Thread-Sensitive Modulo Scheduling for Multicore Processors *

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

Tin-Fook Ngai
Microprocessor Technology Lab, Intel

Abstract

This paper describes a generalisation of modulo scheduling to parallelise loops for SpMT processors that exploits simultaneously both instruction-level parallelism and thread-level parallelism while preserving the simplicity and effectiveness of modulo scheduling. Our generalisation is simple, drops easily into traditional modulo scheduling algorithms such as Swing in GCC 4.1.1 and produces good speedups for SPECfp2000 benchmarks, particularly in terms of its ability in parallelising DOACROSS loops.

1. Introduction

Even as we enter the multicore era, seeking methods to boost the performance of single-threaded applications remains critical [8]. In speculative multicore (SpMT) processors with fast on-chip interconnects [2, 10, 19, 23], inter-thread memory dependences can be tracked by rolling back misspeculated threads and inter-thread register dependences enforced by fast communication and synchronisation. Such SpMT processors can significantly boost the performance of sequential applications that must be parallelised into fine-grain, communicating threads to be executed efficiently.

We present a new compiler technique to parallelise a sequential loop for SpMT processors by breaking the loop iteration boundaries to find a speculative schedule across multiple iterations of the loop. While modulo scheduling is the best known technique to generate non-speculative schedules for loops, this paper generalises it to generate speculative schedules for loops, particularly for DOACROSS loops. When applied directly to SpMT processors, modulo scheduling does not handle the overhead incurred in enforcing both synchronised and speculated inter-thread dependences between the threads that are executing different iterations of a modulo scheduled loop on different cores. Our key observation is that existing modulo scheduling algorithms [12, 15, 9] tend to schedule dependent instructions as close as possible so as to reduce register pressure by means of shortening the lifetimes of loop variants. For example, Swing modulo scheduling (SMS) [12] assigns an instruction to a cycle in its scheduling window, a range of cycles in which the instruction can be scheduled with respect to already scheduled ones, so that the instruction is the closest possible to its already scheduled dependent instructions. When a modulo scheduled loop is executed on multicore processors, these "tightly scheduled" dependences created by such a "lifetime-minimal" scheduling strategy can turn out to be inter-thread dependences.

Our TMS algorithm relies on a new cost model to strengthen the aforementioned "lifetime-minimal" scheduling strategy used in existing modulo scheduling algorithms. For multicore processors, finding the best schedule for a loop by minimising the II of the loop only is no longer adequate. We need to develop a cost model by which we can estimate the execution time of a modulo scheduled loop running on SpMT processors. Guided by this cost model, TMS will schedule an instruction to a cycle in its scheduling window in order to minimise not only the lifetimes of loop variants but also the synchronisation and misspeculation overheads with respect to the already scheduled instructions.

Specifically, this paper makes the following contributions. First, we introduce a new cost model to approximate the execution time of a modulo scheduled kernel loop for SpMT architectures. The execution time of the kernel loop depends on not only the II of the kernel loop but also the overhead incurred in enforcing its inter-thread dependences.

Second, we describe a generalisation of modulo scheduling for parallelising loops, called *thread-sensitive modulo scheduling* (TMS), that exploits both ILP and TLP simultaneously for SpMT multicore processors with fast on-chip interconnects while preserving the simplicity and effectiveness of modulo scheduling. Our generalisation is simple, drops easily into traditional modulo scheduling algorithms and produces good speedups for SPECfp2000 benchmarks, particularly in terms of its ability in parallelising DOACROSS loops. Guide by our cost model, our algorithm aims to minimise the parallel execution time of a loop by minimising the overheads incurred in enforcing synchronised and speculated inter-thread dependences.

Finally, we have implemented TMS on top of Swing modulo scheduling (SMS) [12] adopted in GCC 4.1.1 (since SMS finds the best schedules in general [3]). But our work

* This work is supported by an ARC Grant DP0665581.

is not tied to any existing modulo scheduling algorithms (as made clear in Section 4.1). We have evaluated the performance of TMS on a quad-core SpMT architecture using SPECfp2000 benchmarks. Compared to SMS, TMS can generate kernel loops with significantly reduced synchronisation overhead and negligible misspeculation overhead. In addition, the effectiveness of TMS is also substantiated in terms of its ability in parallelising DOACROSS loops that are difficult to parallelise by existing methods. Our experimental results are also analysed to provide insights into how to improve this work to obtain greater speedups.

The plan of the paper is as follows. Section 2 reviews the related work. Section 3 introduces our speculative parallelisation models. Section 4 presents our generalised modulo scheduling algorithm. Section 5 presents and analyses our experimental results. Section 6 concludes the paper.

2. Related Work

Most of the previous work on speculatively parallelising sequential programs or loops [5, 14] focuses on improving TLP by exploiting thread-level speculation. Typically, inter-core memory dependences are tracked by the hardware by backing up misspeculated threads and inter-core register dependences enforced through memory [5] or by value prediction and validation [5, 14]. However, many sequential programs cannot be parallelised effectively this way if they exhibit a lot of inter-thread register dependences.

Recently, multicore architectures have been extended to support fast inter-core communication and synchronisation of register values [2, 10, 19, 23]. It is demonstrated in [10] that a one-cycle inter-core hop latency is realisable in multicore processors. Based on a mesh interconnect that can route an operand between adjacent cores in one cycle, these researchers have evaluated a 32-core design, called TFlex, that allows cores to be aggregated together dynamically to form more powerful single-threaded processors. Such composability allows a right balance of ILP and TLP to be exploited in a sequential program. Based on [19], the Voltron processor [23] extends a traditional multicore design with a scalar operand network to provide fast inter-core communication to enable fine-grain threads to be executed efficiently. In particular, Voltron allows two adjacent cores to communicate a register value in one cycle when exploiting VLIW-style ILP and three cycles when exploiting fine-grain TLP.

These improvements in inter-core communication have offered new opportunities for improving the performance of many sequential programs by partitioning them for efficient execution in terms of fine-grain, communicating threads. In [18, 21], frequently occurring dependences are synchronised. The `post` and `wait` instructions associated with a synchronised dependence are moved as close as possible. In [4], the work in [16] on modulo scheduling multi-dimensional loops is extended to parallelise non-innermost loops for multicore processors. Thread-level speculation is not used. Instead, all inter-core dependences are synchronised via a software-managed cache. In [13, 20], they exploit TLP not ILP by partitioning a loop iteration into long-running threads and roll back misspeculated long-running threads by means of check-pointing and versioned memory. In contrast, our TMS algorithm executes different iterations of a modulo scheduled loop in different threads and rolls back misspeculated threads using hardware.

Parameter	Values
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
SEND/RECV Latency	3 cycles
Spawn Overhead	3 cycles
Commit Overhead	2 cycles
Invalidation Overhead	15 cycles

Table 1. Architecture simulated.

There have been some research efforts on developing scheduling algorithms for clustered VLIW architectures [17, 22, 1]. These architectures do not support thread-level speculation. Separate register files in different clusters are used to exploit ILP within and across the clusters. A register value can be communicated synchronously between adjacent clusters in one cycle. In these algorithms, an iteration in a kernel loop is sub-divided with different subparts executed in different clusters. In our work, different iterations of a kernel loop are executed speculatively on different cores in a speculative superscalar architecture.

3. Execution Model

As shown in Table 1, an SpMT system consists of multiple cores connected by a uni-directional ring [6]. However, this work is not limited to this architecture. Each core owns its private function units, register file, L1 instruction cache and L1 data cache. All cores share one unified L2 cache. Data dependences through memory-resident values, known as *speculated dependences*, are tracked by the hardware and preserved by backing up any misspeculated threads. Data dependences through register-resident values, known as *synchronised dependences*, are preserved with their values being communicated asynchronously through the ring bus between two cores as in Voltron's queue model [23].

As a generalisation of modulo scheduling algorithms, TMS aims at parallelising innermost loops only. After scheduling, the distances of all inter-iteration register dependences are 1 since all overlapping lifetimes for scalars are handled by introducing register copy instructions in a post-pass. Thus, the values for synchronised register de-

pendences are communicated through adjacent cores.

A loop is executed speculatively in much the same way as in prior work [5, 14, 18]. The iterations of a loop are executed in different threads running in different cores in a round-robin fashion. The first instruction in each thread is a spawn instruction, which contains the start address of the loop. The spawn instruction in iteration i causes a speculative thread for iteration $i + 1$ to be created and executed in the successor core. The oldest thread in the sequential execution order of the loop, called *head thread*, is the only *non-speculative* thread and thus allowed to commit its results. All others are *speculative*. A thread that has completed its execution will check to see if some inter-thread memory dependences were violated in more speculative threads. Let T be the least speculative thread in which a violation is detected. Then T and all its more speculative ones are squashed and T will be re-executed on the core that T was executed before. Upon successful completion, a thread will invalidate all more speculative threads that have been misspeculated. Each invalidation operation that squashes a thread running in a core involves only gang-clearing several bits in MDT and several bits in L1 data cache and flushing its send/receive queues and speculation write buffer. So 15 cycles is more than sufficient.

To communicate a register value between two adjacent cores in the Voltron's queue model [23], the compiler inserts a pair of SEND and RECV instructions to synchronise and forward the value. The latency incurred is 3 cycles: 1 for SEND, 1 per hop to transmit the value and 1 for RECV. As also in Voltron [23], the spawning of a loop iteration thread takes 3 cycles in the same queue model. Just before a loop is executed, the registers holding the live-in values for the loop are copied to all the cores participating in executing the loop. This will happen only once for a loop since the live-ins between iterations are communicated by register communications via SEND and RECV.

All memory dependences are tracked using the memory disambiguation table (MDT) [11], which sits between L1 data cache and L2 cache. As in Hydra [7], a speculation write buffer (of 64 entries) next to L2 cache is available in a core to buffer all speculative writes. Using double buffering, a core can start a new thread with a "new" buffer while the the "old" buffer is draining into L2 cache. So a commit overhead of 2 cycles is assumed.

4. TMS

We describe a motivating example, a cost model for estimating the execution time of a modulo scheduled loop on SpMT processors, and finally our TMS algorithm.

4.1. A Motivating Example

We use an example to highlight the limitation of the "lifetime-minimal" scheduling strategy used in modulo scheduling algorithms if a scheduled loop is executed on SpMT processors. We also explain how we overcome this limitation in TMS. Consider a data dependence graph (DDG) and the architectural parameters relevant to modulo scheduling shown in Figure 1. The resource II is $ResII = 4$ (since the mul has the longest latency). The recurrence II is $RecII = 8$ due to the existence of the recurrence circuit $(n0, n1, n2, n4, n5)$. So the minimum II i.e., MII is $\max(ResII, RecII) = \max(4, 8) = 8$. Figure 2 compares the schedules produced by SMS and TMS and their runtime overheads incurred in enforcing inter-thread dependences.

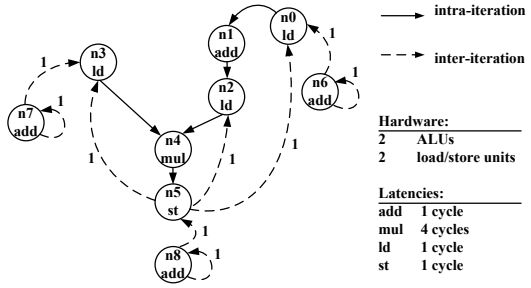


Figure 1. A motivating example. All dependences shown are flow dependences with their distances given. $n5 \rightarrow n0$, $n5 \rightarrow n2$ and $n5 \rightarrow n3$ are memory dependences with small dependence probabilities and the remaining ones are register dependences.

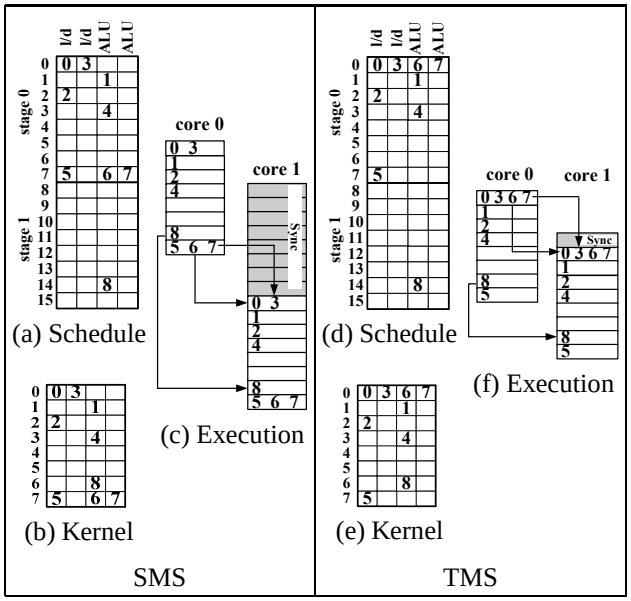


Figure 2. A comparison of SMS and TMS with respect to the execution of two consecutive kernel iterations in a two-core SpMT architecture. Each instruction ni shown in Figure 1 is abbreviated to i here.

SMS According to [12], the nodes in the DDG are scheduled in the order: $n5, n4, n2, n1, n0, n3, n6, n8$ and $n7$. So preference is given to the instructions in the critical path in order to avoid scheduling both an instruction's predecessors and successors at the same time before the instruction itself. By placing each node as close as possible to its predecessors and successors, SMS produces the schedule and kernel as shown in Figures 2(a) and (b), respectively. Consider how $n6$ is scheduled. Its associated scheduling window is $[7, 0]$ with the largest cycle being tried first. So $n6$ is scheduled at cycle 7, which is the first valid choice in the window. The successor node $n0$ of $n6$ has already been scheduled at cycle 0. Since the inter-iteration register dependence $n6 \rightarrow n0$ has a distance of 1, the value produced by $n6$ in one iteration will be consumed by $n0$ in the next iteration at cycle, 0 + 1 * II = 8. As shown in Figure 2(b), both $n6$ and $n0$ are the closest possible to each other (in time).

Modulo scheduling can alter dependence distances in a loop.

Definition 1. *Let $u \rightarrow v$ be a dependence of distance $d(u, v)$ in a loop. The distance in the kernel, denoted $d_{\text{ker}}(u, v)$, is $d_{\text{ker}}(u, v) = d(u, v) + s_v - s_u$, where s_u (s_v) is the stage number of u (v).*

In this example, the inter-iteration dependence $n8 \rightarrow n5$ (with $d(n8, n5) = 1$) has been turned into an intra-iteration dependence in the kernel (with $d_{\text{ker}}(n8, n5) = 0$). The distance of $n6 \rightarrow n0$ remains to be $d_{\text{ker}}(n6, n0) = d(n6, n0) = 1$ since both $n6$ and $n0$ are placed in stage 0. We find that the kernel loop has the following inter-iteration (flow) dependences: $n5 \rightarrow n0$, $n5 \rightarrow n2$, $n5 \rightarrow n3$, $n6 \rightarrow n0$, $n6 \rightarrow n6$, $n7 \rightarrow n3$, $n7 \rightarrow n7$ and $n8 \rightarrow n8$.

Figure 2 (c) shows the execution of the kernel in a two-core architecture. An arrow pointing from core 0 to core 1 symbolises a register communication event via a pair of SENV and RECV. Since $n6 \rightarrow n0$ and $n6 \rightarrow n6$ share one producer, only one communication is required. The same is true for $n7 \rightarrow n3$ and $n7 \rightarrow n7$.

Recall that in our execution model (Section 3), all overlapping lifetimes are implemented via copy instructions. So the non-neighbouring communication when $d(x, y) > 1$ is realised via a sequence of neighbouring communications.

Definition 2. *Let $x \rightarrow y$ be an inter-iteration register dependence, where $d(x, y) = 1$. The synchronisation delay incurred by the dependence is estimated to be:*

$$\begin{aligned} \text{sync}(x, y) \quad = \quad & \text{issue\_slot}(x)\%II - \text{issue\_slot}(y)\%II \\ & + \text{lat}(x) + C_{\text{reg\_com}} \end{aligned} \tag{1}$$

where $\text{issue\_slot}(x)$ and $\text{issue\_slot}(y)$ are the issue cycles of x and y (i.e., those shown in Figures 2(a) and (d)), $\text{lat}(x)$ is the latency of x, and $C_{\text{reg\_com}} = 3$ is the latency incurred in moving a scalar from the producer x to the consumer y.

As discussed in Section 5.1, the number of inserted copies is small for SPECfp2000 benchmarks used in our experiments. In this SMS solution, $\text{sync}(n6, n0) = \text{sync}(n7, n3) = 7\%II - 0\%II + 1 + C_{\text{reg\_com}} = 8 + C_{\text{reg\_com}} = 8 + 3 = 11$. So consecutive threads are sequentialised due to the synchronisation delay caused. As a result, the inter-thread memory dependences $n5 \rightarrow n0$, $n5 \rightarrow n2$ and $n5 \rightarrow n3$ are accidentally preserved.

TMS Instead of scheduling a node at the first available cycle in its scheduling window to aggressively reduce register lifetimes only, TMS finds a scheduling cycle such that some presently acceptable synchronisation and misspeculation overhead thresholds are also satisfied with respect to already scheduled nodes. Figures 2(d) and (e) give the schedule and kernel obtained by TMS, respectively. When scheduling $n6$, the time slot in its scheduling window $[7, 0]$ that leads to the shortest synchronisation delay between $n6$ and the already scheduled successor $n0$ is cycle 0. So it has been placed $n6$ at cycle 1. The node $n7$ is scheduled identically. Figure 2(f) shows the execution of the kernel generated by TMS. Compared to the SMS solution, the synchronisation delay caused by $n6 \rightarrow n0$ has been significantly reduced in the TMS solution. The inter-core memory dependences $n5 \rightarrow n0$, $n5 \rightarrow n2$ and $n5 \rightarrow n3$ are tracked by the hardware. Since their dependence probabilities are assumed to be negligibly small, few misspeculations will occur during the execution of the TMS-generated kernel.

4.2. Cost Model

This section describes a cost model used to approximate the execution time of a modulo scheduled loop running on SpMT processors by taking into account the synchronisation and misspeculation overheads incurred in enforcing inter-thread dependences. We consider not only the II of the loop but also the following four major cost components:

- C_{spn}: the overhead of spawning a thread on a core
- C_{ci}: the commit overhead by the head thread.
- C_{inv}: the invalidation overhead from a thread.
- C_{delay}: the maximal delay incurred by *any* synchronised register dependence in a thread. This parameter approximates the time that a thread spends on waiting for receiving the values for *all* synchronised register dependences from the predecessor thread. It includes the register communication latency $C_{\text{reg\_com}}$ introduced in Definition 1 (line 5 of Figure 3).

It is assumed that the number of iterations in a loop is sufficiently larger than the number of cores: $N \gg ncore$.

The execution time, T, of a modulo scheduled loop is composed of T_{nomiss} (the execution time of the loop in the absence of misspeculations) and $T_{\text{mis\_spec}}$ (the total misspeculation overhead in the loop). They are derived below.

T_{nomiss} Threads are spawned and committed sequentially, their spawn times never overlap, and similarly, their commit times never overlap. If thread i suffers a maximum synchronisation delay, denoted C_{delay}, so will thread $i+1$. The maximum synchronisation delay times of two threads cannot overlap. However, one of three cost components, C_{spn}, C_{ci} and C_{delay}, may cancel one of the other two when threads are executed in parallel. Hence, if a core is always freely available when a new thread is to be spawned, the execution time of a loop is bounded by the serial part of a thread, which is estimated to be: $\max(C_{\text{spn}}, C_{\text{ci}}, C_{\text{delay}}) \times N$. Otherwise, thread spawning can be stalled if no free core is available. In this case, the execution time is approximated by $\frac{T_{\text{lb}}}{ncore} \times N$, where $T_{\text{lb}} = II + C_{\text{ci}} + \max(C_{\text{spn}}, C_{\text{delay}})$ is the lower bound for the execution time of a thread. By combining the two cases, we have:

$$T_{\text{nomiss}} \quad = \quad \max(C_{\text{spn}}, C_{\text{ci}}, C_{\text{delay}}, \frac{T_{\text{lb}}}{ncore}) \times N \quad (2)$$

$T_{\text{mis\_spec}}$ The parallel execution of a loop may be interrupted by misspeculations. After a misspeculated thread is squashed, its execution will be re-started. The penalty paid for one misspeculation is roughly $II + C_{\text{inv}} - \max(0, C_{\text{delay}} - C_{\text{spn}})$, where $II + C_{\text{inv}}$ is the number of cycles wasted in executing and invalidating the squashed thread and $\max(0, C_{\text{delay}} - C_{\text{spn}})$ is the number of cycles gained in re-execution since all inter-thread register dependences of the re-started thread are already satisfied.

The probability value p_d of a memory dependence d is in $[0, 1]$, meaning that for every X writes at the producer, $p_d X$ reads from the consumer will be made to same memory location. As in [5, 14], we assume conservatively that if d is an inter-thread memory dependence, then $p_d X$ of X may be misspeculated. Let M be the set of all inter-thread memory dependences that may be misspeculated in the kernel. The misspeculation probability, denoted P_M, for the kernel is:

$$P_M \quad = \quad 1 - \prod_{e \in M} (1 - p_e) \quad (3)$$

Thus, $P_M \times N$ is the total number of misspeculations. The total misspeculation overhead of a loop is approximated as $T_{\text{mis\_spec}} = (II + C_{\text{inv}} - \max(0, C_{\text{delay}} - C_{\text{spn}})) \times P_M \times N$.

4.3. Algorithm

Figure 3 gives TMS as a generalisation of SMS, where the lines in the SMS code are boxed. Like SMS, TMS finds a schedule iteratively for a loop starting with an empty partial schedule, $\mathcal{PS}$. Whenever a new instruction is added to $\mathcal{PS}$, some new inter-iteration, i.e., inter-thread memory dependence may be introduced. Some of these may be preserved due to the synchronisation delay introduced by already scheduled instructions. This can be checked by applying Definition 3, which makes use of $d_{\text{ker}}(x, y)$ and $\text{sync}(x, y)$ from Definitions 1 and 2.

```
1   #DEFINE P_max = a turnable parameter in [0, 1]
2   TMS()
3   Q_0 ← the ordered node list for scheduling
4   Let F(II, C_delay) = T_nomiss/N = max(C_spn, C_ci, C_delay, T_lb/ncore)
5   Let F_min = F(MII, 1 + C_reg_com)
6   while true do
7       for every (II, C_delay) s.t. F(II, C_delay) = F_min do
8           Q ← Q_0
9           PS ← ∅
10          while Q ≠ ∅ do
11              v ← pop(Q)
12              if ISSUE_SLOT_SELECTION(v, PS) then
13                  Add v to PS
14              else
15                  break; // restart all over again
16          if Q = ∅ then return PS
17      F_min ++
18  ISSUE_SLOT_SELECTION(v, PS)
19  for every slot c in the scheduling window [l, u] of v do
20      if slot c has resource conflicts then continue
21      Let R_PS = RegDep(PS)
22      Let M_PS = MemDep(PS)
23      Let R_v = RegDep(PS ∪ {v}) \ RegDep(PS)
24      Let M_v = MemDep(PS ∪ {v}) \ MemDep(PS)
25      Let M_all be the set of dependences in M_PS ∪ M_v such that they are
        not preserved by R_PS ∪ R_v (Def. 3)
26      if the following two conditions hold then
            C1: ∀ x → y ∈ R_v : sync(x, y) ⩽ C_delay (Def. 2)
            C2: M_v ≠ ∅ ⟹ 1 − ∏_{e∈M_all} (1 − p_e) ⩽ P_max
27          return true
28  return false
```

Figure 3. TMS as a generalisation of SMS (with the new code indicated with boxes).

Definition 3. *For an instruction i, let its latency be denoted by $\text{lat}(i)$ and its issue slot by $\text{issue\_slot}(i)$ as in Definition 2. Let $x \rightarrow y$ be an inter-iteration memory dependence in the kernel of a loop. Let D be a set of some inter-iteration register dependences in the kernel loop. We say that $x \rightarrow y$ is* **preserved** *by D if there exists $u \rightarrow v \in D$, where u is executed earlier than x in the kernel, i.e., $\text{issue\_slot}(u)\%II < \text{issue\_slot}(x)\%II$, such that $\text{sync}(u, v) > \frac{\text{issue\_slot}(x)\%II - \text{issue\_slot}(y)\%II + lat(x)}{d_{\text{ker}}(x,y)}$.*

According to this definition, the producer x is likely to be executed before the consumer y due to some earlier synchronisation delay. So $x \rightarrow y$ is preserved in this sense.

To help us understand **ISSUE_SLOT_SELECTION**, the following definition is introduced.

Definition 4. *Let S be a set of scheduled instructions in the kernel. Then $RegDep(S)$ ($MemDep(S)$) is the set of all inter-iteration register (memory) dependences formed among the instructions in S. Only flow dependences are included in both cases.*

Let us first explain briefly how SMS works. Given an ordered list of the nodes in the DDG of a loop (lines 3 and 8), SMS starts with II being the minimum II, i.e., MII (line 5) and iteratively increases the II until a valid schedule is found (line 16). For a given II, the partial schedule, represented by $\mathcal{PS}$ and initialised to empty (line 9), is incrementally constructed (lines 11 and 13). When assigning an issue slot to

the current node v (line 12), SMS will choose the free available slot c (line 19) that results in no resource conflicts at the cycle (line 20). This ensures that after v has been added to $\mathcal{PS}$ (line 13), all new lifetimes introduced are minimised so that register pressure in the kernel code is reduced.

In addition to iteratively increasing II, TMS relies on two additional parameters, C_{delay} (lines 4, 7 and 26) and $P_{\max}$ (lines 1 and 26), which are also iteratively increased. When a schedule is constructed, the synchronisation delay associated with every inter-iteration register dependence must not be larger than C_{delay} (line 26.C1). The misspeculation frequency caused by the non-preserved inter-iteration memory dependences among all already scheduled instructions must not be larger than $P_{\max}$ (line 26.C2).

Guided by our cost model, we have evolved SMS into TMS by making two major modifications. First, TMS minimises the execution time T of a kernel loop rather than its II. As discussed in Section 4.2, T has two components: T_{nomiss} and $T_{\text{mis\_spec}}$. It is difficult to minimise T directly since P_M in $T_{\text{mis\_spec}}$ is defined for all instructions in the kernel but $\mathcal{PS}$ is only constructed incrementally. So TMS minimises both components separately as follows:

- **Minimisation of T_{nomiss}.** Instead of minimising T, TMS seeks to minimise T_{nomiss} as its objective function (lines 4, 5, 7 and 17). In line 4, minimising $\mathcal{F}(II, C_{\text{delay}})$ is equivalent to minimising T_{nomiss}. In line 5, $\mathcal{F}_{\min}$ is initialised with the minimum of $\mathcal{F}$ attached when II=MII and $C_{\text{delay}} = 1 + C_{\text{reg\_com}}$. By Definition 2, $1 + C_{\text{reg\_com}}$ is the smallest value for C_{delay}. TMS starts with $\mathcal{F}_{\min}$ in line 5 and iteratively increases $\mathcal{F}_{\min}$ (line 17) until a valid schedule is found such that $\mathcal{F}(II, C_{\text{delay}}) = \mathcal{F}_{\min}$ (line 7).

  II can be bounded by the longest critical path in the DDG of the loop. C_{delay} can be bounded by $II/ncore$ since not all cores can be fully utilised otherwise.

- **Minimisation of $T_{\text{mis\_spec}}$.** Instead of $T_{\text{mis\_spec}}$ or P_M given in (3), TMS minimises the misspeculation frequency calculated using (3) for all non-preserved inter-iteration memory dependences (line 26.C2). In practice, several values for $P_{\max}$ (line 1) can be tried so that the best schedule for a loop can be picked.

Second, the issue slot selection strategy in SMS is modified so that when searching for a free slot for the current node v in its scheduling window, both the synchronisation and misspeculation overheads incurred by the instructions in $\mathcal{PS} \cup \{v\}$ are also kept to presently acceptable thresholds (lines 23 – 26). In line 21 (22), $R_{\mathcal{PS}}$ ($M_{\mathcal{PS}}$) denotes the set of inter-iteration register (memory) dependences among the already scheduled instructions in $\mathcal{PS}$ that will appear in the kernel loop. In line 23 (24), R_v (M_v) contains the new ones to be introduced if v is added to $\mathcal{PS}$. In line 25, M is the set of all non-preserved inter-iteration memory dependences in the kernel with respect to the already scheduled instructions in $\mathcal{PS}$ and v. Note that synchronised and speculated dependences are handled differently. For a synchronised dependence $x \rightarrow y$, we only require its delay to be no larger than the presently acceptable threshold C_{delay} (line 26.C1). For a speculated dependence $x \rightarrow y$, some will be unlikely to cause misspeculations if they are preserved in the kernel (Definition 3). For those that are not preserved, we require the misspeculation frequency caused by the instructions in $\mathcal{PS} \cup \{v\}$ to be no larger than $P_{\max}$ (line 26.C2).

After a schedule is built, all overlapping lifetimes are renamed by introducing register copies. Thus, the distances of inter-iteration register dependences in the kernel are 1. All required `SEND` and `RECV` instructions are inserted to synchronise inter-thread register dependences for scalars.

5. Experimental results

We have implemented TMS on top of SMS in GCC 4.1.1 and evaluated the performance on a quad-core SpMT architecture using SPECfp2000 benchmarks by simulation. The effectiveness of TMS in striking a balance between ILP and TLP and in reducing synchronisation and misspeculation overheads (as motivated in Figure 2) is demonstrated with the significant speedups attained by TMS- over SMS-scheduled programs on the quad-core system. In addition, the effectiveness of TMS in parallelising DOACROSS loops is demonstrated by examining in detail the performance impact of its success in parallelising a number of frequently executed DOACROSS loops in some benchmarks.

We excluded `galgel` since it cannot be compiled successfully. All other 13 benchmarks are compiled under "-O2" except `fma3d`, which is compiled under "-O1" since a correct binary was not emitted under "-O2". In GCC 4.1.1, loops with single basic blocks and those whose branches can be converted by compare and move instructions are considered as candidates for modulo scheduling.

All benchmarks are evaluated using a detailed, execution-driven micro-architectural simulator built on top of SimpleScalar. The train input sets are used to collect profiling information. The MinneSPEC large input sets are used for simulation. All benchmarks are simulated to completion (to measure the impact of all scheduled loops).

In Section 5.1, we present the performance results achieved by TMS over SMS. In Section 5.2, we demonstrate the performance advantages of TMS over single-threaded code using a number of selected DOACROSS loops. We also explain how a balanced exploitation of ILP and TLP has contributed to these performance results. Furthermore, by comparing TMS and SMS, we show how TMS has significantly reduced execution stalls at the expense of slightly increased communication overheads by aggressively reducing C_{delay}. Finally, we give evidence that data speculation

Benchmark	#Loops	AVG	AVG	SMS			TMS		
		#Inst	MII	II	MaxLive	C_{delay}	II	MaxLive	C_{delay}
wupwise	16	16.2	4.4	5.4	14.0	5.4	9.5	12.5	3.1
swim	11	25.7	6.0	8.6	14.6	6.5	10.1	15.0	2.0
mgrid	10	34.3	8.3	14.2	15.1	14.2	15.2	26.3	5.0
applu	41	46.8	11.9	19.4	18.9	19.2	23.7	24.2	5.8
mesa	51	24.3	5.7	6.8	13.2	6.3	9.2	15.9	2.6
art	10	16.1	7.6	8.1	7.8	8.1	10.6	8.4	4.0
equake	5	43.6	11.4	12.2	16.2	11.8	16.6	17.8	5.0
facerec	26	31.7	8.0	10.5	17.4	9.5	12.7	16.5	2.9
ammp	11	35.6	9.6	13.4	13.7	13.3	16.3	14.0	4.8
lucas	24	169.6	42.2	59.2	38.7	59.1	65.8	42.2	7.9
fma3d	170	29.0	7.3	8.8	16.8	8.8	11.8	19.4	3.7
sixtrack	340	41.2	10.7	14.1	21.9	13.9	23.0	26.8	6.7
apsi	63	29.0	7.7	10.1	17.6	10.1	13.1	18.2	3.6

Table 2. SMS and TMS compared using traditional metrics for measuring modulo scheduling in single cores.

can be helpful in achieving performance in some loops.

In addition to MII, II, MaxLive and C_{delay}, where MaxLive represents the number of scalar live ranges that are simultaneously live at a program point, the following metrics are also used. The LDP of a loop is referred to the longest dependence path in the DDG of the loop. Intuitively, the MII and LDP for a given loop delineate the range of II values at which a certain degree of ILP is exploitable. The closer the II of a schedule is to MII, the higher the ILP is. So the gap between II and LDP represents roughly the degree of ILP exposed in the scheduled loop. The gap between II and C_{delay} indicates roughly the percentage of a thread that can be executed in parallel with other threads, i.e., the degree of TLP exposed in the scheduled loop.

5.1. TMS vs. SMS

Table 2 compares SMS and TMS using a total of 778 innermost loops from all the 13 SPECfp2000 benchmarks. in terms of some traditional metrics used for measuring the quality of modulo scheduling algorithms for single cores. According to Columns 3 and 4, the average thread sizes for most benchmarks are small, indicating that TMS has been designed to embrace fine-grain, communicating threads.

Let us examine the average values of the three metrics, II, MaxLive and C_{delay}, for all benchmarks by comparing SMS (Columns 5 – 7) and TMS (Columns 8 – 10). For each benchmark, TMS exhibits a larger II but a much smaller C_{delay} than SMS. On the other hand, the TMS solutions are found with respect to our cost model despite their relatively larger II values. Since TMS is aggressive in reducing C_{delay}, the average number of stage counts tends to be slightly larger, resulting in relatively larger MaxLive values. Note that the gaps between II and C_{delay} values are smaller under TMS than SMS. Thus, far more TLP has been exposed in TMS- than SMS-scheduled loops.

The more TLP exposed by TMS over SMS has mostly translated into the speedups as shown in Figure 4. Good loop speedups are observed in all benchmarks except `wupwise`. The loop speedups correlate well with the the

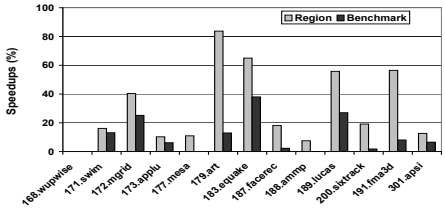


Figure 4. Speedups of TMS over SMS.

statistics given in Table 2. Take `art`, which achieves the largest loop speedup 83%, as an example. The gap between the average II and average C_{delay} achieved by TMS is far greater than that achieved by SMS. So a greater degree of TLP has been exposed by TMS. The II obtained by TMS is slightly larger, indicating that the amount of ILP exposed by TMS is slightly less. Due to a better balance achieved between ILP and TLP, the TMS-scheduled loops run much faster than the SMS-scheduled loops in this benchmark. As for `wupwise`, TMS achieves no loop speedup, and consequently, no program speedup although the TMS-scheduled loops exhibit potentially more TLP. This is because TMS has increased TLP at the expense of ILP (by nearly doubling the average II of SMS-scheduled loops). A detailed examination of the code for `wupwise` reveals that the performance-dominating loop in `wupwise` parallelised by both SMS and TMS has only one non-trivial SCC (Strongly-Connected Component). SMS has opted to fully exploit the inherent ILP with the II being close to the recurrence II while TMS has opted to exploit more TLP at the expense of ILP. As a result, both solutions are on a par in terms of their performance results.

For eight benchmarks, their loop speedups have translated into visible program speedups due to their good loop coverage ratios. The average loop and program speedups for all benchmarks are 28% and 10%, respectively.

5.2. TMS vs. Single-Threaded Code

Seven loops from four benchmarks are selected to evaluate the effectiveness of TMS in parallelising DOACROSS loops. All their enclosing loops are also DOACROSS.

Table 3 lists the benchmarks from which these loops are selected and some statistics about the selected loops and their modulo scheduled loops. Two selected loops in `art` are small (with 11 instructions each) and are thus unrolled four times. The selected loops in these benchmarks account for between 14.3% and 58.5% of the execution times of these benchmarks. These loops are fine-grained and contain between 16 and 102 instructions.

For each benchmark, the average number of SCCs, the average MII and the average length of LDPs for its selected loops are given (Columns 4 – 6). Recall that the gap between II (Column 8) and LDP (Column 7) represents roughly the potential ILP achievable in the TMS-scheduled

Benchmark	#Loops	LC	AVG #Inst	AVG #SCC	AVG MII	LDP	TMS		
							AVG II	AVG ML	AVG D
art	4	21.6%	27	3	11	29	15.5	15	5
equake	1	58.5%	82	3	20	26	27	31	6
lucas	1	33.4%	102	8	62	89	64	15	62
fma3d	1	14.3%	72	3	18	34	20	30	6

LDP: Longest Dep Path LC: Loop Coverage ML: MaxLive D: C_{delay}

Table 3. Selected loops and their TMS-scheduled loops.

loops and the gap between II (Column 8) and C_{delay} (Column 10) represents the potential TLP in the TMS-scheduled loops. Thus, the TMS-scheduled loops for the selected loops in `art` and `fma3d` exhibit good ILP and TLP. The modulo scheduled loop in `equake` exhibits TLP only and the modulo scheduled loop in `lucas` exhibits ILP only.

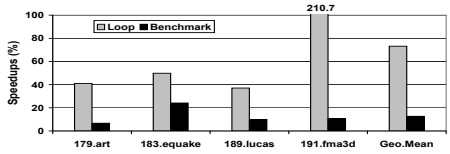
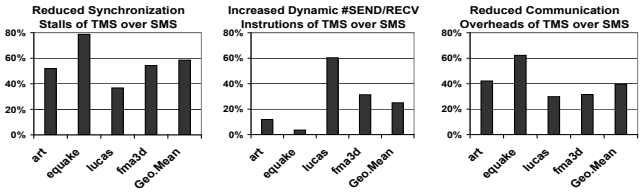


Figure 5. Speedups of TMS over single-threaded code.

Speedups Figure 5 shows that the above-mentioned improvements in ILP or TLP or both in the seven TMS-scheduled loops have resulted in improved execution times over their corresponding single-threaded programs. Both the loop speedups and the program speedups achieved by these loops alone are given. TMS has significantly improved the performance for the selected loops in all four benchmarks, resulting in the speedups between 37% to 210% with an average of 73%. Due to their good loop coverage (Column 3 in Table 3), these loop speedups have all translated into program speedups. The single loop selected in `equake` has the largest loop coverage ratio. The largest program speedup of 24% is observed in this benchmark. The loop in `fma3d` has a relatively small loop coverage ratio. So the program speedup is not as significant as its loop speedup. On average, parallelising these loops has improved the overall program performance by 12%.

Analysis In comparison with SMS and single-threaded code, the performance speedups achieved by TMS are largely due to the significant reductions in C_{delay} values at the expense of some slight increases in register communications. As mentioned earlier, the smaller C_{delay} is, the larger the part of a kernel iteration can be executed in parallel with other iterations. TMS is thus aggressive in reducing C_{delay} to expose more TLP in a loop. As shown in Table 2, TMS-scheduled loops have smaller C_{delay} and larger MaxLive values than SMS-scheduled loops.

At run time, C_{delay} values can be approximated with synchronisation stalls. The *synchronisation stall* for a loop is measured to be the total number of cycles that all its committed threads spend on stalling at a `RECV` instruction (on an empty receive queue). In Figure 6(a), we see that TMS-scheduled loops stall significantly less frequently than SMS-scheduled loops. The average MIIs of the selected loops in `art`, `equake` and fma3d are constrained by resources rather than recurrences. So their C_{delay} values can be small relatively to their IIs (Table 3). As a result, a reduction of more than 50% is observed in the three sets of selected loops. On the other hand, the largest SCC in the selected loop in `lucas` is formed by (flow) dependences whose dependence probabilities are all equal to 1. So its MII is constrained by the recurrence in the SCC. C_{delay} for the loop turns out to be larger than its MII. Thus, the reduced synchronisation overhead is slightly less impressive.



(a) (b) (c)

Figure 6. Synchronisation of TMS vs. SMS.

TMS tradeoffs communication for TLP. By using the smallest C_{delay} possible, TMS-scheduled loops tend to have more stages than SMS-scheduled loops. As a result, TMS-scheduled loops exhibit slightly larger MaxLive values than SMS-scheduled loops (Table 2). Figure 6(b) shows the percentage increases (in dynamic term) of `SEND/REVC` pairs in all committed threads for the four sets of selected loops. Even the increase for the loop in `lucas` is the largest, only three more pairs of `SEND/REVC` instructions are executed per loop iteration under TMS than SMS.

By combining the synchronisation stall cycles and `SEND/RECV` cycles spent in executing all committed threads in a loop, the *communication overhead* for the loop can be approximated. The synchronisation stall is as defined earlier and the number of `SEND/RECV` cycles is simply the product of $C_{\text{reg\_com}}$ and the dynamic number of `SEND/RECV` pairs executed. Despite some more register communications used, TMS, as shown in Figure 6(c), is effective in reducing the communication overhead in a loop relative to SMS, which is important in achieving the performance speedups given earlier in Figures 4 and 5.

Finally, let us briefly examine the impact of speculation on performance (due to space limitations). Our TMS algorithm ensures that the misspeculation frequency (over the total number of all committed threads) in these seven loops is less than 0.1%, resulting in negligible misspeculation overhead. Although misspeculations are infrequent and their performance impact on our benchmarks is negligi-

ble, data speculation can be effective in boosting the performance of certain loops. Without speculation, all inter-thread memory dependences will have to been synchronised, resulting in some loss of TLP in these loops. For example, the performance gain for the loop (program) would be reduced by 19.0% for `equake` and 21.4% for `fma3d` otherwise.

6. Conclusion

In this paper, we present for the first time a generalisation of traditional modulo scheduling for parallelising innermost loops to exploit both ILP and TLP simultaneously on SpMT architectures. We have implemented our algorithm on top of SMS adopted in GCC 4.1.1. Our algorithm is simple, achieves good performance speedups over the SMS-scheduled loops and can be effective in speeding up DOACROSS loops that are difficult to parallelise.

We are working on incorporating loop unrolling into TMS to allow us to tradeoff between communication and parallelism by varying thread granularities. We are also working on extending TMS to also parallelise outer loops.

References

[1] A. Aleta, J. M. Codina, A. Gonzalez, and D. Kaeli. Heterogeneous clustered VLIW microarchitectures. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 354–366, 2007.

[2] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–330, 2004.

[3] J. M. Codina, J. Llosa, and A. González. A comparative study of modulo scheduling techniques. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 97–106, 2002.

[4] A. Douillet and G. R. Gao. Software-pipelining on multi-core architectures. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 39–48, 2007.

[5] 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.

[6] M. Franklin. *The Multiscalar Architecture*. PhD thesis, The University of Wisconsin at Madison, 1993.

[7] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, 1998.

[8] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. Technical Report CS-TR-2007-1593, University of Wisconsin, April 2007.

[9] R. A. Huff. Lifetime-sensitive modulo scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 258–267, 1993.

[10] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *MICRO 50: Proceedings of the 40th annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[11] 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, 1998.

[12] J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, 1996.

[13] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, Washington, DC, USA.

[14] 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.

[15] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, 1994.

[16] H. B. Rong, Z. Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multi-dimensional loops. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 163, 2004.

[17] J. Sánchez and A. González. Modulo scheduling for a fully-distributed clustered VLIW architecture. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 124–133, 2000.

[18] 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.

[19] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Trans. Parallel Distrib. Syst.*, 16(2):145–162, 2005.

[20] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. 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, 2007.

[21] 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.

[22] H. Zhong, K. Fan, S. Mahlke, and M. Schlansker. A distributed control path architecture for vliw processors. In *In Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 197–206, 2005.

[23] H. T. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *HPCA*, pages 25–36, 2007.