Loop Recreation for Thread-Level Speculation

Lin Gao, Lian Li and Jingling Xue
University of New South Wales, Australia

Tin-Fook Ngai
Microprocessor Technology Lab, Intel

Abstract

For some sequential loops, existing techniques that form speculative threads only at their loop boundaries do not adequately expose the speculative parallelism inherent in them. This is because some inter-iteration dependences, which translate into inter-thread dependences at run time, are too costly to synchronize or speculate. This paper presents a novel compiler technique, called loop recreation, to transform a loop into a prologue, a kernel loop — formed with instructions from two adjacent iterations, and an epilogue so that the inter-iteration dependences in the kernel are less costly to enforce at run time than those in the original loop. We prove the concept by giving an algorithm for finding an optimal loop recreation with respect to a simple misspeculation cost model and by demonstrating performance advantages of loop recreation over two recent techniques for speculative multi-core systems running four irregular applications with indirect array accesses.

1. Introduction

The emerging hardware support for speculative multithreading (SpMT) or thread-level speculation allows the compiler to optimistically create speculatively parallel threads for sequential applications 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 and subscripted subscripts). To achieve good performance on SpMT architectures, the management of inter-thread dependences is crucial.

Maximizing speculative thread-level parallelism in loops can often achieve significant performance improvements in many applications. If we turn loop iterations directly into speculative threads as in existing loop-oriented compiler techniques [3, 10, 11, 13, 14, 15], some inter-iteration dependences in the loop, which become inter-thread dependences at run time, can be too costly to enforce. Furthermore, value prediction may not be effective for irregular loops accessing arrays with

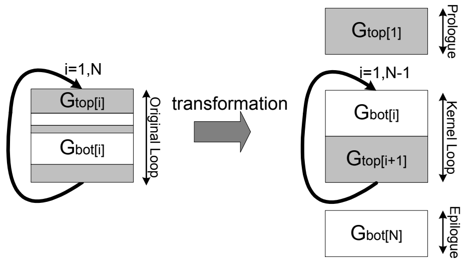
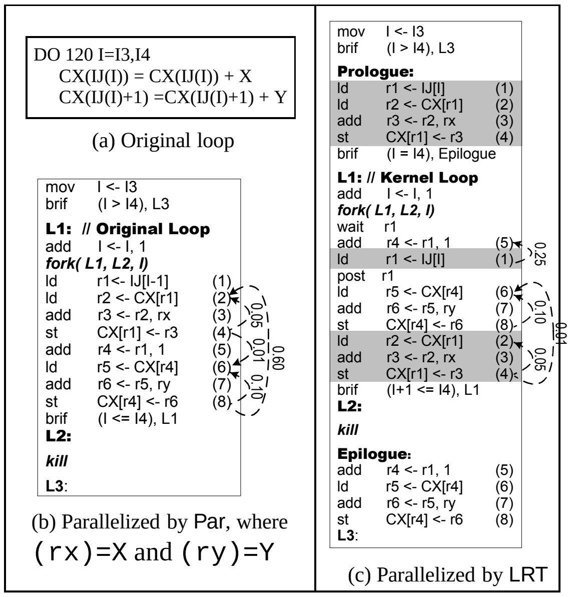


Figure 1. Loop recreation.

pointers and subscripted subscripts. Therefore, loop boundaries are not good to serve as thread boundaries for some loops. Existing compiler techniques [6, 9] for whole programs allow thread boundaries to be formed at control flow edges. But they are not designed specifically to maximize the speculative parallelism in loops. For example, in the important special case when a loop consists of one single basic block, loop boundaries will remain to be thread boundaries.

In this paper, we present a novel compiler technique, called *loop recreation* and illustrated in Figure 1, to speculatively parallelize sequential loops. Our loop recreation transformation is denoted LRT. LRT amounts to finding a partition $G_{\rm par} = \{G_{\rm top}, G_{\rm bot}\}$ of the set of instructions in a loop (with G denoting its data dependence graph) and then transforming the loop into a prologue, a kernel loop and an epilogue. The 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. When the resulting kernel is speculatively executed, its inter-iteration dependences will be less costly to enforce at run time than those in the original loop. As a result, the speculative thread-level parallelism (TLP) in the kernel is improved. A motivating example is presented in Figure 2.

The rest of this paper is organized as follows. Section 2 describes the loop and execution models used. Section 3 presents a loop recreation algorithm. In Section 4, we describe the compiler framework in which our



```
DO 120 I=I3,I4
    CX(IJ(I)) = CX(IJ(I)) + X
    CX(IJ(I)+1) =CX(IJ(I)+1) + Y
```

(a) Original loop

```
mov     I <- I3
brif    (I > I4), L3
L1:  // Original Loop
add     I <- I, 1
fork( L1, L2, I)
ld      r1<- IJ[I-1]        (1)
ld      r2 <- CX[r1]        (2)
add     r3 <- r2, rx        (3)
st      CX[r1] <- r3        (4)
add     r4 <- r1, 1         (5)
ld      r5 <- CX[r4]        (6)
add     r6 <- r5, ry        (7)
st      CX[r4] <- r6        (8)
brif    (I <= I4), L1
L2:
kill
L3:
```

(b) Parallelized by Par, where `(rx)=X` and `(ry)=Y`

```
mov     I <- I3
brif    (I > I4), L3
Prologue:
ld      r1 <- IJ[I]         (1)
ld      r2 <- CX[r1]        (2)
add     r3 <- r2, rx        (3)
st      CX[r1] <- r3        (4)
brif    (I = I4), Epilogue
L1: // Kernel Loop
add     I <- I, 1
fork( L1, L2, I)
wait    r1
add     r4 <- r1, 1         (5)
ld      r1 <- IJ[I]         (1)
post    r1
ld      r5 <- CX[r4]        (6)
add     r6 <- r5, ry        (7)
st      CX[r4] <- r6        (8)
ld      r2 <- CX[r1]        (2)
add     r3 <- r2, rx        (3)
st      CX[r1] <- r3        (4)
brif    (I+1 <= I4), L1
L2:
kill
Epilogue:
add     r4 <- r1, 1         (5)
ld      r5 <- CX[r4]        (6)
add     r6 <- r5, ry        (7)
st      CX[r4] <- r6        (8)
L3:
```

(c) Parallelized by LRT

Figure 2. Loop recreation for a loop abstracted from loop 120 in subroutine `parmvr` of `wave5` in SPECfp95. In (a), the loop code is given. In (b), the loop in assembly code is parallelized by Par, a technique that maps loop iterations directly into threads. In (c), the loop is parallelized by LRT where $G_{\rm top} = \{1,2,3,4\}$ and $G_{\rm bot} = \{5,6,7,8\}$. Only *inter-iteration dependences* are shown, together with their probabilities. In (c), $(1) \rightarrow (5)$ is a register dependence on `r1`. By (1), LRT has reduced the misspeculation probability of a thread from $1-(1-0.05)(1-0.6)(1-0.01)(1-0.10) = 0.66$ in (b) to $1-(1-0.25)(1-0.10)(1-0.01)(1-0.05) = 0.37$ in (c).

algorithm is implemented. Section 5 presents our experimental results. Section 6 reviews the related work. Section 7 concludes the paper.

2. Loop and Execution Models

As in [3], when parallelizing a loop, we presently consider only its data dependences. In addition, an iteration in a parallelized loop will be divided by a thread-spawning instruction into a pre-fork and a post-fork region. The pre-fork region consists of only instructions for computing the loop index for the successor thread to be spawned by the thread-spawning instruction.

The iterations of a parallelized loop are distributed to cores in an SpMT system 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*. Misspeculated threads are squashed and new threads spawned to re-execute their alloted iterations.

This work is not concerned with deciding which dependences should be synchronized or speculated. All inter-iteration register dependences are synchronized using the technique described in [14]. All inter-iteration memory dependences are speculated.

3. A Loop Recreation Algorithm

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.

3.1. Data Dependence Graph (DDG)

The DDG for a loop is standard except that the weights of dependences are so assigned that a misspeculation cost model can be developed for both synchronized and speculated dependences in a unified manner.

The DDG of a loop is a weighted directed multigraph, denoted $G = (V, E, W)$, where V is the set of instructions in the loop, E is the set of directed edges representing the data dependences between instructions and the weight function W is computed as described shortly.

Inter-iteration anti- and output dependences do not appear in the DDG since they are all automatically enforced by the hardware. However, intra-iteration anti- and output dependences must be included. This is necessary to ensure the legality of a loop recreation transformation (Lemma 1). Their weights are set to 0 because they will not cause any overhead to enforce even when they are transformed into inter-iteration dependences.

For a true dependence $u \rightarrow v$, $W(u, v)$ ranges over $[0, 1]$ and is referred to as its *probability* regardless whether it is a register or memory dependence. The intent is that $W(u, v)$ represents the fraction of a loop iteration (in cycles) that is wasted due to synchronization (misspeculation) if $u \rightarrow v$ is synchronized (speculated).

The weight $W(u, v)$ of a memory dependence (u, v) is set as its dependence probability indicating how often the dependence $u \rightarrow v$ actually takes place at run time.

The weight $W(u, v)$ of a register dependence $u \rightarrow v$ in a loop is defined to represent the incurred communication delay as a fraction of the total execution time for one loop iteration in a parallelized version. Note that u represents a write access and v a read access. In addition, u appears lexically after the read v. Let $\mathcal{I}$ be the set of all instructions that lie on a critical dependence path from v to u in the DDG of a loop. When the loop or its parallelized versions are scheduled, all instructions in $\mathcal{I}$ must be scheduled between v and u. Then the weight $W(u, v)$ is estimated (optimistically) to be

$\max(0, ((\sum_{I\in\mathcal{I}\cup\{u,v\}} C_I) - \textit{Spawn\_Overhead})/C_L)$, where C_I is the number of cycles spent on executing instruction I, C_L is the total number of cycles spent on executing one loop iteration and *Spawn_Overhead* is the number of cycles taken to spawn a new thread.

3.2. A Misspeculation Cost Model

It is difficult, if not impossible, to determine accurately the relative timing of dependent instructions in different threads. As in [3, 9], we make some conservative assumptions when estimating misspeculation overhead: (1) different dependences are independent, and (2) an inter-thread memory dependence is always misspeculated.

As shown in Figure 1, a loop recreation for a loop is uniquely specified by a partition of the set of nodes in its DDG, G, into $G_{\rm par} = \{G_{\rm top}, G_{\rm bot}\}$. The transformed loop by $G_{\rm par}$ is referred to as the *recreated loop*.

Let $\mathcal{D}(G_{\rm par})$ be the set of all inter-iteration dependences in the recreated loop generated by a loop recreation $G_{\rm par} = \{G_{\rm top}, G_{\rm bot}\}$. The misspeculation probability of a thread is approximated by:

$$\mathcal{P}(G_{\rm par}) = 1 - \Pi_{(u,v)\in\mathcal{D}(G_{\rm par})}(1 - W(u,v)) \qquad (1)$$

In practice, if there are several inter-thread register dependences, the communication delayed incurred by a thread is roughly equal to the largest communication delay incurred by one of these dependences. For practical loops with hundreds or even thousands of instructions, the weights of register dependences are small. In this case, our cost model will over-approximate slightly the misspeculation probability for a thread. By over-approximating, we ensure that the recreated loop created by any optimal solution is always no worse than the original one (with respect to this cost model used).

3.3. Algorithm

The key idea behind a loop recreation $G_{\rm par} = \{G_{\rm top}, G_{\rm bot}\}$ 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 $G_{\rm top}$ to $G_{\rm bot}$ is increased by 1 while the distance of a dependence from $G_{\rm bot}$ to $G_{\rm top}$ is decreased by 1. Hence, the following two results are immediate.

Lemma 1 *A loop recreation $G_{\rm par} = \{G_{\rm top}, G_{\rm bot}\}$ for a loop is legal if and only if there are no intra-iteration dependences pointing from $G_{\rm bot}$ to $G_{\rm top}$.*

Lemma 2 *A loop recreation $G_{\rm par} = \{G_{\rm top}, G_{\rm bot}\}$ for a loop affects its dependence $u \to v$ spanning $G_{\rm top}$ and $G_{\rm bot}$ as follows. If $u \to v$ is an intra-iteration (inter-iteration) dependence pointing from $G_{\rm top}$ ($G_{\rm bot}$) to $G_{\rm bot}$ ($G_{\rm top}$) in the loop, then $u \to v$ becomes an inter-iteration (intra-iteration) dependence in the recreated loop. Otherwise, an intra-iteration (inter-iteration) dependence in the loop remains so in the recreated loop.*

```
1 LRT($G$) // $G = (V, E, W)$ is a DDG
2 Construct $\mathcal{F}$ from $G$
3 **for** every flow network $G_f = (V_f, E_f, C_f)$ in $\mathcal{F}$ **do**
4     Let $G'_f$ be the simple graph converted from $G_f$
5     $(S_f, T_f) = \text{Find\_Min\_Cut}(G'_f)$
6 Let $(S_{\rm opt}, T_{\rm opt})$ be one of $2^{|\mathcal{C}_{\rm AB}|}$ minimum cuts found
  above such that the capacity of the cut is the smallest
7 **return** $(S_{\rm opt} \setminus \{s\}, T_{\rm opt} \setminus \{t\})$
```

Figure 3. An optimal algorithm.

Definition 1 (Optimality) $G_{\rm par} = \{G_{\rm top}, G_{\rm bot}\}$ is *optimal* if $\mathcal{P}(G_{\rm par})$ is the smallest possible.

Minimizing $\mathcal{P}(G_{\rm par})$ is equivalent to minimizing

$$\sum_{(u,v)\in\mathcal{D}(G_{\rm par})} \ln\left(\frac{1}{1 - W(u,v)}\right) \qquad (2)$$

In our implementation, if $W(u,v) = 1$ for a dependence $u \to v$, then $W(u,v) = 0.99$ will be used instead.

This objective function motivates us to formulate the problem of finding an optimal loop recreation for a loop as one of solving a min-cut problem on a set of *s*-*t* flow networks derived from its DDG. The goal is to find a minimum cut on one of these flow networks so that the cut induces an optimal loop recreation for the loop.

However, we cannot find a loop recreation $G_{\rm par} = \{G_{\rm top}, G_{\rm bot}\}$ for a loop by directly solving a min-cut problem on the DDG of the loop. This is because all inter-iteration dependence edges confined in either $G_{\rm top}$ or $G_{\rm bot}$ do not manifest themselves as cut edges. To overcome this, some inter-iteration dependence edges will be duplicated in flow networks created from G. The notions of A- and B-duplicated edges are defined below.

Definition 2 (A- and B-Duplicated Edges) *Let $x, y \in V$. We write $D^*(x,y) = I$ if there exists a directed path of intra-iteration dependences from x to y in G. Let G_c be the set of inter-iteration dependence edges in G. The sets of only A-duplicated edges, only B-duplicated edges and both A- and B-duplicated edges are:*

$$\begin{aligned}
\mathcal{C}_{\rm A} &= \{(u,v) \in G_c \mid D^*(u,v) \neq I, D^*(v,u) = I\}\\
\mathcal{C}_{\rm B} &= \{(u,v) \in G_c \mid D^*(u,v) = I, D^*(v,u) \neq I\}\\
\mathcal{C}_{\rm AB} &= \{(u,v) \in G_c \mid D^*(u,v) \neq I, D^*(v,u) \neq I\}
\end{aligned}$$

Note that $D^*(u,v) = I \wedge D^*(v,u) = I$ is impossible since u and v do not depend on each other.

We are ready to define $\mathcal{F}$ as a set of $2^{|\mathcal{C}_{\mathrm{AB}}|}$ *s*-*t* flow networks constructed from $G = (V, E, W)$. Each flow network $G_f = (V_f, E_f, C_f)$ is built as follows:

Step 1 The node set V_f is $V \cup \{s\} \cup \{t\}$, where s and t are the unique source and sink, respectively.

Step 2 For every $u \to v \in E$, we add two edges to E_f: $u \xrightarrow{C_f(u,v)} v$ and $v \xrightarrow{C_f(v,u)} u$. If $u \to v$ is an intra-iteration dependence, we set $C_f(u, v) = \ln(\frac{1}{1-W(u,v)})$ and $C_f(v, u) = \infty$. Otherwise, we set $C_f(u, v) = \ln(\frac{1}{1-W(u,v)})$ and $C_f(v, u) = 0$.

Step 3 If $u \to v \in \mathcal{C}_{\mathrm{A}}$, we add $s \xrightarrow{\ln(\frac{1}{1-W(u,v)})} v$ and $u \xrightarrow{\ln(\frac{1}{1-W(u,v)})} t$ to E_f. This is called an *A-duplication*. If $u \to v \in \mathcal{C}_{\mathrm{B}}$, we add $s \xrightarrow{\ln(\frac{1}{1-W(u,v)})} u$ and $v \xrightarrow{\ln(\frac{1}{1-W(u,v)})} t$ to E_f. This is called a *B-duplication*. If $u \to v \in \mathcal{C}_{\mathrm{AB}}$, we add either an A- or B-duplication to E_f.

In Step 2, for every dependence $u \to v$ in G, we add two edges to E_f: $u \to v$ and $v \to u$, where the former edge represents the situation when $u \to v$ spans from G_{top} to G_{bot} and the latter edge represents the situation when $u \to v$ spans from G_{bot} to G_{top}. Therefore, their weights are set as implied by Lemma 2. If $u \to v$ is an intra-iteration dependence, the weight of $v \to u$ in G_f is set to be ∞ to prevent it from becoming a cut edge.

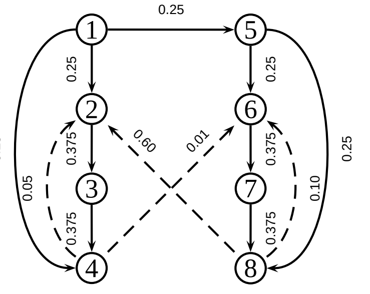
By convention, the absence of an edge in a flow network implies its capacity to be 0. Thus, if $u \to v$ is an inter-iteration dependence, then G_f will consist of implicitly $s \xrightarrow{0} u$, $s \xrightarrow{0} v$, $u \xrightarrow{0} t$ and $v \xrightarrow{0} t$. Therefore, there is no guarantee that $u \to v$ will be a cut edge in a cut. If $u \to v$ is not a cut edge, then its weight will not be included in the capacity of the cut.

In Step 3, two more copies are introduced for every inter-iteration dependence dependence $u \to v$ in $\mathcal{C}_{\mathrm{A}}$ and $\mathcal{C}_{\mathrm{B}}$. Then every *s*-*t* cut is guaranteed to include at least one copy of $u \to v$. Since an edge in $\mathcal{C}_{\mathrm{AB}}$ will be both A- and B-duplicated, $|\mathcal{F}| = 2^{|\mathcal{C}_{\mathrm{AB}}|}$ holds.

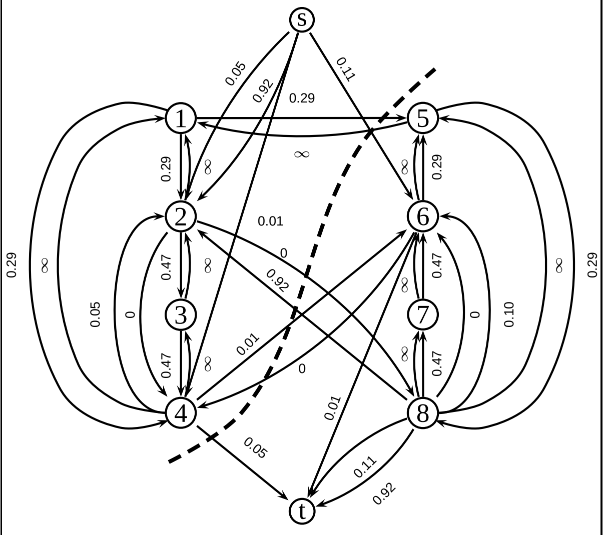
Figure 3 gives our optimal algorithm.

3.4 An Example

We illustrate our algorithm using a DDG abstracted from loop 120 in subroutine `parmvr` of `wave5` in SPECfp95. For the DDG depicted in Figure 4(a), it is easy to check that $\mathcal{C}_{\mathrm{A}} = \{4 \to 2, 8 \to 6\}$, $\mathcal{C}_{\mathrm{B}} = \emptyset$, and $\mathcal{C}_{\mathrm{AB}} = \{8 \to 2, 4 \to 6\}$ since $D^*(2,4) = I \wedge D^*(4,2) \neq I$, $D^*(6,8) = I \wedge D^*(8,6) \neq I$, $D^*(2,8) \neq I \wedge D^*(8,2) \neq I$ and $D^*(4,6) \neq I \wedge D^*(6,4) \neq I$. Hence, $\mathcal{F}$ consists of $2^{|\mathcal{C}_{\mathrm{AB}}|} = 4$ flow networks.



(a) G (where solid (dashed) arrows represent intra-iteration (inter-iteration) dependences)



(b) $G_{8 \xrightarrow{A} 2, 4 \xrightarrow{B} 6}$

Figure 4. An illustration of LRT .

Of the four flow networks in $\mathcal{F}$, $G_{8 \xrightarrow{A} 2, 4 \xrightarrow{B} 6}$, in which $8 \to 2$ is A-duplicated and $4 \to 6$ is B-duplicated, is shown in Figure 4(b). This network is constructed as follows. For each of the nine intra-iteration dependences, its two copies in the network are created in Step 2. For each of the four inter-iteration dependences, its two copies are created in its Step 2 and the other two created in Step 3 (with $4 \to 2$, $8 \to 6$ and $8 \to 2$ being A-duplicated and $4 \to 6$ B-duplicated).

Figure 4(b) also depicts the minimum cut from which the optimal loop recreation $G_{\mathrm{par}} = \{G_{\mathrm{top}}, G_{\mathrm{bot}}\} = \{\{1,2,3,4\}, \{5,6,7,8\}\}$ is found.

3.5 Optimality

Theorem 1 *The partition* $\{S_{\mathrm{opt}} \setminus \{s\}, T_{\mathrm{opt}} \setminus \{t\}\}$ *returned by* LRT *is optimal.*

Proof. The basic idea behind building $\mathcal{F}$ is as follows. If $G_{\mathrm{par}} = \{G_{\mathrm{top}}, G_{\mathrm{bot}}\}$ is a legal loop recreation for G, then $(G_{\text{top}} \cup \{s\}, G_{\text{bot}} \cup \{t\})$ is an *s*-*t* cut that is free of ∞-weighted cut edges in every flow network in $\mathcal{F}$. The converse is true, too. We create $2^{|\mathcal{C}_{\text{AB}}|}$ networks in $\mathcal{F}$ to ensure that all inter-iteration dependences in a recreated loop are captured as cut edges in an *s*-*t* cut. Furthermore, $(S_{\text{opt}} \setminus \{s\}, T_{\text{opt}} \setminus \{t\})$ is optimal if $(S_{\text{opt}}, T_{\text{opt}})$ has the smallest capacity. □

3.6 Time Complexity and Practical Efficiency

We have used Goldberg's implementation of his *push-relabel* HIPR algorithm [4] to find minimum cuts. Its worst-case time complexity when applied to $G = (V, E, W)$ is $O(|V|^2 \times \sqrt{|E|})$. In line 3 of LRT, there can be $|\mathcal{F}| = 2^{|\mathcal{C}_{\text{AB}}|}$ flow networks. So the worst-case time complexity of LRT is $O(|V|^2 \times \sqrt{|E|} \times 2^{|\mathcal{C}_{\text{AB}}|})$.

In practice, LRT is efficient if we adopt the following simple strategy. Let $G_f \in \mathcal{F}$. Let $u_1 \to v_1, \ldots, u_m \to v_m$ 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: (1) $s \in G_{\text{top}}$ and $u, v, t \in G_{\text{bot}}$, (2) $s, u, v \in G_{\text{top}}$ and $t \in G_{\text{bot}}$, (3) $s, u \in G_{\text{top}}$ and $v, t \in G_{\text{bot}}$ and (4) $s, v \in G_{\text{top}}$ and $u, t \in G_{\text{bot}}$. Hence, the capacity of any minimum cut in G_f must be larger than or equal to $\sum_{i=1}^{m} W(u_i, v_i)$. We will first find the minimum cut for the unique flow network in $\mathcal{F}$ in which all edges in $\mathcal{C}_{\text{AB}}$ are A-duplicated and then examine the remaining flow networks in the order in which more and more edges in $\mathcal{C}_{\text{AB}}$ are B-duplicated. We ignore a flow network if $\sum_{i=1}^{m} W(u_i, v_i)$ 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.

The compile times can be reduced further if some edges in $\mathcal{C}_{\text{AB}}$ given in Definition 2 are ignored when their weights are small. This will over-approximate slightly the misspeculation probability $\mathcal{P}$ in (1).

Table 1 gives the compile times for the four benchmarks used in our experiments.

Benchmark	#Nodes	#Edges	$\lvert\mathcal{C}_{\text{AB}}\rvert$	Time (msecs)
Wave5	796	1572	12	21.17
Fmda3d	868	1864	12	12.06
Irreg	85	173	0	0.60
Nbf	80	189	0	0.61

Table 1. Compile times (on a 3.2GHz P4).

4. Implementation

Figure 5 depicts the SUIF/MachSUIF compilation framework in which this work is implemented. Our loop recreation pass is invoked just before MachSUIF's register allocation pass. All virtual registers (i.e., scalars) in the MachSUIF IR are candidates for synchronization. The remaining ones are memory variables, which give rise to speculated memory dependences.

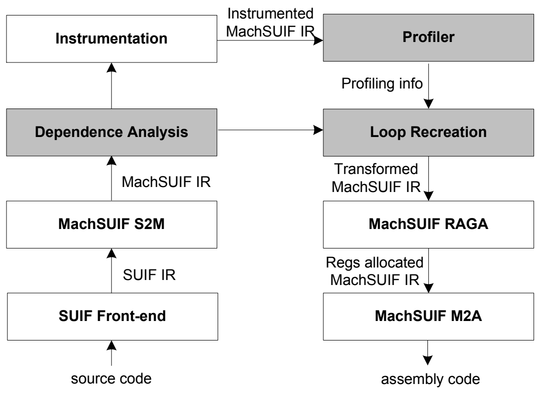


Figure 5. A compilation framework.

We have added the three modules that are highlighted in gray. A program is first converted into the SUIF IR by the SUIF front-end. The SUIF IR is then converted into the MachSUIF IR, on which our loop recreation algorithm operates as a separate pass. The dependence analysis module builds the DDG for a loop. To obtain the probability of a speculated memory dependence, the MachSUIF IR is fed to our profiler. Our loop recreation module reads the MachSUIF IR of a loop and produces the parallelized code in the form of MachSUIF IR.

5. Experimental Results

In this section, we demonstrate performance advantages of loop recreation over two existing parallelization methods using four irregular applications under two different squash mechanisms: *eager squash* and *lazy squash*. In the former case, all misspeculated threads are squashed and restarted as soon as they are detected at some write accesses. In the latter case, these only happen when the squashing thread has run to completion.

5.1. SpMT Architecture

We consider a SpMT multi-core system, where each core has its private function units, register file, separate L1 instruction and data caches. All the cores share a common L2 unified cache. Each core is capable of executing the Alpha ISA with the main architectural parameters listed in Table 2, which are similar to those used in recent papers [3, 9]. Before a loop is executed, registers holding all live-in values for the loop are copied to all cores. This happens only once for a loop since the live-ins between its iterations are synchronized. A latency of

8 cycles for "Copy Overhead" is assumed for one copy operation; this is conservative since a maximum of 13 live-ins for a loop are observed in our experiments.

Parameter	Value
Fetch, Issue, Commit	bandwidth 4, out-of-order issue
L1 I-Cache	16KB, 4-way, 1 cycle (hit)
L1 D-Cache	16KB, 4-way, 3 cycle (hit)
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
Commit Overhead	5 cycles
Invalidation Overhead	12 cycles
Copy Overhead	8 cycles

Table 2. SpMT architecture.

5.2. Benchmarks

We examine four irregular applications accessing arrays with subscripted subscripts: `Irreg` and `Nbf` from [5] and two kernel loops from SPEC benchmarks. `Irreg` is a representative of iterative PDE solvers found in computational fluid dynamics (CFD) applications. `Nbf` is a widely used kernel abstracted from the GROMOS molecular dynamics code. `Wave5` from SPECfp95 solves Maxwell's equations and particle equations of motion on a cartesian mesh with a variety of field and particle boundary conditions. We have chosen loop 120, a one-deep nest, in subroutine `parmvr` and named it `Wave5-120`. To model the fact that `parmvr` is called over 60065 times, an outer loop with an iteration count of 4000 is added for this kernel loop. `Fma3D` from SPECfp2000 is a 3D inelastic, transient dynamic response simulation code based on finite element analysis. The loop chosen, 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.

5.3. Simulations

For the two SPEC benchmark loops, the train inputs are used to collect profiling information (i.e., the probabilities of memory dependences) while the reference inputs are used in simulations. For `Irreg` and `Nbf`, the input graphs used to collect profiling information are different from those used in simulations. In each case, the simulation results for two different input graphs are presented. `Irreg-I32` denotes `Irreg` running with an input graph with 131072 nodes and 3538944 edges and `Irreg-I35` denotes the case when the input graph has 171500 nodes and 4630500 edges. `Nbf-N28` denotes `Nbf` running with an input graph with 87808 nodes and 2370816 edges and `Nbf-N32` denotes the case when the input graph has 131072 nodes and 3538944 edges.

The threaded loops generated by different methods are usually different. To compare their performance results accurately, each program is run for a common set of (consecutive) iterations for all methods compared. All programs are simulated for between 1 to 1.5 billion instructions. For `Wave5a-120` and `Fma3d-NUMP4`, the first 12 and 15 million instructions are skipped, respectively. As for `Irreg-I32`, `Irreg-N35`, `Nbf-N28` and `Nbf-N32`, the number of skipped instructions ranges between 3 and 7 billion.

5.4. Methods Compared

In our experiments, we evaluate the performance improvements of LRT over two methods, Par and SPT [3]. Par simply maps each iteration in a loop into a thread (cf. Figure 2(b)). We will also give the speedups of LRT over sequential programs, which are generated by Non (which stands for none). Unlike SPT as described in [3], full rather than partial re-execution is used.

SPT is guided by a misspeculation cost model to parallelize a given loop [3]. In our experiments, the best solutions it could ideally generate for all four programs are used. For `Irreg` and `Nbf`, 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 6(c) – (f) and 7 are identical. For `Wave5-120`, there are six sections of code exhibiting the same dependence patterns as those shown in Figure 4(a). 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 Figure 4(a) is 12. The four of these end up each being split evenly in the pre-fork and post-fork regions. So the ratio of pre-fork/post-fork is also 1/5.

As for LRT, the pre-fork region of a parallelized loop it generates for any given loop is the same as that generated by Par. For each benchmark, all code sections that share the same dependence characteristics as the loop given in Figure 4(a) have been successfully parallelized. Based on our cost model given in (1), LRT has reduced the misspeculation possibilities for `Wave5`, `Fma3D`, `Irreg` and `Nbf` from 0.82, 0.99, 0.85 and 0.87 to 0.05, 0.00, 0.19 and 0.04, respectively.

5.5. Speedups and Analysis

We present our results for 2-, 4-, 6- and 8-core systems in Figures 6 – 9, where L stands for LRT, P for Par, S for SPT, N for Non, EA for "Eager Squash", and LA for "Lazy Squash". We first take a look at the speedups of LRT over other methods and then describe the reasons behind these performance improvements.

5.5.1 Speedups

Figure 6 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 bars for two methods in a configuration, the one with a shorter bar generates faster code and is thus better than the other.

First of all, we observe that LRT improves scalably the execution time of every program. Note that the performance improvements (of LRT over Non) under both squash mechanisms are nearly the same. Thus, the normalized execution times 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 much better under EA. In the case of `wave5-120` and `Fma3d-NUMP4`, Par is even slightly worse than Non. However, even when EA is assumed, LRT outperforms Par in nearly all configurations. The speedups of LRT over Par on a 8-core system for `Wave5-120`, `Fma3d-NUMP4`. `Irreg-I32`, `Irreg-I35`, `Nbf-N28` and `Nbf-N32` are 1.67, 1.59, 3.71, 3.69, 3.59 and 3.54, respectively. Due to cache effects (as will be explained shortly in Section 5.5.2), some slight performance slowdowns are observed when `Fma3D-NUMP4` is run on 2- and 4-core systems.

Finally, let us examine the performance improvements of LRT over SPT. SPT performs slightly better under EA for all the four programs. When LA is used, the speedups of LRT over SPT on a 8-core system for `Wave5-120`, `Fma3d-NUMP4`. `Irreg-I32`, `Irreg-I35`, `Nbf-N28` and `Nbf-N32` are 1.70, 1.59, 3.75, 3.73, 3.93 and 3.87, respectively. If EA is used instead, these numbers become 1.54, 1.58, 3.71, 3.69, 3.59 and 3.54, respectively. Again, due to cache effects, some slight performance slowdowns are observed when `Fma3D-NUMP4` is run on 2- and 4-core systems.

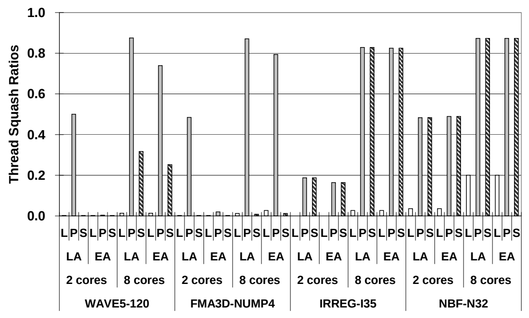


Figure 7. Misspeculation penalties.

5.5.2 Analysis

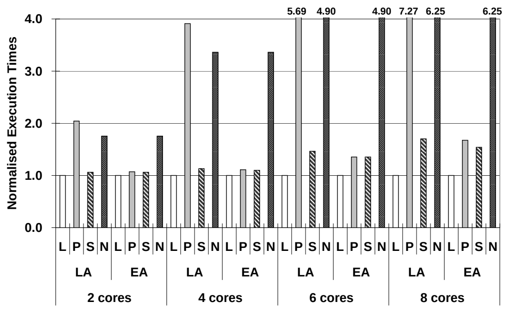
To shed some light on the results given in Figure 6 and understand the impact of the two squash mechanisms on these results, we present some synchronization and misspeculation statistics in Figures 7 and 8.

Misspeculation Figure 7 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 backedge of a parallelized loop is violated, the only form of control dependence violations in this work.

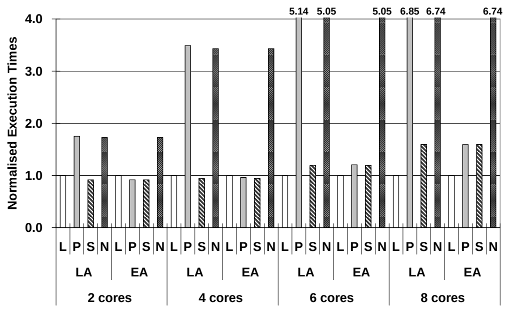
LRT has the smallest squash ratio among all methods compared under either squash scheme. In addition, its squash ratios (due to mainly control misspeculations) are small in all programs except `Nbf-N32`, which attracts a squash ratio of 4% with two cores and of 20% when eight cores are used. These squashes are mainly 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 in our experiments. Note that Par and SPT have parallelized `Nbf` and `Irreg` identically.

Synchronization All sequential loops in our benchmarks are free of inter-iteration register dependences. Thus, there are no synchronized register dependences in the parallelized loops generated by Par and SPT.

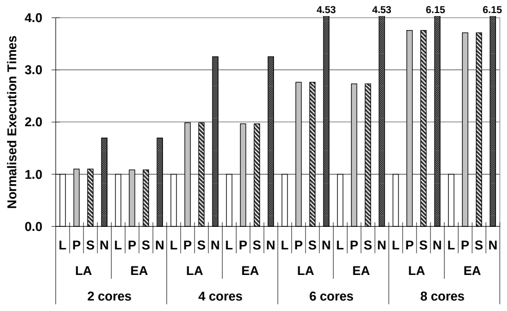
In all the four benchmarks except `Fma3d-NUMP4`, LRT has transformed some intra-iteration register dependences into inter-iteration register dependences. Figure 8 shows the synchronization costs incurred by LRT. The synchronization costs in these benchmarks are small, representing less than 4.3% of their total execution times. `Fma3d-NUMP4` is synchronization-free.



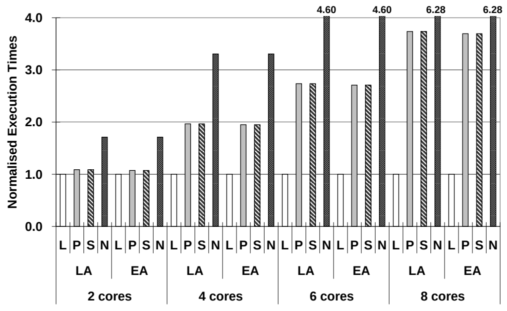
(a) Wave5-120



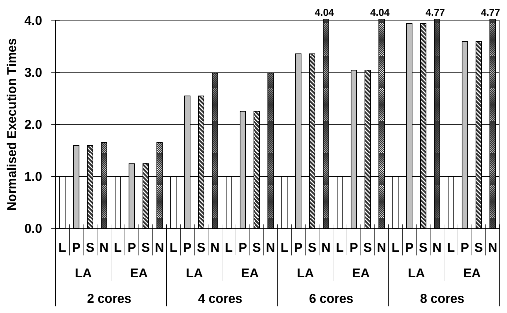
(b) Fma3d-NUMP4



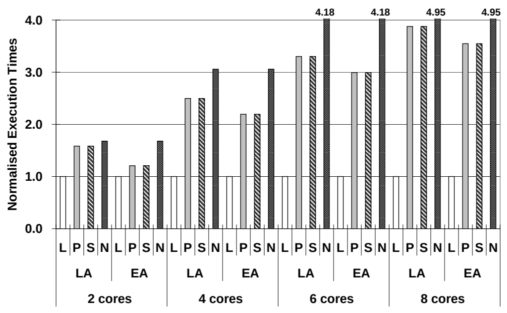
(c) Irreg-I32



(d) Irreg-I35



(e) Nbf-N28



(f) Nbf-N32

Figure 6. Normalized execution times (with respect to LRT).

The percentage synchronization costs for `Wave5-120` are about 4.3% in the worst case. The synchronization costs for `Irreg-I35` and `Nbf-N32` are relatively large considering the small sizes of their loop bodies.

The results under both squash schemes are similar since the misspeculation statistics for LRT are similar.

Eager Squashes vs Lazy Squashes We can now understand better the speedup numbers given in Figure 6 and the impact of two squash mechanisms on these results. Let us examine `Wave5-120` first. Recall that its loop body can be divided into six sections with each sharing the same dependence characteristics as the DDG given in Figure 4(a). LRT has parallelized each section as shown in Figure 4(b). As a result, LRT performs similarly under both squash mechanisms since misspeculations are infrequent (as shown in Figure 7). This fact can be deduced in Figure 6, 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. Therefore, Par suffers frequent misspeculations under LA (as shown in Figure 7) to the extent that the threads are nearly sequentialized (as shown in Figure 6). From Figure 7, we can see that by squashing misspeculated threads earlier for `Wave5-120` under EA, the squash ratio has been reduced significantly. Thus, by squashing misspeculated threads and restarting them earlier, more parallelism has been attained. The parallelized `Wave5-120` from Par runs 1.90 (4.32) 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 spawn-

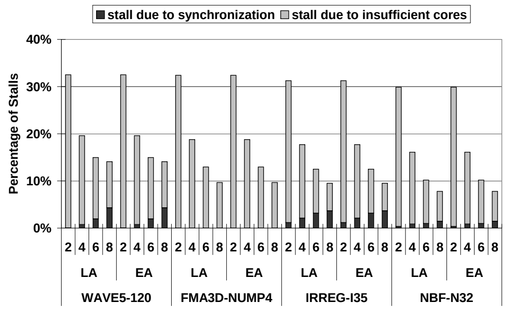


Figure 8. Synchronization cost for LRT.

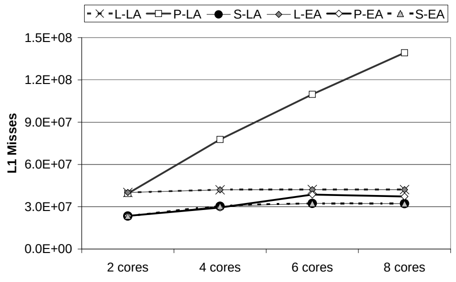


Figure 9. Cache misses of `Fma3D-NUMP4`.

ing threads, the 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 similarly under both squash schemes since the misspeculations for both programs are infrequent. Both Par and SPT generate same parallelized code. So their performance results for each program are identical. Let us examine Par. The dependence violations for `Irreg` are infrequent and usually happen at the end of an iteration. So small performance variations are observed under both squash schemes. The dependence violations for `Nbf` happen slightly earlier in an iteration. Thus, Par performs better under EA for `Nbf`.

Cache Effects When `Fma3d-MUMP4` is run on two- and four-core systems, LRT performs equally as well 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 9. 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 8. 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. 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.

Helper threads [2, 7, 8, 16] are used to speculatively execute a code region to reduce the latency of some expensive instructions in the region. A helper thread for a loop may be formed from any of its instructions in any order since it does not have to be concerned with program correctness. However, as a loop transformation, loop recreation is correctness-preserving.

Software-based value prediction techniques [3, 9] are used to predicate some live-in values for a thread to reduce missspeculation penalties. For example, once a thread is created, the Mitosis compiler [9] 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.

When reviewing existing speculative parallelization techniques, we first consider loop-oriented techniques and then general-purpose ones. To the best of our knowledge, existing loop-oriented techniques [3, 10, 11, 13, 14, 15] form threads only at loop boundaries by turning loop iterations into threads. In [10, 14, 15], frequently occurring dependences are synchronized. 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 [14] to insert the `post` and `wait` instructions required for synchronized dependences.

The SPT compiler [3] attempts to move the producer instructions of some inter-thread dependences into the pre-fork region (subject to their cost model), thereby reducing squashes caused by frequently occurring inter-thread dependences. A software value prediction technique is used to predict some live-in values when their producer instructions are not in the pre-fork region.

Some general-purpose compiler techniques [1, 6, 9, 12] 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 [12] forms threads only at loop boundaries. The follow-up work [1] may allow large loop iterations to be sliced into multiple threads but loop boundaries remain to be thread boundaries. The Mitosis compiler [9] and the work [6] 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 basic block, they will still restrict threads to loop boundaries.

Software pipelining improves instruction level parallelism by overlapping the execution of adjacent loop iterations on single-core processors. However, loop recreation improves TLP by overlapping the execution of two adjacent loop iterations on SpMT multi-core processors.

7. 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 parallelism than the original loop. We present a loop recreation algorithm and demonstrate significant performance advantages of loop recreation over some recent techniques using four irregular applications.

References

[1] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreaded processors. *IEEE Trans. Parallel Distrib. Syst.*, 15(8):713–724, 2004.

[2] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. 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.

[3] Z. H. Du, C. C. 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.

[4] A. Goldberg. Network Optimization Library, 2003. http://www.avglab.com/andrew/soft.html.

[5] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, Jul. 2006.

[6] 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.

[7] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. 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 Press.

[8] C.-K. 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.

[9] 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.

[10] J. G. Steffan, C.B.Colohan, A.Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *International Symposium on High-Performance Computer Architecture*, 2002.

[11] J. Y. Tsai and P. C. 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.

[12] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international Symposium on Microarchitecture*, pages 81–92, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[13] S. Y. Wang, X. R. Dai, K. S. Yellajyosula, A. Zhai, and P. C. Yew. Loop selection for thread-level speculation. In *The 18th International Workshop on Languages and Compilers for Parallel Computing*, 2005.

[14] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2002.

[15] 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.

[16] C. Zilles and G. 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.