Optimal Software Pipelining of Loops with Control Flows*

Han-Saem Yun
School of Computer Science and Engineering
Seoul National University
hsyun@davinci.snu.ac.kr

Jihong Kim
School of Computer Science and Engineering
Seoul National University
jihong@davinci.snu.ac.kr

Soo-Mook Moon
School of Electrical Engineering
Seoul National University
smoon@altair.snu.ac.kr

ABSTRACT

Software pipelining is widely used as a compiler optimization technique to achieve high performance in machines that exploit instruction-level parallelism. However, surprisingly, there have been few theoretical or empirical results on optimal software pipelining of loops with control flows. In this paper, we present three new contributions for this under-investigated problem. First, we propose a necessary and sufficient condition for a loop with control flows to have an optimally software-pipelined program. We also present a decision procedure to compute the condition. Second, we present two software pipelining algorithms. The first algorithm computes an optimal solution for every loop satisfying the condition, but may run in exponential time. The second algorithm computes optimal solutions efficiently for most (but not all) loops satisfying the condition. Third, we present experimental results which strongly indicate that achieving the optimality in the software-pipelined programs is a viable goal in practice with realistic hardware support.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms

Algorithms, Languages

Keywords

Software pipelining, instruction-level parallelism, VLIW

1. INTRODUCTION

Software pipelining refers to a class of fine-grain loop parallelization algorithms which impose no scheduling barrier such as basic block or loop iteration boundaries, thus achieving the effect of fine-grain parallelization with full loop unrolling. Software pipelining computes a static parallel schedule for machines that exploit instruction-level parallelism (ILP) such as superscalar or VLIW processors.

While software pipelining has been used as a major compiler optimization technique to achieve high performance for ILP processors, surprisingly, there have been few theoretical results, let alone practical ones, known on the optimality issue of software pipelined programs. One of the best known open problems is the time optimal software pipelining problem, which can be stated as follows: *given a loop (with or without control flows), 1) decide if the loop has its equivalent time optimal program or not and 2) find a time optimal parallel program if the loop has one, assuming that sufficient resources are available.* A parallel program is said to be time optimal if every execution path p of the program runs in its minimum execution time determined by the length of the longest data dependence chain in p [19].

For straight-line loops (without control flows), the time optimal software pipelining problem is well understood and a time optimal program can be computed in polynomial time [1]. This is because the process of software pipelining can be easily formalized thanks to the strong periodicity of such loops (e.g., a periodic execution model and dependence patterns). For example, the problem of software pipelining of such loops can be modeled by a simple linear formulation [8] and several software pipelining algorithms have been developed using this model.

On the other hand, for loops with control flows, software pipelining algorithms cannot exploit the loop periodicity because execution paths of these loops cannot be modeled by periodic constraints. This irregularity results in numerous complications and makes the formalization very difficult. As a consequence, time optimal software pipelining of such loops has been under-investigated, leaving most of theoretical questions unanswered. In this paper, we focus on loops with control flows.

1.1 Previous Work

Until recently, only two results for loops with control flows were published [19, 20]. The work by Uht [20] proved that the resource requirement necessary for the time optimal execution may increase exponentially for some loops with control flows. The work by Schwiegelshohn *et al.* [19], which is the best known and most significant result on time optimal programs, simply illustrated that certain loops with control flows do not have their equivalent time optimal programs. Since the work by Schwiegelshohn *et al.* was published, no further research results on the problem have been reported for about a decade, possibly having been discouraged by the pessimistic result.

Instead, most researchers focused on developing *better* software pipelining algorithms. To overcome the difficulty of handling control flows, many developed algorithms imposed unnecessarily strict constraints on possible transformations of software pipelining. For example, several software pipelining algorithms first apply transformations that effectively remove control flows before scheduling [4, 12], and recover control flows after scheduling [21]. Although practical, these extra transformations prohibit considerable amount of code motions, limiting the scheduling space exploration significantly.

*This work was supported by grant No.R01-2001-00360 from the Korea Science & Engineering Foundation.

ICS'02, June 22-26, 2002, New York, New York, USA.

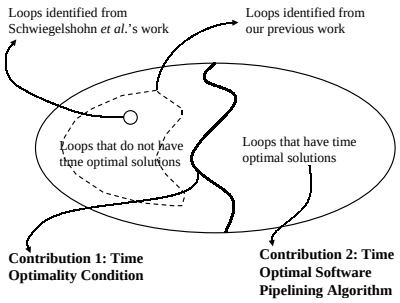

Figure 1: Loop classification based on time optimality

Recently, we have proposed a novel formalization of software pipelining of loops with control flows and, based on the formalization, suggested a necessary condition for loops with control flows to have equivalent time optimal programs [22]. Our previous work can be considered as a generalization of Schwiegelshohn *et al*.'s result, significantly expanding a set of loops that cannot have equivalent time optimal programs.

1.2 Contributions

In this paper, we are to identify exactly what can and cannot be achieved by software pipelining and to empirically evaluate how often software pipelining can generate optimal solutions in real applications. Our contributions can be divided into two parts, theoretical ones and practical ones.

For the theoretical contributions, we further extend our previous results and give answers to the following two fundamental open problems on time optimal software pipelining:

Question 1: Is there a decision procedure that determines if a loop has its equivalent time optimal program or not?

Question 2: For the loops that have the equivalent time optimal programs, is there an algorithm that computes time optimal programs for such loops?

For loops with control flows, these two questions have not been adequately formulated, let alone being solved, until we proposed a new formalization of software pipelining in [22]. In this paper, we call the necessary and sufficient condition for a loop to have its equivalent time optimal program as the *Time Optimality Condition*.

As an answer to the first question, we present the Time Optimality Condition and describe how to compute the Time Optimality Condition. For the second question, we present a software pipelining algorithm that computes time optimal programs for every loop satisfying the Time Optimality Condition.

Figure 1 summarizes our theoretical contributions graphically. The enclosing ellipse represents the set U of all the reducible innermost loops and the bold curve represents the boundary between two sets of loops, one set whose loops have equivalent time optimal programs (i.e., the right region) and the other set whose loops do not have time optimal programs (i.e., the left region). The small circle represents the set of loops shown to have no time optimal solutions by Schwiegelshohn *et al.* [19] while the region closed by the dashed curve represents the set of loops shown to have no time optimal solutions by our previous work [22]. The work described in this paper classifies all the loops in U into one of two sets, proves that the classification is decidable (i.e., each set is recursive) and shows that there exists an algorithm for computing time optimal solutions for eligible loops.

The optimal software pipelining algorithm, which is given to answer Question 2 above, enables us to complete the theoretical treatment on time optimal software pipelining. However, the algorithm

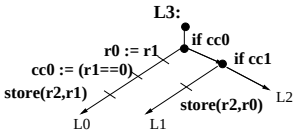

Figure 2: A tree VLIW instruction

is of little practical importance; it suffers from excessive overhead in computation time and code expansion. In the worst case, the overhead is inherently unavoidable [20]. As a practical alternative, we present a more realistic optimal software pipelining algorithm which runs faster with less code expansion and less hardware requirement. Unlike the former optimal algorithm, this algorithm guarantees optimal solutions when loops satisfy a stronger version of the Time Optimality Condition. According to our experimental observations, however, most loops satisfying the Time Optimality Condition satisfy the stronger version as well, which strongly indicates the practical significance of the proposed realistic software pipelining algorithm. (Note that this paper focuses on the theoretical results and that the experimental results are provided to emphasize the importance of the theoretical results in practice.)

The rest of the paper is organized as follows. We explain the machine model assumptions, program representation and dependence representation in Section 2. In Section 3, we present the Time Optimality Condition and describe how to compute it. In Sections 4 and 5, we present two optimal software pipelining algorithms, respectively. Experimental results are given in Section 6 and we conclude with a summary and directions for future work in Section 7.

2. PRELIMINARIES

2.1 Architectural Requirements

In order that the time optimality is well defined for loops with control flows, some architectural assumptions are necessary. In this paper, we assume the following architectural features for the target machine model: First, the machine can execute multiple branch operations (i.e., *multiway branching* [14]) as well as data operations concurrently. Second, it has an execution mechanism to commit operations depending on the outcome of branching (i.e, *conditional execution* [6]). The former assumption is needed because if multiple branch operations have to be executed sequentially, time optimal execution cannot be defined. The latter one is also indispensable for time optimal execution, since it enables to avoid output dependence of store operations which belong to different execution paths of a parallel instruction as pointed out by Aiken *et al.* [3].

As a specific example architecture, we use the tree VLIW architecture model [15], which satisfies the architectural requirements described above. In this architecture, a parallel VLIW instruction, called a tree instruction, is represented by a binary decision tree as shown in Figure 2. A tree instruction can execute simultaneously ALU and memory operations as well as branch operations. The branch unit of the tree VLIW architecture can decide the branch target in a single cycle [14]. An operation is committed only if it lies in the execution path determined by the branch unit [6].

2.2 Program Representation

We represent a sequential program P_s by a control flow graph (CFG) whose nodes are primitive machine operations. If the sequential program P_s is parallelized by a compiler, a *parallel tree VLIW program* P_{tree} is generated. While P_{tree} is the final output from the parallelizing compiler for our target architecture, we represent the parallel program in the *extended sequential representation* for the description purpose.

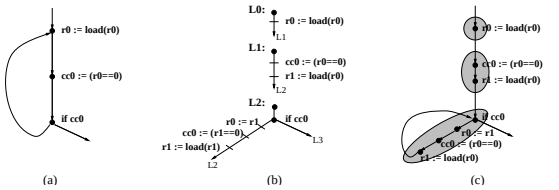

Figure 3: (a) A sequential program, (b) a parallel tree VLIW program, and (c) a parallel program in the extended sequential representation

Under the extended sequential representation, both sequential programs and parallel programs are described using the same notations and definitions used for the sequential programs. Compared to sequential programs, parallel programs include the additional information on operation grouping. Figure 3. (a) shows an input sequential program P_s and Figure 3. (b) shows its corresponding parallel tree VLIW program P_{tree}. Using the extended sequential representation, P_{tree} is represented by Figure 3. (c). The parallel program shown in Figure 3. (c) is based on a sequential representation except that it has the operation grouping information indicated by shaded regions. A group of operations in the shaded area indicates independently executable operations and is called a *parallel group*. A parallel group corresponds to a tree VLIW instruction and can be easily converted into the tree VLIW instruction with some local transformation on copy operations, and vice versa [15].

2.3 Basic Terminology

A program[1] is represented as a triple $\langle G=(N,E),\, O,\, \delta\rangle$. (This representation is due to Aiken *et al*. [3].) The body of the program is a CFG G which consists of a set of nodes N and a set of directed edges E. Nodes in N are categorized into *assignment* nodes that read and write registers or global memory, *branch* nodes that affect the flow of control, and special nodes, *start* and *exit* nodes. The execution begins at the start node and the execution ends at the exit nodes. E represents the possible transitions between the nodes. Except for branch nodes and exit nodes, all the nodes have a single outgoing edge. Each branch node has two outgoing edges while exit nodes have no outgoing edge.

O is a set of operations that are associated with nodes in N. The operation associated with $n \in N$ is denoted by $op(n)$. More precisely, $op(n)$ represents opcode and constant fields only; register fields are not included in $op(n)$.[2] Without loss of generality, every operation is assumed to write to a single register. We denote by $reg_{\text{W}}(n)$ the register to which n writes and by $regs_{\text{R}}(n)$ a set of registers from which n reads .

A configuration is a pair $\langle n,s\rangle$ where n is a node in N and s is a store (i.e., a snapshot of the contents of registers and memory locations). The transition function δ, which maps configurations into configurations, determines the complete flow of control starting from the initial store. Let n_0 be the start node and s_0 an initial store. Then, the sequence of configurations during an execution is $\langle\langle n_0,s_0\rangle,\cdots,\langle n_i,s_i\rangle,\cdots,\langle n_t,s_t\rangle\rangle$ where $\langle n_{i+1},s_{i+1}\rangle = \delta(\langle n_i,s_i\rangle)$ for $0 \le i < t$.

A *path* p of G is a sequence $\langle n_1,\cdots,n_k\rangle$ of nodes in N such that $(n_i,n_{i+1}) \in E$ for all $1 \le i < k$. For a given path p, the length of p is the number of nodes in p and denoted by $|p|$. The i-th ($1 \le i \le |p|$) node of p is addressed by $p[i]$. A path q is said to be a *subpath* of p, written $q \sqsubseteq p$, if there exists j ($0 \le j \le |p|-|q|$) such that $q[i] = p[i+j]$ for all $1 \le i \le |q|$. For a path p and i, j ($1 \le i \le j \le |p|$), $p[i,j]$ represents the subpath induced by the sequence of nodes from $p[i]$ up to $p[j]$. Given paths $p_1 = \langle n_1,n_2,\cdots,n_k\rangle$ and $p_2 = \langle n_k,n_{k+1},\cdots,n_l\rangle$, $p_1 \circ p_2 = \langle n_1,n_2,\cdots,n_k,n_{k+1},\cdots,n_l\rangle$ denotes the concatenated path between p_1 and p_2. A path p forms a cycle if $p[1] = p[|p|]$ and $|p| > 1$. For a given cycle c, c^k denotes the path constructed by concatenating c with itself k times. A path from the start node to one of exit nodes is called an *execution path* and distinguished by the superscript 'e' (e.g., p^{e}). An execution path of parallel program is further distinguished by the extra superscript 'sp' (e.g., $p^{\text{e,sp}}$).

It may incur some confusion to define execution paths for a parallel program because the execution of the parallel program consists of transitions among parallel instructions each of which consists of several nodes. With the conditional execution mechanism described in Section 2.1, however, we can focus on the unique committed path of each parallel instruction while pruning uncommitted paths. Then, like a sequential program, the execution of a parallel program flows along a single thread of control and corresponds to a path rather than a tree.

Some attributes such as redundancy and dependence should be defined in a flow-sensitive manner because they are affected by control flows. Flow-sensitive information can be represented by associating the past and the future control flow with each node. Given a node n and paths p_1 and p_2, the triple $\langle n,p_1,p_2\rangle$ is called a *node instance* if $n = p_1[|p_1|] = p_2[1]$. That is, a node instance $\langle n,p_1,p_2\rangle$ defines the execution context in which n appears in $p_1 \circ p_2$. In order to distinguish the node instance from the node itself, we use a boldface symbol like $\mathbf{n}$ for the former. The node component of a node instance $\mathbf{n}$ is addressed by $node(\mathbf{n})$. A trace of a path p, written $t(p)$, is a sequence $\langle \mathbf{n}_1,\mathbf{n}_2,\cdots,\mathbf{n}_{|p|}\rangle$ of node instances such that $\mathbf{n}_i = \langle p[i],p[1,i],p[i,|p|]\rangle$ for all $1 \le i \le |p|$. The i-th component of $t(p)$ is addressed by $t(p)[i]$ and the index of a node instance $\mathbf{n}$ in the trace $t(p)$ is represented by $pos(\mathbf{n})$. For a node instance $\mathbf{n} = \langle n,p_1,p_2\rangle$ in an execution path p^{e} in a sequential program, an attribute $it(\mathbf{n})$ is defined as the number of iterations which p_1 spans over.

From the formalization of software pipelining in [22], for an execution path $p^{\text{e,sp}}$ in a parallel program, there is a unique execution path $\alpha(p^{\text{e,sp}})$ in the sequential loop that corresponds to $p^{\text{e,sp}}$. Furthermore, for a node instance $\mathbf{n}$ in $p^{\text{e,sp}}$, there is a unique node instance $\beta(\mathbf{n})$ in $\alpha(p^{\text{e,sp}})$ that corresponds to $\mathbf{n}$.

Some of node instances in parallel programs are actually used to affect the control flow or the final store while the others are not. The former ones are said to be *effective* and the latter ones *redundant*. A node is said to be *non-speculative* if all of its node instances are effective. Otherwise it is said to be *speculative* [22].

2.4 Dependence Model

With the sound assumption of regular memory dependences, true dependence information can be easily represented for straight line loops thanks to the periodicity of dependence patterns. For loops with control flows, however, this is not the case and the dependence relationship between two nodes relies on the control flow between them. In order to model this type of dependence, we associate path information with the dependence relation. The dependences carried by registers are defined as follows.

DEFINITION 1. *For nodes n_1 and n_2 and a path p such that $p[1] = n_1, p[|p|] = n_2$, n_2 is said to be dependent on n_1 along p, written $n_1 \prec_p n_2$, if*

$$reg_{\text{W}}(n_1) \in regs_{\text{R}}(n_2) \quad \textit{and}$$
$$reg_{\text{W}}(p^{\text{e}}[i])) \neq reg_{\text{W}}(n_1) \quad \textit{for all } 1 < i < |p| \,.$$

[1] Since a parallel program is represented by the extended sequential representation, the notations and definitions explained in Section 2.3 and 2.4 apply to parallel programs as well as sequential programs.

[2] For two programs to be equivalent, only the dependence patterns of these are needed to be identical but not register allocation patterns. For this reason, register fields are not included in $op(n)$.

Furthermore, we can extend the dependence relation on node instances as follows:

DEFINITION 2. *Given a path p and i, j $(1 \le i < j \le |p|)$, $t(p^e)[j]$ is said to be dependent on $t(p^e)[i]$, written $t(p^e)[i] \prec t(p^e)[j]$, if $p[i] \prec_{p[i,j]} p[j]$.*

The dependence relation between two node instances with memory operations may be irregular even for straight line loops. Existing software pipelining techniques rely on conservative dependence analysis techniques, in which the dependence relationship between two node instances is determined by considering the iteration difference only and is usually represented by *data dependence graphs* [11] or its extensions [7, 18]. In our work, we assume a similar memory dependence relation, in which the dependence relation between two node n_1 and n_2 along p ($p[1] = n_1, p[|p|] = n_2$) rely only on the number of iterations that p spans.

Assuming regular memory dependences, straight-line loops can be transformed so that every memory dependence does not span more than an iteration by unrolling sufficient times. For loops with control flows, we assumed that they are unrolled sufficiently so that memory dependences do not span more than an iteration to simplify notations and the algorithm. This seems to be too conservative but we believe that the claims in this paper can be shown to be still valid in other memory dependence models with slight modifications to the proofs.

Now we are to define a *dependence chain* for sequential and the parallel programs.

DEFINITION 3. *Given a path p, a dependence chain $\mathbf{d}$ in p is a sequence of node instances $\langle \mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_k \rangle$ in $t(p)$ such that $\mathbf{n}_i \prec \mathbf{n}_{i+1}$ for all $1 \le i < k$. A dependence chain is said to be critical if it is the longest one in p. The i-th component of a dependence chain $\mathbf{d}$ is addressed by $\mathbf{d}[i]$ and the number of components in $\mathbf{d}$ is denoted by $|\mathbf{d}|$.*

3. TIME OPTIMALITY CONDITION

In this section, we present the Time Optimality Condition and describe how to compute it. Before presenting the Time Optimality Condition, we first formally define *time optimality*.

3.1 Time Optimality

For each execution path $p^{e,sp}$ in a software pipelined program $\mathcal{L}^{SP}$, the execution time of each node instance $\mathbf{n}$ in $t(p^{e,sp})$ can be counted from the corresponding parallel control flow graph and is denoted by $\tau(\mathbf{n})$. Time optimality of the parallel program $\mathcal{L}^{SP}$ is defined as follows [19, 3, 22]:

DEFINITION 4. *$\mathcal{L}^{SP}$ is time optimal if, for every execution path $p^{e,sp}$ in $\mathcal{L}^{SP}$, $\tau(t(p^{e,sp})[|p^{e,sp}|])$ is the length of the longest dependence chain in the execution path p^e.*

The definition is equivalent to saying that every execution path in $\mathcal{L}^{SP}$ runs in the shortest possible time subject to the true dependences. Note that the longest dependence chain in p^e is used instead of that in $p^{e,sp}$ because the latter may contain speculative nodes which should not be considered for the definition of time optimality. Throughout the remainder of the paper, the length of the longest dependence chain in a path p is denoted by $\|p\|$.

3.2 Time Optimality Condition

In Sections 3.3 and 4, we show that a loop $\mathcal{L}$ has an equivalent time optimal program if and only if the following condition is satisfied:

Condition I (Time Optimality Condition).

(a) There exists a constant $B_1 > 0$ such that for any path p in $\mathcal{L}$,

$$\|p[1,i]\| + \|p[i+1,|p|]\| \;\le\; \|p\| + B_1 \quad \text{for all } 1 \le i < |p| \quad \text{and}$$

(b) there exist constants $B_2, B_3 > 0$ such that for any path p in $\mathcal{L}$, $|p| \;\le\; B_2 \cdot \|p\| + B_3$.

Informally, the Time Optimality Condition requires that every operation be moved within a *bounded range* to yield the time optimal execution for every execution path. Condition I.(a) states that for any path p in $\mathcal{L}$, if the path p is splitted into two subpaths, the sum of the lengths of the longest dependence chains in each subpath can exceed the length of the longest dependence chain in p at most by B_1.

Condition I.(b) is rather trivial. It states that for any path p in $\mathcal{L}$, $|p|$ is bounded by a linear function of $\|p\|$. In other words, if $\mathcal{L}$ has an equivalent time optimal program, there exists a fairly long dependence chain for every path p in $\mathcal{L}$.

THEOREM 5. *Condition I is a necessary and sufficient condition for $\mathcal{L}$ to have an equivalent time optimal program.*

Section 3.3 gives a proof on the necessary part of Theorem 5. We have already proved a condition, which is slightly weaker than Condition I.(a), is a necessary condition in our previous work [22]. In Section 3.3, we prove that the previously proved condition implies Condition I.(a). We prove the sufficient part of Theorem 5 by construction, i.e., the proof for the sufficient part follows from the optimal software pipelining algorithm presented in Section 4. Condition I is intuitive and useful in deriving the theorems, but it is not obvious how to determine if a loop satisfies Condition I or not. If Condition I is to be directly computed from the expressions, every execution path should be enumerated, which is impossible. So we present another condition in Section 3.4 which is equivalent to Condition I and can be computed more easily.

3.3 Necessary Part of Theorem 5

If a loop $\mathcal{L}$ has an equivalent time optimal program $\mathcal{L}^{SP}$ but it does not satisfy Condition I, $\mathcal{L}^{SP}$ must exhibit some anomaly. If Condition I.(a) is not satisfied, an operation n_1 in $\mathcal{L}^{SP}$ should be executed infinitely earlier than n_2 that precedes n_1 in $\mathcal{L}$. In case that Condition I.(b) is not satisfied, infinitely many operations should be executed at the same time slot. We show that no *closed-form* parallel program satisfies this anomalous requirement. In our previous work [22], we have proved the following condition is a necessary condition, which is slightly weaker than Condition I.(a):

Necessary Condition I.

There exists a constant $B > 0$ such that for any execution path p^e in $\mathcal{L}$,

$$\|p^e[1,i]\| + \|p^e[j,|p^e|]\| \;\le\; \|p^e\| + B \quad \text{for all } 1 \le i < j \le |p^e|.$$

THEOREM 6. *Condition I is a necessary condition for $\mathcal{L}$ to have an equivalent time optimal program.*

Proof. To prove the above condition implies Condition I.(a), we first substitute $i+1$ for j in the above condition. Then it remains to show that the inequality also holds for every path, not only for every execution path. For a path p, let p_1 be a simple path from the

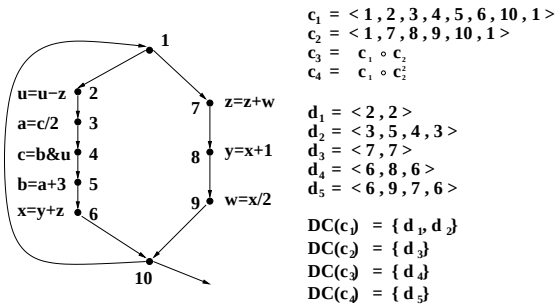

Figure 4: Dependence cycles

loop header to $p[1]$ and let p_2 be a simple path from $p[|p|]$ to an exit of $\mathcal{L}$. Then $p' = p_1 \circ p \circ p_2$ is an execution path of $\mathcal{L}$, and the above inequality holds for p'. Therefore, we have

$$\begin{aligned}
&\|p[1,i]\| + \|p[i+1,|p|]\| \\
&\quad \le \|p'[1,i+|p_1|-1]\| + \|p'[i+|p_1|,|p'|]\| \\
&\quad \le \|p'\| + B \le \|p\| + \|p_1\| + \|p_2\| + B \\
&\quad \le \|p\| + B + 2 \cdot L
\end{aligned}$$

where L is the length of the longest simple path in $\mathcal{L}$.

Suppose $\mathcal{L}$ has an equivalent time optimal program $\mathcal{L}^{\mathrm{SP}}$. Let B_2 be the maximum height among tree parallel instructions of $\mathcal{L}^{\mathrm{SP}}$ and let B_3 be $2 \cdot L \cdot B_2$. For a path p, we define p' to be the same path used for the proof of Condition I.(a). From the fact that $\mathcal{L}^{\mathrm{SP}}$ is time optimal and the definition of B_2, $|p'|$ is bounded by $B_2 \cdot \|p'\|$. Therefore, we have

$$|p| \;\le\; |p'| \;\le\; B_2 \cdot \|p'\| \;\le\; B_2 \cdot (\|p\| + 2 \cdot L) \;= B_2 \cdot \|p\| + B_3 \;. \;\square$$

3.4 Computing Time Optimality Condition

In this section, we explain how to compute the Time Optimality Condition. Directly computing the Time Optimality Condition requires that the infinitely many execution paths be enumerated, which is not possible. So, we derive another equivalent condition that can be checked in a finite number of steps.

Before presenting the new condition, we define a new term, a *dependence cycle*. For straight-line loops the concept of the dependence cycle is well known, but for loops with control flows, the dependence cycle has not been defined formally. We define the dependence cycle for each cyclic path in $\mathcal{L}$ as follows.

DEFINITION 7. *Given a cycle c (may not be simple) in $\mathcal{L}$, d is a dependence cycle with respect to c if there exist $l \ge 1$ and $1 \le i_1 < i_2 < \cdots < i_{|d|} \le l \cdot (|c|-1)$ such that*

$$\begin{aligned}
&i_1 \le |c| - 1 \;\wedge\; i_{|d|} = i_1 + (l-1) \cdot (|c|-1) \quad \textit{and} \\
&d[j] = c^l[i_j] \quad \textit{for} \quad 1 \le j \le |d| \quad \textit{and} \\
&d[j] \prec_{c^l[i_j, i_{j+1}]} d[j+1] \quad \textit{for} \quad 1 \le j < |d| \;.
\end{aligned}$$

Figure 4 shows an example of dependence cycles. We associate several attributes with the dependence cycle, which are defined below.

DEFINITION 8. *For a dependence cycle d, the sum of latencies of $d[1], d[2], \cdots, d[|d|-1]$ is denoted by $\delta(d)$. $span(d)$ denotes l in Definition 7 and $slope(d)$ is defined to be $\delta(d)/span(d)$. Further, $DC(c)$ represents the set of dependence cycles associated with c and $DC_{\mathrm{cr}}(c)$ represents the subset of $DC(c)$ that consists of all the dependence cycles with the maximum slope in $DC(c)$. A dependence cycle in $DC_{\mathrm{cr}}(c)$ is called a critical dependence cycle and its slope value is denoted by $max\_slope(c)$.*

There are a finite number of simple dependence cycles in $DC_{\mathrm{cr}}(c)$ as well as in $DC(c)$ and these dependence cycles can be enumerated using Johnson's algorithm [9]. It is also useful to define dependence relation on dependence cycles. Informally, d_2 is said to be dependent on d_1 if there is a dependence chain from a node in d_1 to one in d_2.

DEFINITION 9. *Given two cycles c_1 and c_2 in $\mathcal{L}$ such that $c_1[i_1] = c_2[i_2]$, d_2 is said to be dependent on d_1 ($d_1 \in DC(c_1)$, $d_2 \in DC(c_2)$), written $d_1 \prec^C d_2$, if*

$$\begin{aligned}
&\exists j_1 < j_2,\; d_1[j_1] \prec_p d_2[j_2] \quad \textit{for some } p \textit{ s.t.} \\
&p \sqsubseteq c_1^{span(d_1)+1} \circ c_1[1,i_1] \circ c_2[i_2,|c_2|] \circ c_2^{span(d_2)+1} \;.
\end{aligned}$$

If $d_1[k_1] = d_2[k_2]$ for some k_1 and k_2, d_1 and d_2 are said to be joined, written $d_1 \bowtie d_2$.[3]

Let $\mathbf{C} = \{c_1, c_2, \cdots\}$ represent the set of all the simple cycles in $\mathcal{L}$ starting from the loop header node and let $\mathbf{C}^k$ ($1 \le k \le |\mathbf{C}|$) and $\mathbf{C}^*$ be defined as follows:

$$\begin{aligned}
&\mathbf{C}^k = \{c_{i_1} \circ c_{i_2} \circ \cdots \circ c_{i_k} \mid \forall j \ne l, i_j \ne i_l \;\wedge\; \forall j > 1, i_1 < i_j\} \\
&\mathbf{C}^* = \cup_{k=1}^{|\mathbf{C}|} \mathbf{C}^k \;.
\end{aligned}$$

Then, the following condition is equivalent to Condition I.

Condition II.

(a) For any cycle c in $\mathbf{C}^*$, $DC(c)$ is not empty and

(b) For each cycle c_i ($1 \le i \le |\mathbf{C}^*|$) in $\mathbf{C}^*$, there exists a dependence cycle $d_i \in DC_{\mathrm{cr}}(c_i)$ such that $d_j \prec^C d_k$ for all $1 \le j < k \le |\mathbf{C}^*|$

It is possible to check if a loop satisfies the Condition II in a finite number of steps because only finite number of cycles need to be enumerated.

Let us consider the example loop shown in Figure 4. There are two simple cycles $c_1 = \langle 1,2,3,4,5,6,10,1\rangle$ and $c_2 = \langle 1,7,8,9,10,1\rangle$ in the loop. So, $\mathbf{C} = \{c_1, c_2\}$ and $\mathbf{C}^* = \mathbf{C}^1 \cup \mathbf{C}^2 = \{c_1, c_2, \} \cup \{c_1 \circ c_2 (= c_3)\} = \{c_1, c_2, c_3\}$. We can easily verify that Condition II.(a) is satisfied but Condition II.(b) is not satisfied; $d_2 = \langle 3,5,4,3\rangle$ and $d_3 = \langle 7,7\rangle$ are the unique elements in $DC_{\mathrm{cr}}(c_2)$ and $DC_{\mathrm{cr}}(c_3)$, respectively, but d_2 is not dependent on d_3.

LEMMA 10. *If a given loop $\mathcal{L}$ satisfies Condition I, it also satisfies Condition II.*

Proof. (a) is obviously satisfied. Suppose (b) is not satisfied for some c_1 and c_2. For $d_1 \in DC_{\mathrm{cr}}(c_1)$, select $d_2 \in DC_{\mathrm{cr}}(c_2)$ and $d_3 \in DC(c_1)$ such that $d_3 \prec^C d_2$ and $slope(d_3)$ is maximum. Note that every $d_2 \in DC_{cr}(c_2)$ may not be dependent on any dependence cycles in $DC(c_1)$ and then d_3 is set to be an imaginary null cycle.

Let $p(i) = c_1^{ai} \circ c_2^{abi} \circ p_f$ where p_f denotes any simple path from the unique loop header node to one exit and a, b are defined as follows.

$$a = \begin{cases} LCM(span(d_1), span(d_2)) & \text{if } d_3 \text{ is null} \;, \\ LCM(span(d_1), span(d_2), span(d_3)) & \text{otherwise} \;. \end{cases}$$

$$b = \lceil slope(d_1)/(slope(d_2) - r) \rceil$$

where r denotes the second largest slope in $DC(c_2)$.

It is evident that one of the longest dependence chain in $p(i)$ can be represented as

$$d_4^{\lfloor ai/span(d_4) \rfloor - 1} \circ p_1^D \circ d_5^{\lfloor abi/span(d_5) \rfloor - 1} \circ p_2^D$$

[3] Note that the $\bowtie$ relation is symmetric.

for some $d_4 \in DC(c_1)$, $d_5 \in DC(c_2)$, and dependence chains p_1^D and p_2^D. Therefore, we have

$$\begin{aligned} \|p(i)\| &\leq \delta(d_4)\cdot(ai/span(d_4)) + \delta(d_5)\cdot(abi/span(d_5)) + \alpha \\ &= slope(d_4)\cdot ai + slope(d_5)\cdot abi + \alpha \end{aligned}$$

for some constant α.
Case 1 : $d_5 \notin DC_{\rm cr}(c_2)$.
$slope(d_5) \leq r$ and $\|p(i)\| \leq slope(d_1)\cdot ai + r\cdot abi + \alpha_2$. From

$$\begin{aligned} & slope(d_1)\cdot a + r\cdot ab - slope(d_3)\cdot a - slope(d_2)\cdot ab \leq \\ & a\cdot(slope(d_1) + b\cdot(r - slope(d_2))) \leq \\ & a\cdot(slope(d_1) - slope(d_1)) = 0 \ , \end{aligned}$$

we have

$$\|p(i)\| \leq slope(d_3)\cdot ai + slope(d_2)\cdot abi + \alpha \ .$$

Case 2 : $d_5 \in DC_{cr}(c_2)$.
From the definition of d_3, $slope(d_4) \leq slope(d_3)$. So we have

$$\|p_1^e(i)\| \leq slope(d_3)\cdot ai + slope(d_2)\cdot abi + \alpha \ .$$

From the assumption, $d_3 \notin DC_{\rm cr}(c_1)$ and $slope(d_3) < slope(d_1)$. But we have

$$\begin{aligned} \|p_1(i)\| &\geq slope(d_1)\cdot ai \\ \|p_2(i)\| &\geq slope(d_2)\cdot abi \ . \end{aligned}$$

where $p_1(i) = p(i)[1,(|c|-1)\cdot ai]$ and $p_2(i) = p(i)[(|c|-1)\cdot ai + 1, \|p(i)\|]$. So,

$$\|p_1(i)\| + \|p_2(i)\| - \|p(i)\| \leq (slope(d_1) - slope(d_3))\cdot i - \alpha \ .$$

Therefore, Condition I is not satisfied, a contradiction. □

Before showing that the inverse proposition also holds, we introduce a new representation for cycles. As will be shown in Lemma 11, it is useful to represent a cycle by a composition of given subcycles. For example, consider a cycle c_5 shown in Figure 5.(a), given the subcycles c_1, c_2, c_3 and c_4.The cycle c_5 can be represented by a tree shown in Figure 5.(b).

Given a cycle c, the tree representation of c, written by $CT(c)$, can be found by the algorithm in [23]. Each node in $CT(c)$ represents a cycle in $\mathbf{C}^*$. Conversely, the sequence of a cycle represented by a tree can be found by the algorithm in [23]. For the sake of convenience, we use the following notation for cycles. Given a cycle c, $c(j)$ represents the same cycle as c but the sequence is shifted such that $c(j)[i] = c[(i+j-1 \bmod |c|)+1]$ for $1 \leq i \leq |c|$.

LEMMA 11. *For any cycle c in $\mathcal{L}$ such that $c \notin \mathbf{C}^*$,*

$$max\_slope(c) = \sum_{c_i \in CT(c)} max\_slope(c_i) \ .$$

Proof. For a critical dependence cycle d in c, we decompose d into critical dependence cycles in $CT(c)$. From Condition II.(b), d can be written as $d_j \circ d_k$, $(d_j \in DC_{\rm cr}(c_j))$ where c_j is a leaf node in $CT(c)$. Then it is obvious that $max\_slope(c) = max\_slope(c_j) + max\_slope(c')$ where $c(l) = c_j \circ c'(l')$ for some l and i'. By applying the same argument to c' recursively, we have $max\_slope(c) = \sum_{c_i \in CT(c)} max\_slope(c_i)$. □

For $|\mathbf{C}|^{|\mathbf{C}|}$ unknowns $\rho_{i_1,i_2,\cdots,i_{|\mathbf{C}|}}$ $(1 \leq i_1, i_2, \cdots, i_{|\mathbf{C}|} \leq |\mathbf{C}|)$, we solve the following linear system of $|\mathbf{C}^*|$ equations in the $|\mathbf{C}|^{|\mathbf{C}|}$ unknowns.

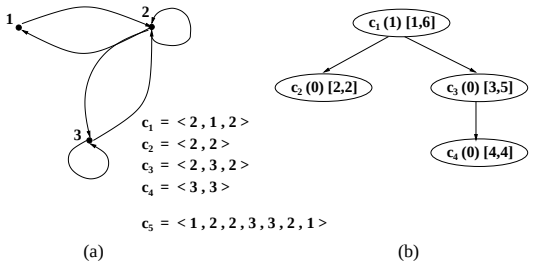

Figure 5: A new representation for a cycle: (a) A graph with cycles and (b) a tree representation of c_5

For each cycle $c = c_{j_1} \circ c_{j_2} \circ \cdots \circ c_{j_k} \in \mathbf{C}^*$,

$$\sum_{h=0}^{k-1} \rho_{j_{(1+h-1 \bmod k)+1}\ ,\ j_{(2+h-1 \bmod k)+1},\ \cdots\ ,j_{(|\mathbf{C}|+h-1 \bmod k)+1}} = max\_slope(c) \ .$$

By using a simple argument based on linear algebraic theorems, we can easily show that the linear system has a solution such that every $\rho_{i_1,i_2,\cdots,i_{|\mathbf{C}|}}$ is positive. (Actually, the solution is not unique and we select any one of them.) Given $\rho_{i_1,i_2,\cdots,i_{|\mathbf{C}|}}$, we can characterize the lengths of critical dependence chains. Let M_1 denote the length of the longest dependence chain in cycles $c_{i_1} \circ c_{i_2} \circ \cdots \circ c_{i_{|C|}-1}$ $(1 \leq i_1, i_2, \cdots, i_{|\mathbf{C}|} \leq |\mathbf{C}|)$ and let M_2 denote the length of the longest dependence chain in simple paths in $\mathcal{L}$.

LEMMA 12. *Given a path $p = p_{\rm s} \circ c_{i_1} \circ c_{i_2} \circ \cdots \circ c_{i_k} \circ p_{\rm f}$ in $\mathcal{L}$ where $k \geq |\mathbf{C}|$, $c_{i_j} \in |\mathbf{C}|$ for all $1 \leq j \leq k$ and both $p_{\rm s}$ and $p_{\rm f}$ are simple paths, let M_3 be*

$$\textstyle\sum_{h=0}^{k-|\mathbf{C}|} \rho_{i_{1+h}\ ,\ i_{2+h}\ ,\ \cdots\ ,\ i_{|\mathbf{C}|+h}} \ .$$

Then, $M_3 \leq \|p\| \leq M_1 + 2\cdot M_2 + M_3$.

Proof. Let $c' = c_{i_{|C|}} \circ c_{i_{|C|+1}} \circ \cdots \circ c_{i_k}$. Then $max\_slope(c')$ is equal to M_3 by Lemma 11. Therefore, we have

$$\begin{aligned} \|p\| &\leq \|p_{\rm s}\| + \|c_{i_1} \circ c_{i_2} \circ \cdots \circ c_{i_{|C|}-1}\| \\ &\quad + \|c_{i_{|C|}} \circ c_{i_{|C|+1}} \circ \cdots \circ c_{i_k}\| + \|p_{\rm f}\| \\ &\leq M_2 + M_1 + M_3 + M_2 \ = \ M_1 + 2\cdot M_2 + M_3 \ . \end{aligned}$$

Similarly,

$$\|p\| \geq \|c_{i_{|C|}} \circ c_{i_{|C|+1}} \circ \cdots \circ c_{i_k}\| = M_3 \ . \quad \square$$

From Lemma 12, we can compute the constants.

LEMMA 13. *If B_1 is selected as $2\cdot M_1 + 4\cdot M_2$, Condition I.(a) is satisfied.*

Proof. For a path $p = p_{\rm s} \circ c_{i_1} \circ c_{i_2} \circ \cdots \circ c_{i_k} \circ p_{\rm f}$ in $\mathcal{L}$ we split p into two subpaths p_1 and p_2. Than p_1 and p_2 can be written as

$$\begin{aligned} p_1 &= p_{s_1} \circ c_{i_1} \circ \cdots \circ c_{i_l} \circ p_{f_1} \quad \text{and} \\ p_2 &= p_{s_2} \circ c_{i_{l+2}} \circ \cdots \circ c_{i_k} \circ p_{f_2} \ . \end{aligned}$$

By Lemma 12 we have

$$\begin{aligned} & \|p_1\| + \|p_2\| - \|p\| \leq \\ & \quad M_1 + 2\cdot M_2 + \textstyle\sum_{h=0}^{l-|\mathbf{C}|} \rho_{i_{1+h}\ ,\ i_{2+h}\ ,\ \cdots\ ,\ i_{|\mathbf{C}|+h}} \ + \\ & \quad M_1 + 2\cdot M_2 + \textstyle\sum_{h=l+1}^{k-|\mathbf{C}|} \rho_{i_{1+h}\ ,\ i_{2+h}\ ,\ \cdots\ ,\ i_{|\mathbf{C}|+h}} \ - \\ & \quad \textstyle\sum_{h=0}^{k-|\mathbf{C}|} \rho_{i_{1+h}\ ,\ i_{2+h}\ ,\ \cdots\ ,\ i_{|\mathbf{C}|+h}} \\ & \qquad \leq 2\cdot M_1 + 4\cdot M_2 \ . \quad \square \end{aligned}$$

LEMMA 14. *If B_2 and B_3 are selected as*

$$B_2 = \max\{ \frac{|c_j|}{\rho_{i_1,i_2,\cdots,i_{|\mathbf{C}|}}} \mid c_j \in |\mathbf{C}|\ ,\ 1 \le i_1, i_2, \cdots, i_{|\mathbf{C}|} \le |\mathbf{C}|\}$$

$$B_3 = 2 \cdot L_C$$

where L_C is the length of the longest simple cycle in $\mathcal{L}$, Condition I.(b) is satisfied.

Proof. For a path $p = p_s \circ c_{i_1} \circ c_{i_2} \circ \cdots \circ c_{i_k} \circ p_f$ in $\mathcal{L}$,

$$\begin{aligned} |p| \le & \sum_{h=1}^{k} (|c_{i_h}| - 1) + 2 \cdot L_C \\ \le & \sum_{h=0}^{k-|\mathbf{C}|} \left(\frac{|c_{i_h}|}{\rho_{i_{1+h}\ ,\ i_{2+h}\ ,\ \cdots\ ,\ i_{|\mathbf{C}|+h}}} \cdot \rho_{i_{1+h}\ ,\ i_{2+h}\ ,\ \cdots\ ,\ i_{|\mathbf{C}|+h}}\right) + B_3 - k \\ \le & B_2 \cdot \sum_{h=0}^{k-|\mathbf{C}|} \rho_{i_{1+h}\ ,\ i_{2+h}\ ,\ \cdots\ ,\ i_{|\mathbf{C}|+h}} + B_3 \\ \le & B_2 \cdot \|p\| + B_3\ . \qquad \text{(By Lemma 12.)} \qquad \square \end{aligned}$$

Note that all the constants B_1, B_2 and B_3 can be computed in finite time.

LEMMA 15. *If a given loop $\mathcal{L}$ satisfies Condition II, it also satisfies Condition I.*

Proof. Directly from Lemmas 13 and 14. □

THEOREM 16. *Condition I is decidable.*

Proof. From Lemmas 10 and 15, Condition I is equivalent to Condition II, whose decision procedure is obvious from the given expression. □

4. TIME OPTIMAL SOFTWARE PIPELINING ALGORITHM

In this section, we present a software pipelining algorithm that computes a time optimal parallel program for every loops satisfying Condition I. (The result in this section also serves as the proof for the sufficient part of Theorem 5.) The time-optimal software pipelining algorithm is mostly based on the algorithm by Aiken *et al*. [3], the latest version of *Perfect Pipelining* [2].

We first present the software pipelining algorithm by explaining our modifications to the Aiken's algorithm. Then, we prove that the output of the algorithm is a time optimal parallel program if the input loop satisfies Condition I.

4.1 The Algorithm

In this section, without loss of generality, we assume that every operation takes 1 cycle to execute. An operation that takes k cycles can be transformed into a chaining of k unit-time `delay` pseudo operations, which can be safely eliminated after scheduling. We assume that an arbitrary but fixed loop $\mathcal{L}$ satisfies Condition I.

Before scheduling, a sequential loop is unrolled infinite times to form an infinite (but recursive) CFG and then the infinite CFG is incrementally compacted by semantic-preserving transformations of Percolation Scheduling [16]. During scheduling, the algorithm finds equivalent nodes n and n' in the infinite CFG, deletes the infinite subgraph below n', and adds backedges from the predecessors of n' to n. In this way, the infinite CFG eventually becomes a finite parallel graph.

The Aiken's original algorithm does not handle false dependences appropriately [3]. An operation node which is blocked by the false dependences but not by true dependences may not be available for scheduling. To compute a time optimal solution, the false dependences should be overcome so that the parallel schedule is constrained by the true dependences only. We modify the Aiken's original algorithm so that the infinite CFG is put into the static single assignment (SSA) form [5], the SSA form is software pipelined into a finite parallel graph, and then the finite parallel graph is translated back out of the SSA form.

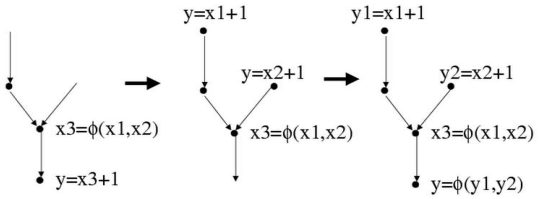

Figure 6: Scheduling above a ϕ-function at the join point

By translating into the SSA form, the false dependences are completely eliminated because every variables are defined by exactly one operation. Moreover, extra ϕ-functions do not incur additional true dependences because the operations that use the target registers of the ϕ-functions can always be combined with the ϕ-functions and be moved above the ϕ-functions. For example, in Figure 6, `y=x3+1` is to be scheduled above `x3=ϕ(x1,x2)`. The operation `y=x3+1` is combined with `x3=ϕ(x1,x2)` and split into `y=x1+1` and `y=x2+1`. Furthermore, to maintain the SSA form even after code motion above the join point, a new ϕ-function is introduced at the join point. In Figure 6, two `y` definitions are replaced by the `y1` and `y2` definitions and a new ϕ-function, `y=ϕ(y1,y2)`, is added.

If an operation is not true-dependent on any operations (except ϕ-functions) in a path, it can always be moved along the path even if it is not free from the false dependences in the original program. When translating a software pipelined program out of the SSA form, some copies may remain, but all the unremovable copy operations can be executed concurrently with any operations that are dependent on the copy operation.

Before describing the algorithm, we define some additional notations. Let $\mathcal{L}^\infty$ represent the infinite recursive graph obtained by unrolling $\mathcal{L}$ infinite times. For a node n in $\mathcal{L}$, let n^i denote the corresponding node in the i-th unrolled copy of $\mathcal{L}$ in $\mathcal{L}^\infty$. For a set X of nodes in $\mathcal{L}^\infty$, X^j is defined to be the set $\{n^{i+j} | n^i \in X\}$. Two sets of nodes in $\mathcal{L}^\infty$, X_1 and X_2, are said to be *equivalent* if $X_1 \equiv X_2^k$ for some k.

The proposed time-optimal software pipelining algorithm begins with $\mathcal{L}^\infty$, an acyclic infinite CFG, and successively transforms $\mathcal{L}^\infty$ into $\mathcal{L}^{SP}$ which consists of parallel groups. Figure 7 describes the overall processing steps of the software pipelining algorithm. The procedure SOFTWARE_PIPELINE calls the SCHEDULE_PARALLEL_GROUP procedure [23] to build a parallel group, and then to build parallel groups for all the branches of that group, and so on. If at any point the algorithm encounters the equivalent set of available operation nodes in the second time, it uses the previously scheduled parallel group.

Before building a parallel group, the COMPUTE_AVAILABLE_OPERATIONS procedure [23] is invoked to compute the set of all available operation nodes that can move into the parallel group without violating the true dependences.[4] In our algorithm, every operation node that is not blocked by the true dependences is always available for scheduling. As in [3], we impose additional constraint on available operations: operations are available at most k iterations. The predetermined constant k is called a *sliding window* [3] and it guarantees the termination of the **while** loop in the SOFTWARE_PIPELINING procedure.

Once the available operation nodes are computed, the SCHEDULE_PARALLEL_GROUP procedure repeatedly moves the opera-

[4] This procedure is functionally equivalent to the same procedure in the Moon's algorithm [15].

```
procedure SOFTWARE_PIPELINE ( L, window_size )
    L' := L^∞
    translate L' into the SSA form
    frontiers := {(n_start, n_root)}
    scheduled_before := { }
    back_edges := { }
    while (∃ (n_p, n_s) ∈ frontiers)
        frontiers := frontiers − {(n_p, n_s)}
        A := COMPUTE_AVAILABLE_OPERATIONS( L', n_s, window_size)
        if (∃A' ∈ scheduled_before s.t. A' and A are equivalent)
            n' := parallel_group_root[A']
            replace (n_p, n_s) by (n_p, n') and
              delete unreachable nodes from L'
            back_edges := back_edges ∪ {(n_p, n')}
        else
            SCHEDULE_PARALLEL_GROUP( L', n_s, A, frontiers)
            scheduled_before := scheduled_before ∪ {A}
        end if
    end while
    foreach ((n_p, n_s) ∈ back_edges)
        INSERT_CONSISTENCY_COPIES( L', n_p, n_s)
    end foreach
    translate L' back out of the SSA form
    remove dead operation nodes
    return L'
end function
```

Figure 7: The time-optimal software pipelining algorithm.

tion nodes to a group boundary [15].[5] When a branch operation node is moved, the group boundary is split into multiple boundaries. When moving up an operation node, ϕ-functions may be encountered. In this case, the scheduled operation node is combined with the ϕ-functions as described in the COMBINE_SOURCE_REGISTERS procedure [23]. The correctness of the algorithm is proved in [23].

From the greediness of the algorithm, along with our modifications in the renaming framework (which has the effect of removing the false dependences), the algorithm exhibits the following property.

LEMMA 17. *Let $\mathcal{L}^{\mathrm{SP}}$ be the result of the software pipelining algorithm with the sliding window of k iterations. Then for an effective node instance $\mathbf{n}$ in an execution path $p^{\mathrm{e,sp}}$ in $\mathcal{L}^{\mathrm{SP}}$ such that $\tau(\mathbf{n}) > 1$, there must exist an effective node instance $\mathbf{n}'$ in $p^{\mathrm{e,sp}}$ such that*

$$\tau(\mathbf{n}') = \tau(\mathbf{n}) - 1 \wedge (\beta(\mathbf{n}') \prec \beta(\mathbf{n}) \vee it(\beta(\mathbf{n})) - it(\beta(\mathbf{n}')) > k) .$$

Proof. Suppose that such $\mathbf{n}'$ does not exist and consider the execution snapshot of the SOFTWARE_PIPELINE procedure when the set of available operations for the predecessor parallel group Ω of $\beta(\mathbf{n})$ is computed. For some path from the group boundary of Ω to $\beta(\mathbf{n})$, there cannot exist any node on which $\beta(\mathbf{n})$ is true-dependent. Otherwise, some node on which $\beta(\mathbf{n})$ is true-dependent should be scheduled into Ω so that $\beta(\mathbf{n})$ can be scheduled into the successor parallel group of Ω, which contradicts the assumption.

Furthermore, $it(\beta(\mathbf{n}))$ can exceed $min\{it(\mathbf{n}'')|\mathbf{n}'' \in \Omega\}$ at most by k. Therefore, when the parallel group Ω is built, the COMPUTE_AVAILABLE_OPERATIONS procedure computes $\beta(\mathbf{n})$ as available and $\beta(\mathbf{n})$ must be scheduled into Ω, a contradiction. □

4.2 Time Optimality of the Algorithm

The software pipelining algorithm described in Figure 7 always generates time optimal parallel programs for loops that satisfy Condition I. The proof is based on the greediness of the algorithm.

[5] Since the transformations in the SCHEDULE_PARALLEL_GROUP procedure can be implemented using transformations described in the Moon's algorithm whose correctness has been already proved [15], they preserve program semantics.

Before presenting the time optimality proof, we prove some miscellaneous properties stated below in Lemmas 18 and 19. (Recall that we have assumed that $\mathcal{L}$ satisfies Condition I and that every operation takes 1 cycle to execute.)

LEMMA 18. *For a path p in $\mathcal{L}$ and $1 = i_1 < i_2 < \cdots < i_l \leq |p|$,*

$$\sum_{j=1}^{l-1} \|p[i_j, i_{j+1}]\| \;\leq\; \|p\| + (l-2)\cdot(B_1+1) .$$

Proof.

$$\begin{aligned}
\|p\| &\geq \|p[i_1,i_2]\| + \|p[i_2+1,i_l]\| - B_1 \\
&\geq \|p[i_1,i_2]\| + \|p[i_2,i_l]\| - 1 - B_1 \\
&\geq \|p[i_1,i_2]\| + (\|p[i_2,i_3]\| + \|p[i_3,i_l]\| - 1 - B_1) - 1 - B_1 \\
&\geq \cdots \;\geq \sum_{k=1}^{l-1} \|p[i_k,i_{k+1}]\| - (l-2)\cdot(B_1+1) \;. \qquad \square
\end{aligned}$$

LEMMA 19. *For node instances $\mathbf{n}_1$ and $\mathbf{n}_2$ in a path p in $\mathcal{L}$ such that $it(\mathbf{n}_2) - it(\mathbf{n}_1) > k$,*

$$\|p[pos(\mathbf{n}_1), pos(\mathbf{n}_2)]\| \geq \lceil \frac{(L-1)\cdot k + 1 - B_3}{B_2} \rceil$$

where L is the length of the shortest cycle in $\mathcal{L}$.

Proof. Since $\mathbf{n}_1$ and $\mathbf{n}_2$ are separated by more than k iterations, the number of node instances between them is at least $(L-1)\cdot k$. From Condition I.(b) we can write

$$\begin{aligned}
\|p[pos(\mathbf{n}_1), pos(\mathbf{n}_2)]\| &\geq \lceil \frac{pos(\mathbf{n}_2) - pos(\mathbf{n}_1) + 1 - B_3}{B_2} \rceil \\
&\geq \lceil \frac{(L-1)\cdot k + 1 - B_3}{B_2} \rceil . \quad \square
\end{aligned}$$

We are now ready to prove the time optimality of the software pipelining algorithm. The SOFTWARE_PIPELINING procedure requires the size of sliding window as an input parameter. To achieve the time optimality, we select the sliding window size as

$$WS = \lceil \frac{2\cdot B_2\cdot(B_1+1) + B_3}{L-1} \rceil \qquad (1)$$

where L is the length of the shortest cycle in $\mathcal{L}$.

LEMMA 20. *Let $\mathcal{L}^{\mathrm{SP}}$ be the result of the software pipelining algorithm with the sliding window of WS iterations. Then $\mathcal{L}^{\mathrm{SP}}$ is time optimal.*

Proof. It suffices to show that for an arbitrary but fixed execution path $p^{\mathrm{e,sp}}$ in $\mathcal{L}^{\mathrm{SP}}$, $\tau(t(p^{\mathrm{e,sp}})[|p^{\mathrm{e,sp}}|]) = \|\alpha(p^{\mathrm{e,sp}})\|$. Let p denote $\alpha(p^{\mathrm{e,sp}})$ and $G_{\mathrm{D}}(N_{\mathrm{D}}, E_{\mathrm{D}})$ be a directed graph such that N_{D} is the set of node instances in $t(p)$ and $E_{\mathrm{D}} = E'_{\mathrm{D}} \cup E''_{\mathrm{D}}$ where

$$\begin{aligned}
E'_{\mathrm{D}} &= \{\, (\mathbf{n}_1, \mathbf{n}_2) \mid \mathbf{n}_1 \prec \mathbf{n}_2 \,\} \\
E''_{\mathrm{D}} &= \{\, (\mathbf{n}_1, \mathbf{n}_2) \mid it(\mathbf{n}_2) - it(\mathbf{n}_1) > WS \}.
\end{aligned}$$

We first show that the length of the longest path in G_{D} is equal to the length of the longest path in $G'_{\mathrm{D}}(N_{\mathrm{D}}, E'_{\mathrm{D}})$, the subgraph of G_{D} induced by E'_{D}. Suppose that there exists a path $\mathbf{p}_{\mathrm{D}} = \mathbf{n}_1 \rightarrow \mathbf{n}_2 \rightarrow \cdots, \rightarrow \mathbf{n}_h$ in G_{D} whose length is larger than the length of the longest path in G'_{D} (which is equal to $\|p\|$). Then, there must exist s (≥ 1) edges $(\mathbf{n}_{i_1}, \mathbf{n}_{i_1+1}), \cdots, (\mathbf{n}_{i_s}, \mathbf{n}_{i_s+1})$ $(i_1 < i_2 < \cdots < i_s)$ in $\mathbf{p}_{\mathrm{D}}$ that come from E''_{D}. So, we have

$$\begin{aligned}
\|p\| \;&<\; |\mathbf{p}_{\mathrm{D}}| = i_1 + \sum_{j=1}^{s-1}(i_{j+1} - i_j) + h - i_s \\
&\leq \|p[1, pos(\mathbf{n}_{i_1})]\| + \sum_{j=1}^{s-1} \|p[pos(\mathbf{n}_{i_j+1}), pos(\mathbf{n}_{i_{j+1}})]\| \\
&\quad + \|p[pos(\mathbf{n}_{i_s+1}), \|p\|]\| \;. \qquad (2)
\end{aligned}$$

From Lemma 18, we can write

$$\begin{aligned}\|p\| \geq\ & \|p[1, pos(\mathbf{n}_{i_1})]\| + \textstyle\sum_{j=1}^{s-1} \|p[pos(\mathbf{n}_{i_j+1}), pos(\mathbf{n}_{i_{j+1}})]\| + \\ & \|p[pos(\mathbf{n}_{i_s+1}), \|p\|]\| + \textstyle\sum_{j=1}^{s} \|p[pos(\mathbf{n}_{i_j}), pos(\mathbf{n}_{i_j+1})]\| \\ & -2s\cdot(B_1+1)\ . \end{aligned} \tag{3}$$

From (2) and (3), we have

$$\textstyle\sum_{j=1}^{s} \|p[pos(\mathbf{n}_{i_j}), pos(\mathbf{n}_{i_j+1})]\| \ <\ 2s\cdot(B_1+1)\ . \tag{4}$$

Since $(\mathbf{n}_{i_j+1}, \mathbf{n}_{i_j}) \in E''_{\mathrm{D}}$, $it(\mathbf{n}_{i_j+1}) - it(\mathbf{n}_{i_j}) > WS$.
Therefore, by Lemma 19, we have for all $1 \leq i \leq s$

$$\begin{aligned}\|p[pos(\mathbf{n}_{i_j}), pos(\mathbf{n}_{i_j+1})]\| &\geq \lceil \frac{(L-1)\cdot WS + 1 - B_3}{B_2} \rceil \\ &\geq 2\cdot B_1 + 2\ , \end{aligned}$$

which contradicts (4). So the assumption is false and the length of the longest path in G_{D} is equal to the length of the longest path in G'_{D}, which is equal to $\|p\|$.

Let $\sigma(\mathbf{n})$ denote the length of the longest path in G_{D} that reaches $\mathbf{n}$. For $1 \leq i \leq \|p^{\mathrm{e,sp}}\|$, we are to show that

$$\tau(t(p^{\mathrm{e,sp}})[i]) \leq \sigma(\beta(t(p^{\mathrm{e,sp}})[i]))$$

when $t(p^{\mathrm{e,sp}})[i]$ is an effective node instance. The proof is by induction on i. Let m be the largest integer such that $\tau(t(p^{\mathrm{e,sp}})[i]) = 1$. Then, the proposition holds trivially for all $1 \leq i \leq m$. For the induction step, assume that the proposition holds for all $1 \leq j < i$. By Lemma 17, there must exist $i' < i$ such that

$$\begin{aligned}& t(p^{\mathrm{e,sp}})[i'] \text{ is an effective node instance} \quad \text{and} \\ & \tau(t(p^{\mathrm{e,sp}})[i']) = \tau(t(p^{\mathrm{e,sp}})[i]) - 1 \quad \text{and} \\ & (\ \beta(t(p^{\mathrm{e,sp}})[i']) \prec \beta(t(p^{\mathrm{e,sp}})[i]) \quad \vee \\ & \qquad it(\beta(t(p^{\mathrm{e,sp}})[i])) - it(\beta(t(p^{\mathrm{e,sp}})[i'])) > WS\ ) \end{aligned} \tag{5}$$

In any cases, $(\ \beta(t(p^{\mathrm{e,sp}})[i'])\ ,\ \beta(t(p^{\mathrm{e,sp}})[i])\ ) \in E''_{\mathrm{D}}$. Therefore, by the definition of σ, we have

$$\sigma(\beta(t(p^{\mathrm{e,sp}})[i])) \ \geq\ \sigma(\beta(t(p^{\mathrm{e,sp}})[i'])) + 1\ . \tag{6}$$

From (5), (6) and the induction hypothesis, we have

$$\begin{aligned}\tau(t(p^{\mathrm{e,sp}})[i]) &= \tau(t(p^{\mathrm{e,sp}})[i']) + 1 \\ &\leq \sigma(\beta(t(p^{\mathrm{e,sp}})[i'])) + 1 \leq \sigma(\beta(t(p^{\mathrm{e,sp}})[i]))\ . \end{aligned}$$

Therefore, we have

$$\tau(t(p^{\mathrm{e,sp}})[k]) \leq \sigma(\beta(t(p^{\mathrm{e,sp}})[k])) = \|p\|$$

where k is the largest integer such that $t(p^{\mathrm{e,sp}})[k]$ is an effective node instance.

To finish the proof, we need to show that redundant node instances do not affect the length of the schedule. Effective node instances are not dependent on redundant node instances. Furthermore, there cannot exist a redundant node instance following the last effective node instance. This is because every node instance following the last effective branch node is guaranteed to be effective by the dead code elimination after the scheduling. □

From Lemma 20, we can state the following theorem.

THEOREM 21. *Condition I is a sufficient condition for $\mathcal{L}$ to have an equivalent time optimal program.*

From Lemma 20, the algorithm in Figure 7 is a time-optimal software pipelining algorithm, provided that the size of sliding window is computable. From Lemmas 13 and 14, B_1, B_2 and B_3 can be computed in a finite number of steps. The size of sliding window can be directly computed from Equation (1). So, we have the following theorem.

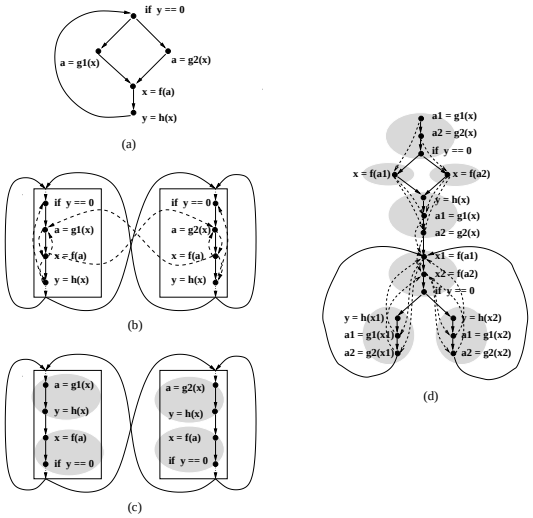

Figure 8: (a) A CFG before scheduling, (b) its corresponding NCFG, (c) the software-pipelined NCFG, and (d) the (time-optimally) software-pipelined CFG. (Solid lines and dashed lines represent control flows and dependences, respectively. Each shaded region represents a parallel group.)

THEOREM 22. *There exists a software pipelining algorithm that computes time optimal programs for loops that satisfy Condition I.*

5. A PRACTICAL SOFTWARE PIPELINING ALGORITHM

In this section, we present a more practical software pipelining algorithm. The software pipelining algorithm uses an intermediate program representation called nondeterministic control flow graph (NCFG)[6] proposed by Milicev [13]. As shown in Figure 8, the original control flow graph (CFG) of a loop (Figure 8.(a)) is transformed into an NCFG (Figure 8.(b)) and the software pipelining algorithm is applied to the NCFG. Then, the software pipelined NCFG (Figure 8.(c)) is transformed back into an equivalent CFG (Figure 8.(d)). In Section 5.2, we present a software pipelining algorithm that computes a time optimal NCFG for every loop satisfying a new condition, which is a stronger version of the Time Optimality Condition. Before describing the software pipelining algorithm, we first explain the NCFG.

5.1 Nondeterministic Control Flow Graph

The NCFG can be understood as a nondeterministic version of the standard control flow graph (CFG).[7] There is a one-to-one correspondence between NCFGs and CFGs, as is the case with nondeterministic finite automata (NFA) and deterministic finite automata (DFA). Given a CFG G of a loop (before software pipelining), let $P^{\mathrm{s}} = \{p^{\mathrm{s}}_1, p^{\mathrm{s}}_2, \cdots\}$ represent the set of all the acyclic paths starting from the loop header to a predecessor of the loop header or a loop exit. Then the corresponding NCFG G^{NCFG} is simply defined as

[6] Milicev used the term 'predicate matrix'. For the rest of the paper, we use 'NCFG' instead of 'predicate matrix', since the former is much more intuitive.

[7] We apply the notations and definitions explained in Sections 2.3, 2.4 and 3.4 to the NCFG as well.

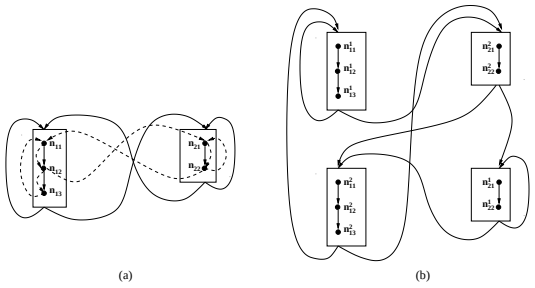

Figure 9: An example of split transformation

follows:

$$N^{\mathrm{NCFG}} = \{n_{i,j} \mid 1 \le i \le |P^{\mathrm{S}}|\ ,\ 1 \le j \le l_i\}$$
$$E^{\mathrm{NCFG}} = \{(n_{i,j}, n_{i,j+1}) \mid 1 \le i \le |P^{\mathrm{S}}|\ ,\ 1 \le j < l_i\} \cup \{(n_{i,l_i}, n_{i',1}) \mid 1 \le i\ ,\ i' < |P^{\mathrm{S}}|\}\ ,$$

where $l_i = |p_i^{\mathrm{S}}|$ and $n_{i,j}$ has the same attributes as $p_i^{\mathrm{S}}[j]$ (e.g., *op*, reg_{W} and $regs_{\mathrm{R}}$). The path $\langle n_{i,1}, n_{i,2}, \cdots, n_{i,|l_i|}\rangle$ forms a *nondeterministic basic block (NBB)*, which is denoted by $\mathbf{b}_i$. Each node $n \in N^{\mathrm{NCFG}}$ belongs to exactly one NBB and the NBB is addressed by $\mathbf{b}(n)$. An NCFG can be abstracted into an *NBB-graph* G^{NBB} whose nodes are the NBBs of the NCFG. Initially, G^{NBB} is a complete graph.

The CFG in Figure 8.(a) has two acyclic paths from the loop header to its predecessor and they correspond to two NBBs of the NCFG in Figure 8.(b). Informally, if a node is contained in more than one path of the CFG, it is copied into the corresponding NBBs of the NCFG.

The original NCFG is expanded by the *split* transformation. the original NCFG G^{NCFG} is transformed into a *k-level split* NCFG G_k^{NCFG} by splitting each NBB of G^{NCFG} into $|N^{\mathrm{NBB}}|^k$ copies. Since each copy of an NBB contains the same operations, we describe the NBB-graph G_k^{NBB} of G_k^{NCFG} to define the split transformation:

$$N_k^{\mathrm{NBB}} = \{\mathbf{b}_i^m \mid 1 \le i \le |N^{\mathrm{NBB}}|\ ,\ 1 \le m \le |N^{\mathrm{NBB}}|^k\}$$
$$E_k^{\mathrm{NBB}} = \{(\mathbf{b}_i^m, \mathbf{b}_{i'}^{m'}) \mid m' = |N^{\mathrm{NBB}}|^{k-1} \cdot (i-1) + \left\lceil \frac{m-1}{|N^{\mathrm{NBB}}|} \right\rceil + 1\ \}\ .$$

Figures 9.(a) and 9.(b) show an NCFG G^{NCFG} and its 1-level split version, G_1^{NCFG}.

A software-pipelined NCFG is transformed back into an executable CFG [13], which is similar to the NFA-to-DFA transformation. A nice property of an NCFG is that the execution time of any path in the NCFG is equal to that in the corresponding CFG. Therefore, it suffices to build a time-optimally software pipelined NCFG.

5.2 The Software Pipelining Algorithm

As with several software pipelining algorithms based on modulo scheduling, our software pipelining algorithm decouples the computation of a schedule and code motions. After computing a schedule, the code motions that are implicit in the schedule are ascertained subsequently.

The scheduling algorithm first determines the *latency* of each NBB of the NCFG based on integer linear programming and then computes each operation's *time offset* from the beginning of its NBB. Informally, the latency of an NBB can be understood as the initiation interval (II). For example, the latency of the left NBB of the NCFG in Figure 8.(b) is the II of the left path of the CFG in Figure 8.(a). We denote the latency of an NBB $\mathbf{b}$ by $\rho(\mathbf{b})$ and the time offset of $n \in N^{\mathrm{NCFG}}$ by $\sigma(n)$. Given an execution path p and a set of the latencies and the time offsets, the execution time of $p[k]$ is given by

$$\tau(t(p)[k]) = \sum_{l=1}^{|\mathbf{bp}(p[1,k])|-1} \rho(\mathbf{bp}(p)[l]) + \sigma(p[k]) \qquad (7)$$

where $\mathbf{bp}(p)$ denotes a path in the NBB-graph which corresponds to p in the NCFG graph. Thus, by determining the latencies and the time offsets, we essentially build a software-pipelined schedule.

The latencies of NBBs are determined such that, for any simple cycle $\mathbf{c}$ in the NBB-graph, the sum of the latencies of NBBs in $\mathbf{c}$ is equal to the slope of the critical dependence cycle in the corresponding cycle c in the NCFG, i.e., $\sum_{i=1}^{|\mathbf{c}|-1} \rho(\mathbf{c}[i]) = max\_slope(c)$. We call such a tuple of latencies as a *tight* tuple. However, a tight tuple does not always exist because the number of equations may be larger than the number of variables (the unknown latencies). This can be resolved by the split transformation, which increases the number of NBBs (equivalently, the variables).

The split transformation also increases the number of simple cycles in the NBB-graph incurring additional equations. But, some of the newly introduced equations may be linearly dependent on other equations and, consequently, the number of variables may exceed the number of equations. In Figure 9.(a), there are three simple cycles in the NCFG but only two nodes in NCFG. Therefore, no solution exists for the linear equations

$$\rho(\mathbf{b}_1) = 3\ ,\ \rho(\mathbf{b}_2) = 2\ ,\ \rho(\mathbf{b}_1) + \rho(\mathbf{b}_2) = 4\ .$$

After splitting with $k = 1$, the number of variables increases to four but all the newly introduced linear equations are linearly dependent on the original equations. The new linear equations are

$$\rho(\mathbf{b}_1^1) = 3\ ,\ \rho(\mathbf{b}_2^2) = 2\ ,\ \rho(\mathbf{b}_1^2) + \rho(\mathbf{b}_2^1) = 4,$$
$$\rho(\mathbf{b}_1^1) + \rho(\mathbf{b}_1^2) + \rho(\mathbf{b}_2^1) = 7\ ,\ \rho(\mathbf{b}_2^2) + \rho(\mathbf{b}_1^2) + \rho(\mathbf{b}_2^1) = 7,$$
$$\rho(\mathbf{b}_1^1) + \rho(\mathbf{b}_1^2) + \rho(\mathbf{b}_2^2) + \rho(\mathbf{b}_2^1) = 9$$

and $(\rho(\mathbf{b}_1^1), \rho(\mathbf{b}_2^2), \rho(\mathbf{b}_1^2), \rho(\mathbf{b}_2^1)) = (3,2,2,2)$ is a solution. Note that the linear dependence comes from the strong dependence relation $\bowtie$ between dependence cycles. If the following condition is satisfied, we can always compute a tight tuple:

Condition III.

(a) For any simple cycle c in G^{NCFG}, $DC(c)$ is not empty and

(b) For each simple cycle c_i in G^{NCFG}, there exists a dependence cycle $d_i \in DC_{\mathrm{cr}}(c_i)$ such that
$d_j \bowtie d_k$ for every pair of simple cycles c_j and c_k.

LEMMA 23. *If G^{NCFG} satisfies Condition III, there is a positive integer k such that there exists a tight tuple of latencies of NBBs of G_k^{NCFG}.*

Proof. The proof can be found in [23]. □

Given an NCFG that satisfies Condition II, the optimal software-pipelined schedule can be computed by the algorithm in Figure 10.

LEMMA 24. *The schedule computed by* COMPUTE_SCHEDULE *meets dependence constraints.*

Proof. We would like to show that

$$\forall p\ \forall k, k'\ (k < k') \text{ such that } p[k] \prec_{p[k,k']} p[k']\ ,$$
$$\tau(t(p)[k]) + \delta(p[k]) \le \tau(t(p)[k'])\ . \qquad (8)$$

procedure COMPUTE_SCHEDULE
if (Condition III is not satisfied)
  return SCHEDULE_NOT_FOUND
else
  compute a tight tuple $\langle \rho(\mathbf{b}_1), \rho(\mathbf{b}_2), \ldots \rangle$
$N^{\mathrm{DG}} := N^{\mathrm{NCFG}} \cup \{n_{\mathrm{DUMMY}}\}$
$E^{\mathrm{DG}} := \{ (n_{ij}, n_{i'j'}) \mid n_{ij} \prec_p n_{i'j'}$ where p is the shortest path from n_{ij} to $n_{i'j'} \}$
$E^{\mathrm{DG}} := E_{DG} \cup (\{n_{\mathrm{DUMMY}}\} \times (N_{DG} - \{n_{\mathrm{DUMMY}}\}))$
foreach $(e = (n_{ij}, n_{i'j'}) \in E^{\mathrm{DG}})$
  $r(e) := \delta(n_{ij}) - d(e) \cdot \rho(\mathbf{b}_i)$ where $\delta(n_{ij})$ is the latency of the operation of n_{ij} and $d(e) = 0$ if $i = i'$, 1 otherwise
end foreach
foreach $(e = (n_{\mathrm{DUMMY}}, n_{ij}) \in E^{\mathrm{DG}})$
  $r(e) := 0$
end foreach
foreach $(n_{ij} \in N^{\mathrm{DG}} - \{n_{\mathrm{DUMMY}}\})$
  $\sigma(n_{ij}) :=$ the length of the longest path in $G^{\mathrm{DG}} = (N^{\mathrm{DG}}, E^{\mathrm{DG}})$ where the length of a path p is the sum of weight $r(e)$ of edges in p
end procedure

Figure 10: The algorithm to compute a software-pipelined schedule.

procedure MOVE_CODE
foreach ($\mathbf{b}_i$)
  $\mathbf{b}_i := \langle\,\rangle$
foreach (n_{ij})
  MOVE_OP(n_{ij}, $\mathbf{b}_i$, $\sigma(n_{ij})$)
end procedure
procedure MOVE_OP(n, $\mathbf{b}$, σ)
if $(0 \le \sigma < \rho(\mathbf{b}))$
  places n on the time-slot σ of $\mathbf{b}$
else if $(\sigma < 0)$ /* move upward */
  foreach ($(\mathbf{b}', \mathbf{b}) \in E^{\mathrm{NBB}}$)
    MOVE_OP (n, $\mathbf{b}'$, $\sigma + \rho(\mathbf{b}')$)
else /* move downward */
  foreach ($(\mathbf{b}, \mathbf{b}') \in E^{\mathrm{NBB}}$)
    MOVE_OP (n, $\mathbf{b}'$, $\sigma - \rho(\mathbf{b}')$)
end procedure

Figure 11: The algorithm to move operations in NCFG.

By virtue of the longest path inequalities, we have

$$\sigma(p[k]) + r((p[k], p[k'])) \le \sigma(p[k']) \text{ , which implies}$$
$$\sigma(p[k]) + \delta(p[k]) - d((p[k], p[k'])) \cdot \rho(\mathbf{b}(p[k])) \le \sigma(p[k']) \,. \quad (9)$$

Therefore, we have

$$\begin{aligned}
&\tau(t(p)[k]) + \delta(p[k]) - \tau(t(p)[k']) \\
&\quad = \sum_{l=1}^{|\mathbf{bp}(p[1,k])|-1} \rho(\mathbf{bp}(p)[l]) + \sigma(p[k]) + \delta(p[k]) - \\
&\qquad \sum_{l=1}^{|\mathbf{bp}(p[1,k'])|-1} \rho(\mathbf{bp}(p)[l]) - \sigma(p[k']) \\
&\quad = -\sum_{l=|\mathbf{bp}(p[1,k])|}^{|\mathbf{bp}(p[1,k'])|-1} \rho(\mathbf{bp}(p)[l]) + \sigma(p[k]) - \sigma(p[k']) + \delta(p[k]) \\
&\quad \le -\sum_{l=|\mathbf{bp}(p[1,k])|}^{|\mathbf{bp}(p[1,k'])|-1} \rho(\mathbf{bp}(p)[l]) + d((p[k], p[k'])) \cdot \rho(\mathbf{b}(p[k])) \quad (10)
\end{aligned}$$

If $\mathbf{b}(p[k]) \equiv \mathbf{b}(p[k'])$, we have

$$\begin{aligned}
&\tau(t(p)[k]) + \delta(p[k]) - \tau(t(p)[k']) \\
&\qquad \le d((p[k], p[k'])) \cdot \rho(\mathbf{b}(p[k])) = 0 \cdot \rho(\mathbf{b}(p[k])) = 0 \,.
\end{aligned}$$

Otherwise, we have

$$\begin{aligned}
&\tau(t(p)[k]) + \delta(p[k]) - \tau(t(p)[k']) \\
&\qquad \le -\rho(\mathbf{b}(p[k])) + d((p[k], p[k'])) \cdot \rho(\mathbf{b}(p[k])) \\
&\qquad = -\rho(\mathbf{b}(p[k])) + 1 \cdot \rho(\mathbf{b}(p[k])) = 0 \,.
\end{aligned}$$

So, the schedule meets dependence constraints. □

Given a schedule, operation nodes are moved by the algorithm in Figure 11. The procedure MOVE_CODE first initializes each NBBs and invokes the MOVE_OP procedure for each operation nodes. The procedure MOVE_OP places each operation node such that the execution time of each operation instance becomes Eq. (7). From the definition of the tight tuple of latencies of NBBs, it can be easily seen that the software-pipelined NCFG is time-optimal.

6. EXPERIMENTAL RESULTS

In order to evaluate how practical the proposed software pipelining algorithms are, we have performed several experiments using a SPARC-based VLIW testbed [17]. We used 1317 innermost loops (with control flows) extracted from SPEC95 integer benchmark programs. We considered loops with up to 64 operations. We assumed that load operations take three cycles while all the other operations take one cycle.

Figure 12.(a) explains an overview of experimental scenarioes. In the first experiment (i.e., E1 in Figure 12.(a)), we measured how many loops satisfy Condition II (i.e., the Time Optimality Condition). Because the computation of Condition II may require excessive time[8], we set the upper bound T_{th} on computing Condition II. If the computation takes longer than T_{th}, the computation gives up, assuming that a loop does not satisfy Condition II. When T_{th} was set to be 30 seconds, we could not determine Condition II within the threshold time for about 3.7% of 1317 loops tested. In Figure 12.(a), the set of such loops is denoted by L1. Among the loops for which Condition II can be checked within T_{th}, 92.5% satisfied Condition II. (That is, 89.1% of the loops tested satisfied Condition II.)

Next, we turned our attention on the practicality of the realistic software pipelining algorithm presented in Section 5. In the second experiment (i.e., E2 in Figure 12.(a)), we measured how many loops satisfy Condition III (i.e., the stronger version of the Time Optimality Condition presented in Section 5). Unlike the first experiment (i.e., E1), we could determine Condition III within the threshold time for all the loops (except those in L1 and L2) since Condition III can be more efficiently evaluated. In the experiment, 79.2% of total loops satisfy Condition III, which indicates that Condition III does not impose a much practical constraint on Condition II. In Figure 12.(a), L3 represents the set of loops that failed Condition III.

In the third experiment (i.e., E3 in Figure 12.(a)), we applied the proposed realistic software pipelining algorithm to the loops satisfying Condition III and measured the running time of the algorithm. In rare cases, the algorithm did not run within the threshold time T_{th}. In Figure 12.(a), the set of such loops is denoted by L4 and the set of loops for which optimally software pipelined loops are computed within T_{th} is denoted by L5, respectively. The portion of loops belonging to L4 and L5 are 2.4% and 76.8% (of total loops), respectively. Figure 12.(b) summarizes graphically the results of three experiments, E1, E2 and E3.

In the final experiment (i.e., E4 in Figure 12.(a)), we were concerned with the resource requirement of optimally software pipelined loops (in L5). We measured the number of functional units and the number of registers in the optimally software-pipelined programs and the results are summarized in Table 1.[9] (We assumed homogeneous FUs.) Among the loops in L5, 42.7% of the loops require

[8] The problem of determining if Condition II, i.e., the Time Optimality Condition, is satisfied or not can be easily proved to be NP-hard by reducing from the 3-satisfiability problem. We omit the proof due to the page limit.

[9] In counting the number of FUs, we omitted copy operations used for renaming. Most of the renaming copy operations can be eliminated by post-pass optimizations such as copy propagation or register coalescing after unrolling [10], which is applicable even to unreducible loops.

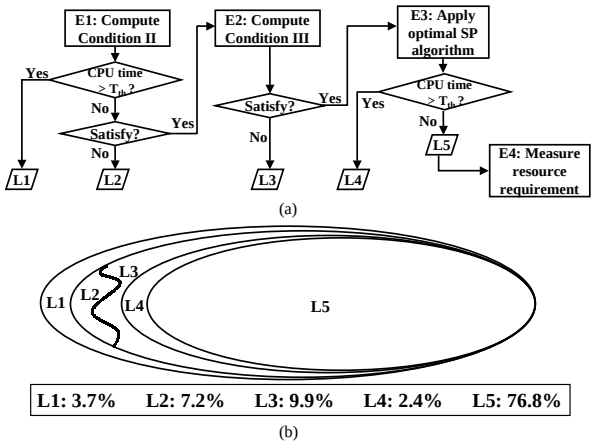

Figure 12: (a) Experiment scenario and (b) loop classification based on the experimental results. (The area of each region roughly represents the relative size of the corresponding set of loops when T_{th} = 30 seconds.)

% of Loops						
	# of FUs	≤ 8	9-12	13-16	> 16	**Total**
# of Regs.	< 32	39.5	8.2	8.0	13.2	**68.9**
	33-64	3.2	3.1	5.3	15.1	**26.7**
	> 64	0	0	0.3	4.1	**4.4**
	Total	**42.7**	**11.3**	**13.6**	**32.4**	100

Table 1: Resource requirement for optimally software-pipelined programs.

at most 8 FUs while only 32.4% of the loops require more than 16 FUs. We believe that the resource requirement can be further reduced if the proposed software pipelining algorithm is augmented by post-pass local code motions (e.g., moving operations in non-critical dependence chains). For the register requirement, we obtained more positive results; 95.6% of the loops require at most 64 registers. Furthermore, for 68.9% of the loops, 32 registers were sufficient without causing any spill.

Our experimental results show that a significant portion of real loops have their time-optimal software-pipelined programs. Furthermore, the time-optimal programs can be computed with realistic levels of hardware support within a reasonable time limit.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented a necessary and sufficient condition for loops with control flows to have their equivalent time optimal programs and described how to compute the condition. We also presented a software pipelining algorithm that computes a time optimal solution for every eligible loop satisfying the condition. The results solve two fundamental open problems on time optimal software pipelining of loops with control flows.

As a practical alternative, we presented a more realistic optimal software pipelining algorithm which covers most eligible loops and runs faster with less code expansion and less resource requirement. Our experimental results strongly indicates achieving the optimality in the software-pipelined programs is a viable goal in practice with realistic hardware support. As a future work, we are interested in developing a resource-constrained near-optimal software pipelining algorithm guided by the results shown in this paper.

8. ACKNOWLEDGEMENT

We would like to thank Prof. Uwe Schwiegelshohn (our shepherd), Dr. Kemal Ebcioğlu and anonymous referees for their helpful comments and suggestions.

9. REFERENCES

[1] A. Aiken and A. Nicolau. Optimal Loop Parallelization. In *Proc. of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 308–317, 1988.

[2] A. Aiken and A. Nicolau. Perfect Pipelining. In *Proc. of the Second European Symposium on Programming, Lecture Notes in Computer Science*, vol. 300, pages 221–235. Springer-Verlag, 1988.

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

[4] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[6] K. Ebcioğlu. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. In *Proc. of IFIP WG 10.3 Working Conference on Parallel Processing*, pages 3–21, 1988.

[7] J. Farrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[8] F. Gasperoni and U. Schwiegelshohn. Generating Close to Optimum Loop Schedules on Parallel Processors. *Parallel Processing Letters*, 4(4):391–403, 1994.

[9] D. Johnson. Finding all the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[10] S. Kim, S.-M. Moon, J. Park, and K. Ebcioğlu. Unroll-Based Register Coalescing. In *Proc. of the 14th International Conference on Supercomputing*, pages 296–305, 2000.

[11] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proc. of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[12] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[13] D. Milicev and Z. Jovanovic. Control Flow Regeneration for Software Pipelined Loops with Conditions. To appear in *International Journal of Parallel Programming*.

[14] S.-M. Moon and S. Carson. Generalized Multi-way Branch Unit for VLIW Microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):850–862, 1995.

[15] S.-M. Moon and K. Ebcioğlu. Parallelizing Non-numerical Code with Selective Scheduling and Software Pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, 1997.

[16] A. Nicolau. Uniform Parallelism Exploitation in Ordinary Programs. In *Proc. of the International Conference on Parallel Processing*, pages 614–618, 1985.

[17] S. Park, S. Shim, and S.-M. Moon. Evaluation of Scheduling Techniques on a SPARC-Based VLIW Testbed. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 104–113, 1997.

[18] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: An Algebraic Approach to Program Dependences. In *Proc. of the 18th ACM Symposium on Principles of Programming Languages*, pages 67–78, 1991.

[19] U. Schwiegelshohn, F. Gasperoni, and K. Ebcioğlu. On Optimal Parallelization of Arbitrary Loops. *Journal of Parallel and Distributed Computing*, 11(2):130–134, 1991.

[20] A. Uht. Requirements for Optimal Execution of Loops with Tests. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):573–581, 1992.

[21] N. Warter, S. Mahlke, W-M. Hwu, and B. Rau. Reverse If-Conversion. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 290–299, 1993.

[22] H.-S. Yun, J. Kim, and S.-M. Moon. A First Step Towards Time Optimal Software Pipelining of Loops with Control Flows. In *Proc. of the 10th Conference on Compiler Construction*, pages 182–199, 2001.

[23] H.-S. Yun, J. Kim, and S.-M. Moon. On Time Optimal Software Pipelining of Loops with Control Flows. Technical report, School of CSE, Seoul National Univ., 2001. Available from http://davinci.snu.ac.kr/Download/opt_sp_techrep.pdf.