Speculative Synchronization and Thread Management for Fine Granularity Threads

Alex Gontmakher* †Avi Mendelson Assaf Schuster Gregory Shklover
Technion, Israel Institute of Technology †Intel Labs, Haifa, Israel
{gsasha,assaf,gkovriga}@cs.technion.ac.il avi.mendelson@intel.com

Abstract

Performance of multithreaded programs is heavily influenced by the latencies of the thread management and synchronization operations. Improving these latencies becomes especially important when the parallelization is performed at fine granularity.

In this work we examine the interaction of speculative execution with the thread-related operations. We develop a unified framework which allows all such operations to be executed speculatively and provides efficient recovery mechanisms to handle misspeculation of branches which affect instructions in several threads.

The framework was evaluated in the context of Inthreads, a programming model designed for very fine grain parallelization. Our measurements show that the speedup obtained by speculative execution of the threads-related instructions can reach 25%.

1 Introduction

Modern processors use a complex of ILP-enhancing mechanisms, such as speculation and multithreaded execution. When combined carefully, these mechanisms complement and amplify each other [28]. However, the mechanisms may interfere and hurt each other's performance. Moreover, the lower the locking granularity, the more significant the performance impact of synchronization [30].

In this work, we investigate the interaction of control speculation with thread management and synchronization instructions. We develop a general framework that enables speculative execution of such instructions and provides an efficient mechanism for misspeculation recovery. The framework, based on identifying and keeping track of the interactions between instructions of different threads, provides a common mechanism that handles speculative execution of synchronization, thread starting and even thread termination. The case of thread termination involves peculiar side effects due to its reverse effect on speculation, as described in Section 3.1.

We apply the framework to *Inthreads* [6], a lightweight threading model that allows parallelization at a resolution comparable to that of speculative execution. Due to the low parallelization granularity, Inthreads programs are highly sensitive to synchronization latency and benefit from the speculative execution of thread-related operations.

The rest of this paper is organized as follows. In Section 2 we develop the framework of speculative execution of thread management and synchronization instructions. In Section 3 we apply the framework to Inthreads. Section 4 discusses the implementation and Section 5 presents the results of the experimental evaluation. Finally, Section 6 describes the related work and Section 7 concludes the paper.

2 Multithreaded Speculative Execution Model

Modern processors use a variety of mechanisms for improving the computation parallelism. Two such mechanisms are *speculative execution*, intended to improve instruction-level parallelism, and *multithreading*, aimed at thread-level parallelism. Usually, these mechanisms operate at different levels and are orthogonal. However, in case of low granularity parallelization, the synchronization may stand in the way of efficient speculative execution.

For an example, consider the program in Figure 1. When the program is executed serially, branch prediction may allow the processor to execute as many iterations in parallel as the hardware can accommodate. However, in the case of parallelized code, the presence of synchronization limits the number of iterations that can be issued speculatively: since a mutex affects execution of other threads' instructions, it is dangerous to enter the mutex before the `if` has been resolved. As a result, in each thread, the `if` must be resolved before the next `if` can be issued. Therefore, the number of iterations that can proceed in parallel is at most one per

*This work was supported in part by a research grant from Intel Israel

Sequential code	Thread t_k of $\mathbb{T}$
`for(i=0; i<N; i++){`	`for(i=k; i<N; i+=T){`
`if(d[i%K].val>i)`	`if(d[i%K].val>i)`
`{`	`{`
`d[i%K].count++;`	`MUTEX_ENTER`
`}`	`d[i%K].count++;`
`}`	`MUTEX_LEAVE`
	`}`
	`}`

Figure 1. Low granularity parallelization example

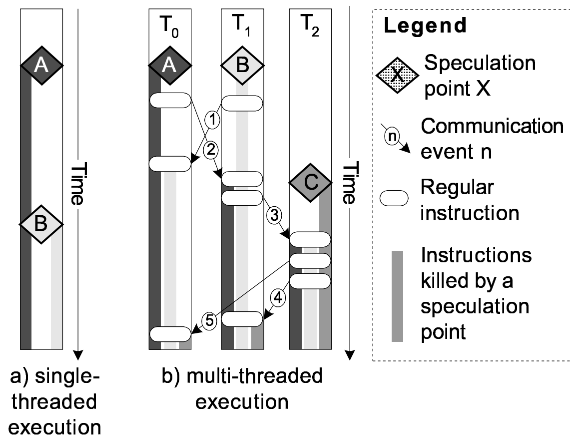

Figure 2. Speculation models for single-threaded and multithreaded execution

active thread, potentially leading to lower ILP than that of the serial, but speculative, code.

To realize the potential of both speculation and multithreading, we must enable speculative execution of instructions that involve communication between threads, such as interactions between instructions involved in a synchronization or between instructions accessing a shared variable. In order to recover from misspeculation, we must keep track of all the *communication events*, and take care of all the instructions, from all the threads, that have been affected by the misspeculated instruction.

Examples of communication events are transfer of a value through a shared variable or an interaction between synchronization instructions. For the purposes of this section, it is enough to note that a communication event consists of a pair of interacting instructions, a *producer* one providing some information to a *consumer* one.

The speculation model of *sequential computation* is linear, as shown in Figure 2a: if an instruction is misspeculated, all the following instructions are squashed. In contrast, the speculation model of *multithreaded computation* is non-linear. Consider the execution in Figure 2b with threads T_0, T_1 and T_2, three speculation points A, B and C, two communication events, 1 and 2, between T_0 and T_1, two events, 3 and 4, between T_1 and T_2, and a communication 5 from T_2 to T_0. It is impossible to arrange the instructions so that all the instructions following a speculation point are those affected by it. For example, A and B are independent, and neither of them should precede the other.

Another observation is that misspeculation recovery is timing sensitive. Consider the misspeculation recovery of B. The misspeculation propagates to T_2 along event 3, squashing the instructions following the consuming instruction of event 3. As a result, the producing instruction of event 4 is squashed, and therefore, the consuming instruction of 4 must be squashed as well. However, that instruction may have been already squashed since it follows B in the program order of T_1, and the pipeline may already contain the instructions for the correct execution path after B. In that case, the processor would squash instructions on the correct execution path. The problem becomes more acute when misspeculation is propagated through longer sequences of events, such as the sequence of 2, 3 and 5 that would result from misspeculation of A.

A straightforward, albeit expensive, approach to this problem would require stopping the front-end during the propagation of branch misprediction. Our approach, described below, performs the recovery in one step without the need for propagation.

2.1 Speculative Execution Framework

The definitions in this section are conceptually similar to the *happens-before relation* defined by Lamport [14] and the related notion of *vector clocks* by Mattern [18]. These works are developed for distributed processors over a not necessarily in-order medium. Furthermore, the feasibility of bounded timestamps is shown in [5, 7, 11], although system is not truly distributed and thus is simpler.

Our framework is defined over a set of threads rather than processors, and directly models the control speculation and the interactions between instructions. The framework can be applied to an architecture by identifying the possible interactions, and handling them as described in Section 2.2. In Section 3.1 we apply the framework to Inthreads.

Let $\mathbb{T}$ be the set of threads supported by the processor. We model the program as a set of instructions organized in *program orders* for each thread $t \in \mathbb{T}$. We denote that instruction i belongs to the program order of thread t by $tid_i = t$. The program order defines a *full order* for instructions of each thread. We denote that instruction i precedes instruction j in the program order by $i \prec j$.

Execution of the program proceeds through *sliding windows* of instructions. There is one sliding window W_t for each thread t. W_t operates as a FIFO: instructions are *intro-*

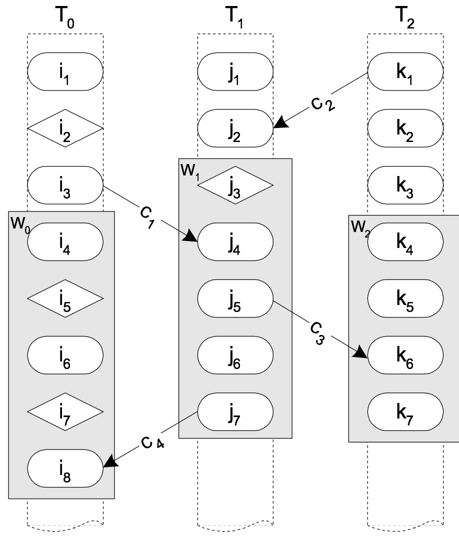

Figure 3. Example execution with speculation and communication events

duced at one end and *retired* at another. $W_t(\tau)$ denotes the contents of W_t at time τ and $W(\tau) = \bigcup_{t \in \mathbb{T}} W_t(\tau)$.

Instructions are classified into *plain* and *speculation point* ones. The speculation point instructions are introduced as *unresolved*. When resolved, a speculation point instruction can be found to be *misspeculated*, in which case all the following instructions must be squashed. We denote the set of speculation point instructions by $\mathbb{S}$ and the set of currently unresolved ones by $\mathbb{S}_?(\tau)$ (clearly, $\mathbb{S}_?(\tau) \subseteq \mathbb{S} \cap W(\tau)$). Also, we define $\mathbb{S}_?(\tau, t) = \mathbb{S}_?(\tau) \cap W_t(\tau)$.

Some instructions participate in *communication events*. A communication event c consists of two instructions, the *producing* one $\overset{\nearrow}{c}$ and the *consuming* one $\overset{\searrow}{c}$ ($c = (\overset{\nearrow}{c}, \overset{\searrow}{c})$). c introduces a *dependency*: if $\overset{\nearrow}{c}$ is squashed, then $\overset{\searrow}{c}$ must be squashed as well. $i \rightsquigarrow j$ denotes that j depends on i through some chain of dependencies. We denote the set of communication events by $\mathbb{C}$.

In the example in Figure 3, $\mathbb{S} = \{i_2, i_5, i_7, j_3\}$, $\mathbb{S}_?(\tau) = \{i_5, i_7, j_3\}$, and $\mathbb{C} = \{c_1 = (i_3, j_4), c_2 = (k_1, j_2), c_3 = (j_5, k_6), c_4 = (j_7, i_6)\}$.

Each instruction i has a *timestamp*, ts_i. The timestamps are assigned independently for each thread, in such a way that $ts_i < ts_j \Leftrightarrow i \prec j$. A *timestamp vector*, tsv_i, defines the latest instruction in each thread t that i depends on:

$$tsv_i(t) = \begin{cases} max_{c \in \mathbb{C}_P(i)} ts_{\overset{\nearrow}{c}}, & t \in \mathbb{T} \setminus \{tid_i\} \\ ts_i, & t = tid_i \end{cases},$$

where $\mathbb{C}_P(i)$ is the set of instructions that affect i through a chain of communication events $c_1, \ldots, c_n$:

$$\mathbb{C}_P(i) = \left\{ c : \begin{array}{l} \exists_{c_1,\ldots,c_n \in \mathbb{C}}, c = c_1 \wedge \overset{\searrow}{c}_n \prec i \wedge \\ \forall_{j \in \{1 \ldots n-1\}} (\overset{\searrow}{c}_j \prec \overset{\nearrow}{c}_{j+1}) \end{array} \right\}.$$

The value of tsv_j can be used to determine if j depends on a given instruction i:

$$i \rightsquigarrow j \Leftrightarrow ts_i \leq tsv_j(tid_i)$$

When a speculation point instruction b is found to be mispredicted, we need to squash all the instructions $\{i \in W(\tau) : tsv_i(tid_b) > ts_b\}$. Consequently, the set of speculative instructions at time τ is $\{i \in W(\tau) : \exists_{b \in \mathbb{S}_?(\tau)} tsv_i(tid_b) > ts_b\}$.

In the example in Figure 3, $tsv_{j_4} = [3, 4, 1]$, $tsv_{j_2} = [\varnothing, 2, 1]$, $tsv_{k_6} = [3, 5, 6]$ and $tsv_{i_8} = [8, 7, 1]$. If j_3 is mispredicted, checking the tsv reveals that both i_8 and k_6 must be squashed. The speculative instructions (i.e., those that might be squashed) are i_8 in T_0, $j_4 \ldots j_7$ in T_1 and, because of communication event c_3, $k_6 \ldots k_7$ in T_2. Note that j_2 is not speculative, since no unresolved speculation point instructions precede k_1.

In the remainder of this section, we will develop efficient ways for computing the set of speculative instructions and for performing misspeculation recovery.

First, if an instruction i preceded by i' does not take part in a communication event, then $tsv_i(t) = tsv_{i'}(t)$ for any thread $t \neq tid_i$. Therefore, in recovering from a misspeculation from a speculation point b, the earliest squashed instruction in every thread except tid_b must be an instruction that takes a consumer part in a communication. As a result, it is sufficient to just scan the instructions in $\mathbb{C} \cap W(\tau)$ in order to determine which instructions must be killed as a result of misspeculation. In addition, it is sufficient to compute tsv only for the communicating instructions.

Second, we note that since the timestamps are monotonically increasing in each thread, we need to scan only the earliest unresolved speculation point in each thread in order to determine which instructions are speculative.

To summarize, the determination of speculative instructions and the misspeculation recovery are performed in the following way. First, we compute the set of earliest speculation points in each thread by scanning all the currently unresolved speculation point instructions:

$$\mathbb{S}_{\mathbb{E}}(t) = ts_b : b \in \mathbb{S}_?(\tau, t) \wedge \nexists_{b' \in \mathbb{S}_?(\tau,t)} b' \prec b$$

Using $\mathbb{S}_{\mathbb{E}}$, we can determine whether a given communication instruction $\overset{\searrow}{c}$ is speculative:

$$spec_{\overset{\searrow}{c}} = \exists_t \mathbb{S}_{\mathbb{E}}(t) < tsv_{\overset{\searrow}{c}}(t)$$

Now we can compute the timestamps of the earliest speculative communication instructions in each thread:

$$\mathbb{C}_{\mathbb{E}}(t) = ts_{\overset{\searrow}{c}} \; : \; \begin{array}{l} c \in \mathbb{C} \cap W_t(\tau) \wedge spec_{\overset{\searrow}{c}} \\ \wedge \nexists_{\overset{\searrow}{c}_1 \prec \overset{\searrow}{c}} : spec_{\overset{\searrow}{c}_1} \end{array}$$

With $\mathbb{S}_{\mathbb{E}}$ and $\mathbb{C}_{\mathbb{E}}$, we can determine whether instruction i is speculative by checking that $ts_i \geq \mathbb{S}_{\mathbb{E}}(tid_i) \vee ts_i \geq \mathbb{C}_{\mathbb{E}}(tid_i)$. The speculative instructions may not be retired even if their execution is completed.

When a speculation instruction b is found to be misspeculated, we determine the set of instructions to be squashed in the following way. For each thread, we scan the communication instructions to determine the earliest squashed one:

$$\mathbb{C}_{\mathbb{E}}^{\mathbb{S}\mathbb{Q}}(b, t) = ts_{\overset{\searrow}{c}} \; : \; \begin{array}{l} c \in \mathbb{C} \cap W_t(\tau) \wedge \\ tsv_{\overset{\searrow}{c}}(tid_b) > ts_b \wedge \\ \nexists_{\overset{\searrow}{c}_1 \prec \overset{\searrow}{c}} : tsv_{\overset{\searrow}{c}_1}(tid_b) > ts_b \end{array}$$

Instruction i must be squashed during the recovery of b if $ts_i \geq \mathbb{C}_{\mathbb{E}}^{\mathbb{S}\mathbb{Q}}(b, tid_i)$.

Finally, the computation of tsv is performed according to its definition by scanning the instructions in $\mathbb{C} \cap W(\tau)$.

Returning to the execution in Figure 3, the computations would proceed in the following way: $\mathbb{S}_{\mathbb{E}} = [5, 3, \varnothing], \mathbb{C}_{\mathbb{E}} = [8, 4, 6]$, exactly as in the original definitions. If j_3 is misspeculated, then $\mathbb{C}_{\mathbb{E}}^{\mathbb{S}\mathbb{Q}}(j_3) = [8, 4, 6]$, implying that the instructions squashed in addition to the instructions following j_3 in T_1 are i_8, k_6 and k_7.

2.2 Handling Communication Instructions

For each instruction i that can potentially take part in a communication event, we must choose one of the two alternative implementations:

- *Issue i speculatively.* In this case, we need to dynamically determine the instructions that participated in a communication with i. The tsv for these instructions must be updated accordingly to enable proper recovery.
- *Delay i until it becomes non-speculative.* Naturally, the instruction that participates in communication with i will not be able to issue before i. On the positive side, squashing of i cannot affect instructions from other threads, and thus the tsv of the instruction receiving the communication need not be updated.

3 Speculation in the Inthreads Model

The Inthreads model is based on a fixed number of threads running over shared registers in the context of a single SMT thread. The model provides an extremely lightweight threading mechanism: the fixed number of threads allows for thread management to be performed entirely in hardware. In addition, the shared registers provide a straightforward and efficient communication mechanism: a value can be transferred by writing it into a register in one thread and reading it from the same register in another.

Each thread has a thread ID (tid) which is determined at thread creation. The *main thread*, identified by $tid = 0$, is always active, while other threads can be started and terminated on demand. Three instructions control the starting and stopping of threads: inth.start, inth.halt and inth.kill. inth.start creates a new thread with a given tid at a given address. To terminate itself, a thread issues an inth.halt instruction. inth.halt is executed synchronously, guaranteeing that all the instructions preceding it will complete. One thread can kill another by issuing an inth.kill instruction.

The synchronization mechanism consists of a set of binary semaphores stored in *condition registers* controlled by three instructions: cond.wait, cond.set and cond.clr. A cond.wait checks whether a given condition is set. If it is, the condition is cleared; otherwise the issuing thread is stalled until some other thread performs a cond.set to that condition. If several cond.wait instructions accessing the same condition are issued in parallel, only one of them will proceed. A cond.set sets the given condition. If there was a cond.wait suspended on the same condition, the cond.wait is awakened and the condition remains cleared. Finally, a cond.clr clears the given condition.

The programming model is described in detail in [6]. A similar architecture, although using a dedicated namespace for shared registers, is described by Jesshope [12].

3.1 Communication Events in the Inthreads Model

A *Synchronization* event occurs between a cond.set and a cond.wait that accesses the same condition. Synchronization events are relatively easy to detect, as the number of synchronization instructions that must be concurrently in flight is low (see Section 5). cond.wait and cond.clr instructions never take a producer part in communication, and can be executed speculatively with no need for recovery.

A *Variable value transfer* event occurs between any instruction writing a value to a variable (whether in a register or at some memory location) and a subsequent instruction reading the value from that variable.

In contrast to the communication through synchronization, communication through regular variables is harder to detect: any two instructions can potentially communicate, and moreover, communication between memory instructions can be detected only after the addresses for both instructions have been computed.

The Inthreads architecture handles the communication

through variables at the architecture level. To this end, the Inthreads-parallelized programs are required to be *data-race-free* [1], i.e., to ensure that any two instructions that access the same location are separated by synchronization. As a result, recovery of synchronization instructions implies correct squashing of all instructions involved in shared-variable communication.

A *Thread starting* event results from the communication between an inth.start and the first instruction of the started thread. Thread starting events are handled similarly to the synchronization ones. The only difference is that the instruction receiving the communication does not belong to a specific instruction type, but is just the first instruction started by the thread. For this, we hold a tsv for every thread, as if the whole thread receives the communication.

A *Thread killing* event occurs between an inth.kill and the first killed instruction of the target thread. While an inth.kill does not supply any "real" information, it does interact with the killed thread since the instructions following the inth.kill should not interact with the ones killed by it.

The situation is further complicated by the fact that inth.kills have a reverse effect on speculation: the instructions in the target thread are squashed only if the inth.kill is not misspeculated. Therefore, it is not sufficient to delay execution of an inth.kill until it becomes non-speculative, as instructions following it might interact with the target thread. Moreover, since an inth.kill terminates other instructions, recovery would be considerably more complex than just killing the dependent instructions. As a result, in this work we consider the inth.kill instructions as communication events, but execute them only non-speculatively.

4 Implementation

In principle, the architecture of Inthreads is quite similar to SMT—both architectures execute multiple independent instruction streams on a shared execution core. Therefore, the microarchitecture re-uses most of the mechanisms present in SMT processors, such as multiple fetch units, multiple ROBs, shared physical register file and functional units and so on (a notable exception is the *Register Allocation Table* (RAT), which is shared between threads in an Inthreads processor). Therefore, we take the SMT microarchitecture as a basis, and only describe the design of the thread management mechanisms.

The mechanisms that support Inthreads execution are outlined in Figure 4. The pipeline includes an additional stage, *Instruction Wait*, which implements delaying instructions of the threads which waiting on a condition. The delayed instructions are stored in the per-thread *Wait Buffers* (WBs). The WBs read the state of condition variables from the *Available Conditions* line to determine which instructions can be released.

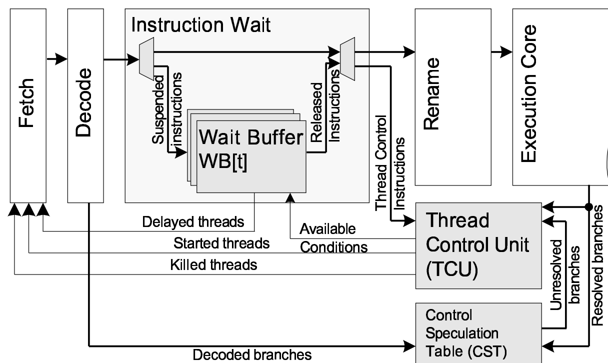

Figure 4. Inthreads microarchitecture outline

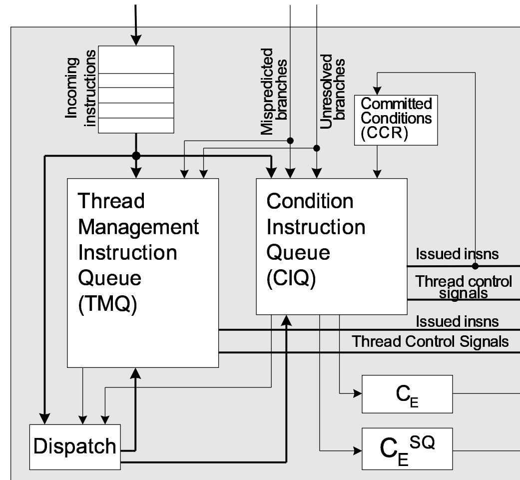

Figure 5. Thread Control Unit

The *Thread Control Unit* (TCU), described in Section 4.1, orchestrates the execution of the threads on the processor. The *Condition Speculation Table* (CST) is used to keep track of the unresolved branch instructions. The computations performed by the TCU and the CST correspond to the multithreading speculation model developed in Section 2 and the Inthreads-specific details in Section 3.

4.1 Thread Control Unit

The TCU, depicted in Figure 5, is used to 1) carry out the side effects of the synchronization and thread management instructions, 2) compute which instructions are speculative, and 3) control the squashing of dependent instructions in all the threads in response to a mispredicted branch.

The TCU holds the instructions in two queues, *CIQ* for the synchronization instructions and *TMQ* for the thread management instructions. Both CIQ and TMQ keep tsv of all the contained instructions. The CIQ and TMQ receive

Parameter	No threads	Inthreads
Pipeline Length	8	9
Supported threads	N/A	8
Fetch policy	8 per cycle	ICOUNT2.8
Branch predictor	bimodal	
L1I size	64KB	
Logical Registers	64GP+64FP	
Physical Registers	512GP,512FP	384GP,384FP
ROB size	512	128*8
Issue Queue size	80	
Memory Queue size	40	
Functional units	6 Int,6FP,4 Branch	
Max outst. misses	16	
Max unresolv. branches	16	
Max active synch insns	16	
Memory ports	2	
L1D size, latency	64KB, 1 cycle	
L2 size, latency	1MB, 20 cycles	
Memory latency	200	

Table 1. Basic processor parameters

from the CST the timestamps $\mathbb{S}_{\mathbb{E}}$ of the currently unresolved branches and use them to determine which of the instructions they contain are non-speculative and can be issued. In addition, the CIQ and TMQ receive information on branch mispredictions, in order to control the squashing of instructions from different threads.

Execution of synchronization instructions in the CIQ involves the computation of the currently available conditions, which are used by the WBs to determine which cond.wait instructions can be released. The *Committed Conditions register* (CCR) holds the committed state of the condition variables. The value of the CCR is fed into the CIQ, which updates it according to the synchronization instructions (cond.wait and cond.clr instructions clean the corresponding bit, and cond.set instructions set it). When instructions are issued from the CIQ, they update the corresponding conditions in the CCR.

5 Evaluation

We have extended the SimpleScalar-PISA model [3] with the Inthreads-related extensions. The basic processor is modelled with a 8-stage pipeline, and the Inthreads-enabled variant has one more stage for synchronization functionality. Fetching was managed by the ICOUNT policy [29]. Table 1 summarizes the parameters.

We assume that the instructions dispatched to the TCU execute in as few as two cycles: one to compute the tsv, and one cycle to issue the instruction if it is ready. The logic involved in the TCU is similar to that of the Issue Queue, while the number of instructions held in the TCU is significantly lower: in our experiments there was no measurable effect to increasing the TCU size over 16 instructions. Still, we measured the effect of increasing the TCU latency, shown in Figures 7 and 8.

The evaluation is based on three benchmarks from the SPEC2K suite [8]: 179.art, 181.mcf and 300.twolf, and four programs from the MediaBench suite [15]: Adpcm encode, Adpcm decode, G721 and Mpeg2. As a measure of the programs' run time we used the number of clock cycles reported by the simulator.

We denote the speculation settings with three letters, one for each speculation mechanism from those described in Section 3.1. For example, TFF means that the speculative execution of synchronization instructions was turned on, and speculative value transfer between variables and speculative thread starting was turned off.

5.1 Microbenchmark

To explore the performance behavior of the speculation mechanisms, we have implemented a microbenchmark with a tunable frequency of thread management events. The benchmark contains two nested loops. The inner loop consists of four independent sequences with heavy branching. The inner loop size, or the *iteration size* of the outer loop, determines the frequency of the synchronization, thread creation and termination events. The number of iterations of the outer loop determines the total number of such events.

The results are summarized in Figure 6. The first row of the graphs shows the speedup in comparison with the serial code, while the second one displays just the speedup that results from turning on speculation. The third row displays the percentage of squashed instructions.

The first three columns plot the performance for a fixed iteration size and varying number of iterations. The speedup remains relatively stable, except at a small number of iterations due to uneven branch prediction. Predictably, the speedup grows with the iteration size as the speedup caused by parallelization overcomes the thread management overhead. A more interesting effect is the growth in the additional speedup that results from speculation (row 2). Speculative execution of thread instructions allows different iterations of the outer loop to proceed in parallel, which is more important with a larger iteration size where the difference in the amount of work performed by the threads grows.

The rightmost three columns of Figure 6 show the behavior of parallelization for a given number of iterations of the outer loop. We can see that the speculation reaches its potential only with a relatively large number of iterations.

Finally, Figure 7 shows the effect of the latency of the TCU on performance. For the smallest possible parameters, a 14-cycle latency cancels out the benefits of parallelization. With larger parameters, the communication becomes less frequent, and the code is less sensitive to the latency.

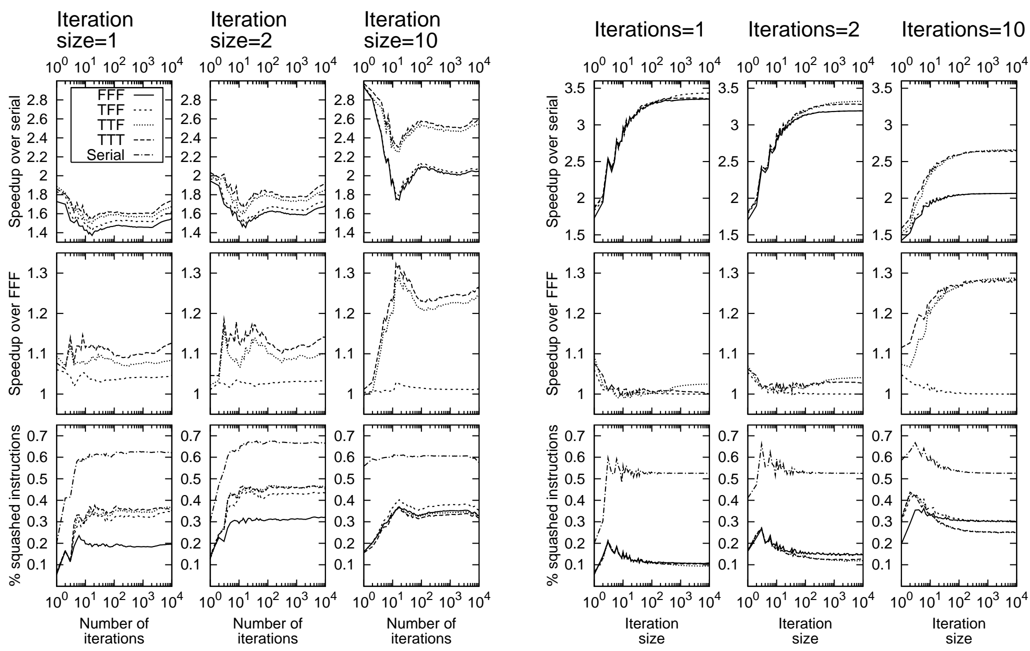

Figure 6. Behavior of the microbenchmark at various size options

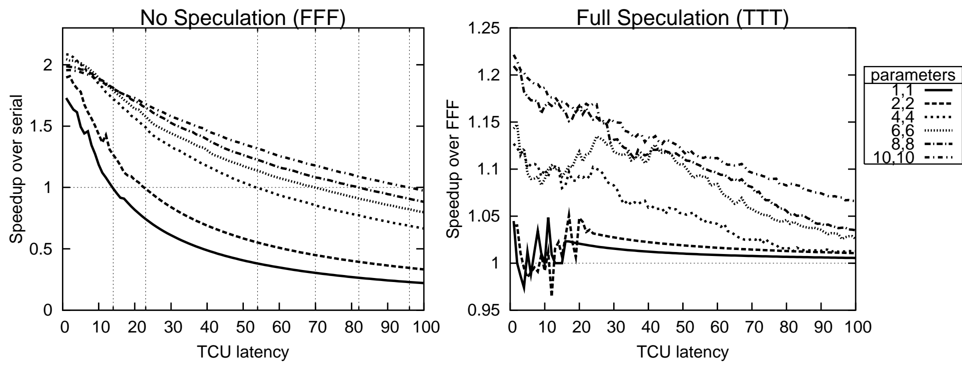

Figure 7. Speedup of the microbenchmark as a function of the TCU latency

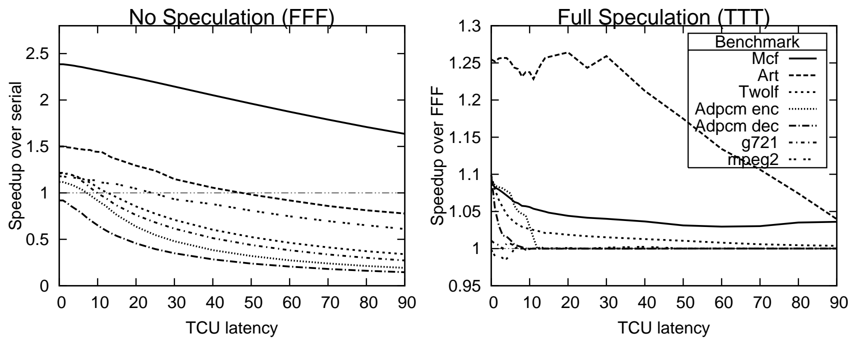

Figure 8. Performance of SPEC and Mediabench benchmarks under varying TCU latency

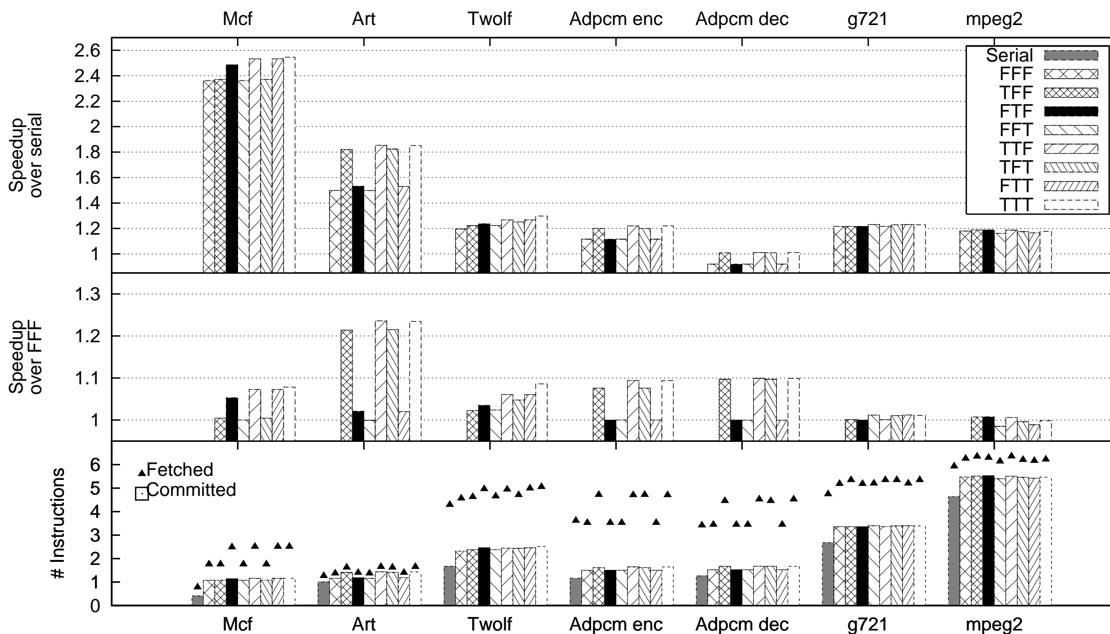

Figure 9. Performance of SPEC and Mediabench benchmarks with speculative execution

5.2 SPEC2000 and Mediabench

The benefit of speculative execution depends on the amount of time saved by earlier execution of thread-related instructions and the frequency of such instructions. Table 2 shows the frequencies and average age of the cond.set and inth.start instructions, measured from the time they enter the TCU and until they are issued.

Benchmark	cond.set		inth.start	
	Age	Freq	Age	Freq
Mcf	41	0.018	7.0	0.003
Art	32	0.04	391	0.000019
Twolf	4.4	0.057	3.3	0.014
Adpcm enc.	6.3	0.059	2.0	0.000029
Adpcm dec.	2.4	0.061	3.0	0.000031
G721	3.7	0.05	3.1	0.012
Mpeg2	1.5	0.014	9.1	0.014

Table 2. Average ages and frequencies of thread-related instructions

The results of applying speculative execution of thread-related instructions to the benchmarks are summarized in Figure 9. The first row shows the overall speedup, and the second one—the speedup increment caused by speculation.

Mcf parallelizes consequent iterations that communicate heavily. This communication is sped up when executed speculatively, explaining the sensitivity of Mcf to speculative variable transfer. Art uses very fine grain synchronization, and thus receives most improvement from speculation on synchronization instructions. Both Art and Mcf perform thread starting relatively unfrequently, and therefore do not benefit from speculation in thread starting.

In contrast, Twolf uses threads to parallelize small independent sections of code with heavy branching, and benefits from all the forms of speculative execution.

Both Adpcm programs split the execution into portions executed by threads arranged in a virtual pipeline, using synchronization at a low granularity. As a result, execution is sped up by speculative synchronization.

Both g721 and mpeg2 are barely affected by the speculation, albeit for opposite reasons. In g721, the hard-to-predict branches are executed just before synchronization, resulting in poor speculation success rate. In contrast, mpeg2 has long non-speculative sequences, obviating the need for speculative synchronization.

The effect of the TCU latency is shown in Figure 8. Mcf and Art are least sensitive due to the relatively long age of the synchronization instructions. The speedup of adding speculation to thread-related operations, shown in Figure 8b, decreases with the latency when it grows larger than the benefit of speculation. It is interesting to note that the speculation speedup for Mcf almost does not decrease with latency. The reason is that when the latency grows, additional parallelism is achieved by an increase in the number of iterations that execute in parallel.

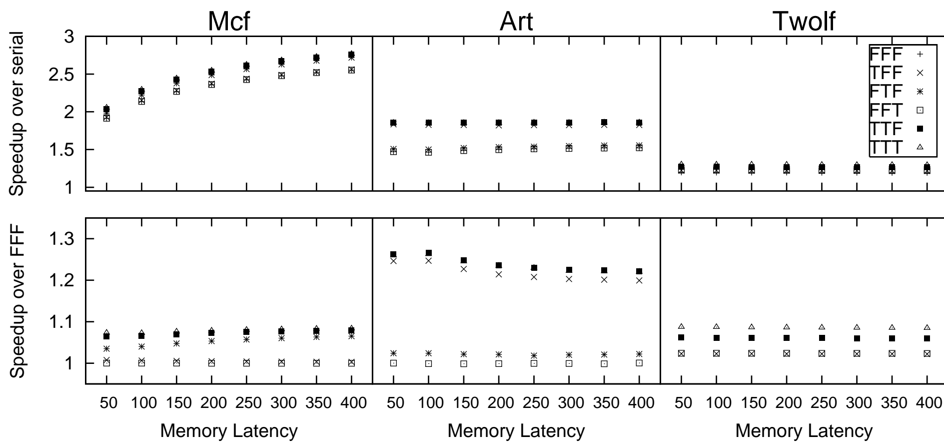

Figure 10. Performance of SPEC benchmarks under varying memory latency

Finally, for the SPEC benchmarks we have measured the effect of memory latency on speedup, shown in Figure 10. While the overall speedup may be sensitive to memory, the additional benefit of speculation remains almost constant.

6 Related Work

Synchronization operations are often redundant and can be ignored speculatively. Examples are *Speculative Synchronization* [17], *Transactional Memory* [9] and *Speculative Lock Elision* [21]. In contrast, just speed synchronization up by executing it speculatively. This avoids the need to detect collisions, and also allows speculation in other thread operations, like starting and terminating.

Thread Level Speculation issues threads derived from a serial program, speculating on a fact that the threads will turn out to be independent. In *Multiscalar processors* [24], the program is partitioned into speculative tasks that execute concurrently while observing data dependencies. *Speculative multithreaded processors* [16] and *Dynamic multithreading processors* [2] generate threads at control speculation points. In those works, the threads are usually very small, often with limited control flow. In contrast, the mechanisms in our work apply to general threads and provide a unified mechanism for speculation of both synchronization and thread management instructions. Other works in this area are [10, 13, 19, 20, 25, 26, 27].

The high cost of branch misprediction has prompted several works that aim at reducing the penalty by retaining those instructions that would execute in the same way regardless of the branch outcome [4, 22, 23]. Our approach can be seen as achieving about the same in software by speculatively starting threads with dependencies marked up by communication instructions.

7 Conclusion

This work provides a unified framework for speculative execution of multithreading-related instructions, including both synchronization primitives and instructions for starting and killing of threads. The framework can be applied to a given architecture by identifying the interactions between instructions of different threads.

The Inthreads architecture has four types of interactions: synchronization, data transfer through variables, thread starting and thread termination. All the types except for data transfer are linked to specific instructions and are easy to detect. The transfer through variables is handled by requiring the programs to be data-race free. As a result, misspeculation recovery of synchronization instructions implies correct recovery shared variable communication.

An additional aspect of this work is that our speculation mechanisms, due to the sharing of the processor resources among several threads, provide a good trade-off between the performance improvement and the increase in the percentage of squashed instructions. This implies that the techniques can be used for improving the power efficiency. We plan to explore this issue in depth in our future work.

References

[1] S. V. Adve and J. K. Aggarwal. A unifi ed formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, 1993.

[2] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 226–236. IEEE Computer Society Press, 1998.

[3] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Re-

port CS-TR-1996-1308, University of Wisconsin-Madison, 1996.

[4] R. Desikan, S. Sethumadhavan, D. Burger, and S. W. Keckler. Scalable selective re-execution for edge architectures. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 120–132, New York, NY, USA, 2004. ACM Press.

[5] D. Dolev and N. Shavit. Bounded concurrrent time-stamp systems are constructible. In *Proceedings of the twenty-fi rst annual ACM symposium on Theory of computing*, pages 454–466, New York, NY, USA, 1989. ACM Press.

[6] A. Gontmakher, A. Schuster, A. Mendelson, and G. Shklover. Inthreads —a new computational model to enhance ILP of sequential programs. Technical Report CS-2005-16, Technion, Israel Institute of Technology, 2005.

[7] S. Haldar and P. Vitányi. Bounded concurrent timestamp systems using vector clocks. *J. ACM*, 49(1):101–126, 2002.

[8] J. L. Henning. Spec cpu2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[10] K. Hiraki, J. Tamatsukuri, and T. Matsumoto. Speculative execution model with duplication. In *Proceedings of the 12th international conference on Supercomputing*, pages 321–328, New York, NY, USA, 1998. ACM Press.

[11] A. Israeli and M. Li. Bounded time-stamps. In *Proceedings of the 28th Annum Symposium on Foundations of Computer Science*, pages 371–382, 1987.

[12] C. Jesshope. Scalable instruction-level parallelism. In *3rd and 4th Intl. Workshops on Computer Systems: Architectures, Modelling and Simulation*, pages 383–392, July 2004.

[13] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 59–70, New York, NY, USA, 2004. ACM Press.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[16] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *Proceedings of the International Conference on Supercomputing (ICS-98)*, pages 77–84, New York, July 13–17 1998. ACM press.

[17] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 18–29, New York, NY, USA, 2002. ACM Press.

[18] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[19] C.-L. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi , and T. N. Vijaykumar. Multiplex: unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the 15th international conference on Supercomputing*, pages 368–380, New York, NY, USA, 2001. ACM Press.

[20] I. Park, B. Falsafi , and T. N. Vijaykumar. Implicitly-multithreaded processors. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 39–51, New York, NY, USA, 2003. ACM Press.

[21] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[22] E. Rotenberg and J. Smith. Control independence in trace processors. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 4–15, Washington, DC, USA, 1999. IEEE Computer Society.

[23] A. Roth and G. S. Sohi. Register integration: a simple and effi cient implementation of squash reuse. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 223–234, New York, NY, USA, 2000. ACM Press.

[24] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, New York, NY, USA, 1995. ACM Press.

[25] G. S. Sohi and A. Roth. Speculative multithreaded processors. *Computer*, 34(4):66–73, Apr. 2001.

[26] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 171–182, Washington, DC, USA, 2002. IEEE Computer Society.

[27] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[28] S. Swanson, L. K. McDowell, M. M. Swift, S. J. Eggers, and H. M. Levy. An evaluation of speculative instruction execution on simultaneous multithreaded processors. *ACM Trans. Comput. Syst.*, 21(3):314–340, 2003.

[29] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, 1995.

[30] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fi ne-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the Fifth Annual International Symposium on High-Performance Computer Architecture*, pages 54–58, Jan 1999.