As explained in Section 5.6, our method currently assumes that rotating register are allocated in a conservative manner. We want to study efficient register allocation methods for SSP.

Last, we want to extend our current code generation scheme to non-rectangular iteration spaces. It is important to consider non-rectangular iteration space, as loop skewing is often applied to a loop nest before software pipelining.

9 RELATED WORK

A number of software pipelining methods have been proposed in the literature [9, 3, 4, 12, 22, 20]. Most of these methods focus on software pipelining only the innermost loop or singly-nested loops. *Hierarchical reduction* [13], *Outer Loop Pipelining (OLP)* [17], and *pipelining-dovetailing* [27], are extensions which apply software pipelining to nested loops. However, all of these approaches start from the innermost loop. This has two serious drawbacks: (1) it may not fully exploit the instruction-level parallelism available at outer loops, and (ii) data reuse potential available at outer loop levels are not exploited due to the innermost-loop centric approach. In contrast, our Single-dimension Software Pipelining of multi-dimension loops (SSP) method, first proposed in [26], can use a combination of instruction-level parallelism and data reuse to choose the most appropriate loop level and software pipeline it. Further, unlike traditional innermost-loop centric approaches which require the construction of modulo reservation table and dependence graph at each loop level, our SSP method requires only the construction of the dependence graph at the chosen loop level, and simplifies it based only on the data dependences at that level. The performance of SSP method was shown to be significantly better than the traditional approaches, viz., simple modulo scheduling (MS) and outer loop software pipelining, referred to as the extended modulo scheduling (xMS) in this paper. At the very least, SSP matches the performance of other methods.

Unroll-and-jam has been applied to improve the performance of software pipelined loops [6]. SSP is different from the above: SSP applies software pipelining to a (selected) outer loop while the work described in [6]. First it unrolls the outer loop and then applies software pipelining to the innermost loop. In other words the RecMII and the MII of the unroll-and-jammed loop still strongly depend on the recurrences in the innermost loop, though reduced by the unroll factor. On the other hand, the RecMII and MII of SSP schedules depend on the recurrences in the chosen loop level. Thus, by choosing an appropriate loop level in the loop nest, the SSP method can achieve a lower initiation interval, while unroll-and-jam has to rely on a large unroll factor for the same.

Unroll-and-squash first applies unroll-and-jam to a nested loop, and then "squash" the jammed inner loops to generate software pipelined code [18]. SSP is different from unroll-and-squash in the following ways: (i) the unroll-and-squash method presented in [18] appears to be limited to 2-level loop nest; (ii) it does not overlap the epilog and prologs inner loop iterations and requires optimizations such as xMS to achieve this; and (iii) it decides on the unroll factor first, and then software pipelines the unrolled(-and-jammed), innermost, loop. An in depth comparison of SSP with unroll-and-jam and unroll-and-squash is left for future work.

Can innermost-loop-centric software pipelining approaches combined with loop transformations such as loop interchange and loop skewing achieve schedules similar to the one produced by the SSP method? First, we argue that such loop transformations are orthogonal to the SSP approach in the sense that the SSP method can also benefit from them. In fact, in all the 6 different versions of the matrix multiply program, which correspond to the application of loop interchange transformation, SSP schedules performed as well as or better than the other schedules for the same version. Second, while it may be true that loop transformation with innermost loop software pipelining can produce schedules similar to SSP schedules in some loop nests, there are loop nests where such transformations may not be legal. In those cases SSP can achieve better schedules, better in terms of data reuse or instruction-level parallelism exploited.

Code generation schemes for single loop modulo scheduling are discussed for VLIW architectures with and without hardware support for modulo scheduling in [23]. The considered hardware support include rotating registers, predicated execution, and iteration control register [24]. The code generation approach followed for modulo scheduling in the Cydra-5 compiler has been discussed in [10] Register allocation for software pipelined loops has been considered in [21]. A number of alternative solutions have been presented for machines with and without hardware support for software pipelining. Code size reduction methods for software pipelined loops have been discussed in [29, 30, 5, 14]. However these work consider software pipelining only the innermost loop.

In contract, this paper deals with code generation issues for the SSP method which deals with multi-dimensional loop nests. Code generation for architectures supporting rotating registers and predicated execution has been considered in this paper. In addition to using rotating register support for handling overlapping live ranges of successive iterations of the outer loop, our code generation scheme uses static register renaming techniques to handle live range overlap in inner loop iterations.

10 CONCLUSIONS

Single-dimension Software Pipelining (SSP) method for multi-dimensional loops, proposed in [26], chooses the most profitable loop level in the loop nest and software pipelines it. This paper discusses a code generation scheme for the SSP method. In particular, it proposes an abstract code generation skeleton and targets it for the IA-64 architecture. It addresses several interesting issues in code generation, including (1) code generation of loop kernels for the chosen loop level as well as its inner loops, (2) code generation of prolog and epilog, (3) register assignment for the selected loop kernel as well as its the inner loops which ensures live ranges of different instances of the same variable do not overlap, (4) code generation using predicated execution, and (5) code size reduction. We have implemented our code generation scheme for the IA-64 architecture. Initial experimental results demonstrate the feasibility and advantages of the proposed scheme.

References

[1]

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

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

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

[5] F. Bodin, Z. Chamski, C. Eisenbeis, E. Rohou, and A. Seznec. GCDS: A compiler strategy for trading code size against performance in embedded applications. Technical Report 1163, Institut de Recherche en Informatique et Systmes Aleatories, Campus de Beaulieu, 35042 Rennes, Cedex, France, 1997.

[6] Steve Carr, Chen Ding, and Philip Sweany. Improving software pipelining with unroll-and-jam. In *Proc. 29th Annual Hawaii International Conference on System Sciences*, pages 183–192, 1996.

[7] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, November 1994.

[8] Standard Performance Evaluation Corp. Spec cpu95 benchmark suite, 1995.

[9] Intel Corporation. *Intel IA64 Architecture Software Developer's Manual.* Intel Corporation, 2001.

[10] James C. Dehnert and Ross A. Towle. Compiling for Cydra 5. *Journal of Supercomputing*, 7:181–227, May 1993.

[11] Guang R. Gao, Qi Ning, and Vincent Van Dongen. Software pipelining for nested loops. ACAPS Technical Memo 53, School of Computer Science, McGill University, Montréal, Québec, May 1993. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[12] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 258–267, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

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

[14] J. Llosa and S.M. Freudenberger. Reduced code size modulo scheduling in the absence of hardware support. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, Istanbul, Turkey, December 2002.

[15] F. McMahon. The livermore fortran kernels: A computer test of the numerical performance range. Technical Report Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[16] Soo-Mook Moon and Kemal Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, November 1997.

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

[18] D. Petkov, R. Harr, and S. Amarasinghe. Efficient pipelining of nested loops: unroll-and-squash. In *16th International Parallel and Distributed Processing Symposium (IPDPS '02 (IPPS & SPDP))*, pages 19–19, Washington - Brussels - Tokyo, April 2002. IEEE.

[19] J. Ramanujam. Optimal software pipelining of nested loops. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 335–342, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

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

[21] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation (PLDI)*, pages 283–299, 1992.

[22] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, San Jose, California, November 30–December2, 1994. ACM SIGMICRO and IEEE-CS TC-MICRO.

[23] B. Ramakrishna Rau, Michael S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proceedings of the 25th annual international symposium on Microarchitecture*, pages 158–169, 1992.

[24] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 departmental supercomputer – design philosophies, decisions, and trade-offs. *Computer*, 22(1):12–35, January 1989.

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

[26] Hongbo Rong, Zhizhi Tang, Alban Douillet, R. Govindarajan, and Guang R. Gao. Single-dimension software pipelining of multi-dimensional loops. CAPSL Technical Memo 49, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, August 2003. In ftp://ftp.capsl.udel.edu/pub/doc/memos/memo049.ps.gz.

[27] Jian Wang and Guang R. Gao. Pipelining-dovetailing: A transformation to enhance software pipelining for nested loops. In *Proceedings of the 6th International Conference on Compiler Construction, CC '96*, Lecture Notes in Computer Science, pages 1–17, Linkoping, Sweden, April 22–26, 1996. Springer-Verlag.

[28] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, June 26–28, 1991. *SIGPLAN Notices,* 26(6), June 1991.

[29] Q. Zhuge, Z. Shao, and E.H-M. Sha. Optimal code size reduction for software pipelinined loops on DSP applications. In *Proceedings of the International Conference on Parallel Processing*, Vancouver, Canada, August 2002.

[30] Q. Zhuge, B. Xiao, Z. Shao, E.H-M. Sha, and C. Chantrapornchai. Optimal code size reduction for software pipelinined and unfolded loops. In *Proceedings of the International Symposium on System Synthesis*, Kyoto, Japan, October 2002.