International Journal of Parallel Programming, Vol. 31, No. 5, October 2003 (© 2003)

Time Optimal Software Pipelining of Loops with Control Flows

Han-Saem Yun,[1] Jihong Kim,[1,3] and Soo-Mook Moon[2]

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

KEY WORDS: Software pipelining; instruction-level parallelism; VLIW; compiler optimization.

[1] School of Computer Science and Engineering, Seoul National University, Seoul, Korea 151–742. E-mail: {hsyun, jihong}@davinci.snu.ac.kr

[2] School of Electrical Engineering, Seoul National University, Seoul, Korea 151-742. E-mail: smoon@altair.snu.ac.kr

[3] To whom correspondence should be addressed

0885-7458/03/1000-0339/0 © 2003 Plenum Publishing Corporation

1. INTRODUCTION

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

While software pipelining has been used as a major compiler optimization technique to achieve high performance for ILP processors, surprisingly, there have been few theoretical results, let alone practical ones, known on the optimality issue of software pipelined programs. One of the best known open problems is the time optimal software pipelining problem, which can be stated as follows: *given a loop (with or without control flows), (1) decide if the loop has its equivalent time optimal program or not and (2) find a time optimal parallel program if the loop has one, assuming that sufficient resources are available*. A parallel program is said to be time optimal if every execution path p of the program runs in its minimum execution time determined by the length of the longest data dependence chain in p.[26]

For straight-line loops (without control flows), the time optimal software pipelining problem is well understood and a time optimal program can be computed in polynomial time.[1] This is because the process of software pipelining can be easily formalized thanks to the strong periodicity of such loops (e.g., a periodic execution model and dependence patterns). For example, the problem of software pipelining of such loops can be modeled by a simple linear formulation and several software pipelining algorithms have been developed using this model.[5, 6, 11, 14]

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.

The time optimality is not only of theoretical interest but also of practical importance for the following reasons: First, ILP available in programs (especially in non-numerical programs, which are the main sources of branch-intensive loops) is inherently limited to an extent that the assumption of unlimited resource does not impose any practical constraint on many time optimal programs computed.[29] Second, knowing the time optimal schedules may result in better parallel schedules (e.g., by local transformations) even under the resource-constrained situations, as is the

case with some software pipelining algorithms for straight-line loops.[6, 11, 19] Third, time optimal schedules may be the only high performance schedule attainable. For example, one can argue for profile-guided scheduling. However, the profile information is not so helpful unless iterations of the hot paths stay in the same acyclic path of a loop for a long time and transitions from one acyclic path to another are infrequent.[27]

1.1. Related Work

For loops *without* control flows, there are several theoretical results.[1, 6, 7, 11, 12] When resource constraints are not present, both the time optimal schedule and the rate optimal one can be found in polynomial time.[1, 11] With resource constraints, the problem of finding the optimal schedule is NP-hard in its full generality[11] but there exist approximation algorithms that guarantee the worst case performance of roughly twice the optimum.[6, 11]

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[13] 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.

Until recently, only two results for loops with control flows were published.[26, 28] The work by Uht[28] proved that the resource requirement necessary for the time optimal execution may increase exponentially for some loops with control flows. The work by Schwiegelshohn *et al.*,[26] which is the best known and most significant result on time optimal programs, simply illustrated that certain loops with control flows do not have their equivalent time optimal programs. Since the work by Schwiegelshohn *et al.* was published, no further research results on the problem have been reported for about a decade, possibly having been discouraged by the pessimistic result.

Instead, most researchers focused on developing *better* software pipelining algorithms. To overcome the difficulty of handling control flows, many developed algorithms imposed unnecessarily strict constraints on

possible transformations of software pipelining. For example, several software pipelining algorithms first apply transformations that effectively remove control flows before scheduling,[4, 19] and recover control flows after scheduling.[30] Although practical, these extra transformations prohibit considerable amount of code motions, limiting the scheduling space exploration significantly.

1.2. Contributions

In this paper, we are to identify exactly what can and cannot be achieved by software pipelining and to empirically evaluate how often software pipelining can generate optimal[4] solutions in real applications. Our contributions can be divided into two parts, theoretical ones and practical ones.

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

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

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

For loops with control flows, these two questions have not been adequately formulated, let alone being solved. In this paper, we call the necessary and sufficient condition for a loop to have its equivalent time optimal program as the *Time Optimality Condition*. As an answer to the first question, we present the Time Optimality Condition and describe how to compute the Time Optimality Condition. For the second question, we present a software pipelining algorithm that computes time optimal programs for every loop satisfying the Time Optimality Condition.

Figure 1 summarizes our theoretical contributions graphically. The enclosing ellipse represents the set U of all the reducible innermost loops and the bold curve represents the boundary between two sets of loops, one set whose loops have equivalent time optimal programs (i.e., the right region) and the other set whose loops do not have time optimal programs (i.e., the left region). The small circle represents the set of loops shown to have no time optimal solutions by Schwiegelshohn *et al.*[26] 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.

[4] In the rest of the paper, we use "optimal" and "time optimal" interchangeably where no confusion arises.

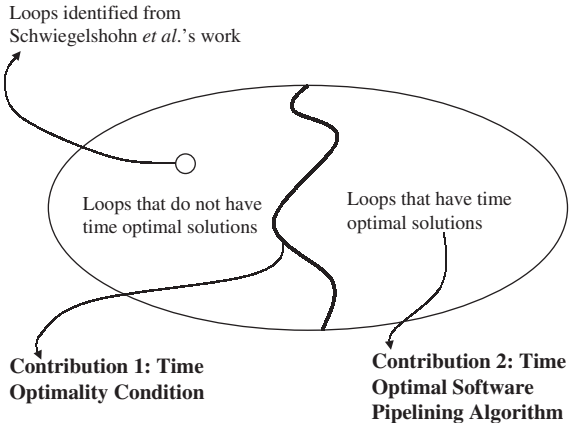


Fig. 1. Loop classification based on time optimality.

The optimal software pipelining algorithm, which is given to answer Question 2 above, enables us to complete the theoretical treatment on time optimal software pipelining. However, the algorithm is of little practical importance; it suffers from excessive overhead in computation time and code expansion. In the worst case, the overhead is inherently unavoidable.[28] As a practical alternative, we present a more realistic optimal software pipelining algorithm which runs faster with less code expansion and less hardware requirement. Unlike the former optimal algorithm, this algorithm guarantees optimal solutions when loops satisfy a stronger version of the Time Optimality Condition. According to our experimental observations, however, most loops satisfying the Time Optimality Condition satisfy the stronger version as well, which strongly indicates the practical significance of the proposed realistic software pipelining algorithm.

Using a series of experimental analysis, we also demonstrate that achieving the optimality in the software-pipelined programs is a viable goal in practice with reasonable levels of hardware support. We have performed experiments using the loops of SPEC95 integer benchmark programs.

The rest of the paper is organized as follows. We explain the machine model assumptions, program representation and dependence representation in Section 2. A formal description of software pipelining is presented in Section 3. In Section 4, we present the Time Optimality Condition and describe how to compute it. In Sections 5 and 6, we present two optimal software pipelining algorithms, respectively. Experimental results are given in Section 7 and we conclude with a summary and directions for future work in Section 8.

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*[21]) 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*[9]). The former assumption is needed because if multiple branch operations have to be executed sequentially, time optimal execution cannot be defined. The latter one is also indispensable for time optimal execution, since it enables to avoid output dependence of store operations which belong to different execution paths of a parallel instruction as pointed out by Aiken *et al.*[3]

As a specific example architecture, we use the tree VLIW architecture model,[22] 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. 2. A tree instruction can execute simultaneously ALU and memory operations as well as branch operations. The branch unit of the tree VLIW architecture can decide the branch target in a single cycle.[21] An operation is committed only if it lies in the execution path determined by the branch unit.[9]

2.2. Program Representation

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

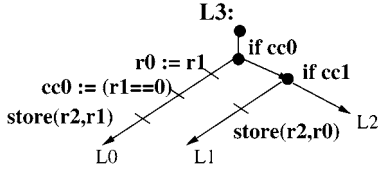


Fig. 2. A tree VLIW instruction.

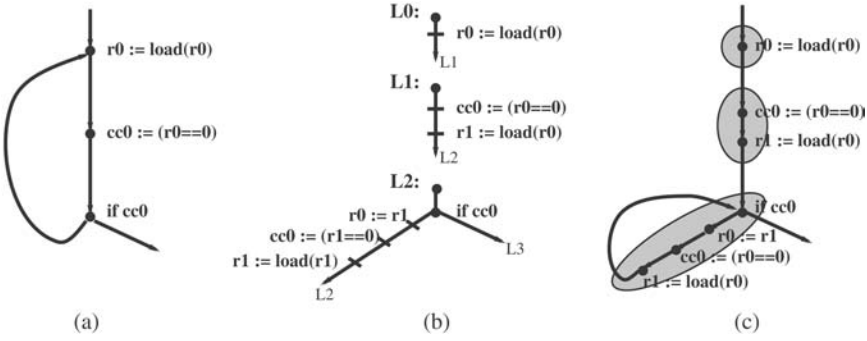


Fig. 3. (a) A sequential program, (b) a parallel tree VLIW program, and (c) a parallel program in the extended sequential representation.

Under the extended sequential representation, both sequential programs and parallel programs are described using the same notations and definitions used for the sequential programs. Compared to sequential programs, parallel programs include the additional information on operation grouping. Figure 3(a) shows an input sequential program P_s and Fig. 3(b) shows its corresponding parallel tree VLIW program P_{tree}. Using the extended sequential representation, P_{tree} is represented by Fig. 3(c). The parallel program shown in Fig. 3(c) is based on a sequential representation except that it has the operation grouping information indicated by shaded regions. A group of operations in the shaded area indicates independently executable operations and is called a *parallel group*. A parallel group corresponds to a tree VLIW instruction and can be easily converted into the tree VLIW instruction with some local transformation on copy operations, and vice versa.[22] An irreducible loop can always be transformed into an reducible one[15] and the input program of software pipelining under consideration is required to be a reducible innermost loop but the corresponding parallel program may consist of several strongly connected components (SCCs) with DAG components.

2.3. Basic Terminology

A program[5] is represented as a triple $\langle G = (N, E), O, \delta \rangle$. (This representation is due to Aiken *et al.*[3]) The body of the program is a CFG G which consists of a set of nodes N and a set of directed edges E. Nodes in

[5] Since a parallel program is represented by the extended sequential representation, the notations and definitions explained in Section 2 apply to parallel programs as well as sequential programs.

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 $op(n)$.[6] 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 $\langle\langle n_0, s_0\rangle,..., \langle n_i, s_i\rangle,..., \langle n_t, s_t\rangle\rangle$ where $\langle n_{i+1}, s_{i+1}\rangle = \delta(\langle n_i, s_i\rangle)$ for $0 \leqslant i < t$.

A *path* p of G is a sequence $\langle n_1,..., n_k\rangle$ of nodes in N such that $(n_i, n_{i+1}) \in E$ for all $1 \leqslant i < k$. For a given path p, the length of p is the number of nodes in p and denoted by $|p|$. The ith $(1 \leqslant i \leqslant |p|)$ 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 $(0 \leqslant j \leqslant |p| - |q|)$ such that $q[i] = p[i+j]$ for all $1 \leqslant i \leqslant |q|$. For a path p and i, j $(1 \leqslant i \leqslant j \leqslant |p|)$, $p[i, j]$ represents the subpath induced by the sequence of nodes from $p[i]$ up to $p[j]$. Given paths $p_1 = \langle n_1, n_2,..., n_k\rangle$ and $p_2 = \langle n_k, n_{k+1},..., n_l\rangle$, $p_1 \circ p_2 = \langle n_1, n_2,..., n_k, n_{k+1},..., n_l\rangle$ 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 $1 \leqslant i \leqslant |p|$.

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

[6] 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 $op(n)$.

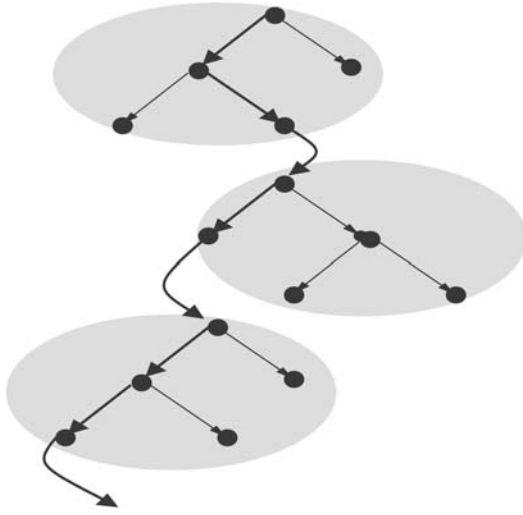


Fig. 4. An execution path in a parallel program.

which the control flows along the execution path. Suppose that a program P is executed with an initial store s_0 and the sequence of configurations is written as $\langle\langle n_0, s_0\rangle, \langle n_1, s_1\rangle, \ldots, \langle n_f, s_f\rangle\rangle$, 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 $\langle n_0, n_1, \ldots, n_f\rangle$ 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 $\forall p^e$ in $\mathscr{P}$, $\exists s$ such that $p^e \equiv ep(\mathscr{P}, s)$.

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 Fig. 4, 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 $\langle \mathbf{n}_1, \mathbf{n}_2, ..., \mathbf{n}_{|p|} \rangle$ of node instances such that $\mathbf{n}_i = \langle p[i], p[1, i], p[i, |p|] \rangle$ for all $1 \leqslant i \leqslant |p|$. The ith 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 ith node instance $\mathbf{n}_i$ of $t(p)$ whose node component is a branch node, a boolean-valued attribute *dir* is defined as follows:

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

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 3.

2.4. 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 Section 2.4.1 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 Section 2.4.2.

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 not the case and the dependence relationship between two nodes relies on the control flow between them as shown in Fig. 5. In Fig. 5(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

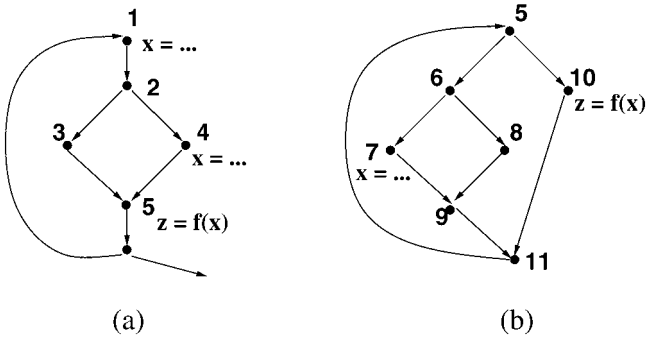


Fig. 5. Path-sensitive dependence examples.

splitted into distinct nodes to be placed in each path. In Fig. 5(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.

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

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

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

Definition 2. Given a path p and $1 \leqslant i < j \leqslant |p|$, $t(p^{\mathrm{e}})[j]$ 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 and is usually represented by *data dependence graphs*[18] or its extensions.[10, 25] In our work, we assume a similar memory dependence relation, in which the dependence relation between two node n_1 and n_2 along p ($p[1] = n_1$, $p[|p|] = n_2$) rely only on the number of iterations that p spans.

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

2.4.2. 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 Ref. 3 and 22 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. 6(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. 6(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.

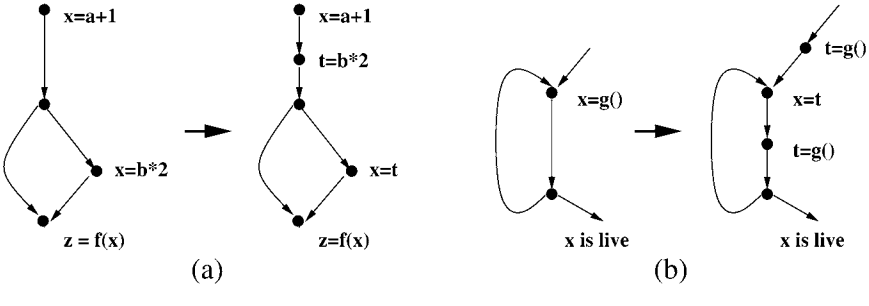


Fig. 6. Copy operations used to overcome false dependences.

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,

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

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

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

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,

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

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 $\langle \mathbf{n}_1, \mathbf{n}_2, ..., \mathbf{n}_k \rangle$ in $t(p)$ such that $\mathbf{n}_i \prec \mathbf{n}_{i+1}$ for all $1 \leqslant i < k$. A dependence chain is said to be critical if it is the longest one in p. The ith 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 $(1 \leqslant i \leqslant j \leqslant |d|)$, $d[i, j]$ represents the sub-chain of d induced by the sequence of node instances from $d[i]$ up to $d[j]$.

3. 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 4. Given an input loop $\mathscr{L}$ and its parallel version $\mathscr{L}^{SP}$, let $\mathbf{P}^e$ and $\mathbf{P}^{e,\,SP}$ denote

the set of all execution paths in $\mathscr{L}$ and the set of those in $\mathscr{L}^{SP}$, respectively. Let us consider a relation $\mathscr{R}$: $\mathbf{P}^{e} \times \mathbf{P}^{e,SP}$ defined by

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

In order to formalize software pipelining, we are to restrict transformations (that map p^{e} into $p^{e,sp}$) 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,[22] 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 $p^{e,sp}$. Therefore $p^{e,sp}$ is required to execute the same operations as p^{e} in an order compatible with the dependences present in p^{e}. The path $p^{e,sp}$, however, may have additional *speculative* nodes[7] from other execution paths that do not affect the final store of $p^{e,sp}$ and copy operations used for overcoming false dependences.[3,22] 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 $t(p^{e})$ and let $\mathbf{N}_2$ represent the set of all effective node instances in $t(p^{e,sp})$. Then, there exists a bijective function f from $\mathbf{N}_1$ to $\mathbf{N}_2$ such that

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

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

Second, the final store[8] of $p^{e,sp}$ should be equal to that of p^{e} to preserve the semantic of $\mathscr{L}$. For this, we require that for any node

[7] In fact, most complications of the nonexistence proof in Section 4 as well as the formalization of software pipelining are due to expanded solution space opened up by branch reordering transformation.

[8] Temporary registers are excluded.

$n = node(\mathbf{n})$, where $\mathbf{n}$ is a node instance in $t(p^e)$, if the target register of n is live at the exit of p^e, the value defined by $node(sp\_ni_{p^e, p^{e,sp}}(\mathbf{n}))$ should be eventually committed to $reg_W(n)$ along $p^{e,sp}$. 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 $p^{e,sp}$ is checked at post-pass dead code elimination optimization phase. Constraint 7 concisely states this condition.

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

$$reg_W(node(\mathbf{n})) = reg_W(node(sp\_ni_{p^e, p^{e,sp}}(\mathbf{n}))) \quad \text{or}$$

$$\exists \mathbf{n}^c \in Prop(sp\_ni_{p^e, p^{e,sp}}(\mathbf{n})), reg_W(node(\mathbf{n})) = reg_W(node(\mathbf{n}^c)).$$

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

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

Now, we are to impose a restriction to preserve the semantic of branches. Let us consider a branch node instance $\mathbf{n} = t(p^e)[i]$ and the corresponding node instance $\mathbf{n}' = t(p^{e,sp})[i'] = sp\_ni_{p^e, p^{e,sp}}(\mathbf{n})$. The role of $\mathbf{n}$ is to separate p^e from the set of execution paths that can be represented by $p^e[1, i] \circ p_f$ where p_f represents any path such that $p_f[1] = p^e[i]$, $p_f[2] \neq p^e[i+1]$ and $p_f[|p_f|]$ is an exit node in $\mathscr{L}$. $\mathbf{n}'$ is required to do the same role as $\mathbf{n}$, that is, it should separate $p^{e,sp}$ from the set of corresponding execution paths. But some of them might already be 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 9. Given an execution path p^e and q^e in $\mathscr{L}$ 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 $(p^e, p^{e,sp}), (q^e, q^{e,sp}) \in \mathscr{R}$, there exists a branch node $p^{e,sp}[j]$ $(j \leqslant i')$ 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 6–9 are all satisfied. (The subscript SA is adapted from the expression "semantically and algorithmically equivalent" in Ref. 26.) Constraint 9 can be used to rule out a pathological case, *unification of execution paths*. Two distinct execution paths $p_1^e = ep(\mathscr{L}, s_1)$ and $p_2^e = ep(\mathscr{L}, s_2)$ in $\mathscr{L}$ are said to be *unified* if $ep(\mathscr{L}^{SP}, s_1) \equiv ep(\mathscr{L}^{SP}, s_2)$. Suppose p_1^e is separated from p_2^e by a branch, then $ep(\mathscr{L}^{SP}, s_1)$ must be separated from $ep(\mathscr{L}^{SP}, s_2)$ by some branch by Constraint 9. So p_1^e and p_2^e cannot be unified.

Let us consider the mapping cardinality of $\mathscr{R}$. 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, $\mathscr{R}$ is a one-to-many relation, and if branch nodes are not allowed to be reordered, $\mathscr{R}$ becomes a one-to-one relation. In addition, the domain and image of $\mathscr{R}$ cover the entire $\mathbf{P}^e$ and $\mathbf{P}^{e,SP}$, respectively. Because of our assumption in Section 2.3 that all the execution paths are executable, $\forall p^e \in \mathbf{P}^e$, $\exists s$, $p^e \equiv ep(\mathscr{L}, s)$ and the domain of $\mathscr{R}$ 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 $\mathscr{R}$ cover the entire $\mathbf{P}^{e,sp}$. To be short, $\mathscr{R}^{-1}$ 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 $\mathscr{L}$ and the set of all effective node instances in all execution paths in $\mathscr{L}^{SP}$, respectively. The following constraint can be derived from the above explanation.

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

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

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

Definition 11. β: $\mathbf{N}^{SP} \Rightarrow \mathbf{N}$ is a surjective function such that

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

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

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

4. TIME OPTIMALITY CONDITION

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

4.1. Time Optimality

For each execution path $p^{e,sp}$ in a software pipelined program $\mathscr{L}^{SP}$, the execution time of each node instance $\mathbf{n}$ in $t(p^{e,sp})$ can be counted from the corresponding parallel control flow graph and is denoted by $\tau(\mathbf{n})$. Time optimality of the parallel program $\mathscr{L}^{SP}$ is defined as follows Ref. 3 and 26.

Definition 12. $\mathscr{L}^{SP}$ is time optimal if, for every execution path $p^{e,sp}$ in $\mathscr{L}^{SP}$, $\tau(t(p^{e,sp})[|p^{e,sp}|])$ is the length of the longest dependence chain in the execution path p^e.

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

4.2. Time Optimality Condition

In Sections 4.3 and 5, we show that a loop $\mathscr{L}$ has an equivalent time optimal program if and only if the following condition is satisfied:

Condition 1 (Time Optimality Condition).

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

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

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

$$|p| \leqslant B_2 \cdot \|p\| + B_3.$$

Informally, the Time Optimality Condition requires that every operation be moved within a *bounded range* to yield the time optimal execution for every execution path. Condition 1(a) states that for any path p in $\mathscr{L}$, if the path p is splitted into two subpaths, the sum of the lengths of the

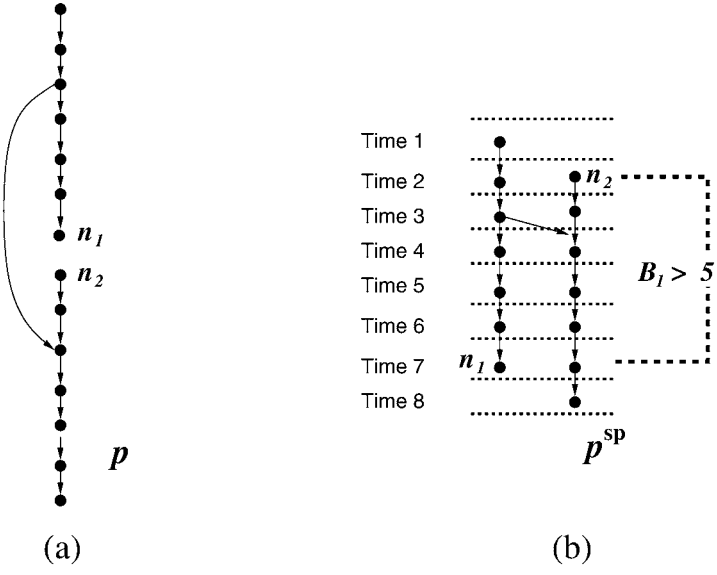


Fig. 7. An example illustrating Condition 1(a).

longest dependence chains in each subpath can exceed the length of the longest dependence chain in p at most by B_1. Figure 7 illustrates Condition 1(a) using an example path p shown in Fig. 7(a) where edges represent true dependences. The path p^{sp} in Fig. 7(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 1(a). Then, we have:

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

Intuitively, the lower bound on B_1 corresponds to the range of the code motion required for the time optimal execution. In Fig. 7, 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 1(b) is rather trivial. It states that for any path p in $\mathscr{L}$, $|p|$ is bounded by a linear function of $\|p\|$. In other words, if $\mathscr{L}$ has an equivalent time optimal program, there exists a fairly long dependence chain for every path p in $\mathscr{L}$.

Let us consider the example loops shown in Fig. 8. These loops were adapted from.[26] The first one (Fig. 8(a)), which was shown to have an

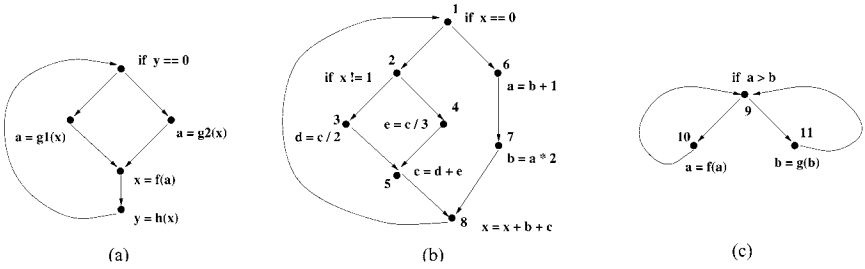


Fig. 8. Example loops used by Schwiegelshohn *et al.*

equivalent time optimal program, satisfies Condition 1. For any execution path p^e that loops k iterations, $\|p^e\| = 2k+1$ and for $1 \leqslant i < j \leqslant |p^e| = 4k$, $\|p^e[1, i]\| \leqslant \lceil i/2 \rceil + 1$ and $\|p^e[j, |p^e|]\| \leqslant \lceil 4k - j/2 \rceil + 2$. So, we have:

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

The second and third loops shown in Figs. 8(b) and 8(c) do not satisfy Condition 1(a), thus having no equivalent time optimal programs as shown in Ref. 26. For the loop in Fig. 8(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 $p^e(k) = c_1^k \circ c_2^k$, we have :

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

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. 8(c) does not satisfy Condition 1 by a similar way.

Theorem 13. Condition 1 is a necessary and sufficient condition for $\mathscr{L}$ to have an equivalent time optimal program.

Section 4.3 gives a proof on the necessary part of Theorem 13. We prove the sufficient part of Theorem 13 by construction, i.e., the proof for the sufficient part follows from the optimal software pipelining algorithm presented in Section 5. Condition 1 is intuitive and useful in deriving the theorems, but it is not obvious how to determine if a loop satisfies Condition 1 or not. If Condition 1 is to be directly computed from the expressions, every execution path should be enumerated, which is impossible. So we present another condition in Section 4.4 which is equivalent to Condition 1 and can be computed more easily.

4.3. Necessary Part of Theorem 13

If a loop $\mathscr{L}$ has an equivalent time optimal program $\mathscr{L}^{SP}$ but it does not satisfy Condition 1, $\mathscr{L}^{SP}$ must exhibit some anomaly. If Condition 1(a) is not satisfied, an operation n_1 in $\mathscr{L}^{SP}$ should be executed infinitely earlier than n_2 that precedes n_1 in $\mathscr{L}$. In case that Condition 1(b) is not satisfied, infinitely many operations should be executed at the same time slot. We show that no *closed-form* parallel program satisfies this anomalous requirement.

Lemma 14. Condition 1(b) is a necessary condition for $\mathscr{L}$ to have an equivalent time optimal program.

Proof. Suppose $\mathscr{L}$ has an equivalent time optimal program $\mathscr{L}^{SP}$. Let B_2 be the maximum height among tree parallel instructions of $\mathscr{L}^{SP}$ 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 1(a). From the fact that $\mathscr{L}^{SP}$ is time optimal and the definition of B_2, $|p'|$ is bounded by $B_2 \cdot \|p'\|$. Therefore, we have

$$|p| \leqslant |p'| \leqslant B_2 \cdot \|p'\| \leqslant B_2 \cdot (\|p\| + 2 \cdot L) = B_2 \cdot \|p\| + B_3. \quad \blacksquare$$

Lemma 15. If there exists a constant $B > 0$ such that for any execution path p^e in $\mathscr{L}$

$$\|p^e[1, i]\| + \|p^e[j, |p^e|]\| \leqslant \|p^e\| + B \qquad \text{for all} \quad 1 \leqslant i < j \leqslant |p^e|, \qquad (1)$$

Condition 1(a) is satisfied.

Proof. In order to prove that (1) implies Condition 1(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 $\mathscr{L}$. Then $p' = p_1 \circ p \circ p_2$ is an execution path of $\mathscr{L}$, and the above inequality holds for p'. Therefore, we have

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

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

Throughout the remainder of this section, we assume that $\mathscr{L}$ does not satisfy (1) and that $\mathscr{L}^{SP}$ 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 proofs are not affected by this transformation.

Lemma 16. For any $l > 0$, there exists an execution path $p^{e,sp}$ in $\mathscr{L}^{SP}$ and dependence chains of length l in $p^{e,sp}$, 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 $1 \leqslant j, k \leqslant l$.

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

$$\begin{aligned} pos(\beta(d_1[j])) &= pos(d_1'[j - l + \|p^e[1, i_1]\|]) \leqslant i_1 < i_2 \leqslant pos(d_2'[k]) \\ &= pos(\beta(d_2[k])). \end{aligned}$$

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

$$\tau(d_1[j]) \geqslant |d_1'[1, j - l + \|p^e[1, i_1]\| - 1]| = j - l + \|p^e[1, i_1]\| - 1$$

$$\tau(d_2[k]) \leqslant \|p^e\| - |d_2'[k, \|p^e[i_2, |p^e|]\|]| + 1 = \|p^e\| - \|p^e[i_2, |p^e|]\| + k.$$

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

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

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 that carries $elr(\mathbf{n})$ at $\mathbf{n}'$.

Lemma 17. Given k assignment node instances $\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_k$ in an execution path in $\mathscr{L}^{SP}$ and a node instance $\mathbf{n}$ in the execution path, if $\mathbf{n}$ is contained in $elr(\mathbf{n}_i)$ for all $1 \leqslant i \leqslant k$, $reg(elr(\mathbf{n}_1), \mathbf{n})$, $reg(elr(\mathbf{n}_2), \mathbf{n}), \ldots$, $reg(elr(\mathbf{n}_k), \mathbf{n})$ 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 $k = h \geqslant 1$. Consider $h+1$ assignment node instances $\mathbf{n}'_1, \mathbf{n}'_2, \ldots, \mathbf{n}'_{h+1}$ 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 $1 \leqslant i \leqslant h$. 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, $reg(elr(\mathbf{n}'_1), \mathbf{n}'_{h+1}), \ldots, reg(elr(\mathbf{n}'_h), \mathbf{n}'_{h+1}))$ 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 that carries part of $elr(\mathbf{n}'_{h+1})$ differs from h distinct registers that 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 span 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.[19] 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.[22] 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 larger number 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 $\mathscr{L}^{SP}$ 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 $\mathscr{L}$.

Proof. Let $p^e = \alpha(p^{e,sp})$ and $M' = \lfloor |d|/(M+1) \rfloor$. From the definition of M, there must exist $pos(\beta(d[1])) \leqslant i_1 < i_2 < \cdots < i_{M'} \leqslant pos(\beta(d[|d|]))$ 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, $\mathbf{n}'$ $(i \leqslant pos(\mathbf{n}') < j)$ such that $\forall k > pos(n)$, $reg_W(p^e[k]) \neq reg_W(node(\mathbf{n}'))$. Thus by Constraint 7, there must exist node instances in d, $\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_{M'-1}$, such that

$$reg_W(node(\beta(\mathbf{n}_i))) = reg_W(node(\mathbf{n}_i)) \quad \text{or}$$

$$\exists \mathbf{n}^c \in Prop(\mathbf{n}_i),\ reg_W(node(\beta(\mathbf{n}_i))) = reg_W(node(\mathbf{n}^c)) \quad \text{for all } 1 \leqslant i \leqslant M'-1.$$

Since $pos(\mathbf{n}_i) < pos(\mathbf{n}_b)$ and $pos(\beta(\mathbf{n}_i)) > pos(\beta(\mathbf{n}_b))$, $node(\mathbf{n}_i)$ is speculative for all $1 \leqslant i \leqslant M'-1$. By Constraint 8, $reg_W(node(\mathbf{n}_i)) \notin \mathbf{R}$ 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 $\mathbf{n}_b$ for all $1 \leqslant i \leqslant M'-1$. ∎

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

$$\mathbf{N}_b(l) = \{\mathbf{n}_b \mid pos(\beta(\mathbf{n}_b)) \leqslant i_1(l) \wedge pos(\mathbf{n}_b) > pos(d_2(l)[1])\}.$$

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

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

Proof. Let $C = (M+1)(R+2)$ where M is defined as in Lemma 18 and R denotes the number of registers used in $\mathscr{L}^{SP}$. Suppose $\tau(\mathbf{n}_b) \geqslant \|p^e(l)[1, i_1(l)]\| - 2 \cdot l + C$. From the proof of Lemma 16, we have

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

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

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

Theorem 20. Condition 1 is a necessary condition for $\mathscr{L}$ to have an equivalent time optimal program.

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

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

where $\mathbf{n}'_b$ represents any branch node instance in $\tau(\mathbf{n}'_b)$ such that $pos(\beta(\mathbf{n}'_b)) \leqslant pos(\beta(d'_1(l)[l]))$. Let $P(\mathbf{n}_b)$ be the set of execution paths in $\mathscr{L}^{SP}$ such that

$$P(\mathbf{n}_b) = \{q^{e,sp} \mid q^{e,sp}[1, pos(\mathbf{n}_b)] \equiv p^{e,sp}(l)[1, pos(\mathbf{n}_b)] \\ \wedge dir(t(q^{e,sp})[pos(\mathbf{n}_b)]) \neq dir(t(p^{e,sp}(l))[pos(\mathbf{n}_b)])\}.$$

Then, $\|q^{e,sp}\| \geqslant \|p^e(l)[1, i_1(l)]\|$ and, by Lemma 17, we have

$$\|q^{e,sp}[pos(\mathbf{n}_b) + 1, \|q^{e,sp}\|]\| > l - C.$$

Since l is not bounded and C is bounded, the length of any path starting from $node(\mathbf{n}_b)$ is not bounded, a contradiction. Therefore the assumption that $\mathscr{L}^{SP}$ is time optimal is not valid and, by Lemma 14, Condition 1 is indeed a necessary condition.

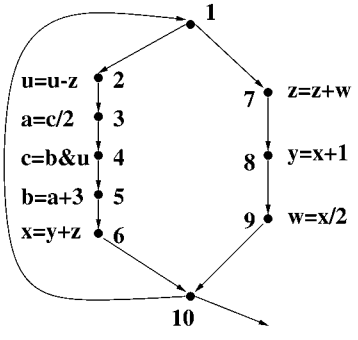
4.4. Computing Time Optimality Condition

In this section, we explain how to compute the Time Optimality Condition. Directly computing the Time Optimality Condition requires that 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 $\mathscr{L}$ as follows.

Definition 21. Given a cycle c (which may not be simple) in $\mathscr{L}$, d is a dependence cycle with respect to c if there exist $l \geqslant 1$ and $1 \leqslant i_1 < i_2 < \cdots < i_{|d|} \leqslant l \cdot (|c| - 1)$ such that

$$i_1 \leqslant |c| - 1 \wedge i_{|d|} = i_1 + (l-1) \cdot (|c| - 1) \qquad \text{and}$$
$$d[j] = c^l[i_j] \qquad \text{for} \quad 1 \leqslant j \leqslant |d| \qquad \text{and}$$
$$d[j] \prec_{c^l[i_j, i_{j+1}]} d[j+1] \qquad \text{for} \quad 1 \leqslant j < |d|.$$



$c_1 = <1, 2, 3, 4, 5, 6, 10, 1>$
$c_2 = <1, 7, 8, 9, 10, 1>$
$c_3 = c_1 \circ c_2$
$c_4 = c_1 \circ c_2^2$

$d_1 = <2, 2>$
$d_2 = <3, 5, 4, 3>$
$d_3 = <7, 7>$
$d_4 = <6, 8, 6>$
$d_5 = <6, 9, 7, 6>$

$DC(c_1) = \{d_1, d_2\}$
$DC(c_2) = \{d_3\}$
$DC(c_3) = \{d_4\}$
$DC(c_4) = \{d_5\}$

Fig. 9. Dependence cycles.

Figure 9 shows an example of dependence cycles. We associate several attributes with the dependence cycle, which are defined below.

Definition 22. For a dependence cycle d, the sum of latencies of $d[1], d[2],..., d[|d|-1]$ is denoted by $\delta(d)$. $span(d)$ denotes l in Definition 21 and $slope(d)$ is defined to be $\delta(d) = span(d)$. Furthermore, $DC(c)$ represents the set of dependence cycles associated with c and $DC_{cr}(c)$ represents the subset of $DC(c)$ that consists of all the dependence cycles with the maximum slope in $DC(c)$. A dependence cycle in $DC_{cr}(c)$ is called a critical dependence cycle and its slope value is denoted by max_slope(c).

There are a finite number of simple dependence cycles in $DC_{cr}(c)$ as well as in $DC(c)$ and these dependence cycles can be enumerated using Johnson's algorithm.[16] 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 23. Given two cycles c_1 and c_2 in $\mathscr{L}$ 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

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

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 $d_1 \bowtie d_2$.[9]

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

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

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

$$\mathbf{C}^* = \bigcup_{k=1}^{|\mathbf{C}|} \mathbf{C}^k.$$

Then, the following condition is equivalent to Condition 1.

Condition 2.

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

(b) For each cycle c_i $(1 \leqslant i \leqslant |\mathbf{C}^*|)$ in $\mathbf{C}^*$, there exists a dependence cycle

$$d_i \in DC_{\text{cr}}(c_i) \text{ such that } d_j \prec^C d_k \qquad \text{for all} \quad 1 \leqslant j < k \leqslant |\mathbf{C}^*|.$$

It is possible to check if a loop satisfies the Condition 2 or not in a finite number of steps because only finite number of cycles need to be enumerated.

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

Lemma 24. If a loop $\mathscr{L}$ satisfies Condition 1, it also satisfies Condition 2.

Proof. Condition 2(a) is obviously satisfied. Suppose that Condition 2(b) is not satisfied for some c_1 and c_2. For $d_1 \in DC_{\text{cr}}(c_1)$, select $d_2 \in DC_{\text{cr}}(c_2)$ and $d_3 \in DC(c_1)$ such that $d_3 \prec^C d_2$ and $slope(d_3)$ is maximum. Note that $d_2 \in DC_{\text{cr}}(c_2)$ 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 $p(i) = c_1^{ai} \circ c_2^{abi} \circ p_{\text{f}}$ where p_{f} denotes any simple path from the unique loop header node to one of the exits 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 written as

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

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

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

for some constant α.

Case 1: $d_5 \notin DC_{cr}(c_2)$. We have $slope(d_5) \leqslant r$ and $\|p(i)\| \leqslant slope(d_1)\cdot ai + r\cdot abi + \alpha_2$. From

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

we have

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

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

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

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

$$\|p_1(i)\| \geqslant slope(d_1)\cdot ai \qquad \text{and} \qquad \|p_2(i)\| \geqslant slope(d_2)\cdot abi$$

where $p_1(i) \stackrel{\text{def}}{=} p(i)[1, (|c|-1)\cdot ai]$ and $p_2(i) \stackrel{\text{def}}{=} p(i)[(|c|-1)\cdot ai+1, \|p(i)\|]$. Thus,

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

which implies that Condition 1 is not satisfied, a contradiction. ▮

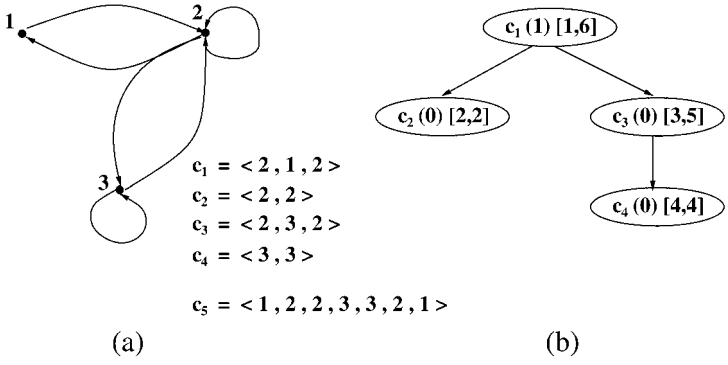


Fig. 10. A new representation for a cycle: (a) A graph with cycles and (b) a tree representation of c_5.

Before showing that the inverse proposition also holds, we introduce a new representation for cycles. As will be shown in Lemma 25, it is useful to represent a cycle by a composition of given subcycles. For example, consider a cycle c_5 shown in Fig. 10(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 Fig. 10(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 with the sequence shifted such that $c(j)[i] = c[(i+j-1 \bmod |c|)+1]$ for $1 \leqslant i \leqslant |c|$.

Lemma 25. For any cycle c in $\mathscr{L}$ 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 2(b), d can be written as $d_j \circ d_k$, $(d_j \in DC_{cr}(c_j))$ 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, l', and c'. 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 $\rho_{i_1, i_2, \ldots, i_{|\mathbf{C}|}}$ $(1 \leqslant i_1, i_2, \ldots, i_{|\mathbf{C}|} \leqslant |\mathbf{C}|)$, we solve the following linear system of $|\mathbf{C}^*|$ equations in the $|\mathbf{C}|^{|\mathbf{C}|}$ unknowns.

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

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

By using a simple argument based on linear algebraic theorems, we can easily show that the linear system has a solution such that every $\rho_{i_1, i_2,\ldots,i_{|\mathbf{C}|}}$ is positive. (Actually, the solution is not unique and we select any one of them.) Given $\rho_{i_1, i_2,\ldots,i_{|\mathbf{C}|}}$, we can characterize the lengths of critical dependence chains. Let M_1 denote the length of the longest dependence chain in cycles $c_{i_1 \circ c_{i2}} \circ \cdots \circ c_{i_{|\mathbf{C}|-1}}$ $(1 \leqslant i_1, i_2,\ldots, i_{|\mathbf{C}|} \leqslant |\mathbf{C}|)$ and let M_2 denote the length of the longest dependence chain in simple paths in $\mathscr{L}$.

Lemma 26. Given a path $p = p_s \circ c_{i_1} \circ c_{i_2} \circ \cdots \circ c_{i_k} \circ p_f$ in $\mathscr{L}$ where $k \geqslant |\mathbf{C}|$, $c_{i_j} \in |\mathbf{C}|$ for all $1 \leqslant j \leqslant k$ and both p_s and p_f are simple paths, let M_3 be

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

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

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 25. Therefore, we have

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

Similarly,

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

From Lemma 26, we can compute the constants B_1, B_2, and B_3 in Condition 1.

Lemma 27. If B_1 is selected as $2 \cdot M_1 + 4 \cdot M_2$, Condition 1(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 $\mathscr{L}$ we split p into two subpaths p_1 and p_2. Then, p_1 and p_2 can be written as

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

By Lemma 26, we have

$$\begin{aligned}\|p_1\|+\|p_2\|-\|p\| &\leqslant \left(M_1+2\cdot M_2+\sum_{h=0}^{l-|\mathbf{C}|}\rho_{i_{1+h},\,i_{2+h},\ldots,\,i_{|\mathbf{C}|+h}}\right)\\ &\quad+\left(M_1+2\cdot M_2+\sum_{h=l+1}^{k-|\mathbf{C}|}\rho_{i_{1+h},\,i_{2+h},\ldots,\,i_{|\mathbf{C}|+h}}\right)\\ &\quad-\left(\sum_{h=0}^{k-|\mathbf{C}|}\rho_{i_{1+h},\,i_{2+h},\ldots,\,i_{|\mathbf{C}|+h}}\right)\leqslant 2\cdot M_1+4\cdot M_2. \quad \blacksquare\end{aligned}$$

Lemma 28. If B_2 and B_3 are selected as

$$\begin{aligned}B_2 &= \max\{|c|/\rho_{i_1,\,i_2,\ldots,\,i_{|\mathbf{C}|}} \mid c\in\mathbf{C}^*,\ 1\leqslant i_1,\,i_2,\ldots,\,i_{|\mathbf{C}|}\leqslant|\mathbf{C}|\}\\ B_3 &= 2\cdot L_C\end{aligned}$$

where L_C is the length of the longest simple cycle in $\mathscr{L}$, Condition 1(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 $\mathscr{L}$,

$$\begin{aligned}|p| &\leqslant \sum_{h=1}^{k}(|c_{i_h}|-1)+2\cdot L_C\\ &\leqslant \sum_{h=0}^{k-|\mathbf{C}|}\left(\frac{|c_{i_h}|}{\rho_{i_{1+h},\,i_{2+h},\ldots,\,i_{|\mathbf{C}|+h}}}\cdot\rho_{i_{1+h},\,i_{2+h},\ldots,\,i_{|\mathbf{C}|+h}}\right)+B_3-k\\ &\leqslant B_2\cdot\sum_{h=0}^{k-|\mathbf{C}|}\rho_{i_{1+h},\,i_{2+h},\ldots,\,i_{|\mathbf{C}|+h}}+B_3\\ &\leqslant B_2\cdot\|p\|+B_3. \qquad \text{(By Lemma 26.)} \quad \blacksquare\end{aligned}$$

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

Lemma 29. If a loop $\mathscr{L}$ satisfies Condition 2, it also satisfies Condition 1.

Proof. Directly from Lemmas 27 and 28. ∎

Theorem 30. Condition 1 is decidable.

Proof. From Lemmas 24 and 29, Condition 1 is equivalent to Condition 2, whose decision procedure is obvious from the given expression. ∎

5. TIME OPTIMAL SOFTWARE PIPELINING ALGORITHM

In this section, we present a software pipelining algorithm that computes a time optimal parallel program for loops satisfying Condition 1. (The result in this section also serves as the proof for the sufficient part of Theorem 13.) The time-optimal software pipelining algorithm is mostly based on the algorithm by Aiken *et al.*,[3] the latest version of *Perfect Pipelining*.[2] We first present the software pipelining algorithm by explaining our modifications to Aiken's algorithm. Then, we prove that the output of the algorithm is a time optimal parallel program if the input loop satisfies Condition 1.

5.1. The Algorithm

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

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.[23] During scheduling, the algorithm finds equivalent nodes n and n' in the infinite CFG, deletes the infinite sub-graph below n', and adds backedges from the predecessors of n' to n. In this way, the infinite CFG eventually becomes a finite parallel graph.

Aiken's original algorithm does not handle false dependences appropriately.[3] An operation node which is blocked by the false dependences but not by true dependences may not be available for scheduling. To compute a time optimal solution, the false dependences should be overcome so that the parallel schedule is constrained by the true dependences only. We modify Aiken's original algorithm such that the infinite CFG is put into the static single assignment (SSA) form Ref. 8, 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 variable is 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. In Fig. 11, `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

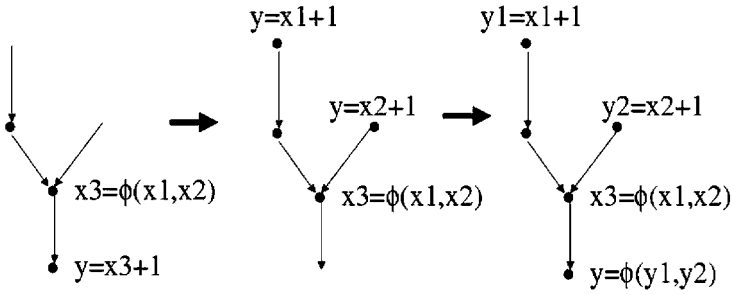


Fig. 11. Scheduling above a ϕ-function at the join point.

`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 Fig. 11, two `y` definitions are replaced by the `y1` and `y2` definitions and a new ϕ-function, `y=`ϕ`(y1, y2)`, is added.

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

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

The proposed time-optimal software pipelining algorithm begins with an acyclic infinite CFG $\mathscr{L}^{\infty}$, and successively transforms $\mathscr{L}^{\infty}$ into $\mathscr{L}^{\mathrm{SP}}$ which consists of parallel groups. Figure 12 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 to build parallel groups for all the branches of that group. 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 the

```
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')
              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 procedure
```

Fig. 12. The time-optimal software pipelining algorithm.

true dependences.[10] In our algorithm, every operation node that is not blocked by the true dependences is always available for scheduling. As in Ref. 3, we impose additional constraint on available operations: operations are available at most k iterations. The predetermined constant k is called a *sliding window*[3] and it guarantees the termination of the **while** loop in the SOFTWARE_PIPELINE procedure.

Once the available operation nodes are computed, the SCHEDULE_PARALLEL_GROUP procedure repeatedly moves the operation nodes into a group boundary.[22] [11] 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).

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 31. Let $\mathscr{L}^{SP}$ 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 $p^{e,sp}$ in $\mathscr{L}^{SP}$ such that $\tau(n) > 1$, there must exist an effective node instance $\mathbf{n}'$ in $p^{e,sp}$ 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 $\min\{it(\mathbf{n}'') \mid \mathbf{n}'' \in \Omega\}$ 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. ∎

[10] This procedure is functionally equivalent to the same procedure in Moon's algorithm.[22]

[11] Since the transformations in the SCHEDULE_PARALLEL_GROUP procedure can be implemented using transformations described in Moon's algorithm whose correctness has been already proved,[22] they preserve program semantics.

5.2. Time Optimality of the Algorithm

The software pipelining algorithm described in Fig. 12 always generates time optimal parallel programs for loops that satisfy Condition 1. The proof is based on the greediness of the algorithm. Before presenting the time optimality proof, we prove some miscellaneous properties stated in Lemmas 32 and 33. (Recall that we have assumed that $\mathscr{L}$ satisfies Condition 1 and that every operation takes 1 cycle to execute.)

Lemma 32. For a path p in $\mathscr{L}$ and $1 = i_1 < i_2 < \cdots < i_l \leqslant |p|$,

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

Proof.

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

Lemma 33. For node instances $\mathbf{n}_1$ and $\mathbf{n}_2$ in a path p in $\mathscr{L}$ such that $it(\mathbf{n}_2) - it(\mathbf{n}_1) > k$,

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

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

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 $(L-1)\cdot k$. Thus, from Condition 1(b), we have

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

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

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

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

Lemma 34. Let $\mathscr{L}^{SP}$ be the result of the software pipelining algorithm with the sliding window of WS iterations. Then $\mathscr{L}^{SP}$ is time optimal.

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

$$E'_D = \{(\mathbf{n}_1, \mathbf{n}_2) \mid \mathbf{n}_1 \prec \mathbf{n}_2\} \qquad \text{and} \qquad E''_D = \{(\mathbf{n}_1, \mathbf{n}_2) \mid it(\mathbf{n}_2) - it(\mathbf{n}_1) > WS\}.$$

We first show that the length of the longest path in G_D is equal to the length of the longest path in $G'_D(N_D, E'_D)$, the subgraph of G_D induced by E'_D. Suppose that there exists a path $\mathbf{p}_D = \mathbf{n}_1 \to \mathbf{n}_2 \to \cdots \to \mathbf{n}_h$ 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 $s(\geq 1)$ edges $(\mathbf{n}_{i_1}, \mathbf{n}_{i_1+1}),\ldots,(\mathbf{n}_{i_s}, \mathbf{n}_{i_s+1})$ $(i_1 < i_2 < \cdots < i_s)$ in $\mathbf{p}_D$ that come from E''_D. So, we have

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

From Lemma 32, we have

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

From (3) and (4), we have

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

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 33, we have for all $1 \leqslant i \leqslant s$

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

which contradicts (5). So the assumption is false and the length of the longest path in $\mathrm{G_D}$ 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 $1 \leqslant i \leqslant \|p^{\mathrm{e,\,sp}}\|$, we are to show that

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

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 $1 \leqslant i \leqslant m$. For the induction step, assume that the proposition holds for all $1 \leqslant j < i$. By Lemma 31, there must exist $i' < i$ such that

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

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

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

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

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

Therefore, we have

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

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

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

From Lemma 34, we can state the following theorem.

Theorem 35. Condition 1 is a sufficient condition for $\mathscr{L}$ to have an equivalent time optimal program.

From Lemma 34, the algorithm in Fig. 12 is a time-optimal software pipelining algorithm, provided that the size of sliding window is computable. From Lemmas 27 and 28, 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 (2). So, we have the following theorem.

Theorem 36. There exists a software pipelining algorithm that computes time optimal programs for loops that satisfy Condition 1.

6. A PRACTICAL SOFTWARE PIPELINING ALGORITHM

In this section, we present a more practical software pipelining algorithm. The software pipelining algorithm uses an intermediate program representation called non-deterministic control flow graph (NCFG)[12] proposed by Milicev.[20] As shown in Fig. 13, the original control flow graph (CFG) of a loop (Fig. 13(a)) is transformed into an NCFG (Fig. 13(b)) and the software pipelining algorithm is applied to the NCFG. Then, the software pipelined NCFG (Fig. 13(c)) is transformed back into an equivalent CFG (Fig. 13(d)). In Section 6.2, we present a software pipelining algorithm that computes a time optimal NCFG for every loop satisfying

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

a new condition, which is a stronger version of the Time Optimality Condition. Before describing the software pipelining algorithm, we first explain the NCFG.

6.1. Nondeterministic Control Flow Graph

The NCFG can be understood as a nondeterministic version of the standard control flow graph (CFG).[13] There is a one-to-one correspondence between NCFGs and CFGs, as is the case with nondeterministic finite automata (NFA) and deterministic finite automata (DFA). Given a CFG G of a loop (before software pipelining), let $P^s = \{p_1^s, p_2^s, \ldots\}$ represent the set of all the acyclic paths starting from the loop header to a predecessor of the loop header or a loop exit. Then the corresponding NCFG G^{NCFG} is simply defined as follows:

$$N^{\text{NCFG}} = \{n_{i,j} \mid 1 \leqslant i \leqslant |P^s|, 1 \leqslant j \leqslant l_i\}$$

$$E^{\text{NCFG}} = \{(n_{i,j}, n_{i,j+1}) \mid 1 \leqslant i \leqslant |P^s|, 1 \leqslant j < l_i\} \cup \{(n_{i,l_i}, n_{i',1}) \mid 1 \leqslant i, i' < |P^s|\},$$

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

The CFG in Fig. 13(a) has two acyclic paths from the loop header to its predecessor and they correspond to two NBBs of the NCFG in Fig. 13(b). Informally, if a node is contained in more than one path of the CFG, it is copied into the corresponding NBBs of the NCFG.

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

$$N_k^{\text{NBB}} = \{\mathbf{b}_i^m \mid 1 \leqslant i \leqslant |N^{\text{NBB}}|, 1 \leqslant m \leqslant |N^{\text{NBB}}|^k\}$$

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

Figures 14(a) and 14(b) show an NCFG G^{NCFG} and its 1-level split version, G_1^{NCFG}.

[13] We apply the notations and definitions explained in Sections 2 and 4.4 to the NCFG as well.

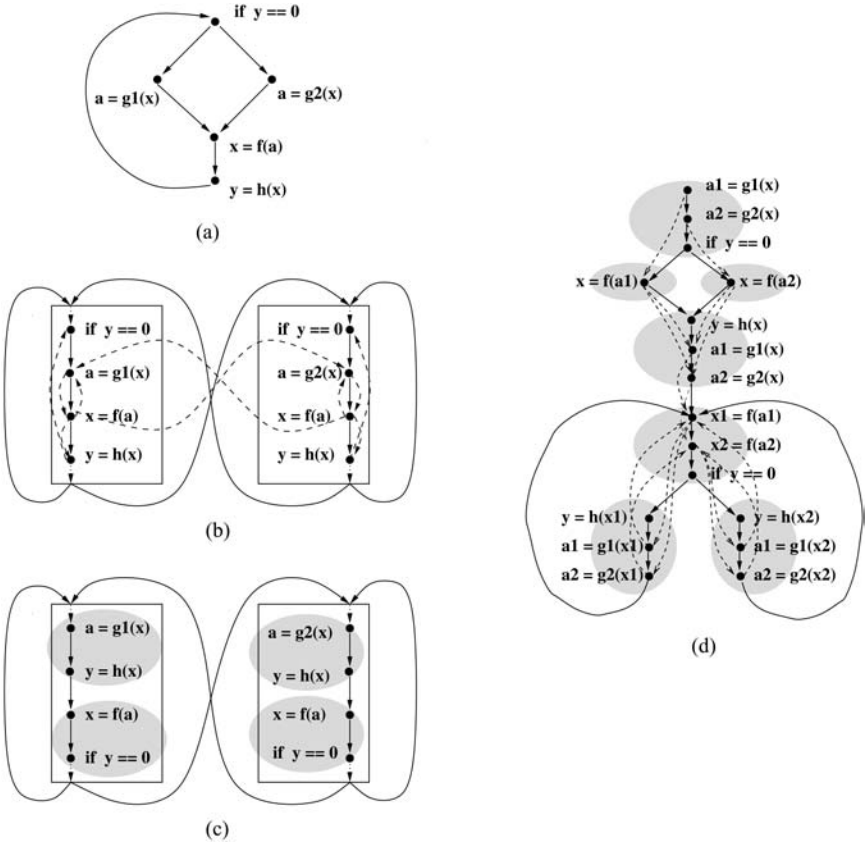


Fig. 13. (a) A CFG before scheduling, (b) its corresponding NCFG, (c) the software-pipelined NCFG, and (d) the (time-optimally) software-pipelined CFG. (Solid lines and dashed lines represent control flows and dependences, respectively. Each shaded region represents a parallel group.)

A software-pipelined NCFG is transformed back into an executable CFG,[20] which is similar to the NFA-to-DFA transformation. A nice property of an NCFG is that the execution time of any path in the NCFG is equal to that in the corresponding CFG. Therefore, it suffices to build a time-optimally software pipelined NCFG.

6.2. The Software Pipelining Algorithm

As with several software pipelining algorithms based on modulo scheduling, our software pipelining algorithm decouples the computation

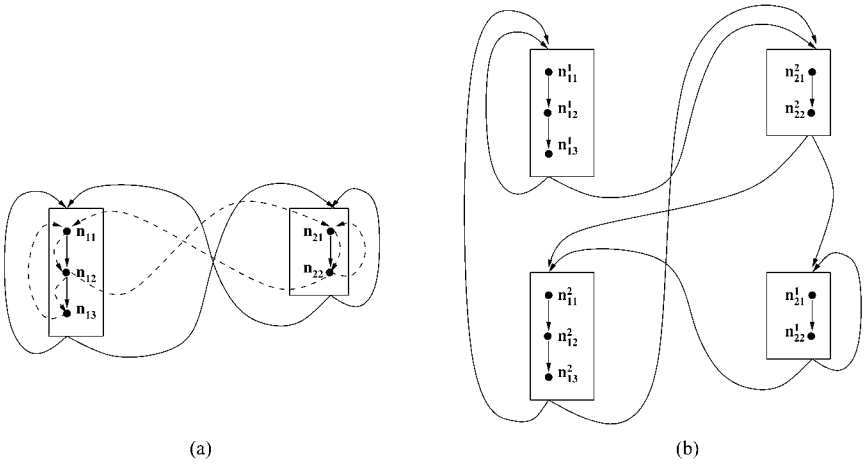


Fig. 14. An example of split transformation.

of a schedule and code motions. After computing a schedule, the code motions that are implicit in the schedule are ascertained subsequently.

The scheduling algorithm first determines the *latency* of each NBB of the NCFG based on linear algebraic theorems and then computes each operation's *time offset* from the beginning of its NBB. Informally, the latency of an NBB can be understood as the initiation interval (II). For example, the latency of the left NBB of the NCFG in Fig. 13(b) is the II of the left path of the CFG in Fig. 13(a). We denote the latency of an NBB **b** by $\rho(\mathbf{b})$ and the time offset of $n \in N^{\mathrm{NCFG}}$ by $\sigma(n)$. Given an execution path p and a set of the latencies and the time offsets, the execution time of $p[k]$ is given by

$$\tau(t(p)[k]) = \sum_{l=1}^{|\mathbf{bp}(p[1,k])|-1} \rho(\mathbf{bp}(p)[l]) + \sigma(p[k]) \tag{8}$$

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

The latencies of NBBs are determined such that, for any simple cycle **c** in the NBB-graph, the sum of the latencies of NBBs in **c** is equal to the slope of the critical dependence cycle in the corresponding cycle c in the

NCFG, i.e., $\sum_{i=1}^{|c|-1} \rho(\mathbf{c}[i]) = max\_slope(c)$. We call such a tuple of latencies as a *tight* tuple. However, a tight tuple does not always exist because the number of equations may be larger than the number of variables (the unknown latencies). This can be resolved by the split transformation, which increases the number of NBBs (equivalently, the variables).

The split transformation also increases the number of simple cycles in the NBB-graph incurring additional equations. But, some of the newly introduced equations may be linearly dependent on other equations and, consequently, the number of variables may exceed the number of equations. In Fig. 14(a), there are three simple cycles in the NCFG but only two nodes in NCFG. Therefore, no solution exists for the linear equations

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

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

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

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

Condition 3.

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

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

Given an NCFG that satisfies Condition 3, the optimal software-pipelined schedule can be computed by the algorithm in Fig. 15.

Lemma 37. The schedule computed by COMPUTE_SCHEDULE meets dependence constraints.

Proof. We would like to show that

$$\begin{aligned} &\forall p\ \forall k, k'(k < k') \text{ s.t. } p[k] \prec_{p[k,k']} p[k'], \\ &\tau(t(p)[k]) + \delta(p[k]) \leqslant \tau(t(p)[k']). \end{aligned} \tag{9}$$

By virtue of the longest path inequalities, we have

$$\sigma(p[k]) + r((p[k], p[k'])) \leqslant \sigma(p[k']), \qquad \text{which implies}$$
$$\sigma(p[k]) + \delta(p[k]) - d((p[k], p[k'])) \cdot \rho(\mathbf{b}(p[k])) \leqslant \sigma(p[k']). \tag{10}$$

---

procedure COMPUTE_SCHEDULE

if (Condition III is not satisfied)

return SCHEDULE_NOT_FOUND

end if

compute a tight tuple $\langle \rho(\mathbf{b}_1), \rho(\mathbf{b}_2), \ldots \rangle$

$N^{\mathrm{DG}} := N^{\mathrm{NCFG}} \cup \{n_{\mathrm{DUMMY}}\}$

$E^{\mathrm{DG}} := \{ (n_{ij}, n_{i'j'}) \mid n_{ij} \prec_p n_{i'j'}$ where p is the shortest path from n_{ij} to $n_{i'j'} \}$

$E^{\mathrm{DG}} := E_{DG} \cup (\{n_{\mathrm{DUMMY}}\} \times (N_{DG} - \{n_{\mathrm{DUMMY}}\}))$

foreach $(e = (n_{ij}, n_{i'j'}) \in E^{\mathrm{DG}})$

$r(e) := \delta(n_{ij}) - d(e) \cdot \rho(\mathbf{b}_i)$ where $\delta(n_{ij})$ is the latency of the operation of n_{ij} and $d(e) = 0$ if $i = i'$, 1 otherwise

end foreach

foreach $(e = (n_{\mathrm{DUMMY}}, n_{ij}) \in E^{\mathrm{DG}})$

$r(e) := 0$

end foreach

foreach $(n_{ij} \in N^{\mathrm{DG}} - \{n_{\mathrm{DUMMY}}\})$

$\sigma(n_{ij}) :=$ the length of the longest path in $G^{\mathrm{DG}} = (N^{\mathrm{DG}}, E^{\mathrm{DG}})$ where the length of a path p is the sum of weight $r(e)$ of edges in p

end foreach

end procedure

Fig. 15. The algorithm to compute a software-pipelined schedule.

Therefore, we have

$$\begin{aligned}
&\tau(t(p)[k]) + \delta(p[k]) - \tau(t(p)[k']) \\
&\quad = \sum_{l=1}^{|\mathbf{bp}(p[1,k])|-1} \rho(\mathbf{bp}(p)[l]) + \sigma(p[k]) + \delta(p[k]) \\
&\qquad - \sum_{l=1}^{|\mathbf{bp}(p[1,k'])|-1} \rho(\mathbf{bp}(p)[l]) - \sigma(p[k']) \\
&\quad = - \sum_{l=|\mathbf{bp}(p[1,k])|}^{|\mathbf{bp}(p[1,k'])|-1} \rho(\mathbf{bp}(p)[l]) + \sigma(p[k]) - \sigma(p[k']) + \delta(p[k]) \\
&\quad \leqslant - \sum_{l=|\mathbf{bp}(p[1,k])|}^{|\mathbf{bp}(p[1,k'])|-1} \rho(\mathbf{bp}(p)[l]) + d((p[k], p[k'])) \cdot \rho(\mathbf{b}(p[k])). \quad (11)
\end{aligned}$$

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

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

Otherwise, we have

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

So, the schedule meets dependence constraints. ▌

Given a schedule, operation nodes are moved by the algorithm in Fig. 16. The procedure MOVE_CODE first initializes each NBBs and invokes the MOVE_OP procedure for each operation nodes. The procedure MOVE_OP places each operation node such that the execution time of each operation instance becomes Eq. (8). From the definition of the tight tuple of latencies of NBBs, it can be easily seen that the software-pipelined NCFG is time-optimal.

7. EXPERIMENTAL RESULTS

In order to evaluate how practical the proposed software pipelining algorithms are, we have performed several experiments using a SPARC-based VLIW testbed.[24] We used 1317 innermost loops (with control flows) extracted from SPEC95 integer benchmark programs. We considered loops

```
procedure MOVE_CODE
    foreach ( b_i )  b_i := ⟨ ⟩  end foreach
    foreach ( n_ij )
        MOVE_OP( n_ij, b_i, σ(n_ij) )
    end foreach
end procedure

procedure MOVE_OP(n, b, σ)
    if (0 ≤ σ < ρ( b ))
        places n on the time-slot σ of b
    else if (σ < 0) /* move upward */
        foreach ( (b', b) ∈ E^NBB )
            MOVE_OP ( n, b', σ + ρ( b' ) )
        end foreach
    else /* move downward */
        foreach ( (b, b') ∈ E^NBB )
            MOVE_OP ( n, b', σ − ρ( b' ) )
        end foreach
    end if
end procedure
```

Fig. 16. The algorithm to move operations in NCFG.

with up to 64 operations. We assumed that load operations take three cycles while all the other operations take one cycle.

Figure 17(a) explains an overview of experimental scenario. In the first experiment (i.e., E1 in Fig. 17(a)), we measured how many loops satisfy Condition 2 (i.e., the Time Optimality Condition). Because the computation of Condition 2 may require excessive time,[14] we set the upper bound T_{th} on computing Condition 2. If the computation takes longer than T_{th}, the computation gives up, assuming that a loop does not satisfy Condition 2. When T_{th} was set to be 30 seconds, we could not determine Condition 2

[14] The problem of determining if Condition 2, i.e., the Time Optimality Condition, is satisfied or not can be easily proved to be NP-hard by reducing from the 3-satisfiability problem.

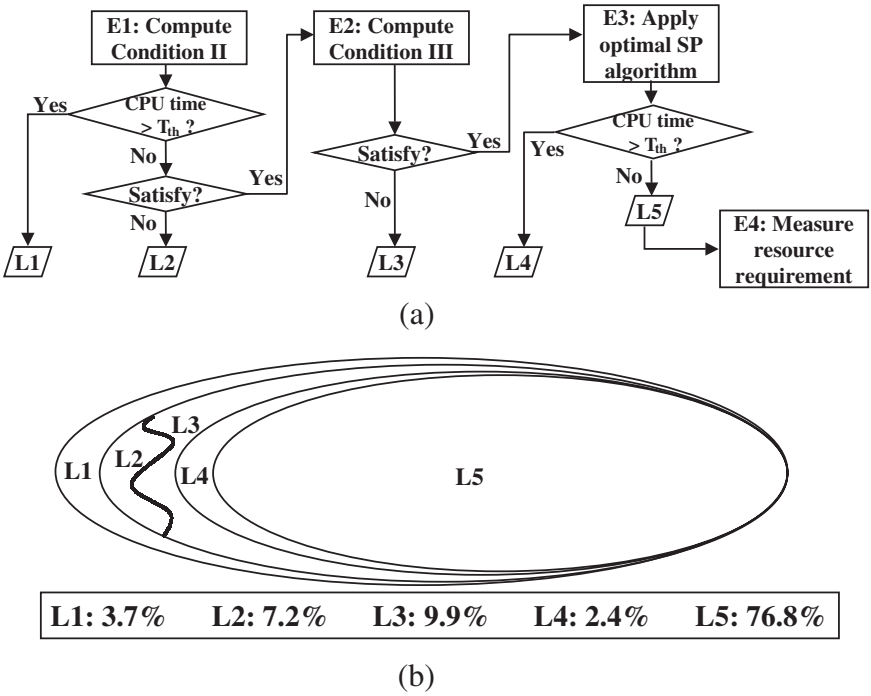


Fig. 17. (a) Experiment scenario and (b) loop classification based on the experimental results. (The area of each region roughly represents the relative size of the corresponding set of loops when $T_{th} = 30$ seconds.)

within the threshold time for about 3.7% of 1317 loops tested. In Fig. 17(a), the set of such loops is denoted by L1. Among the loops for which Condition 2 can be checked within T_{th}, 92.5% of them satisfied Condition 2. (That is, 89.1% of the loops tested satisfied Condition 2.)

Next, we turned our attention on the practicality of the practical software pipelining algorithm presented in Section 6. In the second experiment (i.e., E2 in Fig. 17(a)), we measured how many loops satisfy Condition 3 (i.e., the stronger version of the Time Optimality Condition presented in Section 6). Unlike the first experiment (i.e., E1), we could determine Condition 3 within the threshold time for all the loops (except those in L1 and L2) since Condition 3 can be more efficiently evaluated. In the experiment, 79.2% of total loops satisfy Condition 3, which indicates that Condition 3 does not impose a much additional constraint on Condition 2. In Fig. 17(a), L3 represents the set of loops that do not satisfy Condition 3.

Table I. Resource Requirement for Optimally Software-Pipelined Programs.

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

In the third experiment (i.e., E3 in Fig. 17(a)), we applied the proposed practical software pipelining algorithm to the loops satisfying Condition 3 and measured the running time of the algorithm. In rare cases, the algorithm did not run within the threshold time T_{th}. In Fig. 17(a), the set of such loops is denoted by L4 and the set of loops for which optimally software pipelined loops are computed within T_{th} is denoted by L5, respectively. The portion of loops belonging to L4 and L5 are 2.4% and 76.8% (of total loops), respectively. Figure 17(b) summarizes graphically the results of three experiments, E1, E2, and E3.

In the final experiment (i.e., E4 in Fig. 17(a)), we were concerned with the resource requirement of optimally software pipelined loops (in L5). We measured the number of functional units and the number of registers in the optimally software-pipelined programs and the results are summarized in Table I.[15] (We assumed homogeneous FUs.) Among the loops in L5, 42.7% of the loops require at most 8 FUs while only 32.4% of the loops require more than 16 FUs. We believe that the resource requirement can be further reduced if the proposed software pipelining algorithm is augmented by post-pass local code motions (e.g., moving operations in non-critical dependence chains). For the register requirement, we obtained more positive results, 95.6% of the loops require at most 64 registers. Furthermore, for 68.9% of the loops, 32 registers were sufficient without causing any spill.

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

[15] In counting the number of FUs, we omitted copy operations used for renaming. Most of the renaming copy operations can be eliminated by post-pass optimizations such as copy propagation or register coalescing after unrolling,[17] which is applicable even to unreducible loops.

8. CONCLUSION AND FUTURE WORK

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

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

APPENDIX A: SOFTWARE PIPELINING SUBROUTINES

A.1. Algorithm for Building a Parallel Group

---

procedure SCHEDULE_PARALLEL_GROUP($\mathscr{L}'$, n_{dummy}, A, *frontiers*)
  boundaries := $\{n_{\text{dummy}}\}$
  sort elements in A by the priority order
  foreach ($n \in A$)
    make $\mathscr{L}'$ delete consistent for (r, n)
    n_{new} := COMBINE_SOURCE_REGISTERS($\mathscr{L}'$, 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_{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_new) := n_dummy
        succ_T(n_new) := the duplicate of n_dummy
      end if
    end foreach
    foreach (n_b ∈ boundaries)
      frontiers := frontiers ∪ {(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 if
    end foreach
    return A
  end procedure
```

A.2. 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 if
    end for
    return n_r
  end procedure
```

APPENDIX B: ALGORITHM FOR BUILDING A CYCLE TREE

procedure *build_cycle_tree*(c, $C = \{c_1, c_2, \ldots\}$)
  /* c: the cycle to decompose, C: the set of simple cycles */
  $covered[1..|c|-1] := \{F, \ldots, F\}$
  $V[CT(c)] := \{\}$
  $E[CT(c)] := \{\}$
  while ($\exists i$, $covered[i] \equiv F$)
    $j := 1$
    for ($k := 1$ **to** $|c|-1$)
      if ($covered[k] \equiv F$)
        $orig\_index[j] := k$
        $remained[j{+}{+}] := c[k]$
      end if
    end for
    find the smallest l and m ($l < m < j$) such that $remained[l] \equiv remained[m]$
    find r and s such that
      $c_r(s) \equiv \langle remained[l], remained[l+1], \ldots, remained[m] \rangle$
    $V[CT(c)] := V[CT(c)] \cup \langle c_r(s), [orig\_index[l], orig\_index[m]] \rangle$
    for ($k := l$ **to** $m-1$)
      $covered[orig\_index[k]] := T$
    end for
  end while
  foreach ($x_1 = \langle c_{r_1}(s_1), [j_1, k_1] \rangle$, $x_2 = \langle c_{r_2}(s_2), [j_2, k_2] \rangle \in V[CT(c)]$)
    if ($j_1 < j_2 \wedge k_1 > k_2$)
      $E[CT(c)] := E[CT(c)] \cup [(x_1, x_2)$
    end if
  end foreach
  foreach ($(x_1, x_2) \in E[CT(c)]$)
    if ($\exists x_3$, (x_1, x_3), $(x_3, x_2) \in E[CT(c)]$)
      $E[CT(c)] := E[CT(c)] - (x_1, x_2)$
    end if
  end foreach
  $root[CT(c)]$:= the unique node with no in-edge
  return $CT(c)$
end procedure

ALGORITHM C: ALGORITHM FOR FINDING THE CYCLE FROM A CYCLE TREE

```
procedure found_sequence(T, C = {c1,...})
  /* 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]⟩
end procedure
```

```
procedure post_order(n, sequence, i)
  ⟨c, [i1,i2]⟩ := n/*it must be i ≡ i1*/
  for (j := 1 to outdeg(n))
    ⟨c', [k1, k2]⟩ := child(n, j)
    for (l := i to k1 − 1)
      sequence[l] := c[i1 ++]
    end for
    i = post_order(child(n, j), sequence, k1)
  end for
  for (j := i to i2 − 1)
    sequence[j] := c[i1 ++]
  end for
  return i2
end procedure
```

ACKNOWLEDGMENTS

We would like to thank anonymous referees for their helpful comments and suggestions. This work was supported by Grant No. R01-2001-00360 from the Korea Science and Engineering Foundation. The RIACT at Seoul National University provides research facilities for the study.

REFERENCES

1. A. Aiken and A. Nicolau, Optimal Loop Parallelization, in *Proc. of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 308–317 (1988).
2. A. Aiken and A. Nicolau, Perfect Pipelining, in *Proc. of the Second European Symposium on Programming, Lecture Notes in Computer Science*, Vol. 300, Springer-Verlag, pp. 221–235 (1988).
3. A. Aiken, A. Nicolau, and S. Novack. Resource-Constrained Software Pipelining, *IEEE Trans. Parall. Distr.* **6**(12):1248–1270 (1995).

4. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, Conversion of Control Dependence to Data Dependence, in *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189 (1983).
5. 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*, pp. 139–150 (1995).
6. P.-Y. Calland, A. Darte, and Y. Robert, Circuit Retiming Applied to Decomposed Software Pipelining, *IEEE Trans. Parall. Distr.* **9**(1):24–35 (1998).
7. L.-F. Chao and E. Sha, Scheduling Data-Flow Graphs via Retiming and Unfolding, *IEEE Trans. Parall. Distr.* **8**(12):1259–1267 (1997).
8. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Trans. Progr. Lang. Sys.* **13**(4):451–490 (1991).
9. 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*, pp. 3–21 (1988).
10. J. Farrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Progr. Lang. Sys.* **9**(3):319–349 (1987).
11. F. Gasperoni and U. Schwiegelshohn, Generating Close to Optimum Loop Schedules on Parallel Processors, *Parallel Process. Lett.* **4**(4):391–403 (1994).
12. F. Gasperoni and U. Schwiegelshohn, Optimal Loop Scheduling on Multiprocessors: A Pumping Lemma for *p*-Processor Schedules, in *Proc. of the 3rd Interna-tional Conference on Parallel Computing Technologies*, pp. 51–56 (1995).
13. F. Gasperoni and U. Schwiegelshohn, List Scheduling in the Presence of Branches: A Theoretical Evaluation, *Theoret. Comput. Sci.*, **196**(2):347–363 (1998).
14. R. Govindarajan, E. R. Altman, and G. R. Gao. A Framework for Resource-Constrained Rate-Optimal Software Pipelining, *IEEE Trans. Parall. Distr.* **7**(11):1133–1149 (1996).
15. J. Janssen and H. Corporaal, Making Graphs Reducible with Controlled Node Splitting, *ACM Trans. Progr. Lang. Sys.* **19**(6):1031–1052 (1997).
16. D. Johnson, Finding All the Elementary Circuits of a Directed Graph, *SIAM J. Comput.* **4**(1):77–84 (1975).
17. S. Kim, S.-M. Moon, J. Park, and K. Ebcioğlu, Unroll-based Copy Elimination for Enhanced Pipeline Scheduling, *IEEE Trans. Comput.* **52**(9):977–994 (2002).
18. 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*, pp. 207–218 (1981).
19. 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*, pp. 318–328 (1988).
20. D. Milicev and Z. Jovanovic, Control Flow Regeneration for Software Pipelined Loops with Conditions, *Int. J. Parallel Prog.* **30**(3):149–179 (2002).
21. S.-M. Moon and S. Carson, Generalized Multi-Way Branch Unit for VLIW Microprocessors, *IEEE Trans. Parall. Distr.* **6**(8):850–862 (1995).
22. S.-M. Moon and K. Ebcioğlu, Parallelizing Non-Numerical Code with Selective Scheduling and Software Pipelining, *ACM Trans. Progr. Lang. Sys.* **19**(6):853–898 (1997).
23. A. Nicolau, Uniform Parallelism Exploitation in Ordinary Programs, in *Proc. of the International Conference on Parallel Processing*, pp. 614–618 (1985).
24. S. Park, S. Shim, and S.-M. Moon, Evaluation of Scheduling Techniques on a SPARC-Based VLIW Testbed, in *Proc. of the 30th Annual International Symposium on Microarchitecture*, pp. 104–113 (1997).

25. 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*, pp. 67–78 (1991).
26. U. Schwiegelshohn, F. Gasperoni, and K. Ebcioğlu, On Optimal Parallelization of Arbitrary Loops, *J. Parallel. Distr. Com.* **11**(2):130–134 (1991).
27. S. Shim and S.-M. Moon, Split-Path Enhanced Pipeline Scheduling for Loops with Control Flows, in *Proc. of the 29th Annual Symposium on Microarchitecture*, pp. 93–102 (1998).
28. A. Uht, Requirements for Optimal Execution of Loops with Tests, *IEEE Trans. Parall. Distr.* **3**(5):573–581 (1992).
29. D. W. Wall, Limits of Instruction-Level Parallelism, in *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188 (1991).
30. 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*, pp. 290–299 (1993).