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

Hongbo Rong[†], Alban Douillet[†], R. Govindarajan[‡], Guang R.Gao[†]

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

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

## Abstract

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

*In this paper, we describe our solution to SSP code generation. In contrast to traditional software pipelining, SSP handles two distinct repetitive patterns, and thus requires new code generation algorithms. Further, these two distinct repetitive patterns complicate register assignment and require two levels of register renaming. As rotating registers support renaming at only one level, our solution is based on a combination of dynamic register renaming (using rotating registers) and static register renaming (using code replication). Finally, code size increase, an even more important issue for SSP than for traditional software-pipelining, is also addressed. Optimizations are proposed to reduce code size without significant performance degradation.*

*We first present a code generation scheme and subsequently implement it for the IA-64 architecture, making effective use of rotating registers and predicated execution. We present some initial experimental results, which demonstrate not only the feasibility and correctness of our code generation scheme, but also its code quality.*

## 1. Introduction

Software pipelining for loop nests is a challenging research topic. While numerous algorithms have been proposed for single loops [2, 1, 5, 6, 10], only a few address loop nests [6, 8, 15]. They all modulo schedule a loop nest hierarchically, starting from the innermost loop to the outermost one. This approach, henceforth referred to as *innermost-loop-centric modulo scheduling*, naturally extends the single loop scheduling method to the multi-dimensional domain. However, the approach has two major shortcomings. First, it commits itself to the innermost loop first without considering how much parallelism the other loop levels have to offer. Second, it cannot exploit the data reuse potential that may be present in the outer loops.

In [14], we introduced a *resource-constrained scheduling* method for software pipelining of loop nests, called *Single-dimension Software Pipelining (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 parallelism and data reuse 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.

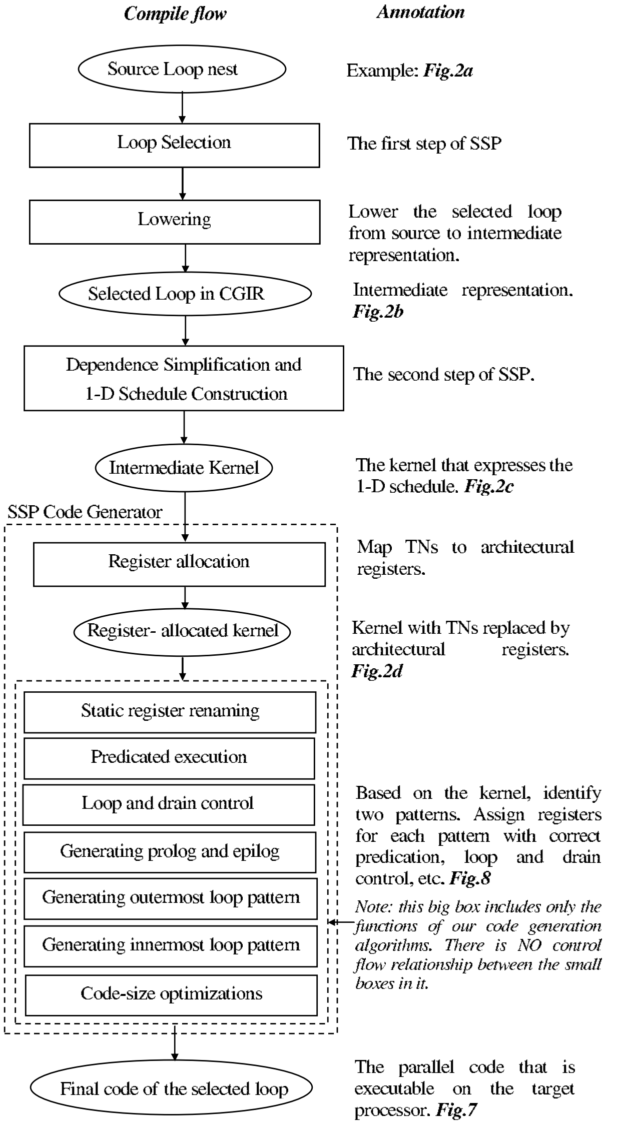
SSP has three steps: (1)Loop selection: select the loop level that may yield the best performance if software pipelining is applied to this level. (2)Dependence simplification and *1-dimensional schedule* construction: simplify the $n$-dimensional ($n$-D) scheduling problem to 1-dimensional (1-D), and then schedule the operations. (3)Final schedule computation: the 1-D schedule is mapped to an $n$-D iteration space to form a *final schedule*, which is semantically equivalent to the selected (serial) loop [1].

This paper presents a code generation scheme for the SSP method. In the context of a modern compiler framework, the scheme is shown in Fig.1. It basically follows the three steps of the SSP method. First, it chooses a prof-

[1] SSP transforms the selected loop only. Its outer loops, if any, remain intact. Therefore, this paper discusses code generation only for the selected loop.

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

itable loop from the source loop nest. The selected loop is then lowered into CGIR (Intermediate Representation for Code Generation). Second, it simplifies dependences and performs scheduling. The output is a kernel – called *intermediate kernel* in the rest of this paper – that expresses the 1-D schedule for the selected loop. Lastly, the SSP code generator (the bigger dotted box in the figure) translates the intermediate kernel into target machine code. This is equivalent to the third step of SSP (final schedule computation). We focus on this step in this paper.



Compile flow
Annotation
Source Loop nest
Example: Fig.2a
Loop Selection
The first step of SSP
Lowering
Lower the selected loop from source to intermediate representation.
Selected Loop in CGIR
Intermediate representation. Fig.2b
Dependence Simplification and 1-D Schedule Construction
The second step of SSP.
Intermediate Kernel
The kernel that expresses the 1-D schedule. Fig.2c
SSP Code Generator
Register allocation
Map TNs to architectural registers.
Register- allocated kernel
Kernel with TNs replaced by architectural registers. Fig.2d
Static register renaming
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, etc. Fig.8
Note: this big box includes only the functions of our code generation algorithms. There is NO control flow relationship between the small boxes in it.
Final code of the selected loop
The parallel code that is executable on the target processor. Fig.7

**Figure 1. Compilation Flow**

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, instead of one, repetitive patterns . These patterns, referred to as the *outermost* and the *innermost loop patterns*, introduce a more challenging code generation problem than traditional software pipelining.
- Because the SSP method overlaps different iterations of an outer loop, their inner loops are also overlapped. Consequently, the live ranges for a TN in each outer loop iteration are overlapped not only in the outer loop, but also in the inner loops. In this case, a two-level rotating register file is required to handle register renaming [13]. In absence of this, in this paper, we combine dynamic register renaming (using rotating registers) and static register renaming (using code replication) to address the problem.
- Code size increase in SSP schedules is more important than in traditional software pipelining. The challenge is how to limit the code size increase while retaining the performance benefits of the SSP method.

In this paper, we discuss the code generation scheme and then target it for the IA-64 architecture. We show how to apply to loop nests the traditional hardware support for software pipelining of single loops, e.g., Intel IA-64 hardware support (rotating registers, predication, and special operations). Initial experimental results demonstrate the feasibility and correctness of our code generation scheme. It also reveals the code quality and performance of the SSP method. Due to the space limitation, this paper only addresses code generation issues and the reader is referred to [14] for details about the SSP method.

The rest of the paper is organized as follows. Section 2 motivates our study by a simple example. Section 3 outlines our code generation method, while Section 4 presents in details the algorithms for the IA-64 architecture. Section 5 presents extensions and optimizations to the basic method. Experimental results are reported in Section 6. A discussion on future work, related work and concluding remarks are then presented in the remaining sections.

## 2. Motivation, Assumption, and Problem Statement

### 2.1. Motivating Example

Fig.2(a) shows a perfect loop nest. Suppose the outermost loop is selected by SSP. After lowering it into an equivalent internal representation for code generation (CGIR), it becomes imperfect (See Fig. 2(b), where the `for` loops are shown in pseudo code 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.

SSP schedules this internal representation of the outermost loop, and outputs an intermediate kernel in the form shown in Fig. 2(c). The scheduling process of an imperfect loop nest is similar to that of a perfect loop nest [14]. Details are documented elsewhere [12] [14] (Technical memo version) [2].

Like traditional software pipelining, the register allocator maps a TN to an architecture register. One possible plan is to allocate $r35$ to TN1, $r45$ to TN2, and $r40$ to TN3. Fig.2(d) shows the corresponding register-allocated kernel. Note that in this kernel, the same TN in adjacent stages, which come from adjacent outermost loop iterations, is allocated registers with successive indexes. For instance, TN1 is allocated $r35$, $r36$, $r37$, and 38, respectively, in each of the stages from right to left. TN1$\{$-1$\}$ in the rightmost stage is the register that will contain the TN1 value in the next outermost loop iteration and therefore is assigned $r34$.

The main problem is then to generate the final executable code in a compact form from the register-allocated kernel.

## 2.2. Assumptions

**2.2.1. Source Loop Nest** In this paper, we assume a $n$-deep ($n > 1$)[3] source loop nest as shown in Fig. 3(a). 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 non-branch 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 similar [12]. For the example in Fig. 2(b), $OPSET_1$ is composed of the two *add* operations, and $OPSET_2$ is composed of *ld4* and *st4* operations.

In the following sections, we **assume that** $OPSET_x$ **(**$2 \leq x \leq n-1$**) is empty to simplify our discussion**. The code generation algorithms are then extended to the general cases when $OPSET_x$ is not necessarily empty in Section 5.2.

**2.2.2. Intermediate Kernel** Given the above loop nest, SSP will generate an intermediate kernel of $S$ different stages, $A_0, A_1, \cdots, A_{S-1}$ from right to left (Fig. 3(b)). Each stage takes $T$ cycles to execute. During each cycle, one or more operations are executed.

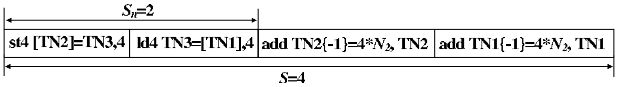
The $S_n$ leftmost stages consist of operations from the innermost loop, i.e., from $OPSET_n$. Other stages consist of operations from the outermost loop, i.e. from $OPSET_1$ (the other $OPSET$s are empty for the time being).

```
int U[$N_1$][$N_2$];
int V[$N_1$][$N_2$];
$L_1$:for ($i_1$=0; $i_1 < N_1$; $i_1$++){
$L_2$:    for ($i_2$=0; $i_2 < N_2$; $i_2$++){
              V[$i_1$][$i_2$]=U[$i_1$][$i_2$];
          }
}
```

(a) Source Loop Nest

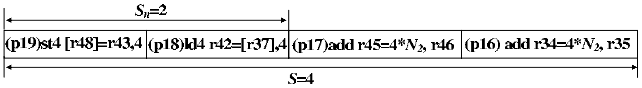
```
$L_1$:for ($i_1$=0; $i_1 < N_1$; $i_1$++){
      $a$: add TN1{-1}=4*$N_2$,TN1
      $b$: add TN2{-1}=4*$N_2$,TN2
$L_2$: for ($i_2$=0;$i_2 < N_2$;$i_2$++){
          $c$: ld4 TN3=[TN1],4
          $d$: st4 [TN2]=TN3,4
      } //end $L_2$
} //end $L_1$
```

(b) Intermediate Representation



| | | | |
|---|---|---|---|
| st4 [TN2]=TN3,4 | ld4 TN3=[TN1],4 | add TN2{-1}=4*$N_2$, TN2 | add TN1{-1}=4*$N_2$, TN1 |

(c) Intermediate Kernel



| | | | |
|---|---|---|---|
| (p19)st4 [r48]=r43,4 | (p18)ld4 r42=[r37],4 | (p17)add r45=4*$N_2$, r46 | (p16) add r34=4*$N_2$, r35 |

(d) Register-allocated Kernel, where r35 is allocated to TN1, r45 to TN2, and r40 to TN3.

**Figure 2. Motivating Example**

## 2.3. 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 SSP final schedule, while reducing code size and loop control overheads.

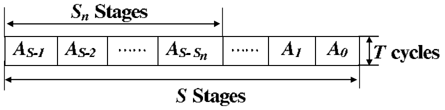
In this paper, we propose to look at a code generation scheme and then target it to the IA-64 architecture to make use of the available hardware support, i.e. rotating registers, predicated execution and specialized ISA (Instruction Set

[2] Code generation discussed in this paper is corresponding to the final schedule with early-issue delay, shown in the Appendix of literature [14] (Technical memo version) and generalized in [12].

[3] When $n = 1$, the loop nest is a single loop, and SSP is equivalent to traditional modulo scheduling [14]. Then the code generation is completely the same as that of modulo scheduling [11]. We do not discuss this case in this paper.

```
$L_1$: for ($i_1$=0; $i_1 < N_1$; $i_1$++) {
        $OPSET_1$
$L_2$:      for ($i_2$=0; $i_2 < N_2$; $i_2$++) {
            $OPSET_2$
            ...
$L_n$:              for ($i_n$=0; $i_n < N_n$; $i_n$++) {
                    $OPSET_n$
                } //end $L_n$
                ...
        } end $L_2$
    } //end $L_1$
```

(a) Generic Source Loop Nest



(b) Generic Intermediate (or Register-allocated) Kernel

**Figure 3. Generic Example**

Architecture), which were originally designed for modulo scheduling of single loops, and show how to apply them to loop nests.

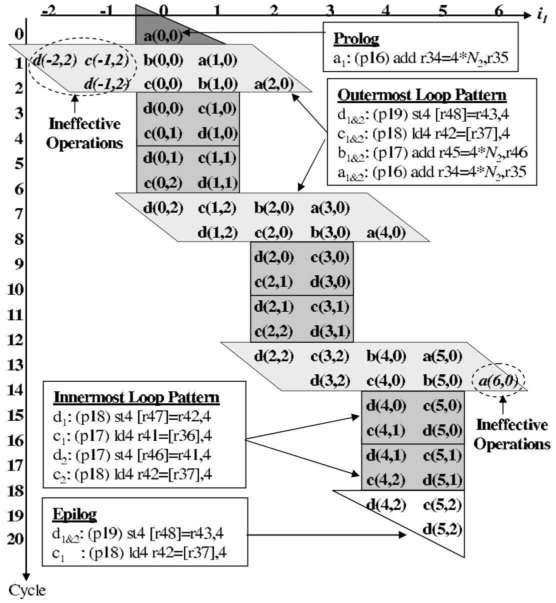
## 3. SSP Code Generation Overview

In this section, we present a high-level overview of the code generation scheme and explain its components, based on the repeating patterns in the SSP final schedule.

### 3.1. Components in a Final Schedule

Let us first identify the different components involved in the SSP final schedule. It consists of 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. 2 have been shown in the SSP final schedule in Fig. 4, where $o(x, y)$ refers to the instance of operation $o$ with loop index $i_1 = x$ and $i_2 = y$. The corresponding IA-64 code for each component is shown nearby, where $o_1$ is operation $o$ in the first cycle of the component under consideration, etc.

There are only two repeating patterns, independently of the number of loops[4]. All operations, i.e. $OPSET_1$ and



**Prolog**
$a_1$: (p16) add r34=4*$N_2$,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*$N_2$,r46
$a_{1\&2}$: (p16) add r34=4*$N_2$,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 4. The SSP Final Schedule for Our Example with $N_1 = 6$ and $N_2 = 3$**

$OPSET_n$, appear in the outermost loop pattern, whereas only operations in the innermost loop, i.e. $OPSET_n$, appear in the innermost loop pattern.

Note that to make the outermost loop pattern appear repetitively, ineffective operations need to be added. The ineffective operations are circled in Fig. 4. 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 Fig. 4). For the IA-64 architecture, predicate registers will be used to make them ineffective.

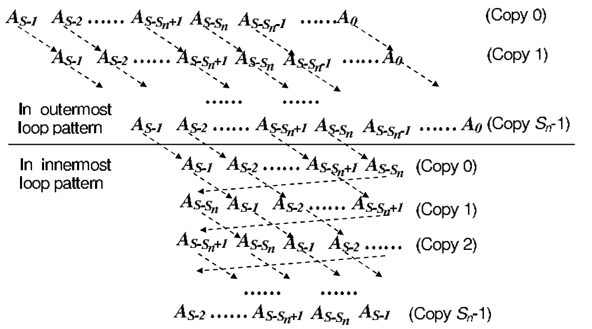
### 3.2. Register Usage Strategy

An invariant in the loop nest can be assigned a nonrotating register in conventional register allocation techniques. In this paper, we discuss allocation of only (predicate, integer and floating-point) rotating registers to variables in the loop nest.

From Fig. 4, we see that after the outermost loop pattern, control will finally transfer to the innermost loop pattern. In general, the code sequence is like that shown in Fig. 5.

It can be seen that the outermost loop pattern is composed of $S_n$ copies of the kernel in a stagger way. This reminds us of the traditional modulo scheduled code. For such code, we can simply repeat the kernel for $S_n$ times, with dynamic register renaming applied after each repetition. It is

[4] It is because of the assumption in Section 2.2.1 that only the outermost and innermost loops have operations that makes only two repetitive patterns appear. However, even for general cases where the other middle loops also have operations, we just need some extra transition code besides these two patterns, as to be described in Section 5.2. In this paper, we will show how to generate such transition code, but will never take them as any repeating pattern, although they do repeat, to simplify our discussion. Therefore, there are only two repeating patterns in any case.



$A_{S-1}$ $A_{S-2}$ ...... $A_{S-S_n+1}$ $A_{S-S_n}$ $A_{S-S_n-1}$ ......$A_0$ (Copy 0)
$A_{S-1}$ $A_{S-2}$ ...... $A_{S-S_n+1}$ $A_{S-S_n}$ $A_{S-S_n-1}$ ......$A_0$ (Copy 1)
...... ......
In outermost loop pattern
$A_{S-1}$ $A_{S-2}$ ...... $A_{S-S_n+1}$ $A_{S-S_n}$ $A_{S-S_n-1}$ ...... $A_0$ (Copy $S_n$-1)
In innermost loop pattern
$A_{S-1}$ $A_{S-2}$ ...... $A_{S-S_n+1}$ $A_{S-S_n}$ (Copy 0)
$A_{S-S_n}$ $A_{S-1}$ $A_{S-2}$ ...... $A_{S-S_n+1}$ (Copy 1)
$A_{S-S_n+1}$ $A_{S-S_n}$ $A_{S-1}$ $A_{S-2}$ ...... (Copy 2)
...... ......
$A_{S-2}$ ...... $A_{S-S_n+1}$ $A_{S-S_n}$ $A_{S-1}$ (Copy $S_n$-1)

**Figure 5. Code Sequence from the Outermost to the Innermost Loop Pattern**

easy to find from Fig. 4 that prolog and epilog are similar to the outermost loop pattern and can use dynamic register renaming as well.

The innermost loop pattern contains $S_n$ copies of the $S_n$ leftmost stages in the kernel. As indicated by the arrows in Fig. 5, the first copy of the kernel (copy 0) is formed by simply shift right by 1 stage the $S_n$ leftmost stages of the last kernel copy in the outermost loop pattern. The next copy (copy 1) is simply a permutation of copy 0. Then copy 2 is a permutation of copy 1, etc. The permutation is to rotate right by 1 stage the current copy. To achieve the effect of permutation, we have to statically rename registers in each copy, unless there were hardware support for dynamic renaming.

In this paper, our strategy for register usage is to combine dynamic (hardware) and static (software) register renaming. Dynamic register renaming, e.g., the rotating register support, is used in the outermost loop pattern, prolog and epilog. Static register renaming is used in the innermost loop pattern.

Let us consider the compile flow in Fig. 1. For the intermediate kernel, we will assign rotating registers to the TNs in this kernel. The first $S$ predicate rotating registers are used to control the issue of the outermost loop iterations,like the traditional modulo scheduling [1]. For the IA-64 architecture, $p16, p17, \cdots, p(16 + S - 1)$, are assigned to each stage in the kernel from right to left. See the example in Fig. 2(d).

For other rotating registers, in our current implementation, we made a simplistic choice of allocating $S$ rotating registers per variable. This method is conservative and some allocated registers might never be used. For instance, in Fig. 2(d), TN1 is allocated rotating register $r35$ whose value is referenced only in the first and the third stages of the kernel, and thus $r36$ and $r38$ are not used by TN1 and are not allocated to other TNs, either. An optimal/tight allocation of rotating registers is left for future work.

After getting the register-allocated kernel, the code generator begins to generate the final schedule. In this process, we will use the kernel directly to form the prolog, the outermost loop pattern, and the epilog, using dynamic register renaming. For the innermost loop pattern, however, the register indexes of the operations in the kernel must be adjusted to reflect the permutation of the kernel, as to be shown in Section 4.

The register-allocated kernel will be used throughout the subsequent code generation process. Thus from now on, when we talk about kernel, we refer to the "register-allocated kernel" by default.

### 3.3. Generated SSP Code Skeleton

Knowing the different components of the final schedule and the register usage strategy, now we can show the skeleton of the generated code in Fig. 6.

The skeleton is written in pseudo-code. Each $L'_i$ corresponds to the $L_i$ loop in the original loop nest. Each *for* loop structure is to be replaced by its equivalent in the target assembly language. This is straightforward and we do not show the details here. The code in bold font is to be generated by the corresponding algorithms in Section 4.

In this skeleton, variable $initial\_i_n$ is used to set the initial value of the innermost loop index $i_n$. When execution reaches $L'_n$ the first time, $initial\_i_n$ is 1; Otherwise, it is 0.

```
        [Initialization]
        [Prolog]
$L'_1$:
        [Outermost loop pattern]
        $initial\_i_n$ = 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 ($i_n = initial\_i_n$;$i_n < N_n$;$i_n$++) {
                    [Innermost loop pattern]
                } //end $L'_n$
                $initial\_i_n$ = 0;
                ...
            } //end $L'_3$
        } //end $L'_2$
        br.ctop $L'_1$;
        [Epilog]
```

**Figure 6. Generated Code Skeleton**

*Br.ctop* is a branch operation in the IA-64 ISA, which rotates registers automatically for dynamic register renaming, and decrements the loop counter register $LC$ if $LC > 0$, or decrements epilog control register $EC$ if $LC = 0$. Fig. 6 shows only one *br.ctop*, which will either branch back to $L'_1$, or fall through to the epilog. Other *br.ctop* operations 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. 2) is shown in Fig. 7. The generated code is shown in IA-64 assembly language and pseudo code. We can distinguish all the components: the initialization (1-6), the prolog (7-9), the outermost loop pattern (10-20), the innermost loop pattern (22-25) and the epilog (28-36). $Initial\_i_n$ is not explicitly shown here, since it is always 1 for double loops, according to Fig. 6.

```
1:              clrrb;;
2:              r35=start address of array U
3:              r45=start address of array V
4:              LC=$N_1 - 1$
5:              EC=3 if $N_1$ is odd, =2 otherwise
6:              mov pr.rot=1<<16;;

7:              (p16)  add    r34=4*$N_2$,r35;;
8:              br.ctop end_prolog_0;;
9:      end_prolog_0:
        $L'_1$:
10:             (p19)  st4    [r48]=r43,4
11:             (p18)  ld4    r42=[r37],4
12:             (p17)  add    r45=4*$N_2$,r46
13:             (p16)  add    r34=4*$N_2$,r35;;
14:             br.ctop end_outermost_pattern_0;;
15:     end_outermost_pattern_0:
16:             (p19)  st4    [r48]=r43,4
17:             (p18)  ld4    r42=[r37],4
18:             (p17)  add    r45=4*$N_2$,r46
19:             (p16)  add    r34=4*$N_2$,r35;;
20:     end_outermost_pattern_1:
21:     $L'_2$:     for($i_2$=1;$i_2 < N_2$;$i_2$++) {
22:             (p18)  st4    [r47]=r42,4
23:             (p17)  ld4    r41=[r36],4;;
24:             (p17)  st4    [r46]=r41,4
25:             (p18)  ld4    r42=[r37],4;;
26:             }

27:             br.ctop $L'_1$;;

28:             LC=0
29:             EC=2;;
30:             (p19)  st4    [r48]=r43,4
31:             (p18)  ld4    r42=[r37],4;;
32:             br.ctop end_epilog_0;;
33:     end_epilog_0:
34:             (p19)  st4    [r48]=r43,4;;
35:             br.ctop end_epilog_1;;
36:     end_epilog_1:
```

**Figure 7. Final Code of Our Example**

# 4. Code Generation for the IA-64 Architecture

The algorithms to generate the different components will now be described in detail in the context of the IA-64 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 SSP final schedule and that we are describing here only one possible solution.

In the following descriptions, we use *emit_op* to emit an operation and *emit_label* to create a label. We will keep using the example from Fig. 2 to illustrate each algorithm.

## 4.1. Prolog

Prolog occurs only once for a given SSP final schedule. It accounts for $S - S_n - 1$ copies of the kernel, with some stages peeled off in each copy. There is no prolog if $S - S_n - 1 = 0$. The prolog for the example appears on lines 7-9 in Fig. 7. The $br.ctop$ operation ensures that the rotating registers are rotated and the $LC$ or $EC$ counter is decremented. Since the branch label ($end\_prolog\_0$) immediately follows the branch, control simply falls through.

The algorithm for generating the prolog is shown in Fig.8, where function $emit\_stages()$ emits operations cycle by cycle from a series of stages, as shown in Fig.8. Here we simply emit a stop bit ";;" when all operations in a cycle are emitted.

## 4.2. The Outermost Loop Pattern

The outermost loop pattern (See Fig. 5) is composed of $S_n$ identical copies of the entire kernel shifted by one outermost loop iteration 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 $br.ctop$ operation to force the rotation of the registers. Once again the $br.ctop$ operation is not used for control flow transfer, but for register rotation.

Furthermore, the last kernel copy issued is not immediately followed by a $br.ctop$ operation. This is to *freeze* the hardware register renaming process until new iterations of the outermost loop are initiated again, which is to happen in the next occurrence of the outermost loop pattern.

Lines 10-20 in Fig. 7 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 $br.ctop$ operation (in line 14). The second copy, however, is not immediately followed by a $br.ctop$, which is delayed to be after the innermost loop pattern and appear in line 27.

The code generation algorithm for the outermost loop pattern is shown in Fig. 8.

### 4.3. The Innermost Loop Pattern

As shown in Fig. 5, after the outermost loop pattern, control will finally transfer to the innermost loop pattern. Since the outermost loop pattern freezes hardware renaming in the end, as said above (Section. 4.2), to keep ensuring that overlapping live ranges of the same TN from different outermost loop iterations do not use the same register, some kind of register renaming must be done. However, the available hardware register renaming is used for the outermost loop pattern, 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.

To equivalently express the innermost loop pattern in Fig. 5, we can perform renaming in this way: From their original values in the register-allocated kernel, the indexes of the rotating registers in the operations of stage $A_j$ $(S - S_n \leq j \leq S - 1)$ in the kernel copy $i (0 \leq i < S_n)$ must be adjusted by:

$$offset(j,i) = \begin{cases} -1 & \text{if } i = 0, \\ (j - i - S)\%S_n - j + S - S_n - 1 & \text{otherwise,} \end{cases}$$

where "%" is the modulo division.

In another word, the first copy of the kernel (copy 0) in the innermost loop pattern is formed by decrementing by 1 the indexes of the rotating registers in each operation in the leftmost $S_n$ stages. From that on, indexes of the rotating registers must be permuted between copies of the kernel (copy 1 to copy $S_n - 1$ in the innermost pattern in Fig. 5).

For our example, the $S_n = 2$ copies of the leftmost 2 stages of the kernel are shown in Fig. 7, lines 22-25. Note how the registers in the original register-allocated kernel have been renamed to make sure each operation uses the correct registers. Take the load operation for instance, which is operation $c$, and appears from the cycle 3 to cycle 4 in Fig. 4 in this form:

```
cycle 3      ...  c(1,0) (in copy 0 of the kernel)
cycle 4     c(0,1) ...   (in copy 1 of the kernel)
```

After mapping to real code, it becomes the following, which corresponds to line 23 and 25 in Fig.7.

```
line 23 ... (p17)ld4 r41=[r36],4(in copy 0 of the kernel)
line 25 (p18)ld4 r42=[r37],4 ...(in copy 1 of the kernel)
```

For the $ld$ operation in copy 0 of the kernel, $offset(j,i) = offset(2,0) = -1 (j = 2$ since the $ld$ operation is in stage $A_2$, as shown in Fig. 2(d). And $i = 0$ since we are solving the offset for copy 0). For the $ld$ operation in copy 1 of the kernel, $offset(j,i) = offset(2,1) = 0$. Therefore, in the first copy, the rotating registers used in the load operation are renamed from p18, r42 and r37 in the register-allocated kernel to p17, r41 and r36. Then in the second copy, they are renamed back to p18, r42 and r37 again.

The corresponding algorithms for generating the innermost loop pattern and adjusting the rotating registers' indexes are shown in Fig. 8, where $index(r)$ refers to the index of register $r$. Function TS() transforms a stage with a given adjustment.

### 4.4. Epilog

The final phase of the SSP schedule is the epilog, which consists of $S_n$ copies of the kernel, except that 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. 8) is also similar.

### 4.5. Initialization

The initialization part in Fig.6 sets 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. 8). 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. 6. According to the behavior of *br.ctop* [1], 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 find the correct value for $EC$, let us reconsider Fig. 4. Since $EC$ is used only by *br.ctop*, which is not used in the innermost loop pattern at all, if we remove all the occurrences of the innermost loop pattern from Fig. 4, we get Fig. 9. To be clear, we have explicitly shown the *br.ctop*s. A *br.ctop* operation is controlled by two registers $LC$ and $EC$. For clarity, we use $br.ctop(LC, EC)$ to represent a *br.ctop* operation with $(LC, EC)$ as its input parameters. For example, $br.ctop(5, 2)$ (in the prolog) means that the current value of the $(LC, EC)$ pair is $(5, 2)$, and from this value, *br.ctop* rotates the registers once, and modifies the value to be $(4, 2)$, according to the semantics of this operation [1]. Therefore, the next *br.ctop* is represented as $br.ctop(4, 2)$, as shown in the first occurrence of the outermost loop pattern in Fig. 9.

We observe the figure vertically and horizontally, in order to find the correct initial value of $EC$.

Vertically, there are $S - 1 + \lceil \frac{N_1}{S_n} \rceil * S_n$ number of *br.ctop*s. The prolog and the first occurrence of the outermost loop

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

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_op("br_ctop end_outermost_pattern_$i$;;");
        }
        emit_label("end_outermost_pattern_$i$:");
    }

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

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

stage TS(stage $STAGE$, int $ofst$):
    copy $STAGE$ to $STAGE'$;
    for each operation $o$ in $STAGE'$ {
        for each rotating register $r$ in $o$ {
                    $index(r) = index(r) + ofst$;
        }
    }
    return $STAGE'$;

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

**Figure 8. SSP Code Generation Algorithms**

pattern account for the first $S - 1$ $br.ctop$s. From that on, there are $\lceil \frac{N_1}{S_n} \rceil - 1$ occurrences of the outermost loop pattern and 1 epilog. Each of them consists of $S_n$ kernels, and each kernel is followed by a $br.ctop$ [5]. This accounts for the remaining $\lceil \frac{N_1}{S_n} \rceil * S_n$ $br.ctop$s.

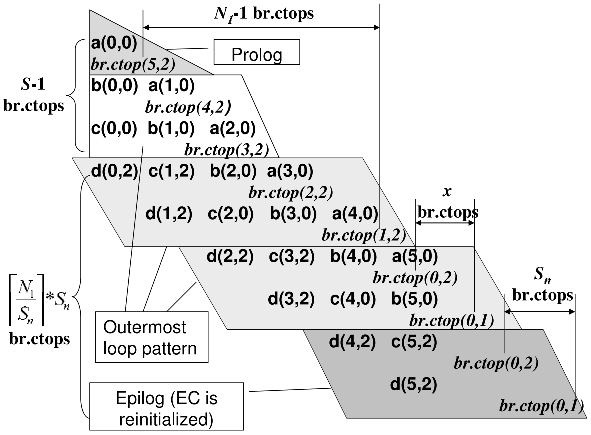


**Figure 9. Find the Setting of EC for Our Example. Here $N_1 = 6$.**

Assume $EC$ has an initial value of $x$. Then horizontally, there are $N_1 - 1 + x + S_n$ $br.ctop$s. First, $(LC, EC)$ is changed from $(N_1 - 1, x)$ to $(1, x)$, and that uses $N_1 - 1$ $br.ctop$s. $LC$ is initialized to this value. Then $(LC, EC)$ is changed from $(0, x)$ to $(0, 1)$, and that uses $x$ $br.ctop$s. $EC$ is initialized to this value. As we said before, we assure that when $(LC, EC)$=$(0, 1)$, we will definitely fall through to epilog, where we need another $S_n$ $br.ctop$s to completely drain the pipelines.

Therefore, we have the following equation:

$$S - 1 + \lceil \frac{N_1}{S_n} \rceil * S_n = N_1 - 1 + x + S_n.$$

From that, we easily find that $EC$ should be initialized to

$$x = S - 1 - ((N_1 - 1)\%S_n).$$

In our example, $S = 4$ and $S_n = 2$. Therefore, $EC = 3 - ((N_1 - 1)\%2)$. That is, $EC = 3$ 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. 7.

## 5. 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.

[5] As said in Section 4.2, the last kernel copy in the outermost loop pattern is not followed by a $br.ctop$ until after the innermost loop pattern. It is so in Fig. 9 because the innermost loop pattern is removed.

### 5.1. Code-Size Optimizations

To facilitate understanding, the code generation algorithms presented in the previous section are not optimized for code size. The prolog, the outermost loop pattern, and the 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. 10(a). In this code, $pd$ designates a non-rotating predicate register used for storing conditional, and $rc$ a non-rotating integer register used as a loop counter.

```
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:");
```

(a) Algorithm for Generating Code-Size Optimized Outermost Loop Pattern

```
Generate_CS-Optimized_Epilog():
    emit_op("$LC = 0$");
    emit_op("$EC = S_n - 1$");
    emit_op("$pe = 1$");
    emit_op("br outermost_begin;;");
    emit_label("exit:");
```

(b) Realizing Draining by Reusing the Outermost Loop Pattern

**Figure 10. Code Size Optimizations**

Note that the above algorithm generates the outermost loop pattern with a single copy of the kernel. The same optimization can also be applied to the prolog and epilog.

To further reduce the code size, we can merge the epilog and the outermost loop pattern. As seen in Fig.4, the epilog and the outermost loop pattern contain the same operations. 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 the operations that do not need to be executed. In order to achieve this, in the initialization phase in Fig. 6, we first initialize a non-rotating 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 algorithm for generating the outermost loop pattern, where $exit$ is a label. Correspondingly, we change the epilog generation algorithm to the one shown in Fig. 10(b), where $pe$ is set to $true$, and the control branches back to reuse the outermost loop pattern.

### 5.2. 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 [6]. So the generated code skeleton is a little different, as shown in Fig.11(a).

To fill the pipelines, the stages from $A_{S-1}$ to $A_{S-S_x}$ need to be permuted using the algorithm shown in Fig. 11(b).

In the algorithm, $offsetx$ is defined as:

$$offsetx(j,i,x) = \begin{cases} -1 & \text{if } i = 0, \\ (j - i - S)\%S_x - j + S - S_x - 1 & \text{otherwise,} \end{cases}$$

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

Function TSx($A_j, ofst$) in the algorithm returns an empty stage if $ofst + j < S - S_n$. In this case, the stage $A_j$ after permutation 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. 8.

## 6. Experiments

### 6.1. Experimental Setup

The code generation algorithms have been implemented as a tool set on an IA-64 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 method [14]) was implemented using a standard modulo scheduling method [5]. We have implemented two versions of SSP code generation, one with

[6] Detailed explanations are omitted here due to the paper size limit. Interested reader may refer to the Appendix in the technical memo version of literature [14] for an example final schedule with the early-issue delay to see the transition code for filling pipelines for an inner loop.

```
          [Initialization]
          [Prolog]
$L'_1$:
          [Outermost loop pattern]
          $initial\_i_n$ = 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($i_n = initial\_i_n$;$i_n < N_n$;$i_n$++)
                      [Innermost loop pattern]
                  } //end $L'_n$
                  $initial\_i_n$ = 0;
                  if ($i_{n-1} < N_{n-1} - 1$ && $S_{n-1} > S_n$) {
                      [Fill $L_{n-1}$ Pipelines]
                      $initial\_i_n$ = 1;
                  }
                  ...
              } //end $L'_3$
              if ($i_2 < N_2 - 1$ && $S_2 > S_n$) {
                  [Fill $L_2$ Pipelines]
                  $initial\_i_n$ = 1;
              }
          } //end $L'_2$
          br.ctop $L'_1$:
          [Epilog]
```

(a) Generated Code Skeleton for the Generic Loop Nest

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

(b) Fill Pipelines for an Inner Loop

**Figure 11. Code Generation for A Generic Loop Nest**

and the other one without code size optimization. We refer to them as 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*) [5], and an extended modulo scheduling method (*xMS*) which overlaps the draining and filling part of an outer loop [8]. We compare the different methods for their performance, code size, and bundling capability.

For the experiments we chose important loops extracted from scientific applications. Because SSP is equivalent to MS when applied to the innermost loop of a loop nest, we considered only loops where SSP would select a loop level other than the innermost one. The following benchmarks extracted from the Livermore Loops suite [9] have been used: matrix multiply (MM), modified 2-D hydrodynamics (HD), LU decomposition (LU) and Successive Over-Relaxation (SOR). For matrix multiply with a loop body of $A[i][j] += B[i][k] * C[k][j]$, 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: $ijk$, $jik$, $ikj$, $kij$, $jki$ and $kji$, according to the order of the indexes of the loop nest. We also applied *loop tiling* to $jki$ with loops $k$ and $i$ tiled, for further comparisons. The chosen tile size was the one giving the best performance. Upon tiling, we further applied *unroll-and-jam*, 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 × 1000, with double precision floating point values. Other matrix sizes were considered in [14].

## 6.2. Results & Analysis

In this section we report the performance results by running the code, generated by our code generation method, on an IA-64 Itanium workstation equipped with a 733MHZ Itanium1 processor, 2GB of main memory, 16KB/96KB/2MB of L1/L2/L3 caches, and running Red Hat Linux 7.2 operating system. In reporting the performance results, our goal is three-fold. First, the 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 final 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 hinders 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 loops on the Itanium workstation. We also report performance numbers relating to code size and bundle density for the code generated by our code generation scheme.

**6.2.1. Correctness** To ensure that our code generation method produces correct code, we compared the outputs produced by SSP, MS and xMS 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. In all benchmarks, the outputs produced by MS, xMS, SSP, and CS-SSP match exactly with those generated by the serial version.

**6.2.2. Performance** As reported in [14] and as shown in Fig 12, 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 SSP performs slightly better for other benchmarks.

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

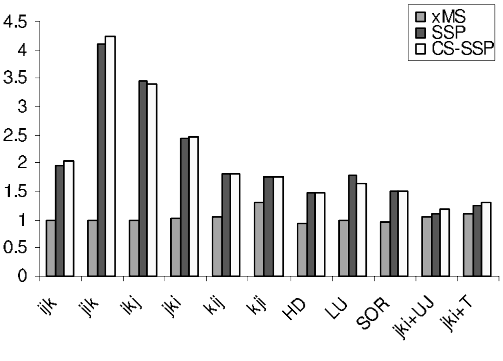


**Figure 12. Speedup**

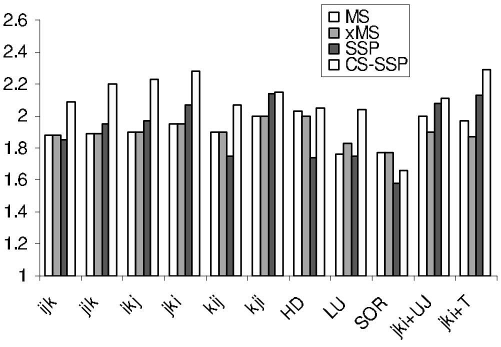


**Figure 13. Bundle Density**

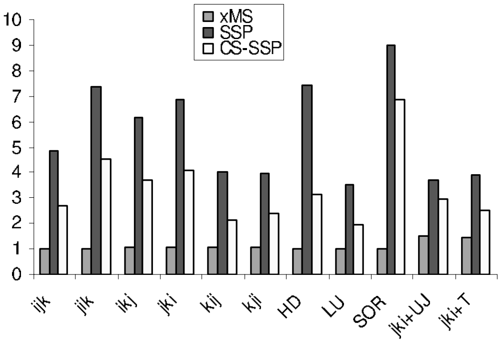


**Figure 14. Code Size**

**6.2.3. Bundle Density** Bundle density is the average number of operations per bundle, excluding NOPs (Null OPerations). Larger bundling density implies 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 exploited at run time.

The bundle density of all schedule methods for the different benchmarks are shown in Fig. 13. While MS and xMS achieve a bundle density of 1.90 on 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.

**6.2.4. Code Size** Last, we compare the code size of the different scheduling methods in Fig 14. 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.

Although the code size increase in SSP and CS-SSP code is high, it is not a surprise. There are several reasons for this code size increase. First, SSP method uses two patterns (the outermost and the innermost loop patterns) instead of one like MS does. Second, and most important, SSP replicates the 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.

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 size of the generated SSP code, among all the benchmarks considered, is less than 4.2KB, which is less than a typical L1 I-cache size. Thus as long as the schedules for the loops 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.

## 7. 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. The copies were introduced because there was no rotating register file available for the inner loops. Therefore one possible future direction is to investigate hardware support and ISA extensions (more affordable than that in [13]) to generate kernel-only code.

As explained in Section 3.2, our method currently allocates rotating register conservatively. More efficient register allocation for SSP will be investigated in the future.

Lastly, we will introduce the extension of our code generation scheme to non-rectangular iteration spaces.

## 8. Related Work

Code generation schemes for modulo scheduling of single loops are discussed for VLIW architectures with and without hardware support in [11]. The considered hardware support include rotating registers, predicated execution, and iteration control registers [3]. The code generation approach for modulo scheduling in the Cydra-5 compiler has been discussed in [3]. Code size reduction for software pipelined loops has been discussed in [7, 4]. All these works consider software pipelining only for 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. Dynamic and static register renaming are combined smoothly to address the issue of life range overlapping at multiple levels.

## 9. Conclusion

The Single-dimension Software Pipelining (SSP) method for a multi-dimensional loop nest [14] 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 a code generation skeleton and targets it for the IA-64 architecture. It addresses several interesting issues in code generation, including (1) code generation of the outermost and the innermost loop patterns, with dynamic and static register renaming to assure that overlapping live ranges of different instances of the same TN use different registers; (2) code generation of the prolog and the epilog; (3) code generation using predicated execution; and (4) 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.

## Acknowledgments

The first author owes a lot to Prof.Zhizhong Tang. We also thank Prof.Bogong Su, Dr. Hongbo Yang, and the anonymous reviewers for their advice.

## References

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

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

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

[4] E. Granston, R. Scales, E. Stotzer, A. Ward, and J. Zbiciak. Controlling code size of software-pipelined loops on the TMS320C6000 VLIW DSP architecture. In *Proceedings of 3rd IEEE/ACM Workshop on Media and Stream Processors*, pages 29–38, 2001.

[5] R. 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.

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

[7] J. Llosa and S. 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.

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

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

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

[11] B. R. Rau, M. 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.

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

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

[14] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software-pipelining for multi-dimensional loops. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, March 2004. A longer version is available as CAPSL Technical Memo 49 from ftp://ftp.capsl.udel.edu/pub/doc/memos/memo049.ps.gz.

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