Time Optimal Software Pipelining of Loops with Control Flows for VLIW Processors

Abstract

Software pipelining is widely used as a compiler optimization technique to achieve high performance in machines that exploit instruction-level parallelism such as superscalar or VLIW processors. However, surprisingly, there have been few theoretical results on the optimality of software pipelined loops with control flows. The problem of time optimal software pipelining of loops with control flows is such an under-investigated theoretical problem. In this paper, we give a complete treatment on the time optimal software pipelining problem solving two fundamental open problems. First, we show that there exists a (computable) decision procedure that can decide if a given loop with control flows has a time optimal parallel program or not. Second, we present a software pipelining algorithm that computes a time optimal parallel program, which is the most significant outcome from the practical point of view. As part of the formal treatment of software pipelining, we propose a new formalization of software pipelining, which provides a basis of our proof as well as a new theoretical framework for software pipelining research.

1 Introduction

Software pipelining [5] 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 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 [30]. Figure 1. (a) and (b) show an sequential loop and its corresponding time optimal parallel program, respectively. Each shaded region in Figure 1. (b) corresponds to a parallel instruction and all

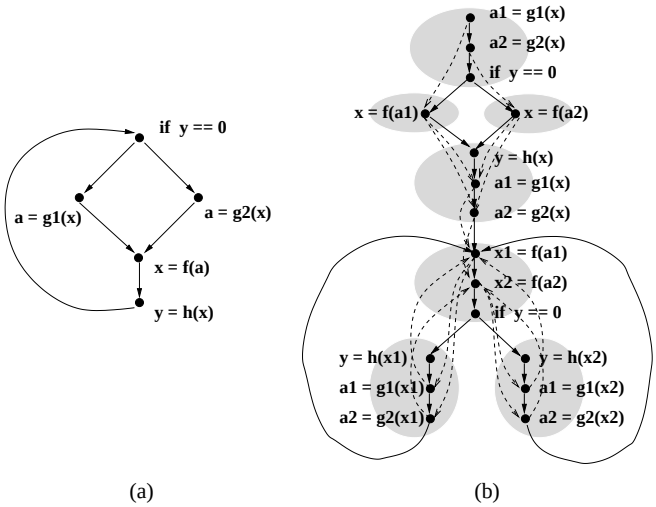

Figure 1. (a) A sequential loop and (b) its time optimal parallel version

operations in a shaded region are executed simultaneously. For any execution path, there exists a dependence chain whose length is equal to the execution time of the execution path.

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 [2]. 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 and several software pipelining algorithms have been developed using this model [28, 14, 15, 7, 17, 9, 8].

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 Related Work

For loops *without* control flows, there exist several theoretical results [2, 14, 15, 9, 8]. When resource constraints are not present, both the time optimal schedule and the rate optimal one can be found in polynomial time [2, 14]. With resource constraints, the problem of finding the optimal schedule is NP-hard in its full generality [14] but there exist approximation algorithms that guarantee the worst case performance of roughly twice the optimum [14, 8].

Given sufficient resources, an acyclic program can be always transformed into an equivalent time optimal program by applying list scheduling to each execution path and then simultaneously executing all the execution paths parallelized by list scheduling. When resources are limited, definitions of time optimality may be based on the average execution time. For acyclic programs, Gasperoni and Schwiegelshohn defined an optimality measure based on the execution probability of various execution paths and showed that a generalized list

scheduling heuristic guarantees the worst case performance of at most $2 - 1/m + (1 - 1/m) \cdot 1/2 \cdot \lceil log_2 m \rceil$ times the optimum [16] where m is the number of operations that can be executed concurrently. For loops with control flows, measures based on the execution probability of paths is not feasible, since there are infinitely many execution paths.

There are few theoretical results for loops with control flows, and, to the best of our knowledge, only two results [30, 31] have been published. The work by Uht [31] proved that the resource requirement necessary for the optimal execution may increase exponentially for some loops with control flows. The Uht's result, however, is based on an idealized hardware model which is not directly relevant to software pipelining. The work by Schwiegelshohn *et al.* [30], which is the most well-known theoretical result on time optimal programs, showed that there are some loops for which no equivalent time optimal programs exist. Although significant, their contribution lacks any formal treatment of the time optimal software pipelining. For example, they do not formally characterize conditions under which a loop does not have an equivalent time optimal program. 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 [6, 20], and recover control flows after scheduling [32]. Although practical, these extra transformations prohibit considerable amount of code motions, limiting the scheduling space exploration significantly.

1.2 Contributions

In this paper, we propose a new formalization of software pipelining and give a complete treatment on the time optimal software pipelining problem. In particular, we 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. In solving two open problems, we take the following four steps:

Step 1: We hypothesize a condition for loops to have equivalent time optimal programs. (the Time Optimality Condition in Section 5)

Step 2: We prove that a loop does not have an equivalent time optimal program unless it satisfies the Time Optimality Condition. (Theorem 20 in Section 6)

Step 3: We describe a software pipelining algorithm that computes a time optimal program for every loop satisfying the Time Optimality Condition. (Theorem 26 in Section 7)

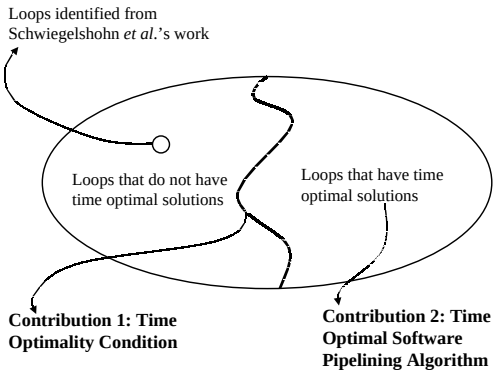

Figure 2. Loop classification based on time optimality

Step 4: We present how to compute the Time Optimality Condition. (Theorem 36 in Section 8)

From Steps 2 and 3, we can conclude that the condition described in Step 1 is indeed a necessary and sufficient condition for loops to have their time optimal programs. We call this condition, *the Time Optimality Condition*. 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.

In Step 2, we prove that a loop does not have its equivalent time optimal program if it does not satisfy the Time Optimality Condition. The rationale behind the proof is that if a loop does not satisfy the Time Optimality Condition but it needs to have a time optimal program, it must have two operations n_1 and n_2 such that n_1 is executed infinitely earlier than n_2 in the time optimal parallel program while n_2 precedes n_1 in the original sequential program. We show that no *closed-form* parallel program satisfies this anomalous requirement.

The results of Step 3, along with those of Step 2, prove that the Time Optimality Condition is indeed a necessary and sufficient condition for a loop to have its time optimal parallel program. The proposed software pipelining algorithm is mostly based on the algorithm by Aiken *et al.* [4] with some modifications in the renaming framework. The proof of the time optimality is based on the greediness of the algorithm.

Although the Time Optimality Condition is intuitive and useful in deriving the theorems, it is not obvious how to compute the condition because the condition includes the universal quantifier. In Step 4, we present an equivalent time optimality condition which can be more directly computed.

Figure 2 summarizes our 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.* [30]. 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.

Although the time optimality results reported in this paper may not have an immediate impact on developing *realistic* time-optimal software pipelining algorithms, we believe that our theoretical results provide an important new insight into the software pipelining problem, which, in turn, stimulates the development of

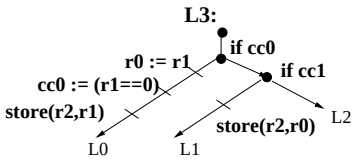

Figure 3. A tree VLIW instruction

higher-quality software pipelining algorithms. For example, our initial experimental results indicate that most loops of SPEC95 benchmark programs have time-optimal parallel programs under the assumption of unlimited resource availability. As with software pipelining algorithms for straight-line loops [20, 14, 8], knowing these time optimal schedules may result in better parallel schedules as well even under the resource-constrained situations.

The rest of the paper is organized as follows. We explain the machine model assumptions, program representation in Section 2. Section 3 discusses the dependence model. A formal description of software pipelining is presented in Section 4. In Section 5, we present the Time Optimality Condition. In Section 6, we prove the necessity part based on the formalization of software pipelining. The sufficiency part is proved by construction of time-optimal software pipelining algorithm in Section 7. In Section 8, we explain how to compute the Time Optimality Condition. We conclude with a summary and directions for future work in Section 9.

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* [25]) 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* [12]). 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*. [4].

As a specific example architecture, we use the tree VLIW architecture model [11, 26], 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 3. 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 [25]. An operation is committed only if it lies in the execution path determined by the branch unit [12].

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

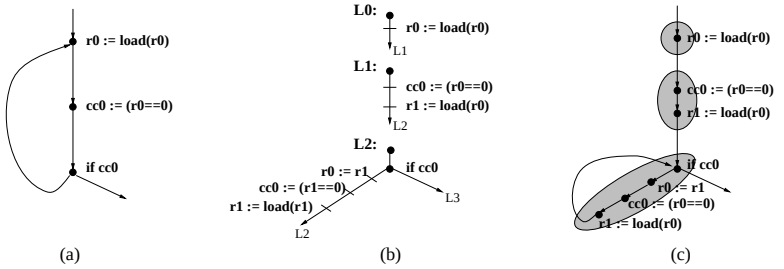

Figure 4. (a) A sequential program, (b) a parallel tree VLIW program, and (c) a parallel program in the extended sequential representation

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.

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 4. (a) shows an input sequential program P_{s} and Figure 4. (b) shows its corresponding parallel tree VLIW program P_{tree}. Using the extended sequential representation, P_{tree} is represented by Figure 4. (c). The parallel program shown in Figure 4. (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 [26].

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.* [4].) 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

[1] Since a parallel program is represented by the extended sequential representation, the notations and definitions 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 $op(n)$.

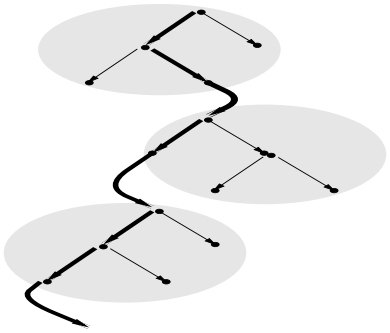

Figure 5. An execution path in a parallel program

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 $0 \leq i < t$.

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 $1 \leq i < k$. For a given path p, the length of p is the number of nodes in p and denoted by $|p|$. The i-th ($1 \leq i \leq |p|$) 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 ($0 \leq j \leq |p| - |q|$) such that $q[i] = p[i+j]$ for all $1 \leq i \leq |q|$. For a path p and i, j ($1 \leq i \leq j \leq |p|$), $p[i, j]$ 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. When c denotes a cycle in the input loop (thus reducible) we assume $c[1]$ represents the unique loop header node. Two paths p and q are said to be equivalent, written $p \equiv q$, if $|p| = |q|$ and $p[i] = q[i]$ for all $1 \leq i \leq |p|$.

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., $p^{\mathrm{e,sp}}$). Each execution path can be represented by an initial store with which the control flows along the execution path. Suppose a program P is executed with an initial store s_0 and the sequence of configurations is written as $\langle\langle n_0, s_0\rangle, \langle n_1, s_1\rangle, \cdots, \langle n_f, s_f\rangle\rangle$, where n_0 denotes the start node and n_f one of exit nodes. Then $ep(P, s_0)$ is defined to be the execution path $\langle n_0, n_1, \cdots, n_f\rangle$ of P. (ep stands for *execution path*.) Compilers commonly performs the static analysis under the assumption that all the execution paths of the program are executable, because it is undecidable to check if an arbitrary path of the program is executable. In this paper, we make the same assumption, That is, we assume $\forall p^{\mathrm{e}}$ in P, $\exists s$ such that $p^{\mathrm{e}} \equiv ep(P, s)$.

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. For example, in Figure 5, the execution path of a parallel program is distinguished by a thick line.

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 $1 \leq i \leq |p|$. 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 the i-th node instance $\mathbf{n}_i$ of $t(p)$ whose node component is a branch node, a boolean-valued attribute dir is defined as follows:

$$dir(\mathbf{n}_i) = \begin{cases} T & \text{if } p[i+1] \text{ is the T-target successor of } p[i] \\ F & \text{otherwise .} \end{cases},$$

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. 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*. These terms are further clarified in Section 4.

3 Dependence Model

Let alone irregular memory dependences, existing dependence analysis techniques cannot model true dependences accurately mainly because true dependences are detected by conservative analysis on the closed form of programs. In Section 3.1 we introduce a path-sensitive dependence model to represent precise dependence information. In order that the schedule is constrained by true dependences only, a compiler should overcome false dependences. We explain how to handle the false dependences in Section 3.2.

3.1 True Dependences

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 as shown in Figure 6. In Figure 6.(a), there are two paths, $p_1 = \langle 1,2,3,5 \rangle$ and $p_2 = \langle 1,2,4,5 \rangle$, from node 1 to node 5. Node 5 is dependent on node 1 along p_1, but not along p_2. This ambiguity cannot be resolved unless node 1 is splitted into distinct nodes to be placed in each path. In Figure 6.(b), node 7 is first used after k iterations of c_1 along $p_3 \circ c_1^k \circ p_4$, where $p_3 = \langle 7,9,11 \rangle$, $p_4 = \langle 5,10 \rangle$ and $c_1 = \langle 5,6,8,9,11,5 \rangle$. However, this unspecified number of iterations, k, cannot be modeled by existing techniques; That is, existing techniques cannot model the unspecified dependence distance. 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.

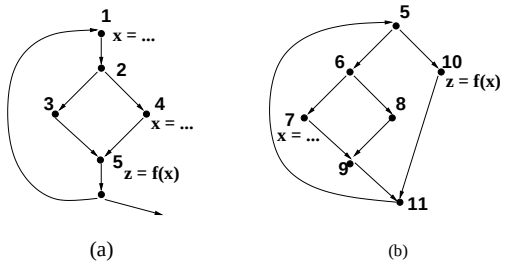

Figure 6. Path-sensitive dependence examples

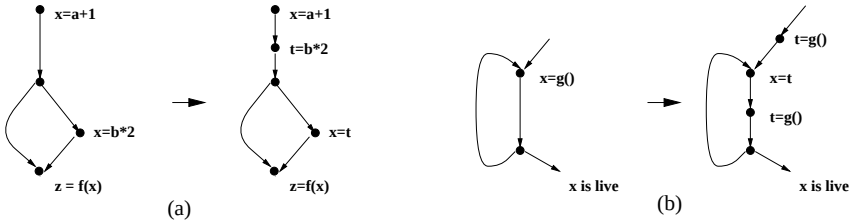

Figure 7. Copy operations used to overcome false dependences

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*

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

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

Definition 2 *Given a path p and i, j $(1 \leq i < j \leq |p|)$, $t(p^{\mathrm{e}})[j]$ is said to be dependent on $t(p^{\mathrm{e}})[i]$, written $t(p^{\mathrm{e}})[i] \prec t(p^{\mathrm{e}})[j]$, 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 [5] and is usually represented by *data dependence graphs* [19] or its extensions [13, 29]. 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 do not lose much generality and that the claims in this paper can be shown to be still valid in other memory dependence models with slight modifications to the proofs.

3.2 False Dependences

For loops with control flows, it is not a trivial matter to handle false dependences. They cannot be eliminated completely even if each live range is renamed before scheduling. For example, the scheduling techniques described in [4, 26] rely on the "on the fly" register allocation scheme based on copy operations so that the schedule is constrained by true dependences only.

In Fig. 7. (a), for the `x=b*2` to be scheduled above the branch node, `x` should not be used for the target register of `x=b*2` and, therefore, the live range from `x=b*2` to `z=f(x)` should be renamed. But the live range from `x=b*2` to `z=f(x)` alone cannot be renamed because the live range from `x=a+1` to `z=f(x)` is combined with the former by `x`. Thus, the live range is splitted by the copy operation `x=t` so that `t` carries the result of `b*2` along the prohibited region and `t` passes `b*2` to `x` the result.

In Fig. 7. (b), `x=g()` is to be scheduled across the exit branch but `x=g()` is used at the exit. So the live range from `x=g()` to exit is expected to be longer than an iteration, but it cannot be realized if only one register is allocated for the live range due to the register overwrite problem. This can be handled by splitting the long live range into ones each of which does not span more than an iteration, say one from `t=g()` to `x=t` and one from `x=t` to the exit.

In the next section, these copy operations used for renaming are distinguished from ones in the input programs which are byproduct of other optimizations such as common subexpression elimination. The true dependence carried by the live range joined by these copy operations is represented by $\overset{*}{\prec}$ relation as follows.

Definition 3 *Given an execution path of a parallel program p^e, let $\mathbf{N}_{p^e}$ represent the set of all node instances in $t(p^e)$. For node instances $\mathbf{n}$ in $t(p^{e,sp})$, $Prop(\mathbf{n})$ represents the set of copy node instances in $t(p^e)$ by which the value defined by $\mathbf{n}$ is propagated, that is,*

$$Prop(\mathbf{n}) = \{\mathbf{n}^c \mid \quad \mathbf{n} \prec \mathbf{n}_1^c \quad , \mathbf{n}_k^c \prec \mathbf{n}^c,\ \mathbf{n}_i^c \prec \mathbf{n}_{i+1}^c \ \textit{for all}\ 1 \le i < k$$
$$\textit{where} \quad \mathbf{n}^c \textit{ and } \mathbf{n}_i^c\ (1 \le i \le k)\ \textit{are copy node instances}\} \ .$$

For node instances $\mathbf{n}_1$ and $\mathbf{n}_2$ in $\mathbf{N}_{p^e}$, we write $\mathbf{n}_1 \overset{*}{\prec} \mathbf{n}_2$ *if*

$$\mathbf{n}_1 \prec \mathbf{n}_2 \ \textit{or}\ \ \exists\, \mathbf{n}^c \in Prop(\mathbf{n}_1),\ \mathbf{n}^c \prec \mathbf{n}_2 \ .$$

Definition 4 *The extended live range of $\mathbf{n}$, written $elr(\mathbf{n})$, is the union of the live range of the node instance $\mathbf{n}$ and those of copy node instances in $Prop(\mathbf{n})$, that is,*

$$elr(\mathbf{n}) = t(p)[pos(\mathbf{n}), max\{pos(\mathbf{n}^c) | \mathbf{n}^c \in Prop(\mathbf{n})\}] \ .$$

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

Definition 5 *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 $|\mathbf{d}|$. For a dependence chain d and i, j $(1 \le i \le j \le |d|)$, $d[i, j]$ represents the sub-chain of d induced by the sequence of node instances from $d[i]$ up to $d[j]$.*

4 A Formalization of Software Pipelining

In this section, we develop a formal account of transformations of software pipelining, which will provide a basis for the proof in Section 6. Given an input loop L and its parallel version L^{SP}, let $\mathbf{P}^{\mathrm{e}}$ and $\mathbf{P}^{\mathrm{e,SP}}$ denote the set of all execution paths in L and the set of those in L^{SP}, respectively. Let us consider a relation $R : \mathbf{P}^{\mathrm{e}} \times \mathbf{P}^{\mathrm{e,SP}}$ defined by

$$(p^{\mathrm{e}}, p^{\mathrm{e,sp}}) \in R \ \text{ iff } \ \exists \text{ a store } s, \ \ ep(L, s) \equiv p^{\mathrm{e}} \ \wedge \ ep(L^{\mathrm{SP}}, s) \equiv p^{\mathrm{e,sp}} \ .$$

In order to formalize software pipelining, we are to restrict transformations (that map p^{e} into $p^{\mathrm{e,sp}}$) by the following five constraints, Constraints 6-10.

First, transformations should exploit only dependence information, that is, they should have only the effect of reordering nodes. Some optimization techniques (e.g., strength reduction and tree height reduction) may reduce the path length by using other semantic properties of programs (e.g., associativity). However, the scheduler is not responsible for such optimizations. These optimizations are performed before/after the scheduling phase.

Additionally, the scheduler is not responsible for eliminating partially dead operation nodes in p^{e}, which are not used in p^{e} but may be used in another execution paths. Partially dead operations may become fully dead by some transformations such as moving branch up and can be eliminated on the fly [26], but we assume that they are not eliminated until a post-pass optimization phase. We require that all operation nodes in p^{e}, dead or not, be also present in $p^{\mathrm{e,sp}}$. Therefore $p^{\mathrm{e,sp}}$ is required to execute the same operations as p^{e} in an order compatible with the dependences present in p^{e}. The path $p^{\mathrm{e,sp}}$, however, may have additional *speculative* nodes[3] from other execution paths that do not affect the final store of $p^{\mathrm{e,sp}}$ and copy operations used for overcoming false dependences [4, 26]. Formally, the first constraint on transformations can be given as follows.

Constraint 6 *Let* $\mathbf{N}_1$ *represent the set of all node instances in* $t(p^{\mathrm{e}})$ *and let* $\mathbf{N}_2$ *represent the set of all effective node instances in* $t(p^{\mathrm{e,sp}})$. *Then, there exists a bijective function* f *from* $\mathbf{N}_1$ *to* $\mathbf{N}_2$ *such that*

$$\forall \mathbf{n} \in \mathbf{N}_1, \qquad op(node(\mathbf{n})) = op(node(f(\mathbf{n}))) \quad \textit{and}$$
$$\forall \mathbf{n}, \mathbf{n}' \in \mathbf{N}_1, \qquad \mathbf{n} \prec \mathbf{n}' \ \textit{iff} \ f(\mathbf{n}) \overset{*}{\prec} f(\mathbf{n}') \ .$$

In this case, $f(\mathbf{n})$ *is said to correspond to* $\mathbf{n}$ *and we use* $sp\_ni_{p^{\mathrm{e}}, p^{\mathrm{e,sp}}}$ *to represent the function* f *for a pair of such execution paths* p^{e} *and* $p^{\mathrm{e,sp}}$.

Second, the final store[4] of $p^{\mathrm{e,sp}}$ should be equal to that of p^{e} to preserve the semantic of L. For this, we require that for any node $n = node(\mathbf{n})$, where $\mathbf{n}$ is a node instance in $t(p^{\mathrm{e}})$, if the target register of n is live at the exit of p^{e}, the value defined by $node(sp\_ni_{p^{\mathrm{e}}, p^{\mathrm{e,sp}}}(\mathbf{n}))$ should be eventually committed to $reg_{\mathrm{W}}(n)$ along $p^{\mathrm{e,sp}}$. For simplicity, we assume that all registers in p^{e} are regarded as being live at the exit of p^{e} during software pipelining. The liveness of each node in $p^{\mathrm{e,sp}}$ are checked at post-pass dead code elimination optimization phase. Constraint 7 concisely states this condition.

[3] In fact, most complications of the nonexistence proof in Section 6 as well as the formalization of software pipelining are due to expanded solution space opened up by branch reordering transformation.

[4] Temporary registers are excluded.

Constraint 7 *For any assignment node instance* $\mathbf{n}$ *in* $t(p^{\mathrm{e}})$ *such that* $\forall i > pos(\mathbf{n}),\ reg_{\mathrm{W}}(p^{\mathrm{e}}[i]) \neq reg_{\mathrm{W}}(node(\mathbf{n}))$,

$$\begin{aligned} reg_{\mathrm{W}}(node(\mathbf{n})) &= reg_{\mathrm{W}}(node(sp\_ni_{p^{\mathrm{e}},p^{\mathrm{e,sp}}}(\mathbf{n}))) \ \textit{or} \\ reg_{\mathrm{W}}(node(\mathbf{n})) &= reg_{\mathrm{W}}(node(\mathbf{n}^c)) \ \textit{for some node instance } \mathbf{n}^c \in Prop(sp\_ni_{p^{\mathrm{e}},p^{\mathrm{e,sp}}}(\mathbf{n})). \end{aligned}$$

It is needed to impose a restriction on registers allocated for speculative nodes. Registers defined by speculative nodes are required to be temporary registers that are not used in L so as not to affect the final store.

Constraint 8 *Let* $\mathbf{R}$ *be the set of registers that are defined by nodes in* L. *Then the target register of each speculative node in* L^{SP} *is not contained in* $\mathbf{R}$.

Now we are to impose a restriction to preserve the semantic of branches. Let us consider a branch node instance $\mathbf{n} = t(p^{\mathrm{e}})[i]$ and the corresponding node instance $\mathbf{n}' = t(p^{\mathrm{e,sp}})[i'] = sp\_ni_{p^{\mathrm{e}},p^{\mathrm{e,sp}}}(\mathbf{n})$. The role of $\mathbf{n}$ is to separate p^{e} from the set of execution paths that can be represented by $p^{\mathrm{e}}[1,i] \circ p_{\mathrm{f}}$ where p_{f} represents any path such that $p_{\mathrm{f}}[1] = p^{\mathrm{e}}[i], p_{\mathrm{f}}[2] \neq p^{\mathrm{e}}[i+1]$ and $p_{\mathrm{f}}[|p_{\mathrm{f}}|]$ is an exit node in L. $\mathbf{n}'$ is required to do the same role as $\mathbf{n}$, that is, it should separate $p^{\mathrm{e,sp}}$ from the set of corresponding execution paths. But some of them might already be separated from $p^{\mathrm{e,sp}}$ earlier than $\mathbf{n}'$ due to another speculative branch node, the instance of which in $p^{\mathrm{e,sp}}$ is redundant, scheduled above $\mathbf{n}'$. This constraint can be written as follows.

Constraint 9 *Given an execution path* p^{e} *and* q^{e} *in* L *such that*

$$q^{\mathrm{e}}[1,i] \equiv p^{\mathrm{e}}[1,i] \ \wedge\ dir(t(q^{\mathrm{e}})[i]) \neq dir(t(p^{\mathrm{e}})[i]) \ ,$$

for any execution path $p^{\mathrm{e,sp}}$ *and* $q^{\mathrm{e,sp}}$ *such that* $(p^{\mathrm{e}}, p^{\mathrm{e,sp}})(q^{\mathrm{e}}, q^{\mathrm{e,sp}}) \in R$, *there exists a branch node* $p^{\mathrm{e,sp}}[j]$ $(j \leq i')$ *such that*

$$q^{\mathrm{e,sp}}[1,j] \equiv p^{\mathrm{e,sp}}[1,j] \ \wedge\ dir(t(q^{\mathrm{e,sp}})[j]) \neq dir(t(p^{\mathrm{e,sp}})[j])$$

where i' *is an integer such that* $t(p^{\mathrm{e,sp}})[i'] = sp\_ni_{p^{\mathrm{e}},p^{\mathrm{e,sp}}}(t(p^{\mathrm{e}})[i])$.

$p^{\mathrm{e,sp}}$ is said to be equivalent to p^{e}, written $p^{\mathrm{e}} \equiv_{SA} p^{\mathrm{e,sp}}$, if Constraints 6-9 are all satisfied. (The subscript *SA* is adapted from the expression "semantically and algorithmically equivalent" in [30].) Constraint 9 can be used to rule out a pathological case, *unification of execution paths*. Two distinct execution paths $p_1^{\mathrm{e}} = ep(L, s_1)$ and $p_2^{\mathrm{e}} = ep(L, s_2)$ in L are said to be *unified* if $ep(L^{\mathrm{SP}}, s_1) \equiv ep(L^{\mathrm{SP}}, s_2)$. Suppose p_1^{e} is separated from p_2^{e} by a branch, then $ep(L^{\mathrm{SP}}, s_1)$ must be separated from $ep(L^{\mathrm{SP}}, s_2)$ by some branch by Constraint 9. So p_1^{e} and p_2^{e} cannot be unified.

Let us consider the mapping cardinality of R. Since distinct execution paths cannot be unified, there is the unique p^{e} which is related to each $p^{\mathrm{e,sp}}$. But there may exist several $p^{\mathrm{e,sp}}$'s that are related to the same p^{e} due to speculative branches. Thus, R is a one-to-many relation, and if branch nodes are not allowed to be reordered, R becomes a one-to-one relation. In addition, the domain and image of R cover the entire $\mathbf{P}^{\mathrm{e}}$ and $\mathbf{P}^{\mathrm{e,SP}}$, respectively. Because of our assumption in Sect. 2.3 that all the execution paths are executable,

$\forall p^{\mathrm{e}} \in \mathbf{P}^{\mathrm{e}}, \exists s,\ p^{\mathrm{e}} \equiv ep(L,s)$ and the domain of R covers the entire $\mathbf{P}^{\mathrm{e}}$. When an execution path $p^{\mathrm{e}} \in \mathbf{P}^{\mathrm{e}}$ is splitted into two execution paths $p_1^{e,sp}, p_2^{e,sp} \in \mathbf{P}^{\mathrm{e,SP}}$ by scheduling some branch speculatively, it is reasonable for a compiler to assume that these two paths are all executable under the same assumption and that the image of R cover the entire $\mathbf{P}^{e,sp}$. To be short, R^{-1} is a surjective function from $\mathbf{P}^{e,sp}$ to $\mathbf{P}^{\mathrm{e}}$.

Let $\mathbf{N}$ and $\mathbf{N}^{\mathrm{SP}}$ represent the set of all node instances in all execution paths in L and the set of all effective node instances in all execution paths in L^{SP}, respectively. The following constraint can be derived from the above explanation.

Constraint 10 *There exists a surjective function* $\alpha : \mathbf{P}^{\mathrm{e,SP}} \Rightarrow \mathbf{P}^{\mathrm{e}}$ *such that*

$$\forall\, p^{\mathrm{e,sp}} \in \mathbf{P}^{\mathrm{e,SP}},\ \alpha(p^{\mathrm{e,sp}}) \equiv_{SA} p^{\mathrm{e,sp}}\ .$$

Using α defined in Constraint 10 above and $sp\_ni_{p^{\mathrm{e}},p^{\mathrm{e,sp}}}$ defined in Constraint 6, another useful function β is defined, which maps each node instance in $\mathbf{N}^{\mathrm{SP}}$ to its corresponding node instance in $\mathbf{N}$.

Definition 11 $\beta : \mathbf{N}^{\mathrm{SP}} \Rightarrow \mathbf{N}$ *is a surjective function such that*

$$\beta(\mathbf{n}^{sp}) = sp\_ni_{\alpha(p^{\mathrm{e,sp}}),p^{\mathrm{e,sp}}}{}^{-1}(\mathbf{n}^{sp})$$

where $p^{\mathrm{e,sp}} \in \mathbf{P}^{\mathrm{e,SP}}$ *is the unique execution path that contains* $\mathbf{n}^{sp}$.

To the best of our knowledge, all the software pipelining techniques reported in literature satisfy Constraints 6-10.

5 Time Optimality Condition

In this section, we present the Time Optimality Condition. Before presenting the Time Optimality Condition, we first formally define *time optimality*. For each execution path $p^{\mathrm{e,sp}} \in \mathbf{P}^{\mathrm{e,SP}}$, the execution time of each node instance $\mathbf{n}$ in $t(p^{\mathrm{e,sp}})$ can be counted from the grouping information associated with L^{SP} and is denoted by $\tau(\mathbf{n})$. Time optimality of the parallel program L^{SP} is defined as follows [30, 4].

Definition 12 (Time Optimality)

L^{SP} *is time optimal, if for every execution path* $p^{\mathrm{e,sp}} \in \mathbf{P}^{\mathrm{e,SP}}$, $\tau(t(p^{\mathrm{e,sp}})[|p^{\mathrm{e,sp}}|])$ *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 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 $p^{\mathrm{e,sp}}$ 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 $\|p\|$.

A loop L has an equivalent time optimal program if and only if the following condition is satisfied:

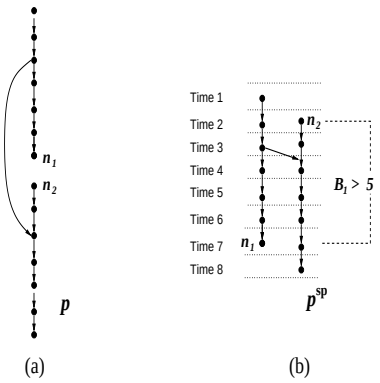

Figure 8. An example illustrating Condition I.(a)

Condition I (Time Optimality Condition)

(a) *There exists a constant* $B_1 > 0$ *such that for any path* p *in* L,

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

(b) *there exist constants* $B_2, B_3 > 0$ *such that for any path* p *in* L,

$$|p| \;\leq\; B_2 \cdot \|p\| + B_3 \,.$$

Condition I.(a) states that for any path p in 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. Figure 8 illustrates Condition I.(a) using an example path p shown in Figure 8.(a) where edges represent true dependences. The path p^{sp} in Figure 8.(b) shows the corresponding path in the time optimal parallel program. Since the length of the longest dependence chain in p is 8, p^{sp} is executed in eight time steps. To compute the lower bound on B_1 for this case, let us substitute 7 for i in Condition I.(a). Then we have:

$$B_1 \geq \|p[1,7]\| + \|p[8,14]\| - \|p\| = 7 + 7 - 8 = 6 \,.$$

Intuitively, the lower bound on B_1 corresponds to the range of the code motion required for the time optimal execution. In Figure 8, n_2 is preceded by n_1 in p, but, for the time optimal execution, n_2 should be executed at least 5 time steps earlier than n_1, which is $B_1 - 1$.

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

Let us consider the example loops shown in Figure 9. These loops were adapted from [30]. The first one (Figure 9. (a)), which was shown to have an equivalent time optimal program, satisfies Condition I. For any execution path p^e that loops k iterations, $\|p^e\| = 2k+1$ and for $1 \leq i < j \leq |p^e| = 4k$, $\|p^e[1,i]\| \leq \lceil i/2 \rceil + 1$ and $\|p^e[j,|p^e|]\| \leq \lceil 4k - j/2 \rceil + 2$. So, we have:

$$\|p^e[1,i]\| + \|p^e[j,|p^e|]\| \leq 2k + 3 - (j-i)/2 \leq \|p^e\| + 2 \;\;.$$

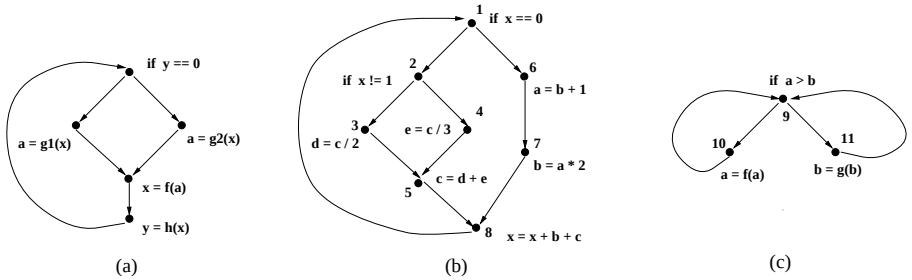

Figure 9. Example loops used by Schwiegelshohn et al.

The second and third shown in Figures 9. (b) and 9. (c) do not satisfy Condition I.(a), thus having no equivalent time optimal programs as shown in [30]. For the loop in Figure 9. (b), let $c_1 = \langle 1,2,4,5,8,1\rangle$ and $c_2 = \langle 1,6,7,8,1\rangle$. For the execution path $p^{\mathrm{e}}(k) = c_1^k \circ c_2^k$, we have :

$$\|p^{\mathrm{e}}(k)[1,5k]\| + \|p^{\mathrm{e}}(k)[5k+1,|p^{\mathrm{e}}(k)|]\| - \|p^{\mathrm{e}}(k)\| =$$
$$(2k+1) + (2k+1) - (3k+1) = k+1 \ .$$

As k is not bounded, there cannot exist a constant B for the loop and it does not satisfy Condition I. It can be also shown that the loop in Fig. 9. (c) does not satisfy Condition I by a similar way. The main result of this paper can be summarized by the following theorem:

Theorem 13 *Condition I is a necessary and sufficient condition for L to have an equivalent time optimal program.*

Section 6 gives a proof on the necessary part of Theorem 13. Section 7 shows that Condition I is also a sufficient condition. The proof is by construction, that is, we present an algorithm to compute the time optimal parallel program for a loop that satisfies Condition I. 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 8 which is equivalent to Condition I and can be computed more easily.

6 Nonexistence of Time Optimal Solution

In this section, we prove that Condition I is a necessary condition for a loop to have an equivalent time optimal parallel program.

Lemma 14 *Condition I.(b) is a necessary condition for L to have an equivalent time optimal program.*

Proof. Suppose L has an equivalent time optimal program L^{SP}. Let B_2 be the maximum height among tree parallel instructions of L^{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 L^{SP} is time optimal and the definition of B_2, $|p'|$ is bounded by $B_2 \cdot \|p'\|$. Therefore, we have

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

□

Lemma 15 *If there exists a constant $B > 0$ such that for any execution path p^e in L*

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

Condition I.(a) is satisfied.

Proof. In order to prove that (1) 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 loop header to $p[1]$ and let p_2 be a simple path from $p[|p|]$ to an exit of L. Then $p' = p_1 \circ p \circ p_2$ is an execution path of L, and the above inequality holds for p'. Therefore, we have

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

where L is the length of the longest simple path in L. □

Throughout the remainder of this section, we assume that L does not satisfy (1) and that L^{SP} is time optimal. Eventually, it is proved that this assumption leads to a contradiction showing that Condition I is indeed a necessary condition. 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 operations. The following proofs are not affected by this transformation.

Lemma 16 *For any $l > 0$, there exists an execution path $p^{e,\mathrm{sp}}$ in L^{SP} and dependence chains of length l in $p^{e,\mathrm{sp}}$, d_1 and d_2, which contain only effective node instances such that $pos(d_1[j]) > pos(d_2[k])$ and $pos(\beta(d_1[j])) < pos(\beta(d_2[k]))$ for any $1 \leq j,\, k \leq l$.*

Proof. From the assumption that L does not satisfy (1), there must exist i_1, i_2 $(i_1 < i_2)$ and p^e such that $\|p^e[1,i_1]\| + \|p^e[i_2,|p^e|]\| >> \|p^e\| + 2 \cdot l$. Note that both the terms of LHS is greater than l because otherwise LHS becomes smaller than or equal to $\|p^e\| + l$, a contradiction.

There exist dependence chains d_1' of length $\|p^e[1,i_1]\|$ and d_2' of length $\|p^e[i_2,|p^e|]\|$ in p^e such that $pos(d_1'[\|p^e[1,i_1]\|]) \leq i_1$ and $pos(d_2'[1]) \geq i_2$. Let $p^{e,\mathrm{sp}}$ be an execution path in L^{SP} such that $\alpha(p^{e,\mathrm{sp}}) = p^e$. By Constraint 6, there exist dependence chains d_1 and d_2 of length l in $p^{e,\mathrm{sp}}$ such that $\beta(d_1[j]) = d_1'[j - l + \|p^e[1,i_1]\|]$ and $\beta(d_2[k]) = d_2'[k]$ for $1 \leq j,\, k \leq l$. Then we have for any $1 \leq j,\, k \leq l$:

$$pos(\beta(d_1[j])) = pos(d_1'[j - l + \|p^e[1,i_1]\|]) \leq i_1 < i_2 \leq pos(d_2'[k]) = pos(\beta(d_2[k]))$$

Next, consider the ranges for $\tau(d_1[j])$ and $\tau(d_2[k])$, respectively :

$$\begin{aligned} \tau(d_1[j]) &\geq \quad |d_1'[1, j - l + \|p^e[1,i_1]\| - 1]| = j - l + \|p^e[1,i_1]\| - 1 \\ \tau(d_2[k]) &\leq \quad \|p^e\| - |d_2'[k, \|p^e[i_2,|p^e|]\|]| + 1 = \|p^e\| - \|p^e[i_2,|p^e|]\| + k \end{aligned}$$

Consequently, we have for any $1 \leq j,\, k \leq l$:

$$\tau(d_1[j]) - \tau(d_2[k]) \geq \|p^e[1,i_1]\| + \|p^e[i_2,|p^e|]\| - \|p^e\| + j - k - l + 1 > 0 \;.$$

Therefore, $pos(d_1[j]) > pos(d_2[k])$. □

For the rest of this section, we use $p^{e,sp}(l)$ to represent an execution path which satisfies the condition of Lemma 16 for a given $l > 0$, and $d_1(l)$ and $d_2(l)$ are used to represent corresponding d_1 and d_2, respectively. In addition, let $i_1(l)$ and $i_2(l)$ be i_1 and i_2, respectively, as used in the proof of Lemma 16 for a given $l > 0$. Finally, $p^e(l)$ represents $\alpha(p^{e,sp}(l))$.

Next, we are to derive the register requirement for "interfering" extended live ranges. $reg(elr(\mathbf{n}), \mathbf{n}')$ is used to denote the register which carries $elr(\mathbf{n})$ at $\mathbf{n}'$.

Lemma 17 *Given k assignment node instances $\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_k$ in an execution path in L^{SP} and a node instance $\mathbf{n}$ in the execution path, if $\mathbf{n}$ is contained in $elr(\mathbf{n}_i)$ for all $1 \leq i \leq k$, $reg(elr(\mathbf{n}_1), \mathbf{n})$, $reg(elr(\mathbf{n}_2), \mathbf{n})$, $\cdots$, $reg(elr(\mathbf{n}_k), \mathbf{n})$ are all distinct.*

Proof. The proof is by induction on k. The base case is trivial. For the induction step, assume the above proposition holds for $k = h \geq 1$. Consider $h+1$ assignment node instances $\mathbf{n}'_1, \mathbf{n}'_2, \cdots, \mathbf{n}'_{h+1}$ in an execution path $p^{e,sp}$ whose extended live ranges share a common node instance $\mathbf{n}'$. Without loss of generality let us assume $pos(\mathbf{n}'_{h+1}) > pos(\mathbf{n}'_i)$ for all $1 \leq i \leq h$. Then the range shared by these extended live ranges can be written as $t(p^{e,sp})[pos(\mathbf{n}'_{h+1}), pos(\mathbf{n}')]$.

By induction hypothesis, $reg(elr(\mathbf{n}'_1), \mathbf{n}'_{h+1})$, $\cdots$, $reg(elr(\mathbf{n}'_h), \mathbf{n}'_{h+1}))$ are all distinct. Moreover, $reg_W(\mathbf{n}'_{h+1})$ must differ from these h registers since the live range defined by $\mathbf{n}'_{h+1}$ interferes with any live ranges carried by these registers. For the same reason at any point in $t(p^{e,sp})[pos(\mathbf{n}'_{h+1}), pos(\mathbf{n}')]$, any register which carries part of $elr(\mathbf{n}'_{h+1})$ differs from h distinct registers which carry extended live ranges of $\mathbf{n}'_i$s. Therefore, the proposition in the above lemma holds for all $k > 0$. □

For loops without control flows, the live range of a register cannot spans more than an iteration although sometimes it is needed to do so. *Modulo variable expansion* handles this problem by unrolling the software-pipelined loop by sufficiently large times such that II becomes no less than the length of the live range [20]. Techniques based on *Enhanced Pipeline Scheduling* usually overcome this problem by splitting such long live ranges by copy operations during scheduling, which is called as dynamic renaming or partial renaming [26]. Optionally these copy operations are coalesced away after unrolling by a proper number of times to reduce resource pressure burdened by these copy operations. Hardware support such as *rotating register files* simplifies register renaming. For any cases, the longer a live range spans, the more registers or amount of unrolling are needed. There is a similar property for loops with control flows as shown below.

Lemma 18 *Given an effective branch node instance $\mathbf{n}_b$ in an execution path $p^{e,sp}$ in L^{SP} and a dependence chain d in $p^{e,sp}$ such that for any node instance $\mathbf{n}$ in d, $pos(\mathbf{n}) < pos(\mathbf{n}_b)$ and $pos(\beta(\mathbf{n})) > pos(\beta(\mathbf{n}_b))$, there exist at least $\lfloor |d|/(M+1) \rfloor - 1$ node instances in d whose extended live ranges contain $\mathbf{n}_b$ where M denotes the length of the longest simple path in L.*

Proof. Let $p^e = \alpha(p^{e,sp})$ and $M' = \lfloor |d|/(M+1) \rfloor$. From the definition of M, there must exist $pos(\beta(d[1])) \leq i_1 < i_2 < \cdots < i_{M'} \leq pos(\beta(d[|d|]))$ such that $p^e[i_1] = p^e[i_2] = \cdots = p^e[i_{M'}]$. If $p^e[i] = p^e[j]$ $(i < j)$, there must exist a node instance in p^e, $\mathbf{n}'$ $(i \leq pos(\mathbf{n}') < j)$ such that $\forall k > pos(\mathbf{n})$, $reg_W(p^e[k]) \neq reg_W(node(\mathbf{n}'))$. Thus

by Constraint 7, there must exist node instances in d, $\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_{M'-1}$, such that

$$reg_{\mathrm{W}}(node(\beta(\mathbf{n}_i))) = reg_{\mathrm{W}}(node(\mathbf{n}_i)) \qquad \text{or}$$
$$reg_{\mathrm{W}}(node(\beta(\mathbf{n}_i))) = reg_{\mathrm{W}}(node(\mathbf{n}^c))$$
$$\text{for some node instance} \qquad \mathbf{n}^c \in Prop(\mathbf{n}_i) \ \text{ for all } \ 1 \le i \le M' - 1 \ .$$

Since $pos(\mathbf{n}_i) < pos(\mathbf{n}_\mathrm{b})$ and $pos(\beta(\mathbf{n}_i)) > pos(\beta(\mathbf{n}_\mathrm{b}))$, $node(\mathbf{n}_i)$ is speculative for all $1 \le i \le M' - 1$. By Constraint 8, $reg_{\mathrm{W}}(node(\mathbf{n}_i)) \notin \mathbf{R}$ and the value defined by $\mathbf{n}_i$ cannot be committed into $r \in \mathbf{R}$ until $\mathbf{n}_b$. So, $elr(\mathbf{n}_i)$ should contain $\mathbf{n}_\mathrm{b}$ for all $1 \le i \le M' - 1$. □

Lemma 19 *Let $\mathbf{N}_\mathrm{b}(l)$ represent the set of effective branch node instances in $p^{\mathrm{e,sp}}(l)$ such that*

$$\mathbf{N}_\mathrm{b}(l) = \{\mathbf{n}_\mathrm{b} \mid pos(\beta(\mathbf{n}_\mathrm{b})) \le i_1(l) \ \wedge \ pos(\mathbf{n}_\mathrm{b}) > pos(d_2(l)[1]) \ \} \ .$$

Then there exists a constant $C > 0$ such that

$$\tau(\mathbf{n}_b) < \|p^{\mathrm{e}}(l)[1, i_1(l)]\| - 2 \cdot l + C.$$

Proof. Let $C = (M+1)(R+2)$ where M is defined as in Lemma 18 and R denotes the number of registers used in L^{SP}. Suppose $\tau(\mathbf{n}_\mathrm{b}) \ge \|p^{\mathrm{e}}(l)[1, i_1(l)]\| - 2 \cdot l + C$.

From the proof of Lemma 16, we have

$$\tau(d_2(l)[C]) \le \|p^{\mathrm{e}}(l)\| - \|p^{\mathrm{e}}(l)[i_2(l), |p^{\mathrm{e}}(l)|]\| + C - 1 < \tau(\mathbf{n}_\mathrm{b}) \ .$$

So at least $\lfloor C/(M+1) \rfloor - 1 = R + 1$ registers are required by Lemmas 17 and 18, a contradiction. Therefore, we have

$$\tau(\mathbf{n}_\mathrm{b}) < \|p^{\mathrm{e}}(l)[1, i_1(l)]\| - 2 \cdot l + C \ . \ \square$$

Theorem 20 *Condition I.(a) is a necessary condition for L to have an equivalent time optimal program.*

Proof. By Lemma 19, there exist an effective branch node instance $\mathbf{n}_\mathrm{b}$ in $p^{\mathrm{e,sp}}(l)$ such that

$$\tau(\mathbf{n}_\mathrm{b}) < \|p^{\mathrm{e}}(l)[1, i_1(l)]\| - 2 \cdot l + C \ \text{ and } \ \tau(\mathbf{n}_\mathrm{b}) > \tau(\mathbf{n}'_\mathrm{b})$$

where $\mathbf{n}'_\mathrm{b}$ represents any branch node instance in $\tau(\mathbf{n}'_b)$ such that $pos(\beta(\mathbf{n}'_\mathrm{b})) \le pos(\beta(d'_1(l)[l])$.

Let $P(\mathbf{n}_\mathrm{b})$ be the set of execution paths in L^{SP} such that

$$P(\mathbf{n}_\mathrm{b}) = \{\ q^{\mathrm{e,sp}} \mid q^{\mathrm{e,sp}}[1, pos(\mathbf{n}_\mathrm{b})] = p^{\mathrm{e,sp}}(l)[1, pos(\mathbf{n}_\mathrm{b})] \ \wedge \ dir(t(q^{\mathrm{e,sp}})[pos(\mathbf{n}_\mathrm{b})]) \ne dir(t(p^{\mathrm{e,sp}}(l))[pos(\mathbf{n}_\mathrm{b})]) \ \}$$

Then $\|q^{\mathrm{e,sp}}\| \ge \|p^{\mathrm{e}}(l)[1, i_1(l)]\|$. By Lemma 2, we have $\|q^{\mathrm{e,sp}}[pos(\mathbf{n}_\mathrm{b}) + 1, \|q^{\mathrm{e,sp}}\|]\| > l - C$. Since l is not bounded and C is bounded, the length of any path starting from $node(\mathbf{n}_\mathrm{b})$ is not bounded, a contradiction. Therefore the assumption that L^{SP} is time optimal is not valid and by Lemma 14 Condition I is indeed a necessary condition. □

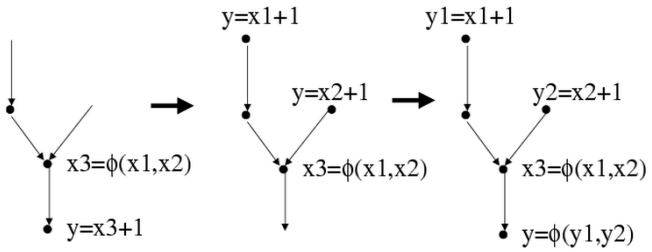

Figure 10. Scheduling above a ϕ-function at the join point

7 Time Optimal Software Pipelining Algorithm

In this subsection, we prove that L has an equivalent time optimal program if L satisfies Condition I. Instead of merely showing the existence of a time optimal program, we construct the time optimal program by a software pipelining algorithm which is based on the algorithm by Aiken *et al*. [4]. 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.

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 L satisfies Condition I.

The time-optimal software pipelining algorithm is mostly based on the algorithm by Aiken *et al*. [4], the latest version of *Perfect Pipelining* [3]. 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 [27]. 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 [4]. 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 [10], 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.

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 10, `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 10, 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

```
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 11. The time-optimal software pipelining algorithm.

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 might be remained, 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 L^∞ represent the infinite recursive graph obtained by unrolling L infinite times. For a node n in L, let n^i denote the corresponding node in the i-th unrolled copy of L in L^∞. For a set X of nodes in L^∞, X^j is defined to be the set $\{n^{i+j} | n^i \in X\}$. Two sets of nodes in L^∞, 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 L^∞, an acyclic infinite CFG, and successively transforms L^∞ into L^{SP} which consists of parallel groups. Figure 11 describes the overall processing steps of the software pipelining algorithm. The procedure SOFTWARE_PIPELINE calls the SCHEDULE_PARALLEL_GROUP procedure (Appendix A.1) 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 (Appendix A.2) is invoked to compute the set of all available operation nodes that can move into the parallel group without violating

```
procedure INFINITE_ACYCLIC_SCHEDULE ( L, window_size )
    L' := L^∞
    frontiers := {(n_start, n_root)}
    /* below condition is always true, thus loops forever */
    while (∃ (n_p, n_s) ∈ frontiers)
        frontiers := frontiers − {(n_p, n_s)}
        A := COMPUTE_AVAILABLE_OPERATIONS( L', n_s, window_size)
        SCHEDULE_PARALLEL_GROUP( L', n_s, A, frontiers)
    end while
    /* cannot reach here */
    return L'
end function
```

Figure 12. An algorithm for building an infinite parallel program.

the true dependences.[5] In our algorithm, every operation node that is not blocked by the true dependences is always available for scheduling. As in [4], we impose additional constraint on available operations: operations are available at most *k* iterations. The predetermined constant *k* is called a *sliding window* [4] 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 operation nodes to a group boundary [26].[6] 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 (Appendix A.3).

The following lemma states the correctness of the proposed algorithm.

Lemma 21 *L^{SP} is equivalent to L.*

Proof. We use the algorithm in Figure 12 to prove the correctness of our software pipelining algorithm. This algorithm is identical to the one in Figure 11 except that it does not reuse previously scheduled parallel groups. Let L_∞^{SP} be the infinite parallel program defined by the algorithm in Figure 12 for a loop L. Then L_∞^{SP} is equivalent to L since all the transformations can be implemented by those in the Moon's algorithm [26]. Thus, we prove the correctness of the software pipelining algorithm of Figure 11 by showing that L^{SP} is equivalent to L_∞^{SP}, i.e., for any initial store s, the final store of $ep(L^{SP}, s)$ is equal to that of $ep(L_\infty^{SP}, s)$. It suffices to show the following properties.

- For any $k > 0$, if there is a sequence of configurations $\langle\langle n_0, s_0\rangle, \cdots, \langle n_k, s_k\rangle\rangle$ of L_∞^{SP}, then there is an sequence of configurations $\langle\langle n'_0, s_0\rangle, \cdots, \langle n'_k, s_k\rangle\rangle$ of L^{SP}.

- When n_i and n'_i are scheduled, the set of available operations in the INFINITE_ACYCLIC_SCHEDULE procedure and that in the SOFTWARE_PIPELINE procedure are equivalent.

[5] This procedure is functionally equivalent to the same procedure in the Moon's algorithm [26].

[6] 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 [24], they preserve program semantics.

- $n_i \equiv {n'_i}^j$ for some j.

The proof is by induction on k. For the base case, let $e = \langle\langle n_0, s_0\rangle\rangle$ be a sequence of configurations in L^{SP}_{∞}. The initial set of available operations A is the same for both L^{SP}_{∞} and L^{SP}. Now in procedure SOFTWARE_PIPELINE, *scheduled_before* is empty, because initially no operation nodes are scheduled. Then the operation to be scheduled is the same as INFINITE_ACYCLIC_SCHEDULE and $e = \langle\langle n_0, s_0\rangle\rangle$ is a equence of configurations in L^{SP}. For the induction step, assume that the proposition in the Lemma holds for k.

If n_k is the exit node, then n'_k is also the exit node and we are done. Otherwise, in the next transition we have $\delta(\langle n_i, s_i\rangle) = \langle n_{i+1}, s_{i+1}\rangle$ and $\delta(\langle n'_i, s_i\rangle) = \langle n'_{i+1}, s_{i+1}\rangle$. The stores must be the same in the two transitions since, by induction hypothesis, n_k and n'_k have the same operations. Let c be the direction of branch taken by the configuration $\langle\langle n_k, s_k\rangle\rangle$. Note c is also taken in the configuration $\langle\langle n'_k, s_k\rangle\rangle$, because n_k and n'_k have the same operations evaluated in the same store. It remains to show that n_{k+1} and n'_{k+1} have the same operations, possibly differing in iteration numbers, i.e., we must show $n_{k+1} \equiv {n'_{k+1}}^j$ for some j.

Consider the available operations B and B' of the two algorithms when n_{k+1} and n'_{k+1} are scheduled. By the induction hypothesis, available operations are equivalent, i.e., $B \equiv B'^l$ for some l. Now there are two cases. For the first case, assume that *scheduled_before* does not include any equivalent set of available operations. Then, the two algorithms work in the same manner and we have $n_{k+1} \equiv {n'_{k+1}}^l$. For the other case, assume that there is a j such that $B^j \in$ *scheduled_before*. Then the parallel group is scheduled in L^{SP} using available operations B^j. The rest can be proved by a similar manner to the above case. □

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 22 *Let L^{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 L^{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. □

7.1 Time Optimality of the Algorithm

The software pipelining algorithm described in Figure 11 always generates time optimal parallel programs for loops that satisfy Condition I. The proof is based on the greediness of the algorithm. Before presenting the

time optimality proof, we prove some miscellaneous properties stated below in Lemmas 23 and 24. (Recall that we have assumed that L satisfies Condition I and that every operation takes 1 cycle to execute.)

Lemma 23 *For a path p in 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.

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

□

Lemma 24 *For node instances $\mathbf{n}_1$ and $\mathbf{n}_2$ in a path p in 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 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

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

□

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

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

where L is the length of the shortest cycle in L.

Lemma 25 *Let L^{SP} be the result of the software pipelining algorithm with the sliding window of WS iterations. Then L^{SP} is time optimal.*

Proof. It suffices to show that for an arbitrary but fixed execution path $p^{\mathrm{e,sp}}$ in L^{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 let $G_{\mathrm{D}}(N_{\mathrm{D}}, E_{\mathrm{D}})$ 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

$$\begin{aligned} \|p\| &< |\mathbf{p}_{\mathrm{D}}| = i_1 + \textstyle\sum_{j=1}^{s-1}(i_{j+1} - i_j) + h - i_s \\ &\leq \|p[1, pos(\mathbf{n}_{i_1})]\| + \textstyle\sum_{j=1}^{s-1}\|p[pos(\mathbf{n}_{i_j+1}), pos(\mathbf{n}_{i_{j+1}})]\| + \|p[pos(\mathbf{n}_{i_s+1}), \|p\|]\| \quad . \end{aligned} \tag{3}$$

From Lemma 23, we can write

$$\begin{aligned} \|p\| \geq\ & \|p[1, pos(\mathbf{n}_{i_1})]\| + \textstyle\sum_{j=1}^{s-1}\|p[pos(\mathbf{n}_{i_j+1}), pos(\mathbf{n}_{i_{j+1}})]\| + \\ & \|p[pos(\mathbf{n}_{i_s+1}), \|p\|]\| + \textstyle\sum_{j=1}^{s}\|p[pos(\mathbf{n}_{i_j}), pos(\mathbf{n}_{i_j+1})]\| - 2s \cdot (B_1 + 1) \quad . \end{aligned} \tag{4}$$

From (3) and (4), we have

$$\textstyle\sum_{j=1}^{s}\|p[pos(\mathbf{n}_{i_j}), pos(\mathbf{n}_{i_j+1})]\| < 2s \cdot (B_1 + 1) \quad . \tag{5}$$

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 24 we have for all $1 \leq i \leq s$

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

which contradicts (5). 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 22, there must exist $i' < i$ such that

$$\begin{aligned} & t(p^{\mathrm{e,sp}})[i'] \text{ is an effective node instance} \quad \text{and} \\ & \tau(t(p^{\mathrm{e,sp}})[i']) = \tau(t(p^{\mathrm{e,sp}})[i]) - 1 \quad \text{and} \\ & (\ \beta(t(p^{\mathrm{e,sp}})[i']) \prec \beta(t(p^{\mathrm{e,sp}})[i]) \quad \vee it(\beta(t(p^{\mathrm{e,sp}})[i])) - it(\beta(t(p^{\mathrm{e,sp}})[i'])) > WS\ ) \end{aligned} \tag{6}$$

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

$$\sigma(\beta(t(p^{\mathrm{e,sp}})[i])) \geq \sigma(\beta(t(p^{\mathrm{e,sp}})[i'])) + 1 \ . \tag{7}$$

From (6), (7) and the induction hypothesis, we have

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

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 $t(p^{e,\mathrm{sp}})[k]$ 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 25, we can state the following theorem.

Theorem 26 *Condition I is a sufficient condition for L to have an equivalent time optimal program.*

Although showing the existence of the constants is sufficient for proving Theorem 13, it does not guarantee that a time-optimal software pipelining algorithm exists, since the algorithm described in Section 7 requires the constants to be computed before parallelization. In the next section, we explain how to compute the constants used in Condition I.

8 Computability of the 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 L as follows.

Definition 27 *Given a cycle c (may not be simple) in L, d is a dependence cycle with respect to c if there exist $l \geq 1$ and $1 \leq i_1 < i_2 < \cdots < i_{|d|} \leq l \cdot (|c|-1)$ such that*

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

Figure 13 shows an example of dependence cycles. We associate several attributes with the dependence cycle, which are defined below.

Definition 28 *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 27 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 $max\_slope(c)$.*

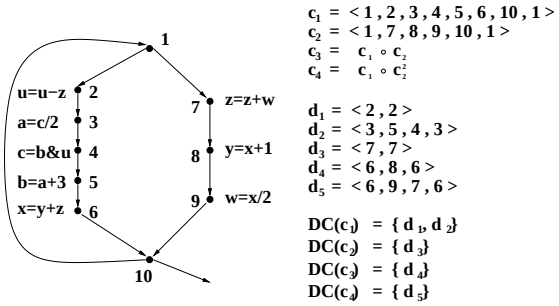

Figure 13. Dependence cycles

There are a finite number of simple dependence cycles in $DC_{\rm cr}(c)$ as well as in $DC(c)$ and these dependence cycles can be enumerated using the Johnson's algorithm [18]. 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 29 *Given two cycles c_1 and c_2 in 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*

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

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 $d_1 \bowtie d_2$.[7]

By unrolling appropriate times, a loop L can always be transformed into L' such that for every dependence cycle in L', its span is 1. For brevity, we assume this property for the rest of this section. Let $\mathbf{C} = \{c_1, c_2, \cdots\}$ represent the set of all the simple cycles in 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:

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

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*

$d_i \in DC_{\rm cr}(c_i)$ *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.

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

Lemma 30 *If a given loop L satisfies Condition I, it also satisfies Condition II.*

Proof. (b) is obviously satisfied. Suppose (a) is not satisfied for some c_1 and c_2. For $d_1 \in DC_{\rm cr}(c_1)$, select $d_2 \in DC_{\rm cr}(c_2)$ and $d_3 \in DC(c_1)$ such that $d_3 \prec^C d_2$ and $slode(d_3)$ is maximal. 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

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

for some $d_4 \in DC(c_1)$, $d_5 \in DC(c_2)$. 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,

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

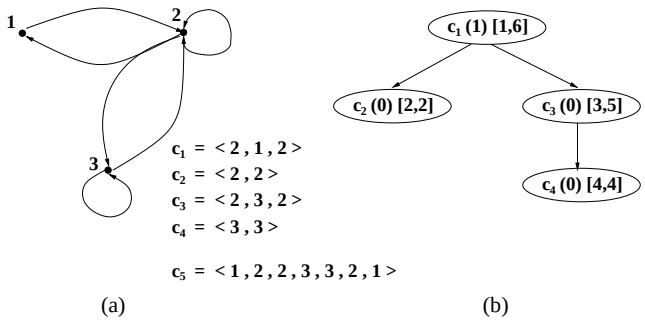

Figure 14. A new representation for a cycle: (a) A graph with cycles and (b) a tree representation of c_5

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 31, it is useful to represent a cycle by a composition of given subcycles. For example, consider a cycle c_5 shown in Figure 14.(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 14.(b).

Given a cycle c, the tree representation of c, written by $CT(c)$, can be found by the algorithm in Appendix B. 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 Appendix C. 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 $1 \le i \le |c|$.

Lemma 31 *For any cycle c in $\bar{L}$ 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 $d_j \circ d_k$, $(d_j \in DC_{\mathrm{cr}}(c_j))$ 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 $\rho_{i_1,i_2,\cdots,i_{|\mathbf{C}|}}$ $(1 \le i_1, i_2, \cdots, i_{|\mathbf{C}|} \le |\mathbf{C}|)$, we solve the following linear system of $|\mathbf{C}^*|$ equations in the $|\mathbf{C}|^{|\mathbf{C}|}$ unknowns.

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 L.

Lemma 32 *Given a path $p = p_s \circ c_{i_1} \circ c_{i_2} \circ \cdots \circ c_{i_k} \circ p_f$ in L where $k \geq |\mathbf{C}|$, $c_{i_j} \in |\mathbf{C}|$ for all $1 \leq j \leq k$ and both p_s and p_f are simple paths, let M_3 be*

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

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 31. Therefore, we have

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

Similarly,

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

□

From Lemma 32, we can compute the constants.

Lemma 33 *If B_1 is selected as $2 \cdot M_1 + 4 \cdot M_2$, Condition I.(a) is satisfied.*

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

$$p_1 = p_{s_1} \circ c_{i_1} \circ \cdots \circ c_{i_l} \circ p_{f_1} \ , \ p_2 = p_{s_2} \circ c_{i_{l+2}} \circ \cdots \circ c_{i_k} \circ p_{f_2} \ .$$

By Lemma 32 we have

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

□

Lemma 34 *If B_2 and B_3 are selected as*

$$\begin{aligned} B_2 &= \max\{ \frac{|c_j|}{\rho_{i_1,i_2,\cdots,i_{|\mathbf{C}|}}} \mid c_j \in |\mathbf{C}| \ , \ 1 \leq i_1, i_2, \cdots, i_{|\mathbf{C}|} \leq |\mathbf{C}| \} \\ B_3 &= 2 \cdot L_C \end{aligned}$$

where L_C is the length of the longest simple cycle in 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 L,

$$\begin{aligned} |p| &\leq \sum_{h=1}^{k} (|c_{i_h}| - 1) + 2 \cdot L_C \\ &\leq \sum_{h=0}^{k-|\mathbf{C}|} \left(\frac{|c_{i_h}|}{\rho_{i_{1+h}, i_{2+h}, \cdots, i_{|\mathbf{C}|+h}}} \cdot \rho_{i_{1+h}, i_{2+h}, \cdots, i_{|\mathbf{C}|+h}} \right) + B_3 - k \\ &\leq B_2 \cdot \sum_{h=0}^{k-|\mathbf{C}|} \rho_{i_{1+h}, i_{2+h}, \cdots, i_{|\mathbf{C}|+h}} + B_3 \\ &\leq B_2 \cdot \|p\| + B_3 \ . \qquad \text{(By Lemma 32.)} \end{aligned}$$

□

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

Lemma 35 *If a given loop L satisfies Condition II, it also satisfies Condition I.*

Proof. Directly from Lemmas 33 and 34. □

Theorem 36 *Condition I is decidable.*

Proof. From Lemmas 30 and 35, Condition I is equivalent to Condition II, whose decision procedure is obvious from the given expression. □

Theorem 37 *There exists a software pipelining algorithm that computes time optimal programs for loops that satisfy Condition I.*

Proof. From Lemma 25, the algorithm in Figure 11 is a time-optimal software pipelining algorithm provided that the size of sliding window is computable. From Lemmas 33 and 34, 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 Eq. (2). □

9 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 showed that it is decidable whether a loop satisfies the condition. Furthermore, we described a software pipelining algorithm that computes a time optimal solution for every eligible loop satisfying the condition. Our results solve two fundamental open problems on time optimal software pipelining of loops with control flows.

Our work can be extended in several directions for developing more efficient realistic software pipelining algorithms. As a short-term research topic, we plan to improve the complexity of the software pipelining algorithm proposed in this paper. We are developing a more efficient version based on the framework by Milicev *et al.* [21, 22, 23]. As a long-term research goal, we are interested in developing a realistic resource-constrained software pipelining algorithm guided by the results shown in this paper.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] 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.

[3] 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.

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

[5] V. Allan, R. Jones, R. Lee, and S. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[6] 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.

[7] E. R. Altman, R. Govindarajan, and G. R. Gao. Scheduling and Mapping : Software Pipelining in the Presence of Structural Hazards. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 139–150, 11995.

[8] P.-Y. Calland, A. Darte, and Y. Robert. Circuit Retiming Applied to Decomposed Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):24–35, 1998.

[9] L.-F. Chao and E. Sha. Scheduling Data-Flow Graphs via Retiming and Unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1259–1267, 1997.

[10] 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.

[11] K. Ebcioğlu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proc. of the 20th Annual Workshop on Microprogramming*, pages 69–79, 1987.

[12] 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.

[13] 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.

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

[15] F. Gasperoni and U. Schwiegelshohn. Optimal Loop Scheduling on Multiprocessors: A Pumping Lemma for p-Processor Schedules. In *Proc. of the 3rd International Conference on Parallel Computing Technologies*, pages 51–56, 1995.

[16] F. Gasperoni and U. Schwiegelshohn. List Scheduling in the Presence of Branches : A Theoretical Evaluation. *Theoretical Computer Science*, 196(2):347–363, 1998.

[17] R. Govindarajan, E. R. Altman, and G. R. Gao. A Framework for Resource-Constrained Rate-Optimal Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1133–1149, 1996.

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

[19] 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.

[20] 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.

[21] D. Milicev and Z. Jovanovic. A Formal Model of Software Pipelining of Loops with Conditions. In *Proc. of the 11st International Parallel Processing Symposium*, pages 554–558, 1997.

[22] D. Milicev and Z. Jovanovic. Predicated Software Pipelining Technique for Loops with Conditions. In *Proc. of the 12nd International Parallel Processing Symposium*, 1998.

[23] D. Milicev and Z. Jovanovic. Code Generation for Software Pipelined Loops with Conditions. Technical report, University of Belgrade, Faculty of Electrical Engineering, 1999.

[24] S.-M. Moon. *Compile-time Parallelization of Non-numerical Code; VLIW and Superscalar*. PhD thesis, University of Maryland, 1993.

[25] S.-M. Moon and S. Carson. Generalized Multi-way Branch Unit for VLIW Microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 850–862, 1995.

[26] S.-M. Moon and K. Ebcioğlu. Parallelizing Non-numerical Code with Selective Scheduling and Software Pipelining. *ACM Transactions on Programming Languages and Systems*, pages 853–898, 1997.

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

[28] Q. Ning and G. R. Gao. A Novel Framework of Register Allocation for Software Pipelining. In *Proc. of the 20th ACM Symposium on Principles of Programming Languages*, pages 29–42, 1993.

[29] 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.

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

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

[32] 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.

Appendix

A Software Pipelining Subroutines

A.1 Algorithm for Building a Parallel Group

---

procedure SCHEDULE_PARALLEL_GROUP(L', n_{dummy}, A, $frontiers$)
  $boundaries := \{n_{\text{dummy}}\}$
  sort elements in A by the priority order
  foreach ($n \in A$)
    make L' delete consistent for (r, n)
    $n_{\text{new}} :=$ COMBINE_SOURCE_REGISTERS(L', n_{dummy}, n)
    if (n is an assigment)
      foreach (non-blocking $n_i \in duplicates(n)$)
        replace every (n', n_i) by $(n', succ(n_i))$ and delete n_i
      end foreach
      insert n_{new} above n_{dummy}
    else /* n is a branch */
      duplicate the subgraph induced by nodes in paths
        from n_{dummy} to $preds(n)$
      $boundaries := boundaries \cup \{$the duplicate of $n_{\text{dummy}}\}$
      replace (n', n) by $(n', succ_F(n))$ for every $n' \in preds(n)$
      insert $(n'', succ_T(n))$ for every duplicate
        n'' of $n' \in preds(n)$
      replace (n_p, n_{dummy}) by (n_p, n_{new})
      $succ_F(n_{\text{new}}) := n_{\text{dummy}}$
      $succ_T(n_{\text{new}}) :=$ the duplicate of n_{dummy}
    end if
  end foreach
  foreach ($n_b \in boundaries$)
    $frontiers := frontiers \cup \{(n_p, n_b)\}$
  end foreach
end procedure

A.2 Algorithm for Computing Available Operations

```
procedure COMPUTE_AVAILABLE_OPERATIONS( L', n_dummy, window_size)
    min_it := it(succ(n_dummy))
    A := {}
    foreach (n s.t. n is reachable from n_dummy and
                                it(n) < min_it + window_size)
        if (∃ n_dummy ~p~> n s.t. n is not blocked along p)
            A := A ∪ {n}
    end foreach
    return A
end procedure
```

A.3 Algorithm for Combining Source Registers with φ-functions

```
procedure COMBINE_SOURCE_REGISTERS( L', n_dummy, n)
    n' := any non-blocking duplicate of n
    p := any path from n_dummy to n'
    n_r := create a duplicate of n
    for (i := |p| to 1)
        if (p[i] is a φ-function)
            combine n_r with p[i]
    end for
    return n_r
end procedure
```

B Algorithm for Building a Cycle Tree

```
procedure build_cycle_tree(c, C = {c_1, ···})
  /* c : the cycle to decompose, C : the set of simple cycles */
    covered[1 .. |c| − 1] := {F, ···, F}
    V[CT(c)] := {}
    E[CT(c)] := {}
    while (∃ i, covered[i] ≡ F)
        j := 1
        for (k := 1 to |c| − 1)
            if (covered[k] ≡ F)
                orig_index[j] := k
                remained[j++] := c[k]
            end if
        end for
        find the smallest l and m (l < m < j)
            such that remained[l] ≡ remained[m]
        find r and s such that
            c_r(s) ≡ ⟨ remained[l], remained[l + 1], ···, remained[m] ⟩
        V[CT(c)] := V[CT(c)] ∪
                        ⟨ c_r(s), [orig_index[l], orig_index[m]] ⟩
        for (k := l to m − 1)
            covered[orig_index[k]] := T
        end for
    end while
    foreach(x_1 = ⟨ c_r1(s_1), [j_1, k_1]⟩, x_2 = ⟨ c_r2(s_2), [j_2, k_2]⟩ ∈ V[CT(c)])
        if (j_1 < j_2 ∧ k_1 > k_2)
            E[CT(c)] := E[CT(c)] ∪ (x_1, x_2)
        end if
    end foreach
    foreach ((x_1, x_2) ∈ E[CT(c)])
        if (∃ x_3, (x_1, x_3), (x_3, x_2) ∈ E[CT(c)])
            E[CT(c)] := E[CT(c)] − (x_1, x_2)
        end if
    end foreach
```

```
    root[CT(c)] := the unique node with no in-edge
    return CT(c)
```

C Algorithm for Finding the Cycle from a Cycle Tree

```
procedure found_sequence(T, C = {c_1, ···})
  /* T : the tree corresponding to a cycle, C : the set of simple cycles */
    sequence[1 .. M]  /* where root[T] = ⟨ x, [1, M] ⟩ */
    post_order(root[T], sequence, 1)
    return ⟨ sequence[1], sequence[2], ···, sequence[M] ⟩
```

```
procedure post_order(n, sequence, i)
    ⟨ c, [i_1, i_2] ⟩ := n  /* it must be i ≡ i_1 */
    for (j := 1 to outdeg(n))
        ⟨ c', [k_1, k_2] ⟩ := child(n, j)
        for (l := i to k_1 − 1)
            sequence[l] := c[i_1++]
        end for
        i = post_order(child(n, j), sequence, k_1)
    for (j := i to i_2 − 1)
        sequence[j] := c[i_1++]
    end for
    return i_2
```