SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **40**:45–72 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Loop Recreation for Thread-Level Speculation on Multicore Processors

Lin Gao[1], Jingling Xue[1], Tin-Fook Ngai[2]

[1] *University of New South Wales, Australia*
[2] *Microprocessor Technology Lab, Intel*

SUMMARY

Inter-iteration dependences in loops can hinder loop-level parallelism. For some loops, existing thread-level speculation (TLS) techniques fail to expose their inherent loop-level parallelism, because some inter-iteration dependences are too costly to synchronize, predict, pre-compute and isolate. This paper presents a compiler technique called *loop recreation* to change the nature of some dependences (by turning some inter-iteration dependences into intra-iteration ones and vice versa) in a loop so that the inter-iteration dependences in the transformed loop are less costly to enforce at run time than those in the original loop. We present an algorithm for finding an optimal loop recreation transformation with respect to a simple misspeculation cost model and demonstrate performance advantages of loop recreation over two recent techniques for multicore systems running nine representative irregular applications.

1. Introduction

As multicore architectures become commonplace, automatic parallelization of sequential programs is required to maximize utilization of the computing resources provided by multicore processors. However, it is difficult for the compiler to create parallel threads for irregular programs on traditional multiprocessor architectures. One promising technique for overcoming this problem is Thread-Level Speculation (TLS) or Speculative MultiThreading (SpMT), which allows the compiler to optimistically create speculatively parallel threads for sequential programs without having to prove they are independent. This can be particularly effective for applications that are difficult to parallelize traditionally due to, for example, their use of irregular data structures (via pointers or subscripted subscripts). Many compiler techniques [34, 27, 13, 9, 16, 2, 30, 22, 7, 31, 12, 3, 5, 24, 25, 29, 20] have been proposed to capitalize such thread-level parallelism (TLP) for either individual program structures (such as loops and procedures) or whole programs. Most of these compiler techniques target loops since substantial amounts of parallelism reside in loops. Despite these research efforts, it remains challenging to develop effective compiler techniques to parallelize individual loops, let alone whole programs. Excessive overhead incurred due to enforcing synchronized and speculated inter-thread flow or true (aka RAW) dependences can lead

Received
Revised
Accepted

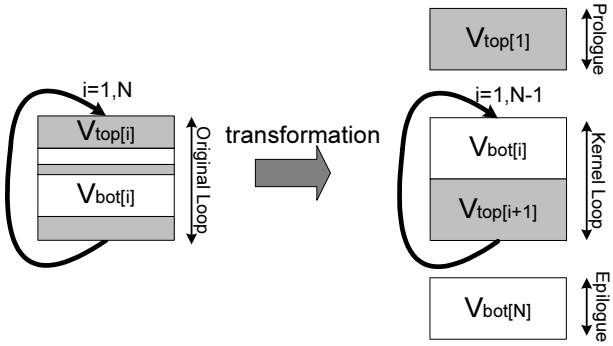


Figure 1: Loop recreation as a loop transformation for a loop. $\{V_{\text{top}}, V_{\text{bot}}\}$ is a partition of the set V of all instructions in the loop. The notation $S[i]$ denotes the set of instructions in S instantiated at iteration i. The instructions in the kernel created can be further scheduled, as illustrated in Figure 2, so that some instructions in $V_{\text{bot}}[i]$ may not appear before those in $V_{\text{top}}[i+1]$.

to significant performance penalties. Therefore, effective management of inter-thread dependences is crucial for achieving good performance on SpMT architectures.

Prior work on TLS for loops in sequential programs with irregular data structures achieves only modest performance improvements. Some inter-iteration dependences in loops are the obstacle to achieving good loop-level parallelism. Techniques such as synchronization, pre-computation, prediction and code isolation have been used to expose the loop-level parallelism inherent in loops. For some recent progress in this research area, POSH [16] yielded approximately 1.2x for a 4-way CMP on SPECint2000 benchmarks when its loop-oriented TLS is applied. Like many other existing TLS compiler techniques [30, 31, 32, 24, 25, 21], the POSH technique turns loop iterations directly into speculative threads. Such an iteration-boundary-preserving loop parallelization technique is referred to as Par in this work. By using Par, all inter-iteration flow dependences in a loop become inter-thread dependences at run time. Intel's effort on developing the SPT compiler for loops [7] achieved 1.08x for a 2-way CMP on SPECint2000. Like Par, SPT also uses loop boundaries as thread boundaries but pre-computes part of loop body to avoid speculating dependences that are likely to happen. One major limitation of prior work is that parallelization is always restricted to loop boundaries so that the nature of dependences (intra- or inter-iteration) remains unchanged. In practice, some inter-iteration dependences in a loop can be too costly to speculate, synchronize, pre-compute and isolate. Furthermore, value prediction may not be effective for irregular loops accessing arrays with pointers or subscripted subscripts. Therefore, loop boundaries are not desirable to serve as thread boundaries for some loops.

In this paper, we present a compiler technique, called *loop recreation transformation* (LRT) and illustrated in Figure 1, to speculatively parallelize sequential loops. Loop recreation amounts to finding a partition $V_{\text{cut}} = \{V_{\text{top}}, V_{\text{bot}}\}$ of the set of instructions in a loop (with V denoting the set of nodes in its data dependence graph) and then transforming the loop into a prologue, a kernel loop and an epilogue. This is where similarity with modulo scheduling ends and differences

begin. When modulo scheduling a loop, the objective is to form a kernel loop by overlapping instructions from multiple iterations in the original loop so that the instruction-level parallelism (ILP) in the kernel is improved. Due to overlapping register lifetimes, register renaming (via, say, rotating registers) is required. Furthermore, the overlapped register lifetimes are likely to span more than one iteration, which indicates more inter-iteration dependences in a modulo-scheduled loop than in the original loop. Therefore, when running iterations of a modulo-scheduled loop in parallel speculatively, its inter-iteration dependences are likely to be more costly to enforce at run time than those in the original loop. In LRT, however, our objective is to form a kernel loop with a different set of inter-iteration dependences from that in the original loop by overlapping instructions from two adjacent iterations in the original loop. LRT changes the nature of some dependences so that some inter-iteration dependences in the original loop are turned into intra-iteration dependences in the kernel and vice versa. When the loop iterations of the resulting kernel is speculatively executed in parallel, its inter-iteration dependences are less costly to enforce at run time than those in the original loop. As a result, the parallelism inherent in the original loop is improved due to the increased speculative thread-level parallelism (TLP) in the kernel. No register renaming is needed. Note that loop recreation differs fundamentally from (partial or full) loop peeling since the instructions in V_{top} or V_{bot} are usually not consecutive in the original loop.

For the loop given in Figures 2(a) and (b), the parallelized loops by three methods (Par, SPT and LRT) are listed in Figures 2(c), (d) and (e), respectively. This example explains how LRT works and why it can expose more parallelism than Par and SPT. It also highlights how speculative threads are spawned by means of a fork instruction. In particular, the fork instruction inserted in a parallelized loop divides each iteration into a *pre-fork* region and a *post-fork* region. The instructions in the pre-fork region pre-compute values for the next thread, and thus are executed sequentially. The key differences among the three methods are as follows. Par aims at maximizing the amount of speculative parallelism achievable in a loop by making the post-fork the largest possible. SPT attempts to reduce the misspeculation overhead of Par by moving the producer instructions of some frequently misspeculated inter-iteration dependences into the pre-fork region to pre-compute the producers for next thread. As a result, the misspeculation penalties that would otherwise be incurred by these dependences in Par are eliminated. To ensure correctness, any instruction on which such a producer depends must also be moved into the pre-fork region. So SPT reduces misspeculation penalties at the expense of parallelism attainable. In comparison with Par and SPT, LRT aims at simultaneously maximizing the parallelism (by keeping the post-fork region the largest possible) and minimizing the misspeculation frequency (by turning frequently occurring inter-iteration dependences into intra-iteration dependences).

For loops whose inter-iteration dependences can be effectively speculated, pre-computed or value-predicted, these properties remain unchanged after LRT has been applied. Therefore, the TLS techniques that use loop boundaries as thread boundaries can be considered as special cases of LRT. Furthermore, LRT is developed to parallelize the loops whose boundaries are not desirable to serve as thread boundaries. Being orthogonal to many existing TLS techniques, LRT can be used as a prepass or postpass to enhance these techniques to extract more TLP from such loops.

In summary, the main contributions of this paper are as follows:

- We introduce a new compiler technique, called loop recreation and denoted LRT, to improve the speculative parallelism inherent in sequential loops with a particular emphasis on reducing the overhead of managing their inter-thread dependences.

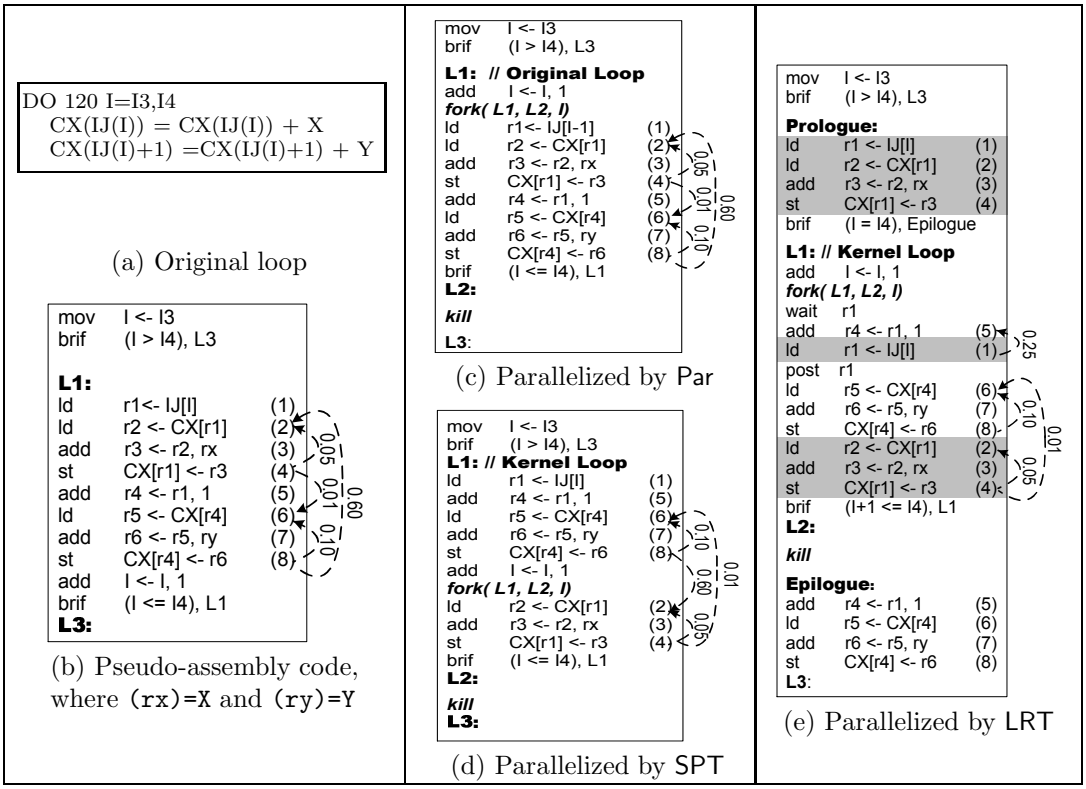


Figure 2: Loop recreation for a loop. In (a), the loop is abstracted from loop 120 in subroutine `parmvr` of `wave5` in SPECfp95. In (b), its pseudo-assembly code is given. In (c), the loop parallelized by Par with iterations being mapped directly into threads is given. In (d), the best solution from SPT is experimentally found according to their cost model. The producer `CX[r4]` of (8) → (2) is moved into the pre-fork region, together with the four other (indirect producer) instructions. In (e), the loop parallelized by LRT is given, where $V_{\text{top}} = \{1, 2, 3, 4\}$ and $V_{\text{bot}} = \{5, 6, 7, 8\}$. In each loop given, all **inter-iteration dependences** are depicted, together with their dependence probabilities. (1) → (5) in (e) is a register dependence on `r1` and all the others shown in (b) – (e) are memory dependences on `CX`. Par and SPT do not change the dependences in the original loop. But LRT has changed the nature of two dependences: (1) → (5) depicted in (e) is an intra-iteration dependence in (b) (and thus not shown in (b)), and (8) → (2) is an inter-iteration dependence in (b) but has been turned into an intra-iteration dependence in (e) (and thus not shown in (e)). Section 3.3 discusses how to represent the communication delays of synchronized dependences such as (1) → (5) also by "probabilities". By "swapping" the roles that (1) → (5) and (8) → (2) play in terms of whether they are intra- or inter-iteration dependences, LRT has selected the less costly (1) → (5) to enforce in the kernel loop. Thus, the misspeculation probability of a thread has been greatly reduced from $1-(1-0.05)(1-0.6)(1-0.01)(1-0.10) = 0.66$ in (c) to $1-(1-0.25)(1-0.10)(1-0.01)(1-0.05) = 0.37$ in (e).

- We present an algorithm for finding an optimal loop recreation for a loop with respect to a simple yet effective misspeculation cost model. This algorithm is efficiently implementable once simple pruning heuristics are used to reduce the search space for an optimal solution.
- We have implemented LRT in SUIF/MachSUIF and demonstrated the performance advantages of LRT over Par and SPT for multicore systems running nine representative irregular applications. Significant performance improvements over Par and SPT are observed under two squash mechanisms.

The rest of this paper is organized as follows. Section 2 describes the loop and execution models used. Section 3 presents our loop recreation transformation, introduces a simple misspeculation cost model and discusses its legality and optimality. Section 4 gives an algorithm for finding an optimal loop recreation transformation with respect to our cost model. Section 5 discusses the evaluation methodology used. Section 6 presents and analyzes our experimental results. Section 7 reviews the related work. Finally, Section 8 concludes the paper.

2. Loop and Execution Models

Like SPT [7], LRT considers only the data dependences when speculatively parallelizing loops. Par is similarly applied to speculate on data dependences. If a loop contains other loops or if branches, each branching structure is either reduced to one single instruction or translated into predicated instructions by if-conversion. Except the back edge of a loop, only data dependences in the loop will be speculated. Therefore, LRT operates on the data dependence graph (DDG) of a loop.

As illustrated in Figure 2(e), a thread consists of the sequence of instructions delineated by **L1** and **L2**. A thread is divided by a *fork* instruction into a pre-fork region (marked by **L1** and *fork*) and a post-fork region (marked by *fork* and **L2**). A *fork* instruction, when executed on a core, spawns a new thread (known as the *successor thread*) on the successor core. The iterations of a parallelized loop are distributed to cores in the SpMT system as described in Section 5.4 in a round-robin fashion. The oldest thread in sequential order is called the *head thread*, which is the only *non-speculative* thread and thus allowed to commit its results. All others are *speculative*.

Inter-iteration anti- and output dependences are implicitly enforced by the hardware. An inter-iteration RAW dependence can be either synchronized or speculated. Deciding which option to take, which was investigated previously [32, 31], is orthogonal to this work. In this work, all *register dependences*, i.e., the dependences involving register variables, are synchronized as in [32] and all *memory dependences*, i.e., the dependences through memory variables, are speculated.

- **Synchronization of Register Dependences**
  For each dependence involving a register value, the dependence is synchronized by communicating the register value between two adjacent threads. To communicate the register value associated with a scalar, whether or not the instruction corresponded to the source of the dependence edge is in the pre-fork region is distinguished. If the corresponding instruction is in the pre-fork region, the register value is already available to the successor thread since all instructions in the pre-fork region of a thread are executed before the successor thread is spawned. Therefore, the *fork* instruction in a thread not only spawns the successor thread, but also forwards all register values defined in the pre-fork region of the thread to the successor thread. Otherwise, the corresponding instruction is in the post-fork region. The compiler inserts a `post` instruction after the last definition of the scalar

in a thread and a `wait` instruction before any use of the scalar in the successor thread. The `wait` instruction stalls execution until the value is produced by the predecessor thread, which communicates the value by the `post` instruction. The insertion of `wait` and `post` instructions, as a step of LRT, operates on the DDG of a loop.

- **Speculation of Memory Dependences**
  We adopt the standard execution model supported by multicore SpMT architectures to handle speculated dependences. Whether a memory dependence is misspeculated or not is detected by the hardware.

Misspeculated threads are squashed and new threads are spawned to re-execute their previously alloted iterations. We will evaluate this work using the following two squash mechanisms:

Eager Squash. Let $T_1, \ldots, T_N$ be all N concurrently running threads ordered from the least speculative to the most speculative. For every write access made in a thread T_i, the hardware checks immediately to see if there exists a dependence violation with respect to an earlier read access made in all more speculative threads $T_{i+1}, \ldots, T_N$. Let T_j be the least speculative thread where a dependence violation is detected. The hardware will squash immediately $T_j, \ldots, T_N$ and spawn a new thread on the same core where T_j was executed to re-execute the same iteration that was previously alloted to T_j.

Lazy Squash. Only after a thread has finished will the hardware squash all more speculative ones that have been misspeculated. Dependence violations can still be detected on the fly.

When a thread reaches the end of the loop marked by `kill`, the thread waits until it becomes the head thread to execute the `kill` instruction, which will cause all speculative threads to be squashed. These threads have been control-misspeculated (on the back edge of the loop).

3. Loop Recreation Transformation

We present an algorithm for finding an optimal loop recreation for a loop with respect to a simple misspeculation cost model. Our algorithm consists of solving a min-cut problem on a set of flow networks derived from the data dependence graph (DDG) of a loop. While the number of flow networks may be large for some loops, our algorithm is practically efficient due to some heuristics that can be readily deployed. Section 3.1 gives a necessary and sufficient condition for the legality of a loop recreation transformation. Section 3.2 describes the cost model used for approximating uniformly the costs incurred in enforcing both synchronized and speculated dependences. Section 3.3 builds the DDG for a loop. Section 3.4 is concerned with the optimality of a loop recreation transformation.

3.1. Legality

As shown in Figure 1, a loop recreation for a loop is uniquely specified by a partition of the set V of its instructions into $V_{\text{cut}} = \{V_{\text{top}}, V_{\text{bot}}\}$. The transformed loop by V_{cut} is referred to as the *recreated loop*. The key idea behind a loop recreation $V_{\text{cut}} = \{V_{\text{top}}, V_{\text{bot}}\}$ is to transform some intra-iteration dependences into inter-iteration dependences and vice versa. As can be observed

in Figure 1, the distance of a dependence from V_{top} to V_{bot} is increased by 1 while the distance of a dependence from V_{bot} to V_{top} is decreased by 1. The distances of other dependences remain unchanged. The standard notion of dependence distance in optimizing compilers [19] is used here (as in modulo scheduling). Therefore, inter-iteration dependences with distances being larger than 1 in the original loop remain to be inter-iteration dependences after loop recreation. However, the nature of a dependence pointing from V_{top} (V_{bot}) to V_{bot} (V_{top}) with distance 0 (1) changes. According to this observation, the legality of a loop recreation transformation is stated below.

Definition 1 (Legality) A loop recreation transformation $V_{\text{cut}} = \{V_{\text{top}}, V_{\text{bot}}\}$ for a loop is *legal* if and only if there are no intra-iteration dependences pointing from V_{bot} to V_{top}.

The results stated below are immediate from Figure 1.

Lemma 1. *A loop recreation $V_{\text{cut}} = \{V_{\text{top}}, V_{\text{bot}}\}$ for a loop affects its dependence $u \rightarrow v$ spanning V_{top} and V_{bot} as follows. If $u \rightarrow v$ is an intra-iteration dependence pointing from V_{top} to V_{bot} in the loop, then $u \rightarrow v$ becomes an inter-iteration dependence with distance 1 in the recreated loop. If $u \rightarrow v$ is an inter-iteration dependence with distance 1 pointing from V_{bot} to V_{top} in the loop, then $u \rightarrow v$ becomes an intra-iteration dependence in the recreated loop. Otherwise, an intra-iteration (inter-iteration) dependence in the loop remains unchanged in the recreated loop.*

3.2. Cost Model

There are many legal loop recreation transformations for a given loop. We will rely on a cost model to rank the relative overheads incurred by different transformations so that the best can be selected. We do not require the cost model to tell us exactly the cost incurred in speculatively executing a loop. In practice, such cost depends on many factors such as the loop trip count, the number of cores available, the architectural parameters of the underlying TLS system and the inter-iteration dependences that are dynamically encountered during program execution. As in prior work [7, 22], we build a simple cost model for a loop by making two common assumptions. First, different dependences in the loop are independent. Second, all incurred memory dependences in the loop are always misspeculated. Both are not always true in practice. However, it is difficult, if not impossible, to know statically the interactions among dependences and determine accurately the relative timings of dependent instructions in different threads.

As mentioned in Section 2, inter-iteration anti- and output register and memory dependences are all automatically enforced in our execution model and incur no runtime overhead. So only inter-iteration RAW (i.e., flow) dependences need to be considered. Each RAW dependence is associated with a *probability* regardless whether it is a register or memory dependence. The probability ranges over [0,1]. The intent is that the probability of an inter-iteration RAW dependence $u \rightarrow v$ represents the fraction of a loop iteration (in cycles) that would be wasted on average due to synchronization (misspeculation) when $u \rightarrow v$ is synchronized (speculated) individually.

The probability p of a RAW memory dependence $u \rightarrow v$ indicates how often the dependence actually takes place at run time. That is, for every N writes issued by the producer instruction u, $p \times N$ reads from the consumer instruction v are made to the same memory location. So p represents the fraction of one iteration that is wasted (in cycles) if the dependence is misspeculated. The probabilities of memory dependences can be estimated by instrumentation or static analysis.

The probability of a RAW register dependence $u \rightarrow v$ represents the incurred communication delay as a fraction of the total execution time for one loop iteration. By convention, we assume that $u \rightarrow v$ may incur some communication delay only when the write access instruction u appears lexically after the read access v. The incurred communication delay is the number of cycles spent on executing instructions that are scheduled between v and u (inclusive).

To compare the costs of different loop recreation transformations for a given loop, we use the *misspeculation probability* of the loop. Let $\mathcal{D}$ be the set of all inter-iteration RAW dependences of a loop: $\mathcal{D} = \mathcal{D}_{\text{mem}} \cup \mathcal{D}_{\mathsf{reg}}$, where $\mathcal{D}_{\text{mem}}$ and $\mathcal{D}_{\mathsf{reg}}$ are the sets of all memory and register dependences contained in $\mathcal{D}$, respectively. As a starting point, we consider the case when $\mathcal{D} = \mathcal{D}_{\text{mem}}$, i.e., $\mathcal{D}_{\mathsf{reg}} = \emptyset$. In this case, the misspeculation probability of a loop can be approximated by:

$$\mathcal{P}(\mathcal{D}) \quad = \quad \mathcal{P}(\mathcal{D}_{\text{mem}}) \;=\; 1 - \Pi_{e \in \mathcal{D}}(1 - Prob_e) \tag{1}$$

where $Prob_e$ is the probability of e (under the assumption that all dependences are independent).

When $\mathcal{D}$ contains one or more register dependences, the misspeculation probability of a loop is refined by distinguishing register dependences from memory dependences as follows:

$$\mathcal{P}(\mathcal{D}) \quad = \quad 1 - \Pi_{e \in \mathcal{D}_{\text{mem}}}(1 - Prob_e) + Prob_{\mathcal{D}_{\mathsf{reg}}^{\max}} \times \Pi_{e \in \mathcal{D}_{\text{mem}}}(1 - Prob_e) \tag{2}$$

where $\mathcal{D}_{\mathsf{reg}}^{\max}$ is the dependence with the largest probability value among all dependences contained in $\mathcal{D}_{\mathsf{reg}}$. In (2), the first addend represents the fraction of a loop iteration lost in cycles due to misspeculation and the second addend represents the fraction of a loop iteration lost (in cycles) due to synchronization. As a result, the communication delay incurred by all dependences in $\mathcal{D}_{\mathsf{reg}}$ is estimated to be the same as the largest communication delay incurred by $\mathcal{D}_{\mathsf{reg}}^{\max}$.

By rearranging (2), we obtain our cost model given below:

$$\mathcal{P}(\mathcal{D}) = 1 - \Pi_{e \in (\mathcal{D}_{\text{mem}} \cup \{\mathcal{D}_{\mathsf{reg}}^{\max}\})}(1 - Prob_e) \tag{3}$$

From this formula, we can see clearly why we also talk about probabilities for register dependences like $\mathcal{D}_{\mathsf{reg}}^{\max}$ just like we do for memory dependences.

3.3. Building the DDG

The DDG of a loop is a weighted directed multigraph, denoted $G = (V, E, Q, K, W)$, where V is the set of instructions in the loop and E is the set of directed edges representing the data dependences between instructions. Let $u \rightarrow v \in E$ such that $u \rightarrow v$ is a dependence from u to v. $Q(u, v) \in \{R, M\}$ represents whether $u \rightarrow v$ is a *R*egister or *M*emory dependence.

$K(u, v)$ represents the (minimum) dependence distance of $u \rightarrow v$, meaning that instruction v at iteration $i + K(u, v)$ may depend on instruction u at iteration i. By convention, $u \rightarrow v$ is an intra-iteration dependence if $K(u, v) = 0$ and inter-iteration dependence otherwise. For a pair of instructions u and v, there can be many dependences $u \rightarrow v$ with varying distances. In order to find an optimal loop recreation for a loop according to Lemma 1, it suffices to distinguish three kinds of distances: (a) $K(u, v) = 0$, (b) $K(u, v) = 1$ and (c) $K(u, v) \geqslant 2$.

Given a path from u to v in G: $P_{u,v} = z_0 \rightarrow z_1 \rightarrow ... \rightarrow z_n$ such that $u = z_0$ and $v = z_n$, where $n \geqslant 1$, we write $K(P_{u,v})$ to represent the sum of dependence distances in the path $P_{u,v}$: $K(P_{u,v}) = \sum_{0 \leqslant i < n} K(z_{i-1}, z_i)$. So $K(P_{u,v})$ indicates that instruction v at iteration $i + K(P_{u,v})$ depends on instruction u at iteration i. By definition, $P_{u,v}$ consists of only intra-iteration

dependences if $K(P_{u,v}) = 0$. Let $V(P_{u,v})$ be the set of instructions in the path $P_{u,v}$. $V(P_{u,v})$ and $K(P_{u,v})$ are used below to compute an estimate for the weight of a register dependence.

For a dependence $u \rightarrow v$, its weight $W(u, v)$ is determined depending on whether it is a memory or register dependence. If $u \rightarrow v$ is a memory dependence, $W(u, v)$ is set to be its dependence probability if it is a RAW dependence and 0 otherwise. Note that anti- and output dependences are included in the DDG only if the IR (Intermediate Representation) representing a loop is not in SSA form. If $u \rightarrow v$ is a register dependence, then we have $W(u, v) = 0$ if $u \rightarrow v$ is anti- or output dependence. Otherwise, we note that when building the DDG for a loop, the loop has not been transformed yet. The weight $W(u, v)$ of a RAW register dependence is then estimated to be $(\sum_{op \in (\mathcal{I} \cup \{u,v\})} C_{op})/C_L$, where C_{op} is the number of cycles spent on executing instruction op, C_L is the total number of cycles spent on executing the entire loop and $\mathcal{I} \subseteq V$ is defined as follows:

$$\mathcal{I} \quad = \quad \begin{cases} \{z \in P_{v,u} \mid u, v \in V, P_{v,u} \text{ is a path in } G \text{ s.t. } K(P_{v,u}) = 0\} & \textbf{if } K(u, v) \geqslant 1 \\ \{z \in P_{v,u} \mid u, v \in V, P_{v,u} \text{ is a path in } G \text{ s.t. } K(P_{v,u}) = 1\} & \textbf{if } K(u, v) = 0 \end{cases}$$

If $K(u, v) \geqslant 1$, i.e., $u \rightarrow v$ is an inter-iteration dependence in a loop L, all instructions in any path $P_{v,u}$ such that $K(P_{v,u}) = 0$ must appear between v and u in L. This is also true in any recreated loop of L in which the dependence distance of $u \rightarrow v$ remains unchanged (Lemma 1). On the other hand, if $u \rightarrow v$ becomes an intra-iteration dependence in a recreated loop of L, then its weight assignment is immaterial. Hence, the incurred communication delay of $u \rightarrow v$ is estimated that way. If $K(u, v) = 0$, i.e., $u \rightarrow v$ is an intra-iteration dependence in L, then $u \rightarrow v$ must point from V_{top} to V_{bot} in any recreated loop of L in order for it to turn into an inter-iteration dependence (Lemma 1). To this end, for every path $P_{v,u}$, at least one edge in the path must point from V_{bot} to V_{top}. If $K(P_{v,u}) = 1$ holds, then the edge pointing from V_{bot} to V_{top} must be the only inter-iteration dependence in $P_{v,u}$ and all the other edges in the path are intra-iteration dependences in the resulting recreated loop. This means that all instructions in $P_{v,u}$ must be scheduled between v and u in the recreated loop. Therefore, the weight assigned to $u \rightarrow v$ is the potentially incurred communication delay when $u \rightarrow v$ represents an inter-iteration dependence in the recreated loop. Otherwise, the weight assignment is immaterial. Our experimental results show that this simple technique for estimating the weights of register dependences suffices in real code.

Recall that $\mathcal{D} = \mathcal{D}_{\text{mem}} \cup \mathcal{D}_{\mathsf{reg}}$ denotes the set of inter-iteration dependences in G introduced in Section 3.2. Obviously, $\mathcal{D}_{\text{mem}}$ and $\mathcal{D}_{\mathsf{reg}}$ are related with G as follows: $\mathcal{D}_{\text{mem}} = \{(u, v) \in E \mid Q(u, v) = M, K(u, v) \geqslant 1\}$ and $\mathcal{D}_{\mathsf{reg}} = \{(u, v) \in E \mid Q(u, v) = R, K(u, v) \geqslant 1\}$.

Figure 3 depicts the DDG for the loop given in Figure 2(b). The DDG has 8 nodes and 13 edges (representing four memory and nine register dependences). The probabilities and distances of the four memory dependences (in dashed lines) are obtained based on profiling information. The weights of the nine register dependences (in solid lines) are estimated as discussed above by assuming that all instructions have unit latency. For register dependence $1 \rightarrow 2$, there is no dependence path in the DDG from instruction 2 to instruction 1. So only the cycles spent on executing instructions 1 and 2 are counted when estimating the weight of $1 \rightarrow 2$. The cases for $1 \rightarrow 4$, $1 \rightarrow 5$, $5 \rightarrow 6$ and $5 \rightarrow 8$ are handled identically. For register dependence $2 \rightarrow 3$, $K(2, 3) = 0$ and the path $P_{3,2} = 3 \rightarrow 4 \rightarrow 2$ is the only path from instruction 3 to 2 in the DDG that satisfies $K(P_{3,2}) = 1$. Therefore, its weight has been set to be $\frac{3}{8} \approx 0.38$. The cases for $3 \rightarrow 4$, $6 \rightarrow 7$ and $7 \rightarrow 8$ are handled identically. So their weights are also 0.38 each.

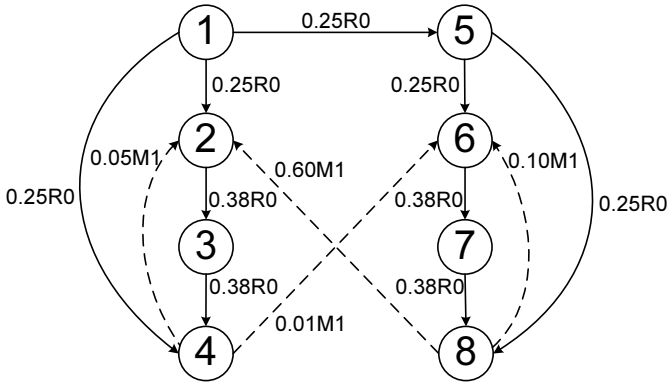


Figure 3: The DDG for the loop in Figure 2(b). Solid (dashed) arrows represent intra-iteration (inter-iteration) dependences with annotations for their dependence nature (e.g., 0.25R0 associated with $1 \rightarrow 2$ means that $Q(1,2) = R$, $K(1,2) = 0$ and $W(1,2) = 0.25$).

3.4. Optimality

Recall that $G = (V, E, Q, K, W)$ denotes the DDG of a loop. Let $\widehat{G} = (\widehat{V}, \widehat{E}, \widehat{Q}, \widehat{K}, \widehat{W})$ be the DDG of the loop recreated by a loop recreation transformation (which will be clear from the context). Clearly, $\widehat{V} = V$, $\widehat{E} = E$, $\widehat{Q} = Q$ and $\widehat{W} = W$. By Lemma 1, the distances of transformed dependences in $\widehat{K}$ can be derived from those in K as follows:

$$\widehat{K}(u,v) \quad = \quad \begin{cases} K(u,v) & \textbf{if } u,v \in V_{\text{top}} \text{ or } u,v \in V_{\text{bot}} \\ K(u,v)+1 & \textbf{if } u \in V_{\text{top}} \text{ and } v \in V_{\text{bot}} \\ K(u,v)-1 & \textbf{if } u \in V_{\text{bot}} \text{ and } v \in V_{\text{top}} \end{cases} \tag{4}$$

Accordingly, we have $\widehat{\mathcal{D}} = \widehat{\mathcal{D}}_{\text{mem}} \cup \widehat{\mathcal{D}}_{\mathsf{reg}}$, where $\widehat{\mathcal{D}}_{\text{mem}} = \{(u,v) \in \widehat{E} \mid \widehat{Q}(u,v) = M, \widehat{K}(u,v) \geqslant 1\}$ and $\widehat{\mathcal{D}}_{\mathsf{reg}} = \{(u,v) \in \widehat{E} \mid \widehat{Q}(u,v) = R, \widehat{K}(u,v) \geqslant 1\}$. In the recreated loop, $\widehat{\mathcal{D}}_{\mathsf{reg}}^{\max}$ is the register dependence with the largest probability value among all the register dependences in $\widehat{\mathcal{D}}_{\mathsf{reg}}$.

Definition 2 (Optimality) A loop recreation transformation $V_{\text{cut}} = \{V_{\text{top}}, V_{\text{bot}}\}$ is *optimal* if

$$\mathcal{P}(\widehat{\mathcal{D}}) \quad = \quad 1 - \Pi_{e \in (\widehat{\mathcal{D}}_{\text{mem}} \cup \{\widehat{\mathcal{D}}_{\mathsf{reg}}^{\max}\})}(1 - Prob_e)$$

is the smallest possible with respect to the cost model (3).

Minimizing $\mathcal{P}(\widehat{\mathcal{D}})$ given in (3) is equivalent to minimizing:

$$\sum_{u \rightarrow v \in (\widehat{\mathcal{D}}_{\text{mem}} \cup \{\widehat{\mathcal{D}}_{\mathsf{reg}}^{\max}\})} \ln(\frac{1}{1 - W(u,v)}) \tag{5}$$

If $W(u,v) = 1$ for a dependence $u \rightarrow v$, then $\frac{1}{1-W(u,v)} = \infty$. In our implementation, this problem is fixed easily by using $W(u,v) = 0.99$ as an accurate approximation instead.

1 $\mathsf{LRT}_{\mathrm{mem}}(G)$ // $G = (V, E, Q, K, W)$
2 Let $G_{\mathrm{ln}} = (V, E, Q, K, W_{\mathrm{ln}})$ such that $W_{\mathrm{ln}}(u, v) = \ln(\frac{1}{1-W(u,v)})$
3 $\mathcal{F} = \mathsf{Cons\_FN}_{\mathrm{mem}}(G_{\mathrm{ln}})$
4 $V_{\mathrm{cut}} = \{V_{\mathrm{top}}, V_{\mathrm{bot}}\} = \mathsf{Process\_FN}(\mathcal{F})$
5 return V_{cut}

Figure 4: An optimal algorithm $\mathsf{LRT}_{\mathrm{mem}}$ for loop recreation when $\mathcal{D} = \mathcal{D}_{\mathrm{mem}}$.

4. Finding an Optimal Transformation

Based on the objective function (5), we find an optimal loop recreation for a loop by solving a min-cut problem on some s-t flow networks derived from the DDG of the loop. The goal is to find a minimum cut on one of these flow networks so that the minimum cut induces an optimal loop recreation for the loop. However, we cannot find a loop recreation $V_{\mathrm{cut}} = \{V_{\mathrm{top}}, V_{\mathrm{bot}}\}$ for a loop by directly solving a min-cut problem on the DDG of the loop. There are two reasons why this is not possible. First, some dependences in $\widehat{\mathcal{D}}$ may not manifest themselves as cut edges. According to (4), $\widehat{\mathcal{D}}$ includes not only the dependences pointing from V_{top} to V_{bot} but also those that are confined in either V_{top} or V_{bot} and those pointing from V_{bot} to V_{top}. However, in a direct min-cut formulation, $\mathcal{P}(\widehat{\mathcal{D}})$ is computed by setting $\widehat{\mathcal{D}}$ as the set of cut edges in a minimum cut. All these cut edges are inter-iteration dependence edges that point from V_{top} to V_{bot} only, but all those inter-iteration dependences that are confined in either V_{top} or V_{bot} and those pointing from V_{bot} to V_{top} are not taken into account. Second, according to Definition 2, only one special register dependence $\widehat{\mathcal{D}}^{\max}_{\mathsf{reg}}$ in $\widehat{\mathcal{D}}$ should be included in $\mathcal{P}(\widehat{\mathcal{D}})$. However, in a direct min-cut formulation, all register dependences that point from V_{top} to V_{bot} are considered when $\mathcal{P}(\widehat{\mathcal{D}})$ is computed.

Following our two-step process used in developing our cost model, we will also present our algorithm in finding optimal loop recreation transformations in two steps. In Section 4.1, we assume that all dependences in the DDG of a loop are memory dependences, i.e., $\mathcal{D} = \mathcal{D}_{\mathrm{mem}}$. Therefore, the cost model given in (1) is used. In Section 4.2, all register dependences in a loop are also considered, i.e., $\mathcal{D} = \mathcal{D}_{\mathrm{mem}} \cup \mathcal{D}_{\mathsf{reg}}$. So the cost model given in (3) is used.

4.1. An Algorithm for the Special Case When $\mathcal{D} = \mathcal{D}_{\mathrm{mem}}$

Given a loop such that its DDG $G = (V, E, Q, K, W)$ consists of memory dependences only, $\mathsf{LRT}_{\mathrm{mem}}$ given in Figure 4 will find an optimal loop recreation from G. $\mathsf{LRT}_{\mathrm{mem}}$ operates on $G_{\mathrm{ln}} = (V, E, Q, K, W_{\mathrm{ln}})$, which is the same as G except that the dependence weights are redefined so that we can formulate the problem of finding an optimal loop recreation for a loop as a min-cut problem (line 2). In $\mathsf{Cons\_FN}_{\mathrm{mem}}$, we create a set, $\mathcal{F}$, of flow networks from G_{ln} (line 3). In $\mathsf{Process\_FN}$, we obtain an optimal loop recreation from a minimum cut with the smallest capacity found from one of the flow networks in $\mathcal{F}$ (line 4).

Section 4.1.1 describes the basic idea behind the development of our min-cut-based algorithm $\mathsf{LRT}_{\mathrm{mem}}$. Section 4.1.2 explains this algorithm in detail. Section 4.1.3 illustrates $\mathsf{LRT}_{\mathrm{mem}}$ by our motivating example. Section 4.1.4 discusses the efficiency of our min-cut-based algorithm.

4.1.1. A Min-Cut-Based Formulation

A set of flow networks $\mathcal{F}$ is constructed from $G_{\mathrm{ln}} = (V, E, Q, K, W_{\mathrm{ln}})$. Let $G_f = (V_f, E_f, C_f)$ be a flow network in $\mathcal{F}$. Then $V_f = V \cup \{s, t\}$, i.e., a flow network will consist of all the nodes in V, the source s and the sink t. Therefore, the key to constructing G_f lies in building the edge set E_f and a capacity function C_f that assigns a non-negative real value $C_f(u, v)$ to each edge $u \rightarrow v \in E_f$. Every flow edge $u \rightarrow v \in E_f$ is always created as a copy of some dependence edge $x \rightarrow y$ in G_{ln}.

Let (S, T) be an s-t cut in G_f that partitions V_f into S and T such that $s \in S$ and $t \in T$:

$$(S, T) \quad = \quad \{u \rightarrow v \in E_f \mid u \in S, v \in T\} \tag{6}$$

In other words, (S, T) consists of only the edges in E_f pointing from S to T. The sum of the capacities of all edges, i.e., *cut edges* in (S, T) is known as the *capacity*, $\mathrm{CAP}(S, T)$, of the cut:

$$\mathrm{CAP}(S, T) \quad = \quad \sum_{u \rightarrow v \in (S,T)} C_f(u, v) \tag{7}$$

For convenience, (S, S), (T, T) and (T, S) are defined similarly in this paper: $(S, S) = \{u \rightarrow v \in E_f \mid u, v \in S\}$, $(T, T) = \{u \rightarrow v \in E_f \mid u, v \in T\}$, $(T, S) = \{u \rightarrow v \in E_f \mid u \in T, v \in S\}$.

The basic idea behind building $\mathcal{F}$ is as follows. If $\{V_{\mathrm{top}}, V_{\mathrm{bot}}\}$ is a legal loop recreation for G_{ln} (or equivalently for G), then $(V_{\mathrm{top}} \cup \{s\}, V_{\mathrm{bot}} \cup \{t\})$ is an s-t cut in every flow network in $G_f \in \mathcal{F}$. If (S, T) is a minimum cut in a flow network G_f, then $\{S \setminus \{s\}, T \setminus \{t\}\}$ is a legal loop recreation for G_{ln}. The problem is that the quality, i.e., cost of a minimum cut (S, T) is measured by its capacity $\mathrm{CAP}(S, T)$, which is defined only by the cut edges pointing from S to T. This implies that when constructing G_f, we must make sure that all and only inter-iteration dependences in the loop recreated by $\{S \setminus \{s\}, T \setminus \{t\}\}$ are the cut edges in (S, T). Therefore, we define:

$$\mathcal{P}(S, T) \quad = \quad 1 - \Pi_{e \in (S,T)}(1 - Prob_e) \tag{8}$$

Unless indicated otherwise, $\widehat{\mathcal{D}}$ is associated with the loop recreated by $\{S \setminus \{s\}, T \setminus \{t\}\}$. Given the assumption that $\mathcal{D} = \mathcal{D}_{\mathrm{mem}}$, the misspeculation probability for the loop recreated by $\{S \setminus \{s\}, T \setminus \{t\}\}$ can be simplified from Definition 2 to:

$$\mathcal{P}(\widehat{\mathcal{D}}) \quad = \quad 1 - \Pi_{e \in \widehat{\mathcal{D}}_{\mathrm{mem}}}(1 - Prob_e)$$

For the reasons explained at the beginning of this section, we cannot find an optimal loop recreation for a loop by directly solving a min-cut problem on the DDG of the loop. Instead, we will create a set of flow networks $\mathcal{F}$ to ensure that every inter-iteration dependence in the loop recreated by a cut $\{S \setminus \{s\}, T \setminus \{t\}\}$ is captured by at least one cut edge in the s-t cut (S, T). To achieve this, some dependence edges are duplicated and may be counted more than once so that $\mathcal{P}(\widehat{\mathcal{D}}) \leqslant \mathcal{P}(S, T)$ holds. Hence, $\mathcal{P}(\widehat{\mathcal{D}}) < \mathcal{P}(S, T)$ may hold for a particular minimum cut (S, T). However, our construction ensures that $\{S \setminus \{s\}, T \setminus \{t\}\}$ is optimal such that $\mathcal{P}(\widehat{\mathcal{D}}) = \mathcal{P}(S, T)$ if $\mathrm{CAP}(S, T)$ is the smallest among all minimum cuts in all flow networks in $\mathcal{F}$.

4.1.2. Finding Minimum Cuts

To ensure that every inter-iteration dependence edge in $\widehat{\mathcal{D}}$ for a recreated loop can be identified as a cut edge, some dependences in G may be mapped into several edges in G_f. Three sets of edges, A-, B- and AB-duplicated edges, that require different ways of duplication are defined below.

Let $x, y \in V$ be two instructions in the DDG G of a loop. We write $K(x,y)^* = 0$ if there exists a path, $P_{x,y}$, of intra-iteration dependences from x to y in G, i.e., $K(P_{x,y}) = 0$. If no such path can be found, we write $K(x,y)^* \neq 0$.

The following result that is immediate from Definition 1 is used to reduce the number of flow networks that will otherwise need to be built in line 3 of the $\mathsf{LRT}_{\mathrm{mem}}$ algorithm.

Lemma 2. *Let $x, y \in V$ such that $K(x,y)^* = 0$. If $V_{\mathrm{cut}} = \{V_{\mathrm{top}}, V_{\mathrm{bot}}\}$ is a legal loop recreation transformation, then either $x \notin V_{\mathrm{bot}}$ or $y \notin V_{\mathrm{top}}$ holds.*

Proof. if both $x \in V_{\mathrm{bot}}$ and $y \in V_{\mathrm{top}}$ hold at the same time, then at least one intra-iteration dependence on a dependence path from x to y must be pointing from V_{bot} to V_{top}. So $V_{\mathrm{cut}} = \{V_{\mathrm{top}}, V_{\mathrm{bot}}\}$ cannot be legal by Definition 1. □

The notions of A-, B- and AB-duplicated edges are formally defined below.

Definition 3 (A-, B- and AB-Duplicated Edges) *The sets of A-duplicated edges, B-duplicated edges and AB-duplicated edges for dependence edges with distance 1 are defined by:*

$$\begin{aligned}
\mathcal{C}_{\mathrm{A}}(G) &= \{(u,v) \in E \mid u \neq v, K(u,v) = 1, W(u,v) \neq 0, K(u,v)^* \neq 0, K(v,u)^* = 0\} \\
\mathcal{C}_{\mathrm{B}}(G) &= \{(u,v) \in E \mid u \neq v, K(u,v) = 1, W(u,v) \neq 0, K(u,v)^* = 0, K(v,u)^* \neq 0\} \\
\mathcal{C}_{\mathrm{AB}}(G) &= \{(u,v) \in E \mid u \neq v, K(u,v) = 1, W(u,v) \neq 0, K(u,v)^* \neq 0, K(v,u)^* \neq 0\}
\end{aligned}$$

$K(u,v)^* = 0 \wedge K(v,u)^* = 0$ is impossible since u and v would then depend on each other.

4.1.2.1. $\mathsf{Cons\_FN}_{\mathrm{mem}}$ Its four steps are explained below. The notation $u \xrightarrow{c} v$ indicates that $u \rightarrow v$ is an edge with the edge capacity c. In Step 1, we introduce two new nodes as the source s and sink t, respectively. That is, every flow network consists of all nodes in V as well as s and t. The construction of weighted edges for each flow network is carried out in Steps 2 – 4.

In Step 2, we note that every dependence $u \rightarrow v$ in a loop remains to be an inter-iteration dependence after a loop recreation transformation if either $K(u,v) \geqslant 2$ or $u \rightarrow v$ is a self edge. Therefore, $u \rightarrow v$ must be included in an s-t cut. As a result, an edge pointing from s to t for $u \rightarrow v$ is added. After Step 2, every dependence considered in Steps 3 and 4 is neither a self-edge nor a dependence with a distance larger than 1 unless indicated otherwise.

In Step 3, for every dependence $u \rightarrow v$ in G_{ln}, we add two edges to G_f: $u \xrightarrow{C_f(u,v)} v$ and $v \xrightarrow{C_f(v,u)} u$, where the former edge represents the situation when $u \rightarrow v \in (S,T)$ and the latter edge represents the situation when $u \rightarrow v \in (T,S)$. Therefore, $C_f(u,v)$ and $C_f(v,u)$ are set as implied by Lemma 1. In line 8, we set $C_f(v,u) = \infty$ if $K(u,v) = 0$ to prevent $v \rightarrow u$ from becoming a cut edge. In particular, we set $C_f(v,u) = \infty$ if $K(u,v) = 0$ because $u \rightarrow v \notin (T,S)$ holds for all legal loop recreation transformations by Lemma 1.

After Step 3, every dependence has been mapped to two copies in a flow network. This is sufficient for intra-iteration but not for inter-iteration dependences. An intra-iteration dependence $u \rightarrow v$ in a loop remains so in a recreated loop if $u \rightarrow v \in (S,S) \cup (T,T)$ by Lemma 1. In this case,

1 Cons_FN$_{\text{mem}}(G_{\text{ln}})$ // $G_{\text{ln}} = (V, E, Q, K, W_{\text{ln}})$

// Build the part of $G_f = (V_f, E_f, C_f)$ that is common in all networks (lines 2 – 14)

// Step 1: create the source s and the sink t

2 Set $V_f = V \cup \{s, t\}$

// Step 2: handle dependences that cross iterations before and after transformation

3 **for** $u \to v \in E$ such that $K(u, v) \geqslant 2$ or $u = v$ **do**

4 Add an edge to E_f: $s \xrightarrow{C_f(s,t)} t$ and set $C_f(s, t) = W_{\text{ln}}(u, v)$

// Step 3: handle dependences spanning an s-t cut

5 **for** $u \to v \in E$ such that $K(u, v) \leqslant 1$, where $u \neq v$ **do**

6 Add two edges to E_f: $u \xrightarrow{C_f(u,v)} v$ and $v \xrightarrow{C_f(v,u)} u$

7 **if** $K(u, v) = 0$ **then**

8 Set $C_f(u, v) = W_{\text{ln}}(u, v)$ and $C_f(v, u) = \infty$

9 **else** // $K(u, v) = 1$

10 Set $C_f(u, v) = W_{\text{ln}}(u, v)$ and $C_f(v, u) = 0$

// Step 4: duplicate inter-iteration dependences of distance 1 not spanning an s-t cut

// A-duplication

11 **for** $u \to v \in \mathcal{C}_{\text{A}}(G_{\text{ln}})$ **do**

12 Add $s \xrightarrow{C_f(s,v)} v$ and $u \xrightarrow{C_f(u,t)} t$ to E_f and set $C_f(s, v) = C_f(u, t) = W_{\text{ln}}(u, v)$

// B-duplication

13 **for** $u \to v \in \mathcal{C}_{\text{B}}(G_{\text{ln}})$ **do**

14 Add $s \xrightarrow{C_f(s,u)} u$ and $v \xrightarrow{C_f(v,t)} t$ to E_f and set $C_f(s, u) = C_f(v, t) = W_{\text{ln}}(u, v)$

// Build distinct parts for all flow networks in $\mathcal{F}$ (lines 15 – 18)

15 Let $u_1 \to v_1, \ldots, u_{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|} \to v_{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|}$ be all edges in $\mathcal{C}_{\text{AB}}(G_{\text{ln}})$

// AB-duplications

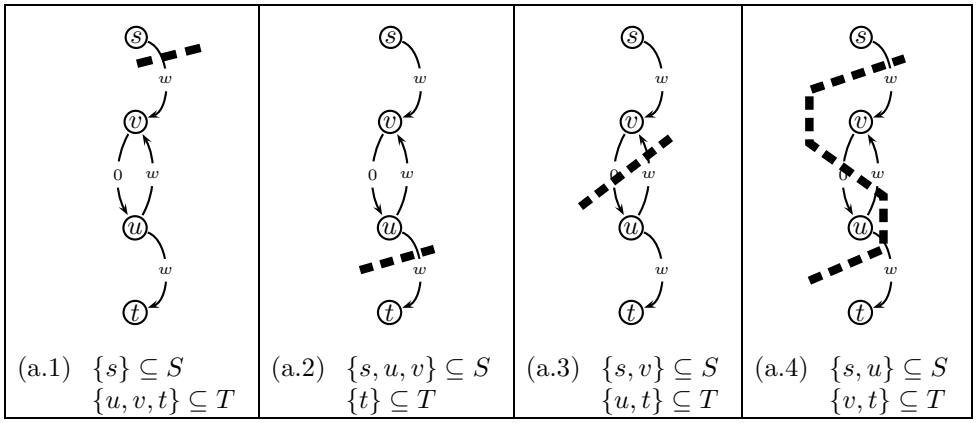
16 **for** $(X_1, \ldots, X_{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|}) \in \{A, B\}^{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|}$ **do**

17 Let $G_{u_1 \xrightarrow[X_1]{} v_1, \ldots, u_{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|} \xrightarrow[X_{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|}]{} v_{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|}}$ be initialized with G_f and then augmented with every $u_i \to v_i$ being X_i-duplicated

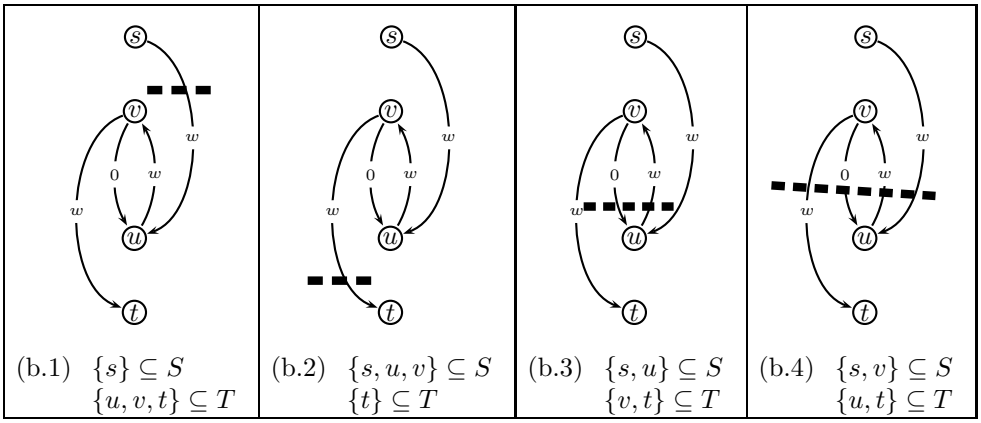
18 $\mathcal{F} \cup= \{G_{u_1 \xrightarrow[X_1]{} v_1, \ldots, u_{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|} \xrightarrow[X_{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|}]{} v_{|\mathcal{C}_{\text{AB}}(G_{\text{ln}})|}}\}$

19 **return** $\mathcal{F}$

Figure 5: Constructing $\mathcal{F}$ from G_{ln} when $\mathcal{D} = \mathcal{D}_{\text{mem}}$.



(a) A-duplicated (w stands for $W_{\ln}(u, v)$)



(b) B-duplicated (w stands for $W_{\ln}(u, v)$)

Figure 6: Four different minimum cuts for $u \to v$ duplicated in two duplication schemes.

the edge should not be a cut edge (since it is absent in $\widehat{\mathcal{D}}$). However, the situation is different if $u \to v$ is an inter-iteration dependence. In this case, if $u \to v \in (S, S) \cup (T, T)$, then $u \to v$ remains to be an inter-iteration dependence in $\widehat{\mathcal{D}}$. However, in this case, $u \to v$ may not be a cut edge in a cut although it should be.

In Step 4, two more copies are introduced for every inter-iteration dependence $u \to v$ in a flow network. We do so by duplicating $u \to v$ so that if $u \to v \in (S, S) \cup (T, T)$, then every s-t cut is guaranteed to include at least one copy of $u \to v$. As illustrated in Figure 6, $u \to v$ is either A-duplicated (lines 11 and 12) or B-duplicated (lines 13 and 14). When both are needed

```
1 Process_FN(F)
2 for every G_f = (V_f, E_f, C_f) in F do
3     Let G'_f be the simple graph converted from the multigraph G_f by merging all parallel
      edges into one single edge whose weight is the sum of those of the parallel edges)
4     (S_f, T_f) = Find_Min_Cut(G'_f)
5 Let (S_opt, C_opt) be one of the |F| minimum cuts found in line 4 with the smallest capacity
6 return (S_opt \ {s}, T_opt \ {t})
```

Figure 7: Finding a minimum cut in $\mathcal{F}$.

(lines 15 – 18), there are $2^{|\mathcal{C}_{\rm AB}(G_{\rm ln})|}$ different ways for duplicating all edges in $\mathcal{C}_{\rm AB}(G_{\rm ln})$. Hence, $|\mathcal{F}| = 2^{|\mathcal{C}_{\rm AB}(G_{\rm ln})|}$. Let us explain why an edge $u \rightarrow v \in \mathcal{C}_{\rm A}(G_{\rm ln})$ needs not also to be B-duplicated in lines 11 and 12. By the definition of $\mathcal{C}_{\rm A}(G_{\rm ln})$, we know that $K(v,u)^* = 0$. Thus, $u \rightarrow v \notin (S,T)$ (Lemma 2). Due to line 8, the cut shown in Figure 6(a.4) must also include an ∞-weighted edge. This cut, which corresponds to an illegal loop recreation, is not a minimum cut. Similarly, $u \rightarrow v \in \mathcal{C}_{\rm B}(G_{\rm ln})$ is not also A-duplicated in lines 13 and 14.

By duplicating every inter-iteration dependence $u \rightarrow v$, a flow network in $\mathcal{F}$ contains four copies of that edge, three of which have the same weight, $W_{\rm ln}(u,v)$. As shown in Figure 6, an s-t cut is guaranteed to include at least one copy of $u \rightarrow v$. However, in Figure 6(a.4) (Figure 6(b.4)), the weight $W_{\rm ln}(u,v)$ of $u \rightarrow v$ is counted twice (one time) too many. Due to such over-counting, the underlying minimum cut may or may not induce an optimal loop recreation. By creating $2^{|\mathcal{C}_{\rm AB}(G_{\rm ln})|}$ flow networks, however, we are guaranteed that such over-counting will not occur in at least one minimum cut found in one of these flow networks.

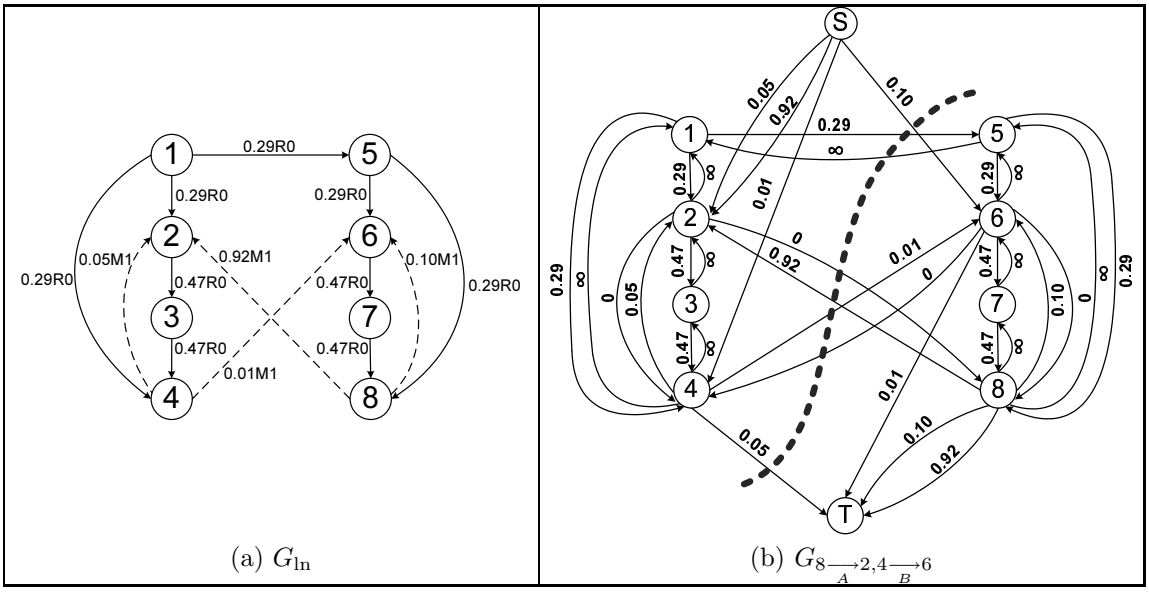
4.1.2.2. Process_FN This procedure, which is given in Figure 7, returns an optimal loop recreation derived from the minimum cut found in one of the flow networks in $\mathcal{F}$.

A proof for the optimality of $\mathsf{LRT}_{\rm mem}$ is given in Appendix A.

4.1.3. Example

The DDG for our example is depicted in Figure 3. There are 13 dependence edges. For illustration purposes, let us assume that these are all memory dependences. There are four inter-iteration dependences. We find that $\mathcal{C}_{\rm A}(G_{\rm ln}) = \{4 \rightarrow 2, 8 \rightarrow 6\}, \mathcal{C}_{\rm B}(G_{\rm ln}) = \emptyset$ and $\mathcal{C}_{\rm AB}(G_{\rm ln}) = \{8 \rightarrow 2, 4 \rightarrow 6\}$ since $K(2,4)^* = 0 \wedge K(4,2)^* \neq 0$, $K(6,8)^* = 0 \wedge K(8,6)^* \neq 0$, $K(2,8)^* \neq 0 \wedge K(8,2)^* \neq 0$ and $K(4,6)^* \neq 0 \wedge K(6,4)^* \neq 0$. Hence, $\mathcal{F}$ has $2^{|\mathcal{C}_{\rm AB}(G_{\rm ln})|} = 4$ flow networks.

Let us trace the execution of our algorithm $\mathsf{LRT}_{\rm mem}$ with respect to the loop briefly. $G_{\rm ln}$ for the loop is shown in Figure 8(a). Of the four flow networks in $\mathcal{F}$, $G_{8 \underset{A}{\longrightarrow} 2,4 \underset{B}{\longrightarrow} 6}$, in which $8 \rightarrow 2$ is A-duplicated and $4 \rightarrow 6$ is B-duplicated, is shown in Figure 8(b). This network is constructed by applying $\mathsf{Cons\_FN}_{\rm mem}$ to $G_{\rm ln}$ as follows. For each of the nine intra-iteration dependences, its two copies in the network are created in lines 7 and 8. For each of the four inter-iteration dependences,



(a) G_{ln} (b) $G_{8 \underset{A}{\longrightarrow} 2,4 \underset{B}{\longrightarrow} 6}$

Figure 8: An illustration of $\mathsf{LRT}_{\text{mem}}$ for the DDG shown in Figure 3 by assuming that all edges are memory dependences.

its two copies are created in lines 9 and 10 and the other two created in lines 11 – 18 (with $4 \rightarrow 2$, $8 \rightarrow 6$ and $8 \rightarrow 2$ being A-duplicated and $4 \rightarrow 6$ B-duplicated).

Figure 8(b) also depicts the minimum cut from which the optimal loop recreation $\{V_{\text{top}}, V_{\text{bot}}\} = \{\{1, 2, 3, 4\}, \{5, 6, 7, 8\}\}$ with respect to the objective function (5) is found (on the simple graph reduced from the multigraph shown in Figure 8(b)).

4.1.4. Time Complexity and Practical Efficiency

We have used Goldberg's implementation of his *push-relabel* HIPR algorithm [10] to find minimum cuts. Its worst-case time complexity when applied to $G = (V, E, Q, K, W)$ is $O(|V|^2 \times \sqrt{|E|})$. In line 2 of Process_FN, there can be $|\mathcal{F}| = 2^{|\mathcal{C}_{\text{AB}}(G)|}$ different flow networks. So the worst-case time complexity of our algorithm is $O(|V|^2 \times \sqrt{|E|} \times 2^{|\mathcal{C}_{\text{AB}}(G)|})$.

In practice, $\mathsf{LRT}_{\text{mem}}$ is efficient if we adopt the following simple strategy in our implementation. Let $G_f \in \mathcal{F}$. Let $u_1 \rightarrow v_1, \ldots, u_m \rightarrow v_m$ be all the B-duplicated edges in G_f. For any minimum cut in G_f, a B-duplicated edge will be cut in one of the four possible ways as depicted in Figure 6(b). Hence, the capacity of any minimum cut (S, T) in G_f must be larger than or equal to $\sum_{i=1}^{m} W_{\text{ln}}(u_i, v_i)$, i.e., $\text{CAP}(S, T) \geqslant \sum_{i=1}^{m} W_{\text{ln}}(u_i, v_i)$. We will first find the minimum cut for the unique flow network in $\mathcal{F}$ in which all edges in $\mathcal{C}_{\text{AB}}(G_{\text{ln}})$ are A-duplicated and then examine the remaining flow networks in the order in which more and more edges in $\mathcal{C}_{\text{AB}}(G_{\text{ln}})$ are B-duplicated.

1 $\mathsf{LRT}(G)$ // $G = (V, E, Q, K, W)$
2 Let $G_{\ln} = (V, E, Q, K, L)$ such that $W_{\ln}(u, v) = \ln(\frac{1}{1-W(u,v)})$
3 $\mathcal{F} = \mathsf{Cons\_FN}(G_{\ln})$
4 $V_{\text{cut}} = \{V_{\text{top}}, V_{\text{bot}}\} = \mathsf{Process\_FN}(\mathcal{F})$
5 return V_{cut}

Figure 9: An optimal algorithm LRT for loop recreation when $\mathcal{D} = \mathcal{D}_{\text{mem}} \cup \mathcal{D}_{\mathsf{reg}}$.

We ignore a flow network if $\sum_{i=1}^{m} W_{\ln}(u_i, v_i)$ for all its B-duplicated edges $u_1 \to v_1, \ldots, u_m \to v_m$ is larger than or equal to the capacity of the best minimum cut found so far. This pruning strategy is efficient as validated in our experiments discussed in Section 6.1.1.

4.2. A General Algorithm When $\mathcal{D} = \mathcal{D}_{\text{mem}} \cup \mathcal{D}_{\mathsf{reg}}$

In the general case, memory dependences and register dependences must be distinguished according to our cost model given in Definition 2. We generalize $\mathsf{LRT}_{\text{mem}}$ to LRT so that LRT is optimal with respect to the objective function (5), where only the register dependences $\widehat{\mathcal{D}}_{\mathsf{reg}}^{\max}$ is relevant. LRT is the same as $\mathsf{LRT}_{\text{mem}}$ except a different module, $\mathsf{Cons\_FN}$, for constructing $\mathcal{F}$ is used. In fact, $\mathsf{Cons\_FN}$ given in Figure 10 is a simple extension of $\mathsf{Cons\_FN}_{\text{mem}}$ to deal with register dependences. The new lines added to $\mathsf{Cons\_FN}_{\text{mem}}$ are highlighted by boxed lines. A proof for the optimality of LRT can be found in Appendix B.

Unlike $\mathsf{Cons\_FN}_{\text{mem}}$, $\mathsf{Cons\_FN}$ may construct a maximum of $2^{|\mathcal{C}_{\text{AB}}(G)|} \times (N_{\mathsf{reg}} + 1)$ flow networks, where N_{reg} is the number of distinct non-zero weights of register dependences in $G_{\ln}$. Let $l_0 = 0$ and $l_1, \ldots, l_{N_{\mathsf{reg}}}$ be N_{reg} distinct non-zero weights of register dependences in $G_{\ln}$ (line 2). All flow networks in $\mathcal{F}$ are divided into $N_{\mathsf{reg}} + 1$ subsets $\mathcal{F}_{l_0}, \ldots, \mathcal{F}_{l_{N_{\mathsf{reg}}}}$ (lines 3, 23 and 24) so that for the flow networks in $\mathcal{F}_{l_i}$, only register dependences whose weights are no larger than l_i can be cut edges (lines 4 and 5). Effectively, l_i is the weight of $\widehat{\mathcal{D}}_{\mathsf{reg}}^{\max}$ referred to in our cost model given in Definition 2. In the case of $\mathcal{F}_{l_0}$, only memory dependences can be cut edges. In fact, $\mathcal{F}_{l_0}$ is exactly the set of flow networks constructed for all memory dependences in G (Section 4.1).

By Lemma 1, every dependence $u \to v$ with a distance $K(u, v) \geqslant 2$ must be included in a minimum cut. Therefore, we avoid building $\mathcal{F}_{l_i}$ if there is a register dependence $u \to v$ with a distance larger than or equal to 2 such that $W_{\ln}(u, v) > l_i$ (line 4). In this case, achieving a minimum synchronization delay l_i for $\widehat{\mathcal{D}}_{\mathsf{reg}}^{\max}$ is not possible. Since $\mathcal{F}_{l_{N_{\mathsf{reg}}}}, \ldots, \mathcal{F}_{l_0}$ are built in that order (line 3), the construction of $\mathcal{F}$ terminates immediately at the i-th iteration when there exists a register dependence $u \to v$ such that $K(u, v) \geqslant 2$ and $W_{\ln}(u, v) > l_i$ (line 4).

In line 5, $G_{\ln}^i$ is derived from $G_{\ln}$ with different register dependence weights so that only register dependences with weights no larger than l_i can be cut edges. For a register dependence $u \to v$, we set $W_{\ln}^i(u, v) = \infty$ if $W_{\ln}(u, v) > l_i$ to prevent it from becoming a cut edge. In lines 6 – 23, $2^{|\mathcal{C}_{\text{AB}}(G_{\ln}^i)|}$ flow networks in $\mathcal{F}_{l_i}$ are constructed from $G_{\ln}^i$ in the same way as explained in Section 4.1.2 except that an additional edge is added in line 7 to represent the weight of $\widehat{\mathcal{D}}_{\mathsf{reg}}^{\max}$.

1 Cons_FN($G_{\ln}$) // $G_{\ln} = (V, E, Q, K, W_{\ln})$

2 Let $l_0 = 0$ and $l_1, \ldots, l_{N_{\mathsf{reg}}}$ be N_{reg} distinct non-zero weights $W_{\ln}(u, v)$ of all register dependences $u \to v$ in $G_{\ln}$ in increasing order of their values

3 **for** $i = N_{\mathsf{reg}}$ to 0

// Build s-t flow networks in $\mathcal{F}_{l_i}$

4 **if** $\exists\ u \to v \in E$ such that $Q(u, v) = R$, $W_{\ln}(u, v) > l_i$ and $K(u, v) \geqslant 2$ **then break**

5 Let $G^i_{\ln} = (V, E, Q, K, W^i_{\ln})$ where $W^i_{\ln}(u, v)$ is defined as follows:

$W^i_{\ln}(u, v) = W_{\ln}(u, v)$ if $Q(u, v) = M$

$W^i_{\ln}(u, v) = \infty$ if $Q(u, v) = R$ and $W_{\ln}(u, v) > l_i$

$W^i_{\ln}(u, v) = 0$ if $Q(u, v) = R$ and $W_{\ln}(u, v) \leqslant l_i$

6 Set $V_f = V \cup \{s, t\}$

7 **if** $l_i \neq 0$ **then** add $s \xrightarrow{C_f(s,t)} t$ to E_f and set $C_f(s, t) = l_i$

8 **for** $u \to v \in E$ such that $K(u, v) \geqslant 2$ or $u = v$ **do**

9 Add an edge to E_f: $s \xrightarrow{C_f(s,t)} t$ and set $C_f(s, t) = W^i_{\ln}(u, v)$

10 **for** $u \to v \in E$ such that $K(u, v) \leqslant 1$, where $u \neq v$ **do**

11 Add two edges to E_f: $u \xrightarrow{C_f(u,v)} v$ and $v \xrightarrow{C_f(v,u)} u$

12 **if** $K(u, v) = 0$ **then**

13 Set $C_f(u, v) = W^i_{\ln}(u, v)$ and $C_f(v, u) = \infty$

14 **else** // $K(u, v) = 1$

15 Set $C_f(u, v) = W^i_{\ln}(u, v)$ and $C_f(v, u) = 0$

16 **for** $u \to v \in \mathcal{C}_{\mathrm{A}}(G^i_{\ln})$ **do**

17 Add $s \xrightarrow{C_f(s,v)} v$ and $u \xrightarrow{C_f(u,t)} t$ to E_f and set $C_f(s, v) = C_f(u, t) = W^i_{\ln}(u, v)$

18 **for** $u \to v \in \mathcal{C}_{\mathrm{B}}(G^i_{\ln})$ **do**

19 Add $s \xrightarrow{C_f(s,u)} u$ and $v \xrightarrow{C_f(v,t)} t$ to E_f and set $C_f(s, u) = C_f(v, t) = W^i_{\ln}(u, v)$

20 Let $u_1 \to v_1, \ldots, u_{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|} \to v_{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|}$ be all edges in $\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})$

21 **for** $(X_1, \ldots, X_{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|}) \in \{A, B\}^{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|}$ **do**

22 Let $G^i_{u_1 \xrightarrow[X_1]{} v_1, \ldots, u_{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|} \underset{X_{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|}}{\longrightarrow} v_{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|}}$ be initialized with G_f and then augmented with every $u_i \to v_i$ being X_i-duplicated

23 $\mathcal{F}_{l_i} \cup= \{G^i_{u_1 \xrightarrow[X_1]{} v_1, \ldots, u_{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|} \underset{X_{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|}}{\longrightarrow} v_{|\mathcal{C}_{\mathrm{AB}}(G^i_{\ln})|}}\}$

24 $\mathcal{F} = \bigcup_{0 \leqslant i \leqslant N_{\mathsf{reg}}} \mathcal{F}_{l_i}$

25 **return** $\mathcal{F}$

Figure 10: Constructing $\mathcal{F}$ from $G_{\ln}$ when $\mathcal{D} = \mathcal{D}_{\mathrm{mem}} \cup \mathcal{D}_{\mathsf{reg}}$.

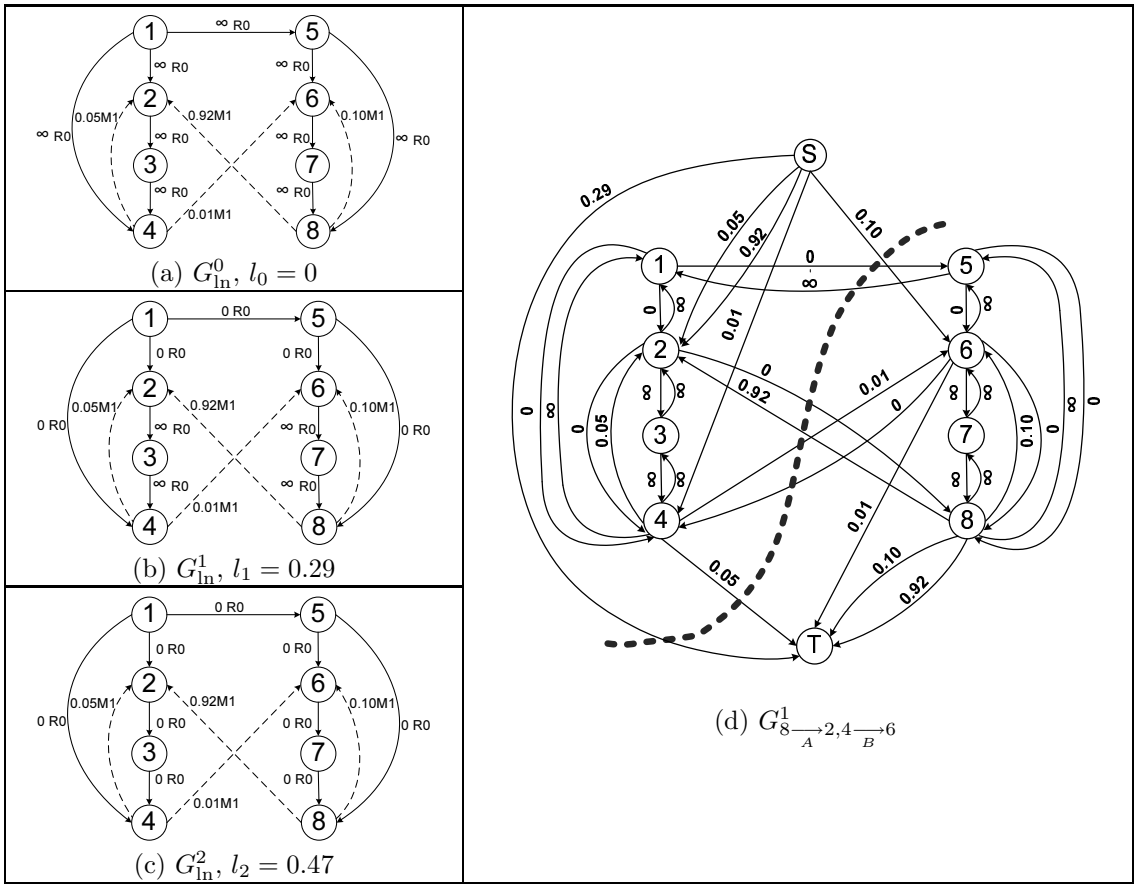


Figure 11: An illustration of LRT. In (a) – (c) $G^0_{\ln}$, $G^1_{\ln}$ and $G^2_{\ln}$ derived from G in Figure 3 are shown. In (d), a flow network in $\mathcal{F}_{l_1}$ with $8 \rightarrow 2$ A-duplicated and $4 \rightarrow 6$ B-duplicated is given. The minimum cut shown by a dashed line represents the optimal solution found by LRT.

In terms of its worst-case time complexity, LRT is seemingly more expensive than $\mathsf{LRT}_{\mathrm{mem}}$. The worst-case time complexity of finding a minimum cut on $G = (V, E, Q, K, W)$ remains to be $O(|V|^2 \times \sqrt{|E|})$. Since there can be at most $|\mathcal{F}| = 2^{|\mathcal{C}_{\mathrm{AB}}(G)|} \times (N_{\mathsf{reg}} + 1)$ flow networks, the worst-case time complexity of LRT is $O(|V|^2 \times \sqrt{|E|} \times 2^{|\mathcal{C}_{\mathrm{AB}}(G)|} \times (N_{\mathsf{reg}} + 1))$. As discussed in Section 3.2, the weights of register dependences in a loop are often small. N_{reg} can be made smaller if the register dependences with small weights are ignored in line 2 of Cons_FN. In addition, there are typically only a few different weight values for the remaining register dependences. Therefore, N_{reg} is often very small in practice. When N_{reg} is relatively large in some pathological cases, we can always bound it from above to a certain value by rounding off register dependence weights.

In our motivating example with its $G_{\ln}$ shown in Figure 8(a), there are nine register dependences with two distinct weights. Therefore, $l_0 = 0$, $l_1 = 0.29$ and $l_2 = 0.47$. Figure 11 shows the three graphs, $G^0_{\ln}$, $G^1_{\ln}$ and $G^2_{\ln}$, built in line 5 of Cons_FN from $G_{\ln}$. Let us consider $G^1_{\ln}$ constructed for l_1. The weights of $2 \rightarrow 3$, $3 \rightarrow 4$, $6 \rightarrow 7$ and $7 \rightarrow 8$ in $G^1_{\ln}$ are all ∞ since $W_{\ln}(2,3) = W_{\ln}(3,4) = W_{\ln}(6,7) = W_{\ln}(7,8) = 0.47 > 0.29 = l_1$. As a result, these four edges cannot be cut edges for all flow networks in $\mathcal{F}_{l_1}$ by construction. The weights of the other five register dependences are 0 in $G^1_{\ln}$. By applying lines 6 – 23 of Cons_FN to $G^1_{\ln}$, we have obtained a flow network $G^1_{8 \underset{A}{\longrightarrow} 2,4 \underset{B}{\longrightarrow} 6}$ with $8 \rightarrow 2$ A-duplicated and $4 \rightarrow 6$ B-duplicated, shown in Figure 11(d). The flow edge $S \xrightarrow{0.29} T$ added in line 7 ensures that weight of $\mathcal{D}^{\max}_{\mathsf{reg}}$ is also included in the capacity of any cut. The optimal solution returned by LRT is found on $G^1_{8 \underset{A}{\longrightarrow} 2,4 \underset{B}{\longrightarrow} 6}$. The optimal loop recreation $\{V_{\text{top}}, V_{\text{bot}}\} = \{\{1,2,3,4\},\{5,6,7,8\}\}$ happens to be same as the one found in Figure 8(b). This transformation is illustrated earlier in Figure 2(e).

5. Evaluation Methodology

We first discuss our implementation of loop recreation in the SUIF/MachSUIF compilation framework. We then describe the benchmarks and simulator used for evaluating our work. Finally, we discuss the two TLS-based loop parallelization methods against which LRT is compared.

5.1. Loop Recreation Implementation

Loop recreation is implemented in the SUIF/MachSUIF compilation framework as illustrated in Figure 12. A program is first converted into the SUIF IR by the SUIF frontend. The SUIF IR is then converted into the MachSUIF IR. Several MachSUIF passes such as register allocation can be applied on the MachSUIF IR. Finally, the MachSUIF backend is used to generate the threaded alpha assembly code, which is fed to a cycle-accurate simulator. Our loop recreation is implemented in the three new modules that are highlighted in gray.

Our three new modules all operate on the MachSUIF IR. The dependence analysis module builds the DDG for a loop. The MachSUIF IR is not in SSA form. So both anti and output intra-iteration dependences are included in the DDG. The data dependences for scalars are found using the standard def-use information. The dependences for memory variables such as arrays and pointers are analyzed in the SUIF IR and passed to the MachSUIF IR. The "probability" of a synchronized register dependence is calculated as described in Section 3.3. The "probability" of a memory dependence is obtained by the profiling module. Our loop recreation module reads the MachSUIF IR of a loop and produces the parallelized code in the form of MachSUIF IR.

In our implementation, the loop recreation pass is invoked just before MachSUIF's register allocation pass. All virtual registers in the MachSUIF IR are candidates for synchronization. All spilled scalars may be synchronized as described in [18] or speculated as memory variables. For all the applications used in our experiments, no spilling has occurred. Loop recreation can also be invoked after MachSUIF's register allocation pass. But some anti- and output intra-dependences artificially introduced on physical registers may reduce some loop recreation opportunities.

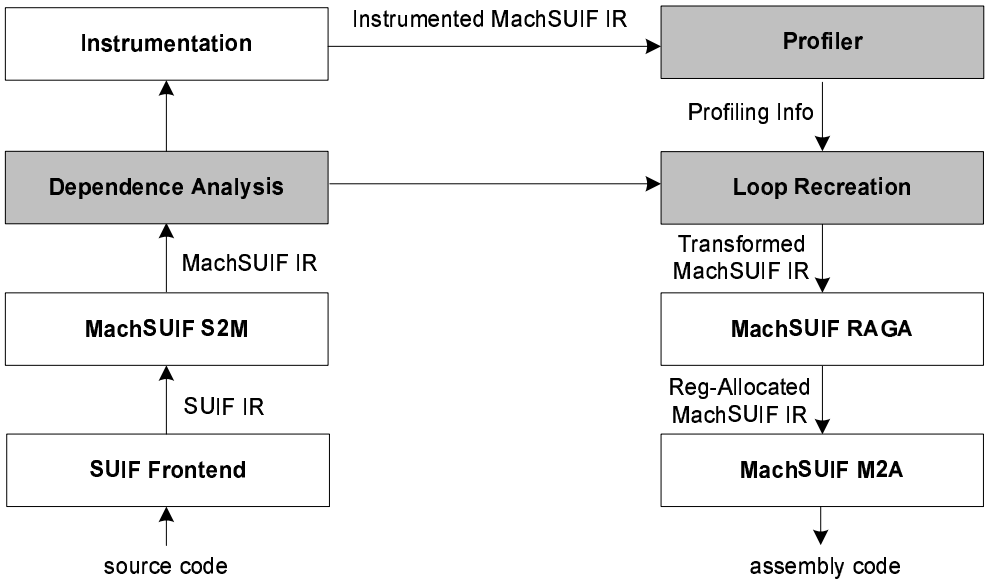


Figure 12: A compilation framework for loop recreation.

5.2. Benchmarks

We examine nine irregular applications to evaluate our work, two benchmarks from [26], two kernel loops from SPEC benchmarks, and five benchmarks from the SparseBench suite.

`Irreg` and `Nbf` are from [26]. `Irreg` is a representative of iterative PDE solvers found in computational fluid dynamics (CFD) applications. In such applications, unstructured meshes are used to model physical structures. The computation kernel is a two-deep nest, where the outer loop is a time loop and the inner loop is parallelized. `Nbf` is a widely used kernel abstracted from the GROMOS molecular dynamics code. It computes a force between two modules and applied to their velocities. The computation kernel is a three-deep nest, where the outermost loop is a time loop and the middle loop enumerates all the atoms available. Its innermost loop is parallelized. For these two benchmarks, the input graphs used to collect profiling information (i.e., the probabilities of memory dependences) are different from the input graphs used in simulation. For `Irreg`, an input graph with 131072 nodes and 3538944 edges is used in simulation. About 1 billion instructions are simulated after about 5 billion instructions have been fast-forwarded. For `Nbf`, an input graph with 87808 nodes and 2370816 edges is used in simulations. About 1 billion instructions are simulated after about 3 billion instructions have been skipped.

`Wave5` and `Fma3d` are two kernel loops from SPEC95 and SPEC2000 benchmarks, respectively. `Wave5` solves Maxwell's equations and particle equations of motion on a Cartesian mesh with a variety of field and particle boundary conditions. We have selected loop 120 in subroutine

`parmvr` and will refer to it as `Wave5-120` from now on. To model the fact that there are tens of thousands of calls to `parmvr`, an outer loop with an iteration count of 4000 is added for this kernel loop. `Fma3d` is a 3D inelastic, transient dynamic response simulation code based on finite element analysis. The loop selected, named `Fma3d-NUMP4`, is the one with the upper bound `NUMP4` in subroutine `SCATTER_ELEMENT_NODAL_FORCE_PLATD`, which has 721 invocations. The inner loop, which has eight iterations, are fully unrolled. To simulate these many subroutine invocations, an outer loop with an iteration count of 20 has been added. For these two SPEC benchmark loops, the train inputs are used to collect profiling information while the reference inputs are used in simulations. `Wave5a-120` and `Fma3d-NUMP4` are simulated for about 0.5 billion instructions each after 12 and 15 million instructions have been skipped, respectively.

`Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu` and `Long-vector` are taken from SparseBench [1]. SparseBench is a benchmark suite of iterative solvers for sparse linear systems used in numerical analysis. SparseBench comes with several different matrix storage formats, preconditioners and iterative methods. All our Sparsebench benchmarks use the Diagonal storage format, the ILU preconditioner, and two iterative methods for sparse systems of linear equations: the GMRES (Generalized Minimum Residual) method and the CG (Conjugate Gradients) method. `Reference` is the reference code. `Naive-ilu` is similar to `Reference` except that a naive coding for a regular ILU solver is used. `Long-vector` provides a contiguous storage of diagonals in regular storage (a generalization of diagonal storage). In these three benchmarks, the CG method is employed. On the other hand, `Bulkgmres` and `Classical-gs` use the GMRES method with different GMRES orthogonalization algorithms: the former uses QR factorization after building a Krylov subspace while the latter uses the classical Gram-Schmidt orthogonalization. The parallelized loops reside in the preconditioners of these benchmarks. For all five benchmarks, a $40^3 \times 40^3$ matrix is used in simulations and a $38^3 \times 38^3$ matrix is used to collect profiling information. All SparseBench benchmarks are simulated for about 100 million instructions after about 2.2 billion instructions have been skipped initially.

Note that all the loops being parallelized except those in `Naive-ilu` have one single basic block each. Therefore, existing whole-program techniques such as [11, 22] will parallelize them by mapping their loop iterations directly into threads as Par does. `Naive-ilu` consists of four control-intensive loops. Existing whole-program techniques may parallelize it as Par does due to the high dependence probability between adjacent loop iterations or in the same way as LRT does if they use the cost model introduced in this work for LRT. Once again, value prediction is ineffective for all these applications since memory accesses are made mostly via subscripted subscripts and there are no obvious patterns among the values accessed. Furthermore, in each loop nest being parallelized, no DOALL loops are found.

Note that for all simulations in this work, each program is run for the same set of (consecutive) iterations of the threaded loops generated by three different methods, LRT, Par and SPT, as discussed in Section 5.3. As a result, a fair comparison for their performance results can be made.

5.3. Methods Compared

We evaluate the performance improvements of LRT over two methods, Par and SPT as described in Section 1. We will also give the speedups of LRT over sequential programs (denoted by Non).

Table I. Configuration of SpMT architecture.

Parameter	Value
Fetch, Issue, Commit	bandwidth 4, out-of-order issue
LD/ST queue	32/32
Function Units	4 int, 2 fp, 1 branch
Branch Prediction	2K BTB, mispred. penalty 11 cycles
L1 I-Cache	16KB, 4-way, 1 cycle (hit)
L1 D-Cache	16KB, 4-way, 3 cycles (hit)
MDT [14]	16KB, 4-way, 2 cycles (L1 to MDT)
L2 Cache (shared)	1MB, 4-way, 12 cycles (hit), 80 cycles (miss)
Local Register File	1 cycle
Interconnect Latency	3 cycles
Spawn Overhead	5 cycles
Squash Overhead	15 cycles

For Par and LRT, the pre-fork region of a parallelized loop serves only to compute the value of the loop variable for the successor thread. On the other hand, SPT may keep more instructions in the pre-fork region to further reduce misspeculation penalties as described in [7].

In our simulation of SPT, full rather than partial re-execution is used. in [7], SPT is guided by a misspeculation cost model to parallelize a given loop. In particular, the ratio of pre-fork/post-fork is a user-controlled rather than compiler-determined parameter. In our experiments, the best solutions it could ideally generate for all nine programs are experimentally found and used. For `Irreg`, `Nbf`, `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu` and `Long-vector`, SPT generates the same parallelized loops as Par. In each case, the pre-fork region serves only to compute the value of loop variable for the successor thread. Therefore, the results for SPT and Par as depicted in Figures 13 and 15(c) – (i) are identical. For `Wave5-120`, there are six sections of code exhibiting the same dependence patterns as those in our motivating example (given in Figure 2). SPT achieves the best result when the ratio of pre-fork/post-fork is 1/5, in which case about half of a code section in two of the six sections are moved into the pre-fork region. For `Fma3d-NUMP4`, the number of code sections with similar dependence patterns as those in the motivating example is 12. The four of these code sections end up each being split evenly in the pre-fork and post-fork regions. So the ratio of pre-fork/post-fork also happens to be 1/5.

5.4. SpMT Architecture Simulated

We consider a multicore SpMT system where all cores are connected through a unidirectional ring-type network. Each core has its private function units, register file, L1 instruction data and L1 data cache. All the cores share a common L2 unified cache. Each core is capable of executing the Alpha ISA with the main parameters of the architecture listed in Table I.

The register dependences are synchronized by asynchronous communication through the ring bus between two cores as in Voltron's queue model [33]. As in Section 2, a pair of `post` and `wait` instructions is inserted by the compiler to communicate a register value defined in the post-fork region between adjacent cores. The interconnect latency is 3 cycles: 1 for `post`, 1 per hop to transmit the value and 1 for `wait`. The `fork` instruction uses the same queue model for

forwarding inter-iteration register values. A latency of 5 cycles for "Spawn Overhead" is assumed for the spawning of a thread. This suffices for LRT and Par since `fork` forwards only the value of the loop variable to the successor core but may be an underestimate for SPT for some loops.

In our simulator (developed based on SimpleScalar), we detect misspeculated dependences using the memory disambiguation table (MDT) [14], which sits between the L1 data cache and L2 cache. When there is a memory-based data dependence violation, MDT will detect the violation and notify the misspeculated thread to be squashed. The latency for accomplishing this is 15 cycles as given in Table I, including the time on sending a squash signal to the misspeculated thread, the time on flushing its speculative writes and related entries in the MDT, the time on squashing all more speculative threads, and the time on restoring the machine state for thread re-execution (with some of these being done in parallel). When a thread commits, any dirty line that has not been displaced from the corresponding private L1 cache is flushed. So the commit overhead includes the overhead for updating the system state and writing such lines to the L2 cache.

Before a loop is executed, the registers holding all live-in values for the loop are copied to all cores. This happens only once for a loop since the inter-iteration register dependences in a loop are synchronized. The time spent on this single copy operation associated with a loop is negligible compared to the total execution time of the loop.

6. Experimental Results

We first compare all methods by presenting some static and dynamic statistics about our benchmarks. We then demonstrate and analyze the performance advantages of LRT over Par and SPT under two squash mechanisms described in Section 2. The performance results are obtained and compared on two-, four-, six- and eight-core systems. In the figures presented below, L stands for LRT, P for Par, S for SPT, N for Non, EA for "Eager Squash", and LA for "Lazy Squash".

6.1. Benchmark Statistics

6.1.1. Compiler-Time Statistics

Table II presents some compile-time statistics about the nine benchmarks used in our experiments. Columns 2 – 5 are concerned with the time complexity of LRT while Columns 6 – 9 compare all the methods in terms of their misspeculation probabilities. As expected, N_{reg} is small for all nine benchmarks. For `Irreg`, `Nbf`, `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu`, and `Long-vector`, $\mathcal{C}_{\mathrm{AB}}(G)$ is empty. So the numbers of flow networks built for each of these seven benchmarks is no larger than $N_{\mathsf{reg}} + 1 = 4$. For `Wave5-120` and `Fma3d-NUMP4`, the number of flow networks built by LRT is 12288 each. However, by applying the pruning strategy discussed in Sections 4.1.4 and 4.2, LRT is efficient in finding the optimal solution in each case. The compile times by LRT for `Wave5-120`, `Fma3d-NUMP4`, `Irreg`, `Nbf`, `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu`, and `Long-vector` are 54.93, 33.25, 1.71, 1.83, 7.04, 7.02, 7.03, 3.26 and 7.02 msecs, respectively. So our algorithm is efficient for handling loops in real applications.

According to Columns 6 – 9, the misspeculation probabilities $\mathcal{P}(D)$ of the original loops in the nine benchmarks are high. While Par never changes $\mathcal{P}(D)$, SPT is effective only for `Wave5-120`. On the other hand, LRT has significantly reduced $\mathcal{P}(D)$ for each benchmark. To get a feel about

Table II. Compile-time statistics for benchmarks.

Benchmark	#Nodes	#Edges	$\lvert\mathcal{C}_{\mathrm{AB}}(G)\rvert$	N_{reg}	$\mathcal{P}(D)$	$\mathcal{P}(\widehat{\mathcal{D}})$		
					Non	LRT	SPT	Par
Wave5-120	796	1572	12	2	0.82	0.05	0.76	0.82
Fmda3d	868	1864	12	2	0.99	0.00	0.99	0.99
Irreg	85	173	0	2	0.85	0.19	0.85	0.85
Nbf	80	189	0	2	0.87	0.04	0.87	0.87
Bulkgmres	178	610	0	3	1	0.28	1	1
Classical-gs	178	610	0	3	1	0.28	1	1
Reference	178	610	0	3	1	0.28	1	1
Naive-ilu	89	223	0	3	0.97	0.33	0.97	0.97
Long-vector	175	605	0	3	1	0.27	1	1

the optimal loop partitions found for the nine benchmarks, the size of G_{top} (relative to that of a single iteration) is 47% for `Wave5-120`, 67% for `Fma3d-NUMP4`, 28% for `Irreg`, 11% for `Nbf`, 76% for `Bulkgmres`, 76% for `Classical-gs`, 76% for `Reference`, 81% for `Naive-ilu`, and 76% for `Long-vector`.

6.1.2. Runtime Statistics

Figure 13 compares the thread squash ratios of a program parallelized by LRT, Par and SPT under the LA and EA squash mechanisms. The squash ratio of a program is referred to as the percentage of the number of squashed threads over the total number of spawned threads. In each benchmark, a thread is squashed mainly due to data dependence violations. However, some threads are also squashed when the back edge of a parallelized loop is violated, which is the only form of control dependence violations in this work.

6.1.2.1. Misspeculation As mentioned in Section 5.3, the pre-fork region of a loop parallelized by LRT and Par serves only to compute the value of its loop variable for the successor thread. So the squash ratio difference between LRT and Par represents a rough estimate of the misspeculation possibility dynamically reduced by LRT for the loop. This degree of runtime reduction is consistent with the misspeculation possibility statically reduced by LRT for the loop given in Table II.

The parallelized loops for `Fma3d-NUMP4`, `Irreg`, `Nbf`, `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu`, and `Long-vector` by LRT are free of inter-thread memory dependences. Thus, LRT has the smallest squash ratio among all methods compared under either squash scheme. In addition, its squash ratios (due to mainly control misspeculations) are relatively small in all programs. The largest LRT squash ratio are in `Nbf`, which attracts a squash ratio of 4% with two cores and of 20% when eight cores are used. These squashes are caused by control misspeculations since the iteration count of the parallelized loop, which is nested inside two other loops, is small. In general, Par suffers the highest squash ratios for all programs in all configurations. Note that Par and SPT have parallelized `Nbf`, `Irreg`, `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu`, and `Long-vector` in exactly the same way, as explained earlier.

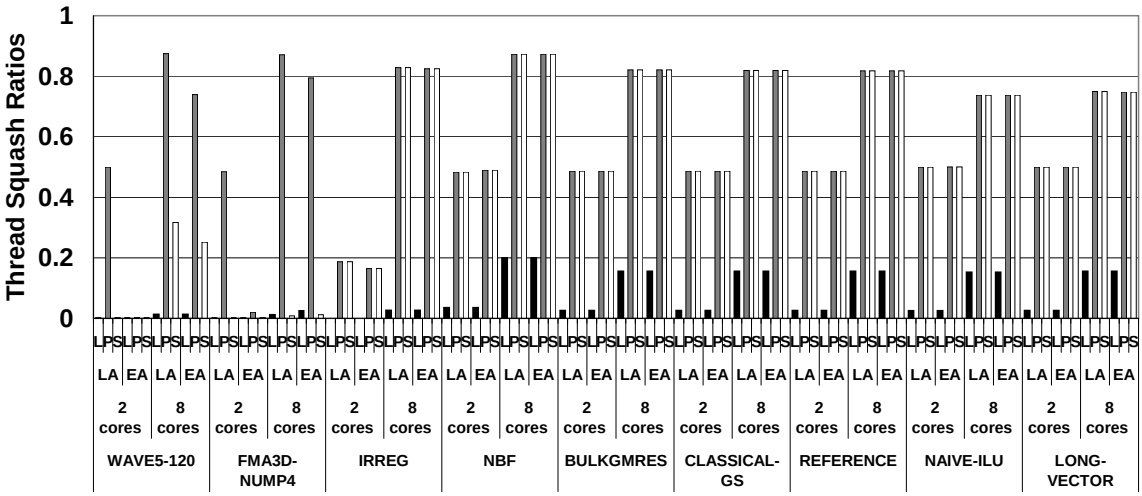


Figure 13: Misspeculation penalties of LRT, Par and SPT.

6.1.2.2. Synchronization Initially, all sequential loops in our benchmarks are free of inter-iteration register dependences. Therefore, there are no synchronized register dependences in the parallelized loops generated by Par and SPT. So these two algorithms need not to be analyzed.

In all the nine benchmarks except `Fma3d-NUMP4`, LRT has transformed some intra-iteration register dependences into inter-iteration register dependences. Figure 14 shows the synchronization costs incurred by LRT. `Fma3d-NUMP4` is synchronization-free. The synchronization costs in `Wave5-120`, `Irreg` and `Nbf` are small, representing less than 4.3% of their total execution times. In the remaining five benchmarks, `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu` and `Long-vector`, relatively larger synchronization costs are observed. In each of these five benchmarks, LRT has introduced an inter-iteration register dependence with a relatively large weight (indicating the desired synchronization delay) in order to avoid speculating some inter-iteration memory dependences in the original loop that always or nearly always happen. Despite the large synchronization costs incurred in these five benchmarks, LRT outperforms Par and SPT as discussed shortly below due to the excessive misspeculation overhead successfully eliminated by LRT (Figure 13). These results show that the optimal solution found by LRT for a program tends to minimize the total synchronization and speculation overhead for the program.

The results under both squash schemes are similar for LRT due to similar misspeculation statistics observed.

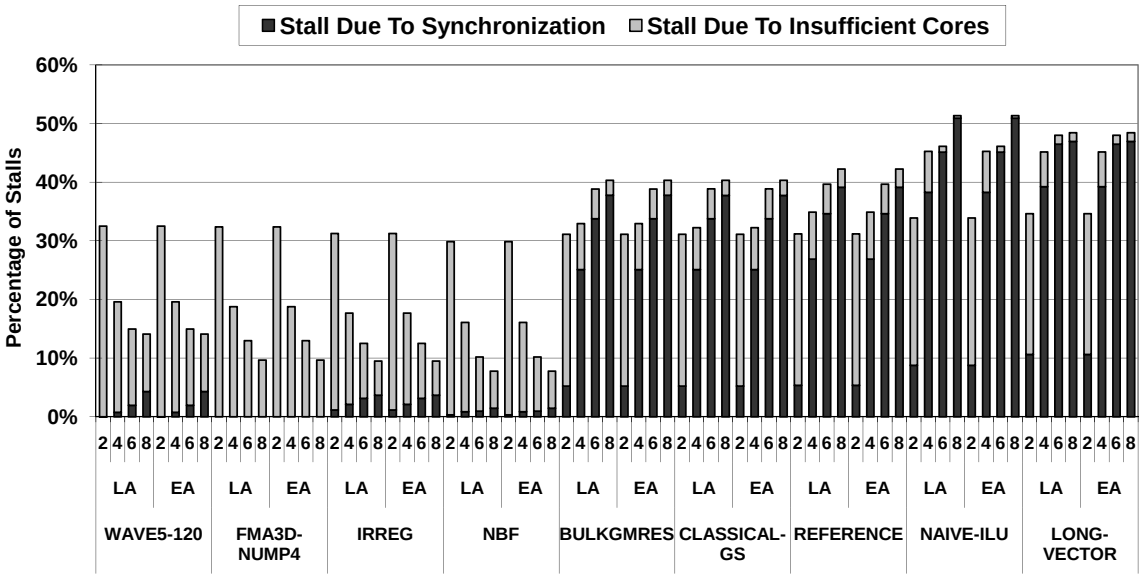


Figure 14: Per-thread synchronization penalties for LRT.

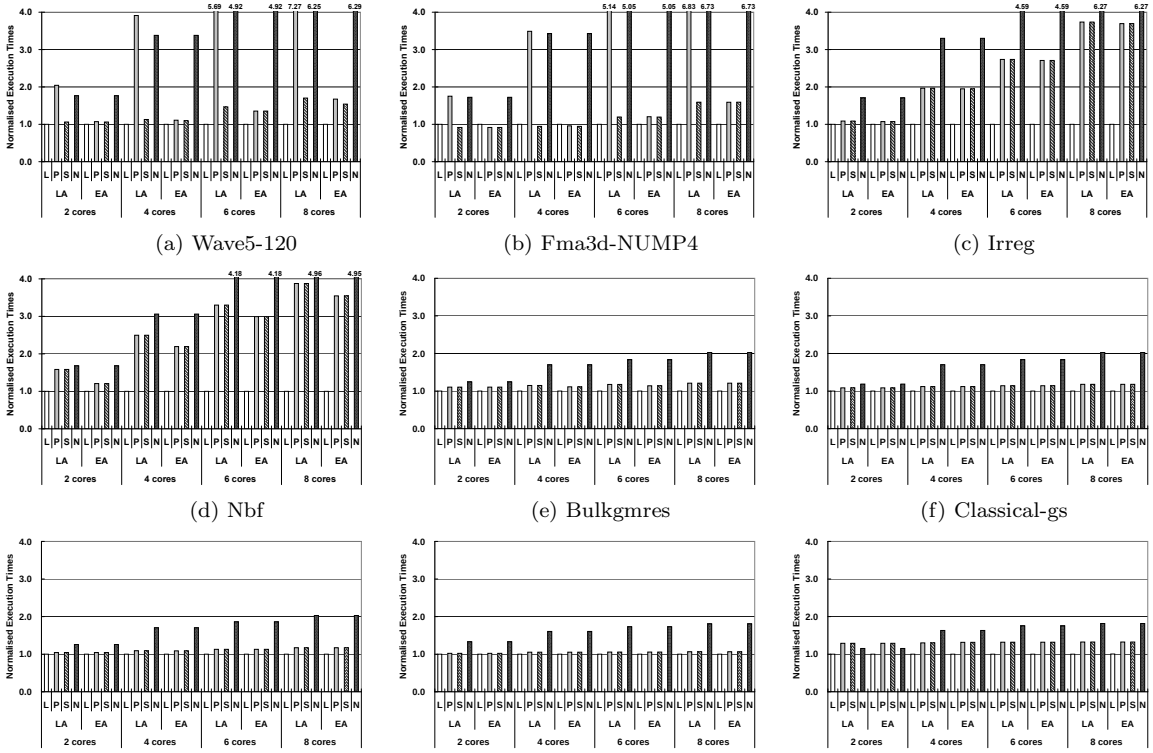
6.2. Speedups and Analysis

6.2.1. Speedups

Figure 15 compares all the methods in terms of their performance results. All the execution times are normalized with respect to LRT. So the execution time for a particular method represents the speedup of LRT over that method. As a result, when comparing the execution time bars of a pair of methods in a configuration, the method with a lower bar generates faster code and is thus better than the other method.

First of all, we observe that LRT improves scalably the execution time of every sequential program. Note that the performance improvements (of LRT over Non) under both squash mechanisms are nearly the same. Thus, the normalized execution times presented in Figure 15 happen to also allow us to find out how well a particular method works for a program under the two different squash mechanisms.

Next, let us take a look at the performance improvements of LRT over Par. For every benchmark, the speedup of LRT over Par under each squash scheme generally increases as the number of cores increases. In general, Par performs better under EA than under LA. In the case of `Wave5-120` and `Fma3d-NUMP4`, Par has almost degenerated into Non. In the case of `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu` and `Long-vector`, the dependences that cause most misspeculations are located at the end of each parallelized loop. Therefore, the Par results under EA and LA are nearly



(a) Wave5-120 (b) Fma3d-NUMP4 (c) Irreg

(d) Nbf (e) Bulkgmres (f) Classical-gs

(g) Reference (h) Naive-ilu (i) Long-vector

Figure 15: Normalized execution times (with respect to LRT).

the same for these five benchmarks. However even when EA is assumed, LRT outperforms Par in nearly all configurations. The speedups of LRT over Par on an eight-core system for `Wave5-120`, `Fma3d-NUMP4`. `Irreg`, `Nbf`, `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu` and `Long-vector` are 1.68, 1.59, 3.69, 3.55, 1.21, 1.18, 1.17, 1.06 and 1.32, respectively. Due to cache effects (as explained in Section 6.2.2), some slight performance slowdowns are observed when `Fma3d-NUMP4` is run on two- and four-core systems.

Finally, let us examine the performance improvements of LRT over SPT. SPT achieves nearly the same results for all the nine programs except `Nbf` under both squash mechanisms. When LA is used, the speedups of LRT over SPT on an eight-core system for `Wave5-120`, `Fma3d-NUMP4`, `Irreg`, `Nbf`, `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu` and `Long-vector` are 1.70, 1.59, 3.74, 3.88, 1.21, 1.18, 1.17, 1.06 and 1.32 respectively. If EA is used instead, these numbers become 1.54, 1.59, 3.69, 3.55, 1.21, 1.18, 1.17, 1.06 and 1.32, respectively. Again, due to cache effects, some slight performance slowdowns are observed when `Fma3d-NUMP4` is run on two- and four-core systems.

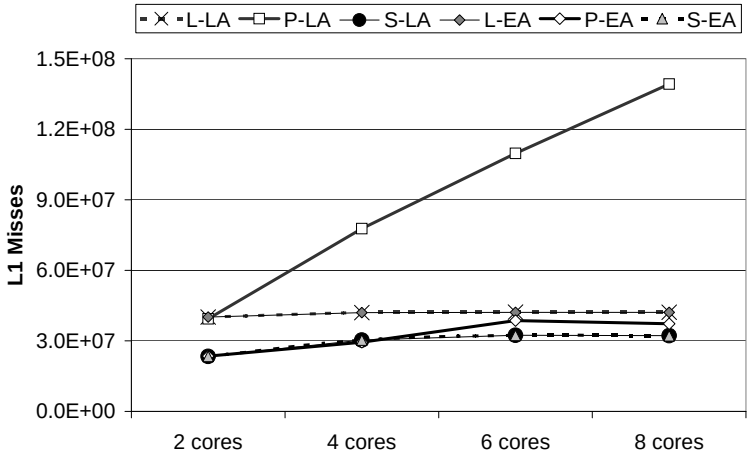


Figure 16: L1 data cache misses of `Fma3d-NUMP4`.

6.2.2. Cache Effects

When `Fma3d-MUMP4` is run on two- and four-core systems, LRT performs equally as or slightly worse than Par under EA and than SPT under both LA and EA. As mentioned in Section 5.2, the `Fma3d-MUMP4` loop is parallelized only after its eight-iteration inner loop has been fully unrolled. This creates possibly accesses to all fields of eight different structure elements of an array, named `FORCE`. In contrast with Par and SPT, LRT forms a new loop iteration from the instructions in two adjacent iterations in the `Fma3d-MUMP4` loop. As a result, LRT has happened to decrease the amount of spatial reuse among these accesses in the L1 data cache private to each core. The performance slowdowns of LRT in two- and four-core cases are due to increased L1 data cache misses shown in Figure 16. As the number of cores increases, LRT outperforms Par and SPT since the more parallelism exposed by LRT has significantly more than offset the cache effects. The lack of sufficient cores makes it difficult to harness the amount of parallelism exposed by our loop recreation technique, as indicated in Figure 14. We have verified that LRT will slightly outperform Par and SPT if the extra L1 cache misses had not occurred in the two- and four-core configurations.

6.2.3. Eager Squashes vs Lazy Squashes

Now, let us examine the impact of two squash mechanisms on the performance of all nine benchmarks in Figure 15. Let us consider `Wave5-120` first. Recall that its loop body can be divided into six sections with each sharing the same dependence characteristics as the motivating loop given in Figure 2(b). LRT has parallelized each section as shown in Figure 2(e). As a result, LRT performs similarly under both squash mechanisms since misspeculations are infrequent (as shown

in Figure 13). This fact can be deduced in Figure 15, where the two N bars in each configuration have the same height (up to two decimal points). In the case of Par, the loop body of its parallelized loop consists of a frequently occurring memory dependence in each of its six above-mentioned code sections (as illustrated in Figure 2(c)). Therefore, Par suffers frequent misspeculations under LA (as shown in Figure 13) to the extent that the threads are nearly sequentialized (as shown in Figure 15). From Figure 13, we can see that by squashing misspeculated threads earlier for `Wave5-120` under EA, the squash ratio has been reduced in the two-core case but suffers a slight increase when eight cores are used. In either case, by squashing misspeculated threads and restarting them earlier, more parallelism can be attained. The parallelized `Wave5-120` from Par runs 1.97 (3.95) faster under EA than under LA in the two-core (eight-core) configuration. In the case of SPT, the ratio of pre-fork/post-fork is 1/5. Due to this delay in spawning threads, speculated memory dependences behave similarly in both squash schemes. So the performance variations in both cases are small.

The situation for `Fma3d-NUMP4` is similar to that for `Wave5-120`.

For `Irreg` and `Nbf`, LRT behaves identically under both squash schemes since its parallelized loops for both programs are free of inter-thread memory dependences. Both Par and SPT generate the same parallelized code. So their performance results for each program are identical. Let us examine Par. The dependence violations for `Irreg` usually happen at the end of a loop iteration. So small performance variations are observed under both squash schemes. The dependence violations for `Nbf` happen slightly earlier in a loop iteration. As a result, Par performs better under EA than under LA for `Nbf`. The performance results for `Bulkgmres`, `Classical-gs`, `Reference`, `Naive-ilu` and `Long-vector` are similar to that for `Irreg`.

7. Related Work

Loop recreation works on any architecture that provides hardware support for speculative multithreading (SpMT). Therefore, we will review only some compiler techniques related to this work. The seminal work known as Multiscalar on SpMT started more than one decade [8]. Since then, a large amount of research work has been done to exploit parallelism from sequential programs. We first review loop-oriented speculative parallelization techniques and then general-purpose ones. To the best of our knowledge, most existing loop-oriented techniques such as [7, 24, 25, 30, 32, 31, 34] turn loop iterations into different speculative threads. For SpMT systems that support value synchronization, frequently occurring dependences are synchronized [24, 32, 31]. The `post` and `wait` instructions associated with a synchronized dependence are moved as close as possible. As a result, the time for communicating the required values can be reduced. In this work, we have adopted the technique described in [32] to insert the `post` and `wait` instructions required for synchronized register dependences.

Loop transformations and optimizations may be applied to uncover the loop-level parallelism hindered by some inter-iteration dependences. The SPT compiler [7] attempts to pre-compute some inter-iteration dependences by moving the producer instructions of the dependences into the pre-fork region (subject to their cost model), thereby reducing squashes caused by frequently occurring inter-thread dependences. In [34], some loop transformations such as loop fission are employed to remove, isolate or pre-compute the inter-iteration dependences that are likely to be misspeculated. In [24, 31], the researchers optimize speculative threads by move the instructions

associated with a frequently occurring inter-iteration dependence as close as possible. However, all these transformations and optimizations do not attempt to alter the nature of a dependence. This means that whether a dependence is an intra- or inter-iteration dependence is not changed.

The work of [28] (Speculative DSWP) represents a different way to extract threads from loops. It pipelines one single loop iteration across multiple cores so that same code slice of all loop iterations (a pipeline stage) forms a long-running thread. Misspeculated long-running threads are rolled back by means of checkpointing and versioned memory. Hence, load balance is crucial for their technique. By allowing a pipeline stage itself to be parallelized, their later work [23] alleviates part of the load balancing problem for non-speculative parallelization. In [4], an extension of their parallel-stage DSWP to SpMT is briefly mentioned, but the required hardware support on misspeculation detection and recovery is much more complicated than that introduced in [28]. Sequential and parallel stages have to be handled differently due to the acyclic dependences assumed among the sequential stages and the round-robin distribution of the instances of a parallel stage. Furthermore, parallel stages may need to be grouped together if more than one stage is parallelized. Note that LRT can also be used to improve the speculative parallelism inherent in their parallel stages if they are allowed to be speculatively parallelized. In general, as mentioned earlier, LRT is typically used as a prepass or postpast to further improve such existing TLS techniques.

Some general-purpose compiler techniques [3, 11, 22, 29, 16, 13] can walk through the CFG of a program and form threads at the boundaries of control flow edges. Let us examine how loops are handled. The earlier algorithm used in the Multiscalar project [29] forms threads only at loop boundaries. The follow-up work [3, 16, 13] may allow large loop iterations to be sliced into multiple threads but loop boundaries remain to be thread boundaries. The Mitosis compiler [22] and the work [11] may turn some basic blocks in a loop into a thread. But they are not designed to specifically maximize the speculative parallelism in loops. For instance, when a loop has one single basic block, they will still restrict threads to loop boundaries.

Both loop-oriented or general-purposed speculative parallelization techniques may be augmented by some TLS enhancement techniques. Value prediction techniques [7, 22, 16, 13] are used to predicate some live-in values for a thread to reduce misspeculation penalties. For example, once a thread is created, the Mitosis compiler [22] will generate a piece of code (called P-slice) to predict the live-in values for each speculative thread. P-slices are not necessarily part of the original program and the values they produce do not have to be correct. However, these techniques may not be effective for irregular applications accessing arrays via pointers and indirection arrays.

Helper threads [6, 15, 17, 35] are used to speculatively execute of a code region to prefetch some expensive instructions (i.e. instructions with large latency value) in the region. A helper thread for a loop may be formed from any of its instructions in any order since the helper thread does not have to be concerned with program correctness. However, as a loop transformation, loop recreation is correctness-preserving.

8. Conclusion

The development of speculative parallelization techniques for improving the performance of sequential programs is very challenging. In this paper, we present a new compiler technique, called loop recreation, for restructuring a loop into a prologue, a kernel loop, an epilogue so that

the kernel can yield higher speculative loop-level parallelism than the original loop on SpMT architectures. We present an algorithm for finding an optimal loop recreation transformation with respect to a simple cost model. We demonstrate significant performance advantages of loop recreation over some recent techniques using nine representative irregular applications. Our work is orthogonal to many existing TLS techniques and can thus be implemented as a prepass or postpass to enhance these existing techniques.

REFERENCES

1. Sparsebench. `http://www.netlib.org/benchmark/sparsebench/`.
2. Saisanthosh Balakrishnan and Gurindar S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 302–313, Washington, DC, USA, 2006. IEEE Computer Society.
3. Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreaded processors. *IEEE Trans. Parallel Distrib. Syst.*, 15(8):713–724, 2004.
4. Matthew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David I. August. Revisiting the sequential programming model for the multicore era. *IEEE Micro*, 28(1):12–20, 2008.
5. Michael K. Chen and Kunle Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 434–446. ACM, 2003.
6. Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, New York, NY, USA, 2001. ACM Press.
7. Z. H. Du, C. Ch. Lim, X. F. Li, C. Yang, Q. Zhao, and T. F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of Conference on Programming Language Design and Implementation*, 2004.
8. M. Franklin. *The Multiscalar Architecture*. PhD thesis, The University of Wisconsin at Madison, 1993.
9. Lin Gao, Lian Li, Jingling Xue, and Tin-Fook Ngai. Loop recreation for thread-level speculation. In *Parallel and Distributed Systems, 2007 International Conference on*, 2007.
10. Andrew Goldberg. Network optimization library. 2001.
11. T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of Conference on Programming Language Design and Implementation*, 2004.
12. Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 59–70. ACM, 2004.
13. Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 205–214. ACM, 2007.
14. V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Proceedings of the 12th International Conference on Supercomputing*, pages 85–92. ACM Press, 1998.
15. Steve S.W. Liao, Perry H. Wang, Hong Wang, Gerolf Hoflehner, Daniel Lavery, and John P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2002. ACM.
16. Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. Posh: a tls compiler that exploits program structure. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167. ACM, 2006.
17. Chi-Keung Luk. Tolerating memory latency through Software-Controlled Pre-Execution in simultaneous multithreading processors. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, New York, NY, USA, 2001. ACM Press.
18. Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, New York, NY, USA, 1997. ACM Press.

19. Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
20. Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, 1997.
21. Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303, 1999.
22. C. G. Quinones, C. Madrile, J. Sanchez, P. Marcuello, A. Gonzalez, and D. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In *Proceedings of Conference on Programming Language Design and Implementation*, 2005.
23. Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, New York, NY, USA, 2008. ACM.
24. J. G. Steffan, C.B.Colohan, A.Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA '08: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002.
25. J. Y. Tsai and P. Ch. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *International Conference on Parallel Architecture and Compiler Techniques*, pages 35–46, 1999.
26. Chau-Wen Tseng. Exploiting locality for irregular scientific codes. *IEEE Trans. Parallel Distrib. Syst.*, 17(7):606–618, 2006. Member-Hwansoo Han.
27. Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, 2007.
28. Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.
29. T. N. Vijaykumar. *Compiling for the multiscalar architecture*. PhD thesis, 1998. Supervisor-Gurindar S. Sohi.
30. S. Y. Wang, X. R. Dai, K. S. Yellajyosula, A. Zhai, and P. Ch. Yew. Loop selection for thread-level speculation. In *The 18th International Workshop on Languages and Compilers for Parallel Computing*, 2005.
31. A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of memory-resident value communication between speculative threads. In *International Symposium on Code Generation and Optimization*, 2004.
32. Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 171–183. ACM, 2002.
33. Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *HPCA*, pages 25–36, 2007.
34. Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA '08: Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.
35. Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, New York, NY, USA, 2001. ACM Press.

APPENDIX A. Optimality of $\mathsf{LRT}_{\mathrm{mem}}$

Lemma 3. *Let $G_f = (V_f, E_f, C_f) \in \mathcal{F}$. If $\{V_{\mathrm{top}}, V_{\mathrm{bot}}\}$ is a legal loop recreation for G, then $(V_{\mathrm{top}} \cup \{s\}, V_{\mathrm{bot}} \cup \{t\})$ is a s-t cut that is free of ∞-weighted cut edges in G_f, and conversely.*

Proof. By Lemma 1, G_f is constructed so that if $u \to v$ is an intra-iteration dependence in G (or equivalently in G_{ln}), $C_f(v, u) = \infty$ (line 8). Hence, "$\Longrightarrow$" holds. To prove "$\Longleftarrow$", we note further

that no s-t cut (S,T) in G_f that is free of ∞-weighted cut edges can include $v \rightarrow u$ as a cut edge. In other words, $u \rightarrow v \notin (T,S)$ holds. Hence, $\{S \setminus \{s\}, T \setminus \{t\}\}$ is legal for G. □

Theorem 1. *In* $\mathsf{LRT}_{\mathrm{mem}}$*,* $\{S_{\mathrm{opt}} \setminus \{s\}, T_{\mathrm{opt}} \setminus \{t\}\}$ *returned by* Process_FN *is optimal.*

Proof. $\mathsf{LRT}_{\mathrm{mem}}$ is developed under the assumption that $\mathcal{D} = \mathcal{D}_{\mathrm{mem}}$. Due to (5) and Lemma 3, finding an optimal loop recreation in G amounts to solving min-cut problems for the flow networks in $\mathcal{F}$. Let $G_f \in \mathcal{F}$ a flow network. Let G'_f be defined in line 3 of Process_FN. It suffices to show that no inter-iteration dependence has had its weight counted more than once in $(S_{\mathsf{opt}}, T_{\mathsf{opt}})$, which is the minimum cut found, say, in $G_{\mathrm{opt}} \in \mathcal{F}$. Assume to the contrary that there exists a dependence, $u \rightarrow v$, that is multiply counted, which must happen as shown in Figure 6(a.4) or (b.4). For reasons of symmetry, let us assume the former is the case. Then $(S_{\mathsf{opt}}, T_{\mathsf{opt}})$ must include both $s \rightarrow v$ and $u \rightarrow t$. Let $G'_{\mathsf{opt}} \in \mathcal{F}$ be another flow network that is the same as G_{opt} except that $u \rightarrow v$ is duplicated as shown in Figure 6(b.4). Then $(S'_{\mathsf{opt}}, T'_{\mathsf{opt}})$, which is derived from $(S_{\mathsf{opt}}, T_{\mathsf{opt}})$ with its $s \rightarrow v$ and $u \rightarrow t$ removed and $u \rightarrow v$ added, must be an s-t cut as illustrated in Figure 6(b.3). This implies that $(S'_{\mathsf{opt}}, T'_{\mathsf{opt}})$ has a smaller capacity than $(S_{\mathsf{opt}}, T_{\mathsf{opt}})$, i.e., $\mathrm{CAP}(S'_{\mathsf{opt}}, T'_{\mathsf{opt}}) < \mathrm{CAP}(S_{\mathsf{opt}}, T_{\mathsf{opt}})$ since the weight of every duplicated edge is positive by Definition 3. A contradiction has been reached. Therefore, The loop recreation $\{S_{\mathrm{opt}} \setminus \{s\}, T_{\mathrm{opt}} \setminus \{t\}\}$ returned by Process_FN is optimal. □

APPENDIX B. Optimality of LRT

Theorem 2. *In* LRT*,* $\{S_{\mathrm{opt}} \setminus \{s\}, T_{\mathrm{opt}} \setminus \{t\}\}$ *returned by* Process_FN *is optimal.*

Proof. Applying Lemma 1 and proceeding exactly as in the proof of Theorem 1, we find that every memory dependence in $\widehat{\mathcal{D}}_{\mathrm{mem}}$ has exactly one copy in $(S_{\mathsf{opt}}, T_{\mathsf{opt}})$. Let $(S_{\mathsf{opt}}, T_{\mathsf{opt}})$ be a minimum cut found in a flow network contained in $\mathcal{F}_{l_i}$. It suffices to show that l_i is the smallest synchronization delay incurred. Assume to contrary that there exists another minimum cut $(S'_{\mathsf{opt}}, T'_{\mathsf{opt}})$ in some $\mathcal{F}_{l_j}$ satisfying $\mathrm{CAP}(S'_{\mathsf{opt}}, T'_{\mathsf{opt}}) < \mathrm{CAP}(S_{\mathsf{opt}}, T_{\mathsf{opt}})$ such that $l_j < l_i$. If this were the case, then $(S'_{\mathsf{opt}}, T'_{\mathsf{opt}})$ would have been returned as the optimal solution by Process_FN. □