A First Step Towards Time Optimal Software Pipelining of Loops with Control Flows

Han-Saem Yun[1], Jihong Kim[1], and Soo-Mook Moon[2]

[1] School of Computer Science and Engineering
[2] School of Electrical Engineering
Seoul National University, Seoul, Korea

Abstract. We address the problem of time optimal software pipelining of loops with control flows, one of the most difficult open problems in the area of parallelizing compilers. We present a necessary condition for loops with control flows to have equivalent time optimal programs, generalizing the result by Schwiegelshohn *et al.*, which has been the most significant theoretical result on the problem. As part of the formal treatment of the problem, 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. Being the *first* generalized result on the problem, our work described in this paper forms an important first step towards time optimal software pipelining.

1 Introduction

Software pipelining is a loop parallelization technique for machines that exploit instruction-level parallelism (ILP) such as superscalar or VLIW processors. It transforms a sequential loop so that new iterations can start before preceding ones finish, thus overlapping the execution of multiple iterations in a pipelined fashion. Since most of the program execution time is spent in loops, much effort has been given to developing various software pipelining techniques.

One of the important theoretical open problems in software pipelining is how to test if a loop with control flows has its equivalent time optimal program or not. A program is 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 [1]. If decidable, time optimality can be used as a useful measure in evaluating (or improving) existing software pipelining algorithms.

Although there were several related research investigations [2,3,4] on the problem of time optimal software pipelining of loops with control flows, there have been few significant theoretical results published. The work by Schwiegelshohn *et al.* [1] is the best known and most significant result so far, which simply illustrated that, for some loops with control flows, there cannot exist time optimal parallel programs. Their work lacks *a formalism* required to develop generalized results. To the best of our knowledge, since the work by Schwiegelshohn *et al.* was published, no further research results on the problem

R. Wilhelm (Ed.): CC 2001, LNCS 2027, pp. 182–199, 2001.

has been reported, possibly having been discouraged by the pessimistic result by the Schwiegelshohn *et al.*'s work.

In this paper, we describe a necessary condition for loops with control flows to have equivalent time optimal programs. Our result is the *first* general theoretical result on the problem, and can be considered as a generalization of the Schwiegelshohn *et al.*'s result. In order to prove the necessary condition in a mathematically concrete fashion, we propose a new formalization of software pipelining, which provides a basis of our proof. The proposed formalization of software pipelining has its own significance in that it provides a new theoretical framework for software pipelining research.

We believe that the work described in this paper forms an important first step towards time optimal software pipelining. Although we do not formally prove in the paper, we strongly believe that the necessary condition described in this paper is also the sufficient condition for time optimal programs. Our short-term research goal is to verify this claim so that a given loop with control flows can be classified depending on the existence of time optimal program. Ultimately, our goal is to develop a time optimal software pipelining algorithm for loops that satisfy the condition.

The rest of the paper is organized as follows. In Sect. 2, we briefly review prior theoretical work on software pipelining. We explain the machine model assumptions and program representation in Sect. 3. Section 4 discusses the dependence model. A formal description of software pipelining is presented in Sect. 5 while the proof of a necessary condition is provided in Sect. 6. We conclude with a summary and directions for future work in Sect. 7.

2 Related Work

For loops *without* control flows, there exist several theoretical results [5,6,7,8, 9]. When resource constraints are not present, both the time optimal schedule and the rate optimal one can be found in polynomial time [5,6]. With resource constraints, the problem of finding the optimal schedule is NP-hard in its full generality [6] but there exist approximation algorithms that guarantee the worst case performance of roughly twice the optimum [6,9].

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 [10] 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.

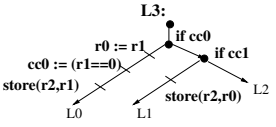

Fig. 1. A tree VLIW instruction

There are few theoretical results for loops with control flows, and, to the best of our knowledge, only two results [1,11] have been published. The work by Uht [11] 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.* [1], 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.

3 Preliminaries

3.1 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* [12]) 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* [13]). 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.* [14].

As a specific example architecture, we use the tree VLIW architecture model [3,15], which satisfies the architectural requirements described above. In this architecture, a parallel VLIW instruction, called a tree instruction, is represented by a binary decision tree as shown in Fig. 1. 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 [12]. An operation is committed only if it lies in the execution path determined by the branch unit [13].

3.2 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

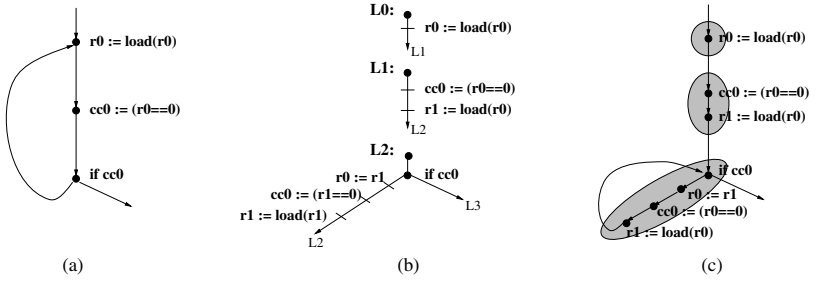

Fig. 2. (a) A sequential program, (b) a parallel tree VLIW program, and (c) a parallel program in the extended sequential representation

parallelized by a compiler, a *parallel tree VLIW program* P_{tree} is generated. While P_{tree} is the final output from the parallelizing compiler for our target architecture, we represent the parallel program in the *extended sequential representation* for the description purpose. 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 2. (a) shows an input sequential program P_{s} and Fig. 2. (b) shows its corresponding parallel tree VLIW program P_{tree}. Using the extended sequential representation, P_{tree} is represented by Fig. 2. (c). The parallel program shown in Fig. 2. (c) is based on a sequential representation except that it has the operation grouping information indicated by shaded regions. The operations belonging to the same group (i.e., the same shaded region) are executed in parallel. A parallel tree VLIW program can be easily converted into the parallel program in the extended sequential representation with some local transformation on copy operations, and vice versa [15].

3.3 Basic Terminology

A program[1] is represented as a triple $\langle\, G = (N, E)\,,\, \mathcal{O}\,,\, \delta\,\rangle$. (This representation is due to Aiken *et al.* [14].) 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.

$\mathcal{O}$ 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

[1] Since a parallel program is represented by the extended sequential representation, the notations and definitions explained in Sect. 3.3 and Sect. 4.1 apply to parallel programs as well as sequential programs.

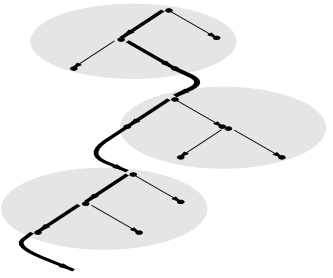

Fig. 3. An execution path in a parallel program

opcode only; Constant fields and register fields are not included in $op(n)$. Without loss of generality, every operation is assumed to write to a single register. We denote by $reg_{\mathrm{W}}(n)$ the register to which n writes and by $regs_{\mathrm{R}}(n)$ a set of registers from which n reads .

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

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 number of nodes in p is denoted by $|p|$ and 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. 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}). Each execution path can be represented by an initial store with which the control flows along the execution path. Suppose a program $\mathcal{P}$ 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 $ep(\mathcal{P}, s_0)$ is defined to be the execution path $\langle n_0, n_1, \cdots, n_f\rangle$. (*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 $\mathcal{P}$, $\exists\, s$ such that $p^{\mathrm{e}} \equiv ep(\mathcal{P}, s)$.

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 Sect. 3.1, 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 Fig. 3, 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:

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

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 Sect. 5.

4 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 Sect. 4.1 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 Sect. 4.2

4.1 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

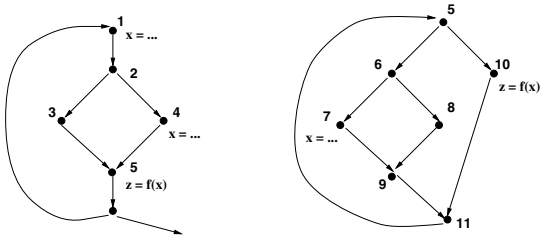

Fig. 4. Path-sensitive dependence examples

not the case and the dependence relationship between two nodes relies on the control flow between them as shown in Fig. 4. In Fig. 4. (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 Fig. 4. (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, it is necessary to define the dependence relation on node instances rather than on nodes themselves. The dependences between node instances carried by registers are defined as follows.

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

$$\begin{aligned} &reg_{\mathrm{W}}(p^{\mathrm{e}}[i]) \in regs_{\mathrm{R}}(p^{\mathrm{e}}[j]) \text{ and} \\ &reg_{\mathrm{W}}(p^{\mathrm{e}}[k])) \neq reg_{\mathrm{W}}(p^{\mathrm{e}}[i]) \text{ for all } i < k < j. \end{aligned}$$

The relation $\prec$ models *true* dependence along p^{e}, which corresponds to actual definition and uses of values during execution. The relation $\prec$ precisely captures the flow of values through an execution of a program. From Definition 1, we can easily verify the following property on the relation $\prec$: For execution paths p_1^{e} and p_2^{e}, where $1 \le i_1 < j_1 \le |p_1^{\mathrm{e}}|$ and $1 \le i_2 < j_2 \le |p_2^{\mathrm{e}}|$,

$$\begin{aligned} j_1 - i_1 = j_2 - i_2 \ \wedge \ p_1^{\mathrm{e}}[i_1 + k] = p_2^{\mathrm{e}}[i_2 + k] \text{ for all } 0 \le k \le j_1 - i_1 \\ \Longrightarrow \ t(p_1^{\mathrm{e}})[i_1] \prec t(p_1^{\mathrm{e}})[j_1] \text{ iff } t(p_2^{\mathrm{e}})[i_2] \prec t(p_2^{\mathrm{e}})[j_2] \ . \end{aligned}$$

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 [16] and is usually represented by *data dependence graphs* [17] or its extensions [18,19]. The above property holds for

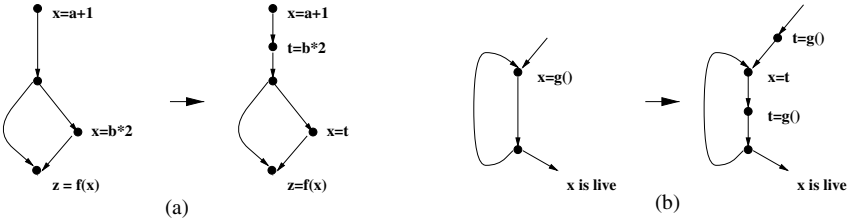

Fig. 5. Copy operations used to overcome false dependences

these representation techniques. In our work, we assume a similar memory dependence relation , in which the dependence relation between node instances with memory operations also satisfies the above property.

4.2 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 [14,15] 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. 5. (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. 5. (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 2. *Given an execution path of a parallel program* p^{e}*, let* $\mathbf{N}_{p^{\mathrm{e}}}$ *represent the set of all node instances in* $t(p^{\mathrm{e}})$*. For node instances* $\mathbf{n}$ *in* $t(p^{\mathrm{e,sp}})$*,* $Prop(\mathbf{n})$ *represents the set of copy node instances in* $t(p^{\mathrm{e}})$ *by which the value defined by* $\mathbf{n}$ *is propagated, that is,*

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

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*

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

Definition 3. *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 ready to define a *dependence chain* for sequential and the parallel programs.

Definition 4. *Given a path* p*, a dependence chain* d *in* p *is a sequence of node instances in* $t(p)$ $\langle \mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_k \rangle$ *such that* $\mathbf{n}_i \overset{*}{\prec} \mathbf{n}_{i+1}$ *for all* $1 \leq i < k$*.*

The i-th component of a dependence chain d is addressed by $d[i]$ and the number of components in d is denoted by $|d|$.

5 Requirements 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 Sect. 6. Given an input loop $\mathcal{L}$ and its parallel version $\mathcal{L}^{\mathrm{SP}}$, let $\mathbf{P}^{\mathrm{e}}$ and $\mathbf{P}^{\mathrm{e,SP}}$ denote the set of all execution paths in $\mathcal{L}$ and the set of those in $\mathcal{L}^{\mathrm{SP}}$, respectively. Let us consider a relation $\mathcal{R} : \mathbf{P}^{\mathrm{e}} \times \mathbf{P}^{\mathrm{e,SP}}$ defined by

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

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 1-5.

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 [15], 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[2] from other execution paths that do not affect the final store of $p^{\mathrm{e,sp}}$ and copy operations used for overcoming false dependences [14,15]. Formally, the first constraint on transformations can be given as follows.

Constraint 1. *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*

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

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[3] of $p^{\mathrm{e,sp}}$ should be equal to that of p^{e} to preserve the semantic of $\mathcal{L}$. 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 2 concisely states this condition.

Constraint 2. *For any assignment node instance* $\mathbf{n}$ *in* $t(p^{\mathrm{e}})$ *such that* $\forall i > pos(\mathbf{n}),\ reg_{\mathrm{W}}(p^{\mathrm{e}}[i]) \neq reg_{\mathrm{W}}(node(\mathbf{n}))$*,*

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

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 $\mathcal{L}$ so as not to affect the final store.

Constraint 3. *Let* $\mathbf{R}$ *be the set of registers that are defined by nodes in* $\mathcal{L}$*. Then the target register of each speculative node in* $\mathcal{L}^{\mathrm{SP}}$ *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 $\mathcal{L}$. $\mathbf{n}'$ 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

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

[3] Temporary registers are excluded.

separated from $p^{e,sp}$ earlier than $\mathbf{n}'$ due to another speculative branch node, the instance of which in $p^{e,sp}$ is redundant, scheduled above $\mathbf{n}'$. This constraint can be written as follows.

Constraint 4. *Given an execution path* p^e *and* q^e *in* $\mathcal{L}$ *such that*

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

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

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

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

$p^{e,sp}$ is said to be equivalent to p^e, written $p^e \equiv_{SA} p^{e,sp}$, if Constraints 1-4 are all satisfied. (The subscript SA is adapted from the expression "semantically and algorithmically equivalent" in [1].) Constraint 4 can be used to rule out a pathological case, *unification of execution paths*. Two distinct execution paths $p_1^e = ep(\mathcal{L}, s_1)$ and $p_2^e = ep(\mathcal{L}, s_2)$ in $\mathcal{L}$ are said to be *unified* if $ep(\mathcal{L}^{SP}, s_1) \equiv ep(\mathcal{L}^{SP}, s_2)$. Suppose p_1^e is separated from p_2^e by a branch, then $ep(\mathcal{L}^{SP}, s_1)$ must be separated from $ep(\mathcal{L}^{SP}, s_2)$ by some branch by Constraint 4. So p_1^e and p_2^e cannot be unified.

Let us consider the mapping cardinality of $\mathcal{R}$. Since distinct execution paths cannot be unified, there is the unique p^e which is related to each $p^{e,sp}$. But there may exist several $p^{e,sp}$'s that are related to the same p^e due to speculative branches. Thus, $\mathcal{R}$ is a one-to-many relation, and if branch nodes are not allowed to be reordered, $\mathcal{R}$ becomes a one-to-one relation. In addition, the domain and image of $\mathcal{R}$ cover the entire $\mathbf{P}^e$ and $\mathbf{P}^{e,SP}$, respectively. Because of our assumption in Sect. 3.3 that all the execution paths are executable, $\forall p^e \in \mathbf{P}^e, \exists s, \ p^e \equiv ep(\mathcal{L}, s)$ and the domain of $\mathcal{R}$ covers the entire $\mathbf{P}^e$. When an execution path $p^e \in \mathbf{P}^e$ is splitted into two execution paths $p_1^{e,sp}, p_2^{e,sp} \in \mathbf{P}^{e,SP}$ 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 $\mathcal{R}$ cover the entire $\mathbf{P}^{e,sp}$. To be short, $\mathcal{R}^{-1}$ is a surjective function from $\mathbf{P}^{e,sp}$ to $\mathbf{P}^e$.

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

Constraint 5. *There exists a surjective function* $\alpha : \mathbf{P}^{e,SP} \Rightarrow \mathbf{P}^e$ *such that*

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

Using α defined in Constraint 5 above and $sp\_ni_{p^e,p^{e,sp}}$ defined in Constraint 1, another useful function β is defined, which maps each node instance in $\mathbf{N}^{SP}$ to its corresponding node instance in $\mathbf{N}$.

Definition 5. $\beta : \mathbf{N}^{\mathrm{SP}} \Rightarrow \mathbf{N}$ *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 1-5.

6 Nonexistence of Time Optimal Solution

In this section, we prove a necessary condition for a loop to have an equivalent time optimal parallel program. Before a formal proof, we first 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 $\mathcal{L}^{\mathrm{SP}}$ and is denoted by $\tau(\mathbf{n})$. Time optimality of the parallel program $\mathcal{L}^{\mathrm{SP}}$ is defined as follows [1,14].

Definition 6. (Time Optimality)
$\mathcal{L}^{\mathrm{SP}}$ *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 $\mathcal{L}^{\mathrm{SP}}$ 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 renaming of the paper, the length of the longest dependence chain in a path p is denoted by $\|p\|$.

For time optimal programs, there have been no significant theoretical results reported, since Schwiegelshohn *et al.* showed that no time optimal parallel programs exist for some loops with control flows. [1]. In this section, we prove a *strong* necessary condition for a loop to have the equivalent time optimal parallel program. Our work is the *first* theoretical result on time optimality of loops with control flows.

The necessary condition for $\mathcal{L}$ to have its equivalent time optimal parallel program is as follows.

Condition 1. *There exists a constant* $B > 0$ *such that for any execution path* p^{e} *in* $\mathcal{L}$

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

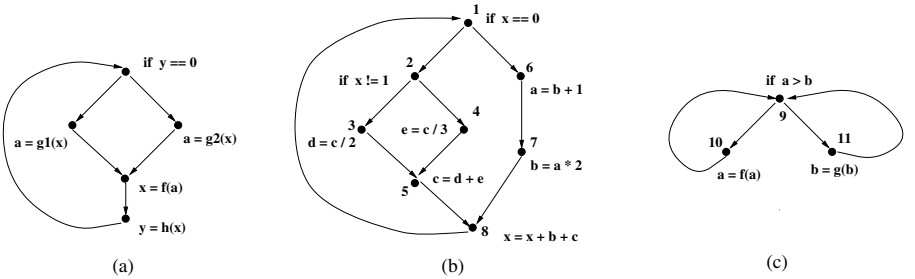

Fig. 6. Example loops used in [1] by Schwiegelshohn *et al.*

Let us consider the example loops shown in Fig. 6. These loops were adapted from [1]. The first one (Fig. 6. (a)), which was shown to have an equivalent time optimal program, satisfies Condition 1. For any execution path p^{e} that loops k iterations, $\|p^{\mathrm{e}}\| = 2k + 1$ and for $1 \leq i < j \leq |p^{\mathrm{e}}| = 4k$, $\|p^{\mathrm{e}}[1, i]\| \leq \lceil i/2 \rceil + 1$ and $\|p^{\mathrm{e}}[j, |p^{\mathrm{e}}|]\| \leq \lceil 2k - j/2 \rceil + 2$. So,

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

The second and third shown in Figs. 6. (b) and 6. (c) do not satisfy Condition 1, thus having no equivalent time optimal programs as shown in [1]. For the loop in Fig. 6. (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 :

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

As k is not bounded, there cannot exist a constant B for the loop and it does not satisfy Condition 1. It can be also shown that the loop in Fig. 6. (c) does not satisfy Condition 1 by a similar way.

Throughout the remaining of this section, we assume that $\mathcal{L}$ does not satisfy Condition 1 and that $\mathcal{L}^{\mathrm{SP}}$ is time optimal. Eventually, it is proved that this assumption leads to a contradiction showing that Condition 1 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 proof is not affected by this transformation.

Lemma 1. *For any $l > 0$, there exists an execution path $p^{\mathrm{e,sp}}$ in $\mathcal{L}^{\mathrm{SP}}$ and dependence chains of length l in $p^{\mathrm{e,sp}}$, 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 $\mathcal{L}$ does not satisfy Condition 1, there must exist i_1, i_2 $(i_1 < i_2)$ and p^{e} such that $\|p^{\mathrm{e}}[1, i_1]\| + \|p^{\mathrm{e}}[i_2, |p^{\mathrm{e}}|]\| \ > \ \|p^{\mathrm{e}}\| + 2 \cdot l$. Note

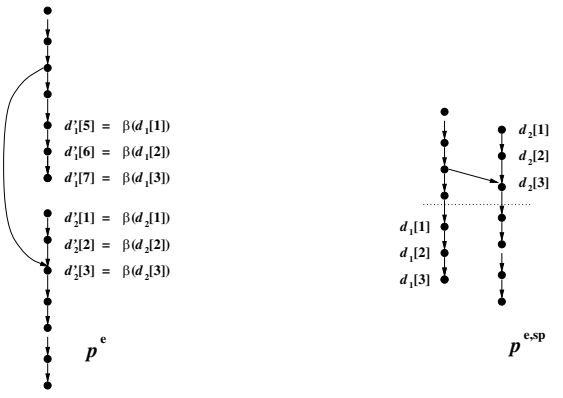

Fig. 7. An example illustrating Lemma 1

that both the terms of LHS is greater than l because otherwise LHS becomes smaller than or equal to $\|p^{\mathrm{e}}\| + l$, a contradiction.

There exist dependence chains d'_1 of length $\|p^{\mathrm{e}}[1, i_1]\|$ and d'_2 of length $\|p^{\mathrm{e}}[i_2, |p^{\mathrm{e}}|]\|$ in p^{e} such that $pos(d'_1[\|p^{\mathrm{e}}[1, i_1]\|]) \leq i_1$ and $pos(d'_2[1]) \geq i_2$. Let $p^{\mathrm{e,sp}}$ be an execution path in $\mathcal{L}^{\mathrm{SP}}$ such that $\alpha(p^{\mathrm{e,sp}}) = p^{\mathrm{e}}$. By Constraint 1, there exist dependence chains d_1 and d_2 of length l in $p^{\mathrm{e,sp}}$ such that $\beta(d_1[j]) = d'_1[j - l + \|p^{\mathrm{e}}[1, i_1]\|]$ and $\beta(d_2[k]) = d'_2[k]$ for $1 \leq j,\ k \leq l$. Then we have for any $1 \leq j,\ k \leq l$:

$$pos(\beta(d_1[j])) = pos(d'_1[j - l + \|p^{\mathrm{e}}[1, i_1]\|]) \leq i_1 < i_2 \leq pos(d'_2[k]) = pos(\beta(d_2[k]))$$

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

$$\begin{aligned} \tau(d_1[j]) &\geq\ |d'_1[1, j - l + \|p^{\mathrm{e}}[1, i_1]\| - 1]| = j - l + \|p^{\mathrm{e}}[1, i_1]\| - 1 \\ \tau(d_2[k]) &\leq\ \|p^{\mathrm{e}}\| - |d'_2[k, \|p^{\mathrm{e}}[i_2, |p^{\mathrm{e}}|]\|]| + 1 = \|p^{\mathrm{e}}\| - \|p^{\mathrm{e}}[i_2, |p^{\mathrm{e}}|]\| + k \end{aligned}$$

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

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

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

Figure 7 illustrates Lemma 1 using an example where $l = 3$.

For the rest of this section, we use $p^{\mathrm{e,sp}}(l)$ to represent an execution path which satisfies the condition of Lemma 1 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 1 for a given $l > 0$. Finally, $p^{\mathrm{e}}(l)$ represents $\alpha(p^{\mathrm{e,sp}}(l))$.

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 2. *Given k assignment node instances $\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_k$ in an execution path in $\mathcal{L}^{\mathrm{SP}}$ 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 $p^{\mathrm{e,sp}}$ 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 $t(p^{\mathrm{e,sp}})[pos(\mathbf{n}'_{h+1}), pos(\mathbf{n}')]$.

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, $reg_{\mathrm{W}}(\mathbf{n}'_{h+1})$ 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 $t(p^{\mathrm{e,sp}})[pos(\mathbf{n}'_{h+1}), pos(\mathbf{n}')]$, 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 [15]. 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 3. *Given an effective branch node instance $\mathbf{n}_{\mathrm{b}}$ in an execution path $p^{\mathrm{e,sp}}$ in $\mathcal{L}^{\mathrm{SP}}$ and a dependence chain d in $p^{\mathrm{e,sp}}$ such that for any node instance $\mathbf{n}$ in d, $pos(\mathbf{n}) < pos(\mathbf{n}_{\mathrm{b}})$ and $pos(\beta(\mathbf{n})) > pos(\beta(\mathbf{n}_{\mathrm{b}}))$, there exist at least $\lfloor |d|/(M+1) \rfloor - 1$ node instances in d whose extended live ranges contain $\mathbf{n}_{\mathrm{b}}$ where M denotes the length of the longest simple path in $\mathcal{L}$.*

Proof. Let $p^{\mathrm{e}} = \alpha(p^{\mathrm{e,sp}})$ 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 $p^{\mathrm{e}}[i_1] = p^{\mathrm{e}}[i_2] = \cdots = p^{\mathrm{e}}[i_{M'}]$. If $p^{\mathrm{e}}[i] = p^{\mathrm{e}}[j]$ $(i < j)$, there must exist a node instance in p^{e}, $\mathbf{n}'$ $(i \leq pos(\mathbf{n}') < j)$ such that $\forall\ k > pos(\mathbf{n})$, $reg_{\mathrm{W}}(p^{\mathrm{e}}[k]) \neq reg_{\mathrm{W}}(node(\mathbf{n}'))$. Thus by Constraint 2, there must exist node instances in d, $\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_{M'-1}$, such that

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

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 $1 \leq i \leq M' - 1$. By Constraint 3, $reg_{\mathrm{W}}(node(\mathbf{n}_i)) \notin \mathbf{R}$ 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 $1 \leq i \leq M' - 1$. □

Lemma 4. *Let $\mathbf{N}_\mathrm{b}(l)$ represent the set of effective branch node instances in $p^{\mathrm{e,sp}}(l)$ such that for any $\mathbf{n}_\mathrm{b} \in \mathbf{N}_\mathrm{b}(l)$, $pos(\beta(\mathbf{n}_\mathrm{b})) \leq i_1(l)$ and $pos(\mathbf{n}_\mathrm{b}) > pos(d_2(l)[1])$. 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 3 and R denotes the number of registers used in $\mathcal{L}^{\mathrm{SP}}$. Suppose $\tau(\mathbf{n}_\mathrm{b}) \geq \|p^{\mathrm{e}}(l)[1, i_1(l)]\| - 2 \cdot l + C$.

From the proof of Lemma 1, $\tau(d_2(l)[C]) \leq \|p^{\mathrm{e}}(l)\| - \|p^{\mathrm{e}}(l)[i_2(l), |p^{\mathrm{e}}(l)|]\| + C - 1 < \tau(\mathbf{n}_\mathrm{b})$. So at least $\lfloor C/(M+1) \rfloor - 1 = R + 1$ registers are required by Lemmas 2 and 3, a contradiction. So, $\tau(\mathbf{n}_\mathrm{b}) < \|p^{\mathrm{e}}(l)[1, i_1(l)]\| - 2 \cdot l + C$. □

Theorem 1. *Condition 1 is a necessary condition for $\mathcal{L}$ to have an equivalent time optimal program.*

Proof. By Lemma 4, there exist an effective branch node instance $\mathbf{n}_\mathrm{b}$ in $p^{\mathrm{e,sp}}(l)$ such that $\tau(\mathbf{n}_\mathrm{b}) < \|p^{\mathrm{e}}(l)[1, i_1(l)]\| - 2 \cdot l + C$ 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 $pos(\beta(\mathbf{n}'_\mathrm{b})) \leq pos(\beta(d'_1(l)[l])$.

Let $P(\mathbf{n}_\mathrm{b})$ be the set of execution paths in $\mathcal{L}^{\mathrm{SP}}$ such that $q^{\mathrm{e,sp}} \in P(\mathbf{n}_\mathrm{b})$ if $q^{\mathrm{e,sp}}[1, pos(\mathbf{n}_\mathrm{b})] = p^{\mathrm{e,sp}}(l)[1, pos(\mathbf{n}_\mathrm{b})]$ and $dir(t(q^{\mathrm{e,sp}})[pos(\mathbf{n}_\mathrm{b})]) \neq dir(t(p^{\mathrm{e,sp}}(l))[pos(\mathbf{n}_\mathrm{b})])$. Then $\|q^{\mathrm{e,sp}}\| \geq \|p^{\mathrm{e}}(l)[1, i_1(l)]\|$. 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 $\mathcal{L}^{SP}$ is time optimal is not valid and Condition 1 is indeed a necessary condition. □

7 Conclusion

In this paper, we presented a necessary condition for loops with control flows to have their equivalent time optimal programs. The necessary condition described in the paper generalizes the Schwiegelshohn *et al.*'s work, which was lacking for a formalism to produce such a general condition. Based on a newly proposed formalization of software pipelining, we proved the necessary condition in a mathematically concrete fashion.

Our result, which is the first general theoretical result on time optimal software pipelining, is an important first step towards time optimal software pipelining. We strongly believe that the necessary condition presented in the paper is also the sufficient condition. Our immediate future work, therefore, includes the verification of this claim. As a long-term research goal, we plan to develop a time optimal software pipelining algorithm that can generate time optimal programs for eligible loops.

References

1. U. Schwiegelshohn, F. Gasperoni, and K. Ebcioğlu. On Optimal Parallelization of Arbitrary Loops. *Journal of Parallel and Distributed Computing*, 11(2):130–134, 1991.
2. A. Aiken and A. Nicolau. Perfect Pipelining. In *Proceedings of the Second European Symposium on Programming*, pages 221–235, June 1988.
3. K. Ebcioğlu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming (Micro-20)*, pages 69–79, 1987.
4. A. Zaky and P. Sadayappan. Optimal Static Scheduling of Sequential Loops with Tests. In *Proceedings of the International Conference on Parallel Processing*, pages 130–137, 1989.
5. A. Aiken and A. Nicolau. Optimal Loop Parallelization. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 308–317, 1988.
6. F. Gasperoni and U. Schwiegelshohn. Generating Close to Optimum Loop Schedules on Parallel Processors. *Parallel Processing Letters*, 4(4):391–403, 1994.
7. F. Gasperoni and U. Schwiegelshohn. Optimal Loop Scheduling on Multiprocessors: A Pumping Lemma for p-Processor Schedules. In *Proceedings of the 3rd International Conference on Parallel Computing Technologies*, pages 51–56, 1995.
8. 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.
9. 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.
10. F. Gasperoni and U. Schwiegelshohn. List Scheduling in the Presence of Branches: A Theoretical Evaluation. *Theoretical Computer Science*, 196(2):347–363, 1998.
11. A. Uht. Requirements for Optimal Execution of Loops with Tests. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):573–581, 1992.
12. 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.
13. K. Ebcioğlu. Some Design Ideas for a VLIW Architecture for Sequential Natured Software. In *Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing*, pages 3–21, 1988.
14. A. Aiken, A. Nicolau, and S. Novack. Resource-Constrained Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1248–1270, 1995.
15. S.-M. Moon and K. Ebcioğlu. Parallelizing Non-numerical Code with Selective Scheduling and Software Pipelining. *ACM Transactions on Programming Languages and Systems*, pages 853–898, 1997.

16. V. Allan, R. Jones, R. Lee, and S. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.
17. D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 207–218, 1981.
18. 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.
19. K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graphs: An Algebraic Approach to Program Dependences. In *Proceedings of the 1991 Symposium on Principles of Programming Languages*, pages 67–78, 1991.
20. M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.