Parallel Techniques of the Sequential Codes Based on Multi-core

Xiang Li and Jing Zhang

1School of Computer Science and Engineering, Xi’an University of Technology, Xi’an, 710048, China
2School of Electrical and Information Engineering, Shaanxi University of Science and Technology, Xi’an, 710021, China

Abstract: Multi-core processors are becoming ubiquitous with the continuous development of hardware technology. But many applications are sequential applications and they receive no benefits running on multi-core processors. Addressing this problem, using parallel techniques improve the sequential program running on the multi-core processors. In this paper, we introduced three parallel types of thread level parallelism. Typical DOALL, DOACROSS, DSWP and PS-DSWP techniques are described. These techniques can explore the parallel from sequential application, but much dependence is not easily predictable or manifests them infrequently by the non-speculative transformation. So many speculative techniques, such as thread level speculation (TLS), Speculation DSWP (SpecDSWP), Speculative PS-DSWP (SpecPS-DSWP) and Interprocedural SpecPS-DSWP (iSpecPS-DSWP), are proposed to break problematic dependences to enhance parallelism. We introduced these speculative parallel techniques and described their execution. SpecDSWP, SpecPS-DSWP and iSpecPS-DSWP are compared from supporting speculation types, memory version and implement steps. At last, some extended TM systems which support TLS techniques are analyzed from thread spawning mechanism, context passing mechanism and sequential ordering.

Key words: Multi-core, speculative parallel techniques, thread level speculation, decoupled software pipelining, transactional memory

INTRODUCTION

Multi-core processors are becoming ubiquitous with the continuous development of hardware technology. The number of cores per chip is expected to rise in future, such as Intel’s 4-core Core i7, IBM 8-core Power 7, AMD’s 16-core Interlagos, Tilera’s 64-core Tile 64 and Intel’s 80-core Teraflops processors. Multi-core processors can improve system throughput and speed up multi-threaded applications. It can be seen from the Fig. 1 that many important applications are single threaded and single-threaded applications receive no benefits running on multi-core processors (Vachharajani et al., 2007). Therefore, how to using the hardware parallelism of multi-core processors to speed single applications is a challenge. Since loops of sequential applications are the largest source of parallelism considerable attention on multi-core processors (Hursen et al., 1997; Zhang et al., 2012a). This study presents and analyzes several thread partitioning techniques based on multi-core and some extended TM systems which support TLS techniques.

BACKGROUNDS AND RELATED WORK

Parallel types of thread-level parallelism: Independent Multi-Threading (IMT), Cyclic Multi-Threading (CMT) and Pipelined Multi-Threading (PMT) are three categories techniques for extracting parallelism from loops of TLP. IMT transformation prohibits cross-thread dependences and has restricted iteration level parallelism. The typical IMT transformation is DOALL technique which usually used to parallelize loops in array-based scientific programs. CMT transformation which is suitable for general purpose applications allows cross-thread dependences and maintains correct execution dependences among threads by synchrony. DOACROSS technique is the major technique in this category. PMT also allows cross-thread dependences but it only allows dependence of one direction. The first proposed PMT technique is DOPipe (Davies, 1981) which is restricted to only loops with limited control flow. Decoupled Software Pipelining (DSWP) (Bridges, 2008) is a more general PMT technique to extract pipelined parallelism from loops with arbitrary control flow. It can be seen the difference of DOALL, DOACROSS and DSWP techniques from the Fig. 2.

Dependence graphs: There have been several ways to perform parallelism in a sequential application. The first way is Data Dependence Graphs (DDG), which adopt a node to represent the execution of an instruction and adopt edges to represent the dependences between
TRADITIONAL THREAD PARTITIONING TECHNIQUES BASED ON MULTI-CORE

Thread-level program parallelization is the key for exploiting the parallelism of the multi-core processors. Several techniques have been proposed for program multithreading, typically DOALL, DOACROSS, DSWP, PS-DSWP (Prabhu et al., 2011).

DOALL parallelization: DOALL parallelization methods of the typical IMT technique are highly efficient because of none the cross-thread communication among iterations. DOALL executes loop iterations in separate threads to prove none inter-iteration dependences. The requirement that the iteration space be divisible prohibits pointer-chasing loops, which are common in general purpose applications, so DOALL parallelization methods usually is used to parallel scientific programs.

DOALL loop algorithms are divided into two categories, shared memory with uniform memory and non-uniform memory access (NUMA). DOALL algorithms that shared memory with uniform memory DOALL have static and dynamic methods. Block Scheduling (BS), static chunking and cyclic scheduling is typical static DOALL loop algorithm. The advantage of static DOALL loop algorithm is lower runtime scheduling overhead while the disadvantage is unbalanced distribution of load among processors. Dynamic loop scheduling algorithms for DOALL loops are self-scheduling (SS), fixed-size chunking (FS), guided self-scheduling (GSS), factoring, trapezoid self-scheduling (TSS) and etc. The advantage of dynamic scheduling is better load balance because of determines the division of iterations among processors at runtime while disadvantage is high overhead. DOALL algorithms
Table 1: Typical DOALL algorithms comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Shared Memory</th>
<th>Scheduling</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block scheduling</td>
<td>Uniform memory</td>
<td>Static</td>
<td>Assign iterations 1-N/P to the first processor, iteration NT=1-2*N/P to the second processor</td>
</tr>
<tr>
<td>Cyclic scheduling</td>
<td>Uniform memory</td>
<td>Static</td>
<td>Allocate iterations i, i+P, i+2P, ..., to processor i (1, i+P)</td>
</tr>
<tr>
<td>Fixed-size chunking</td>
<td>Uniform memory</td>
<td>Dynamic</td>
<td>Scheduling steps is P, Chunk size is N/P</td>
</tr>
<tr>
<td>Guided self-scheduling</td>
<td>Uniform memory</td>
<td>Dynamic</td>
<td>Scheduling steps is P*ln(N/P), Chunk size is N/P</td>
</tr>
<tr>
<td>Factoring</td>
<td>Uniform memory</td>
<td>Dynamic</td>
<td>Scheduling steps is P*(N/P), Chunk size is N/P</td>
</tr>
<tr>
<td>Affinity scheduling</td>
<td>NUMA</td>
<td>/</td>
<td>The iterations of a loop are divided into chunks of size N/P iterations and each chunk is statically assigned to a different processor</td>
</tr>
<tr>
<td>Dynamic partitioned</td>
<td>NUMA</td>
<td>/</td>
<td>Similar to AFS, except that it balances the load by readjusting the sizes of the allocated partitions on subsequent executions of a loop</td>
</tr>
<tr>
<td>affinity scheduling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locality-based dynamic</td>
<td>NUMA</td>
<td>/</td>
<td>Adapt to any data partitioning methods, including cyclic or block-cyclic; it computes the chunk size as R*(2/P)</td>
</tr>
<tr>
<td>scheduling algorithm</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Typical DOACROSS algorithms comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Model</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclic</td>
<td>Regular</td>
<td>Exploit the parallelism.</td>
<td>Higher inter-processor communication cost, lower hardware utilization</td>
</tr>
<tr>
<td>PSS</td>
<td>Regular/irregular</td>
<td>Balance the load / have fully parallel inspector and executor phases</td>
<td>Difficult to arrive at an optimal chunk size / require many memory accesses; does not treat input dependencies differently</td>
</tr>
<tr>
<td>ZYRP</td>
<td>Irregular</td>
<td>Can handle all types of dependencies, tightly connected the inspector and the executor, simple algorithm</td>
<td>Un-reused inspector of the same loop; the execution of iterations with dependencies cannot be overlapped even partially</td>
</tr>
<tr>
<td>PRPS</td>
<td>Regular/irregular</td>
<td>Use the inspector to simply detect whether or not the loop is fully parallel</td>
<td>Cannot exploit partially parallel loops</td>
</tr>
<tr>
<td>Chen’s CYT</td>
<td>Regular/irregular</td>
<td>Can reuse the inspector results across loop invocations. It allows the partial overlap of dependent iterations.</td>
<td>Have unnecessary overhead; this algorithm does not treat input dependencies differently</td>
</tr>
</tbody>
</table>

DOACROSS parallelism: Because data dependences and control dependences among the iterations of the loop are widespread in many applications, so DOACROSS parallelism is aim for these applications. DOACROSS parallelization executes adjacent iterations in coarse-grain parallelism and fine-grained reference-level parallelism by the concurrent execution of parts of each loop iterations across multiple cores. At first one thread executes the loop’s critical path on the core. And then execution of the loop’s next iteration begins on the next core while the current iterations are being executed. The advantages of DOACROSS parallelization are reducing overhead and handling more general class of loop. The disadvantage of DOACROSS is lower parallel efficiency (Oplinger et al., 1997).

DOACROSS Loop Scheduling (Thulasiraman et al., 1995). Some typical DOACROSS techniques are compared in Table 2.

DSWP techniques: DSWP techniques can extract threads from general purpose applications, because it uses the fine-grained pipeline parallelism to handle complicated control dependence and memory dependence. DSWP parallelizes a loop by partitioning the body of the loop into a sequence of pipeline stages. Each stage is then executed by a separate thread. The threads communicate either through special hardware structures or through memory. The DSWP algorithm has three main steps. First, the Program Dependence Graph (PDG) which contains all register, memory and control dependences in the loop is constructed. Second, Strongly Connected Components (SCC) are established by founding all dependence recurrences in the PDG. Third, DAGSCC is formed by allocating each SCC to a thread while ensuring that no cyclic dependences are formed between the threads (Vachharajam et al., 2007).

The DSWP techniques were first proposed to effectively explore parallel of complicated loops (Rangan et al., 2004). The DSWP techniques which included a generic algorithm were proposed to automatically apply for the general-purpose applications. The AutoDSWP techniques can identify more loops for the universal applications and ensure that cross-thread iterations does not exist (Rangan, 2007). Because the PS-DSWP techniques integrated the applicability of
Table 3: Typical DSWP algorithms comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compiler</th>
<th>Key insight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rangan DSWP / Ottoni DSWP</td>
<td>IMPACT</td>
<td>It identifies two paths of critical path and off-critical path handle parallelized recursive data structure codes.</td>
</tr>
<tr>
<td>AutoDSWP</td>
<td>VELOCITY</td>
<td>It uses general algorithm to depart from the notion of identifying critical and off-critical paths of the regions. It achieves pipelined parallelism by that the loop critical path dependence need not even once be routed from core to core. It can improve the thread balance to achieving optimal speedup.</td>
</tr>
<tr>
<td>PS-DSWP</td>
<td>VELOCITY</td>
<td>It's algorithm based on Ottoni DSWP. After isolating the recurrences in their own stages in DSWP, it adopts DOALL techniques to explore the portions loop.</td>
</tr>
</tbody>
</table>

DSWP with the scalability of DOALL (Raman, 2009), so PS-DSWP techniques implementation are two steps. First, it adopts DSWP to divided the threads, these threads are referred as DSWPed thread. Second using DOALL techniques parallel executed DSWPed thread (De Lima Ottoni, 2008). The execution of DSWP/PS-DSWP techniques is respectively described in Fig. 3 and 4 some typical DSWP techniques are compared in Table 3.

**SPECULATIVE THREAD PARTITIONING TECHNIQUES BASED ON MULTI-CORE**

The automatic parallelization techniques of DOACROSS, DSWP and PS-DSWP can explore the parallel from sequential application, but much dependence is difficult pre-detection by the non-speculative transformation. So many speculative techniques, such as thread level speculation (TLS), Speculation DSWP (SpecDSWP), Speculative PS-DSWP (SpecPS-DSWP) and Interprocedural SpecPS-DSWP (iSpecPS-DSWP) are proposed to break dependences to increase the parallelism.

All speculative loop parallelization techniques use speculation to break dependencies among threads. To ensure the application correct execution, speculative threads maintain speculative state separately from non-speculative state. This allows them to rollback to a known good state when occur mis-speculation (Li and Zhang, 2012).

Speculative techniques based on DOACROSS- TLS: TLS techniques are extended the basic DOACROSS concept with speculation (Hall et al., 2005) to aim at reducing inter-thread communication. TLS techniques allow cross-thread dependences and adopt synchrony to maintain correct execution dependences among threads.
(Nemanich, 2009; Bridges, 2008). Like DOACROSS techniques, TLS techniques adopt a separate thread to implement each iteration and parallel execute multiple iterations of a loop. Unlike DOACROSS, TLS techniques execute the oldest thread in sequential application order with non-speculation and execute other threads with speculation (Rauchwerger and Padua, 1999; Zhai, 2005). During the program execution, the sequence of instructions can be split into a set of ordered threads to guarantee the threads committing in original order. Using hardware or software to track all related threads, TLS can ensure that sequential programs are correct of parallel execution on multi-core processors. If the conflicts of the parallel execution is found during the program execution, one of the conflict threads is revocation and then implement the compensation operation and re-execute thread (Ohukotun et al., 2009). The inter-thread data dependence, inter-thread control-flow mis-prediction and inter-thread load imbalance are the key factors of runtime overhead in TLS techniques (Wang et al., 2008; Tian, 2010; Venkatesan, 2009). According to architectures, TLS techniques are divided into tightly coupled TLS architectures, chip multiprocessor TLS, multithreaded processor TLS, shared-memory multiprocessor TLS and software-only TLS (Wang, 2006; Packirisamy, 2009). Trace processor, the superthread architecture, Multiscalar, speculative multithreaded (SM) processor and point are typical tightly coupled TLS architectures. Hydra, STAMPede and POSH compiler are typical chip multiprocessor TLS. Dynamic multithreading (DMT) and implicitly multithreaded processors (IMT) are multithreaded processor TLS. Knight and the superthread architecture are typical shared-memory multiprocessor TLS. Typically software-only TLS compilers are Lazy Privatizing Dowl test (LPRD), Behavior-oriented Parallelization (BOF) and Safe Future (Rotenberg et al., 1997; Akkary and Driscoll, 1998; Sohi, 2001; Oancea and Mycroft, 2008; Gao et al., 2010, Zhong et al., 2007; Wele et al., 2005). The execution of TLS techniques is described in Fig. 5 and some typical TLS techniques are compared from focus, software and hardware in Table 4.
Table 4: Comparison of typical TLS techniques

<table>
<thead>
<tr>
<th>Systems</th>
<th>Year</th>
<th>Focus</th>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace processor</td>
<td>1997</td>
<td>Instruction trace</td>
<td>/</td>
<td>Processing units and local registers</td>
</tr>
<tr>
<td>SM</td>
<td>1998</td>
<td>Loops</td>
<td>/</td>
<td>Thread units</td>
</tr>
<tr>
<td>Multiscalar</td>
<td>1995</td>
<td>Tasks</td>
<td>Identify and compile tasks</td>
<td>Address resolution buffer and temporal or sequenced cache buffers</td>
</tr>
<tr>
<td>point</td>
<td>2005</td>
<td>Generic</td>
<td>Compiler infrastructure</td>
<td>Speculative multi-threading processor</td>
</tr>
<tr>
<td>Hydra</td>
<td>1996</td>
<td>Loops and module-level parallelism</td>
<td>Profile</td>
<td>Speculative execution of threads</td>
</tr>
<tr>
<td>STAMPede</td>
<td>2000</td>
<td>Loops</td>
<td>Synchronization</td>
<td>Speculative cache coherence</td>
</tr>
<tr>
<td>POSH</td>
<td>1998</td>
<td>Loops and module-level threads</td>
<td>Profile</td>
<td>Shared memory processor</td>
</tr>
<tr>
<td>DMT</td>
<td>2003</td>
<td>Loop and module continuations</td>
<td>Identify and compile tasks</td>
<td>Multi-threaded hardware</td>
</tr>
<tr>
<td>INT</td>
<td>2005</td>
<td>Generic</td>
<td>Multithreaded hardware with speculative execution of threads</td>
<td></td>
</tr>
<tr>
<td>Super threaded</td>
<td>1998</td>
<td>Loops</td>
<td>No data speculation in threads</td>
<td>Multi-processor with instruction set extensions</td>
</tr>
<tr>
<td>LPRD</td>
<td>1995</td>
<td>Loops</td>
<td>Run-time support mark and test phase</td>
<td>/</td>
</tr>
<tr>
<td>Safe Future</td>
<td>2005</td>
<td>Loops</td>
<td>Run-time, object copy, byte rewrite</td>
<td>/</td>
</tr>
</tbody>
</table>

Fig. 5(a-d): TLS example, (a) Sequential code, (b) Speculated sequential code, (c) Execution of code in figure a, (d) Execution of code in figure b

Speculative techniques based on DSWP: Speculation DSWP (SpecDSWP) (Shi, 2007; Vachhramji, 2008) SpecPS-DSWP and iSpecPS-DSWP are typical techniques which add speculative techniques based on DSWP technique.

SpecDSWP extended on DSWP to focus on removing dependences that form large pipeline stages. As in DSWP, SpecDSWP execute stages to achieve pipelined parallelism; Unlike DSWP, SpecDSWP speculative executes each iteration to allow intra-iteration speculation at frequently silent store. SpecDSWP techniques generally checkpoint memory at the beginning of each iteration and executes iteration on multiple threads or multiple processors. SpecDSWP allows multiple threads to execute inside the same version at the same time by a version memory system. If Mis-speculation occurs, SpecDSWP techniques use two recovery mechanisms. One is that worker threads run a non-speculative multithread version of the loop. The other is that commit thread run the original single thread loop. The main
The advantage of SpecDSWP is its latency tolerant property realized by decoupling inter-process communication delay from the critical path to off the critical path.

The SpecPS-DSWP's performance is limited by the static analyses. So that, SpecPS-DSWP techniques add speculation to the PS-DSWP algorithms to extract large amount of parallelism. SpecPS-DSWP can use speculation to remove inter-iteration dependences to prevent the extraction of data-level parallelism. Reifying the speculation and then applying the PS-DSWP algorithm to the resulting code assigns the SpecPS-DSWP thread. After generating PS-DSWP multi-threaded code, checkpoint and recovery code are inserted into each thread and the Commit Thread. SpecPS-DSWP relies upon a version memory system with hierarchical memory versions. The SpecPS-DSWP technique is a novel parallelization framework to optimize larger loops than existing parallelization techniques.

The iSpecPS-DSWP technique is formed that add inter-procedural analysis and optimization scope to the existing PMT techniques, including DSWP, PS-DSWP, Spec-DWSP and SpecPS-DWSP (Bridges, 2008). iSpecPS-DSWP builds interprocedural PDG(iPDG) which is derivative of the system dependence graph(SDG). iPDG contains all local dependence and any reachable procedures. Deciding the loop-carriedness of each edge in the iPDG will occur register dependence, parameter dependence, control dependence, call dependence and memory dependence. If register dependence, control dependence and memory dependence are in the DSWPed loop, these dependences are treated as SpecPS-DSWP.

Otherwise, they are marked as intra-iteration (INTRA). Parameter dependence and call dependence are always marked as INTRA because they can not involve the loop back edge. Parameter dependence is handled similar to register dependences. Call dependences are treated in the same as the control dependences. The execution of SpecDSWP/SpecPS-DSWP techniques is respectively described in Fig. 6 and 7.

SpecDSWP, SpecPS-DSWP and iSpecPS-DSWP are typical speculative DSWP techniques. They are compared from based techniques, recovery mechanism, support speculative types and complement steps. In addition to control speculation, SpecDSWP, SpecPS-DSWP and iSpecPS-DSWP support other speculative types, such as biased branch speculation, infrequent basic block speculation, memory value speculation and silent store speculation. Biased branches speculation can break the control dependences between the branch and other instructions and break data dependences by altering the control flow of the program. The compiler inserts a new basic block on each control flow edge from the branch to each speculated target to realize biased branches speculation. Infrequent block speculation can be mapped to a set of biased branch speculation to break all incoming control flow edges in blocks. To enable silent store speculation, the compiler inserts a load of the same memory and a branch to compare the loaded value with the one about to be stored. The memory dependences are removed by silent store speculation when silent stores write to a location but they do not change the location's value. To realize memory value speculation, the compiler...
Fig. 7(a-d): SpecPS-DSWP example, (a) Code, (b) PDG, (c) DAGsec and (d) Execution of code in figure a with DSWP.

Fig. 8(a-c): TLS and Spec-DSWP schedule, (a) Code, (b) Execution of code in figure a with SpecDSWP and (c) Execution of code in figure a with TLS.

comparses two value and use the same way of biased branch speculation to treat the differ value. The loop-carried invariant speculation, parameter dependence speculation and call dependence speculation are processed by the same methodology as memory value speculation. Some typical Speculative techniques based on DSWP are compared in Table 5.

Although either TLS or SpecDSWP technique could extract parallelism from loops, the TLS and SpecDSWP technique has their advantages. For TLS, it more suitable for general program and its speed not limited by the size of largest pipeline stage. For SpecDSWP, it can better handle synchronizing dependences among iterations and has not communication latency that caused by the critical path for the synchronized portion (Bridges, 2008). The difference of TLS and SpecDSWP techniques is described in Table 6 and using Fig. 8 describe the execution of the both techniques.
Table 5: Comparison speculative DSWP algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Based Techniques</th>
<th>Recovery mechanism</th>
<th>Support speculative types</th>
<th>Complement Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Infrequent basic block speculation</td>
<td>2. Select speculation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. Silent store speculation</td>
<td>3. Build speculative PDG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4. Memory value speculation</td>
<td>4. Form SCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5. Detect misprediction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6. Determine need speculation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Infrequent block speculation</td>
<td>2. Select speculation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. Silent store speculation</td>
<td>3. Build speculative PDG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4. Memory value speculation</td>
<td>4. Form and assign SCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5. Determine need specification</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6. Rebuild speculative PDG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7. Form and assign SCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8. Determine memory versions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9. Determine synchronization</td>
</tr>
<tr>
<td>iSpecPS-DSWP</td>
<td>DSWP</td>
<td>Hierarchical memory versions</td>
<td>1. Biased branch speculation</td>
<td>1. Build PDG</td>
</tr>
<tr>
<td></td>
<td>PS-DSWP</td>
<td></td>
<td>2. Infrequent block speculation</td>
<td>2. Select speculation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4. Memory value speculation</td>
<td>4. Form and assign SCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5. Determine need speculation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6. Rebuild speculative PDG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7. Form and assign SCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8. Determine memory versions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9. Determine synchronization</td>
</tr>
</tbody>
</table>

Table 6: Comparison TLS and SpecDSWP algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>TLS</th>
<th>SpecDSWP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus on iteration</td>
<td>Remove inter-iteration</td>
<td>Reduce inter-thread communication</td>
</tr>
<tr>
<td>Limited in speed</td>
<td>Stay local on a single thread processor</td>
<td>Stay on multiple processors</td>
</tr>
<tr>
<td>Submitted order</td>
<td>The size of the longest cross-Iteration dependence cycle communication latency between processor cores</td>
<td>The size of largest pipeline stage</td>
</tr>
<tr>
<td>Implement model</td>
<td>Older iteration's commit before younger iterations when conflict occurs, younger iteration roll back</td>
<td>When conflict occurs, each thread is restored to its recovery code Restore threads state and re-executing iteration non-speculatively</td>
</tr>
<tr>
<td></td>
<td>Execute each iteration in a different thread, forming pipeline only in a single direction All iterations execute speculatively</td>
<td></td>
</tr>
</tbody>
</table>

THRESHOLD SPECULATION TECHNIQUES ON TRANSACTIONAL MEMORY

Transactional Memory (TM): Transactional Memory (TM) systems adopt implicit locks to exploit more parallelism and check the read/write conflict before committing a transaction. When TM systems occur the conflicts, TM systems aborts the transaction and rolls back and then processes the transaction again until they have no conflict. TM systems can be classified into hardware TM (HTM), software TM (STM) and hybrid TM (HyTM) according to the different implementations (Nasir, 2009). HTM system implement conflict detection and consistency among concurrent transactions by hardware cache. The advantages of HTM are the higher speed, easier programming and lower overhead. The disadvantage of HTM is that it must be redesigned the caches and the coherence protocol (Gruhn, 2010). Some HTM designs were proposed, such as, transactional coherence and consistency (TCC) (Hammond et al., 2004), Bulk (Ceze et al., 2006), unbounded TM (UTM), large TM (LTM) (Aranian et al., 2005) and token TM (Bobba, 2010) systems. STM systems can be designed to work on existing systems without adding extra special hardware. STM systems adopt software to deal conflict detection, implement version management, guarantee the synchronization and etc. The advantages of STM are easier modification and extending. But STM execution has higher overhead and lower performance than HTM. Some STM designs were proposed, such as, Rochester software transactional memory (RSTM) (Marathe et al., 2006), word based software transactional memory (WSTM) (Harris and Fraser, 2003), Intel's McRTM (Guerraoui et al., 2007), Dynamic Software Transactional Memory (DSTM) (Herlihy et al., 2003) and transactional locking II (TLM) (Dice et al., 2006). HyTM aims at using the advantages of both HTM and STM. HyTM adopt software to execute some larger transaction while HyTM adopt hardware to execute part functions of systems and some small transaction. Signature-accelerated transactional memory (SigTM) (Minh et al., 2007), Virtualized TM (VTM) (Rajwar et al., 2005), log-based
transactional memory (LogTM) (Damron et al., 2006),
logTM signature edition (LogTM-SE) and non-blocking
zero-indirection transactional memory (NZTM)
(Al Tabba, 2011) are some typical HyTM (Li et al., 2010).

Extended TM to support TLS: There are many differences
semantic between TLS and TM but both of them require
similar underlying support, such as dependency violation
detection, result buffering, check pointing and replay. TM
systems lack thread spawning mechanism, context
passing mechanism and sequential ordering mechanism
compared to TLS. Therefore, combined the TLS and TM
can more effectively improve the serial program
running on the multi-core processors (Zhong et al., 2007;
Zhang et al., 2012). Typical systems which support
to TLS and TM techniques are TCC, STMlite, Helper
Transaction and PTT. They all adopt compiler to support
the context passing mechanism. TCC adopts hardware
to support sequential ordering commit and adopts manual
and hardware to implement thread spawning. Helper
Transactions (Yoo and Lee, 2008) is an in-depth design
that can effectively exploit out-of-order procedure
speculation. Helper Transactions uses binary tree to
describe the sequential ordering mechanism, while it uses
compiler to spawn thread and context passing mechanism.
STMlite (Mehra et al., 2009 support TLS technology
based on light-weight STM model. STMlite adopts hash
read and write sets to spawn thread and employs transaction
commit manager to guarantee sequential
ordering commit. Profile of a guided TLS and TM system
(PTT) system (Wang, 2010) which expands the token in
LogTM adopts determination priority, modification cache
coherence protocol and token mechanism to support
sequential committing. PTT modifies cache coherence
protocol and hardware to spawn thread (Li et al., 2012).

CONCLUSION

In this study, we introduced three parallel types of
thread level parallelism, such as IMT, CMT, PMT and
described their execution model. We compared typical
DOALL, DOACROSS, DSWP and PS-DSWP techniques.
These techniques can explore the parallel from sequential
application but much dependence is not easily predictable
or manifests them infrequently by the non-speculative
transformation. So many speculative techniques, such as
TLS, SpecDSWP, SpecPS-DSWP and iSpecPS-DSWP, are
proposed to further improve parallelism. We introduced
these speculative parallel techniques and described their
execution model. SpecDSWP, SpecPS-DSWP and
iSpecPS-DSWP are compared from supporting
speculation types, memory version and implement steps.

We compared TLS and SpecDSWP techniques from
iteration, limited in speed, submitted order and implement
model. While TLS more suitable for general program,
SpecDSWP can better handle synchronizing dependences
among iterations. Some opportunities for future research
in parallel sequential program are novel thread partitioning
algorithms, inter-procedural extensions, speculative
optimization and etc. At last, some extended TM to
support TLS systems are analyzed from thread spawning
mechanism, context passing mechanism and sequential
ordering.

ACKNOWLEDGMENTS

This work is supported by the 2007 Nationality 863
Project “Software Product Line based on Workflow”
(No. 2007AA010305), the 2013 Shaanxi Provincial
Department of Education project “Research of technology
and system of thread level speculation based on
transactional memory”(2013JK1197), the State Key
Laboratory for Manufacturing Systems Engineering
Open Project (No. sklms201011). The authors also
gratefully acknowledge the helpful comments and
suggestions of the reviewers, who have improved the
presentation.

REFERENCES

multithreading processor. Proceedings of the 31st
Annual ACM/IEEE International Symposium on
Microarchitecture, November 30-December 2, 1998,
Dallas, TX., USA., pp. 226-236.
Al Tabba, F.A.W., 2011. A study of hybrid transactional
memory. Doctoral Dissertation, The University of
Auckland, Auckland, New Zealand.
Ananian, C.S., K. Asanovic, B.C. Kuszmaul,
C.E. Leiserson and S. Lie, 2005. Unbounded
transactional memory. Proceedings of the 11th
International Symposium on High-Performance
Computer Architecture, February 12-16, 2005, San
Francisco, CA., USA., pp. 316-327.
Bobba, J., 2010. Hardware support for efficient
transactional and supervised memory systems. Ph.D.
Thesis, University of Wisconsin-Madison, USA.
Bridges, M.J., 2008. The velocity compiler: Extracting
efficient multicore execution from legacy sequential
codes. Ph.D. Thesis, Princeton University, USA.
disambiguation of speculative threads in
News, 34: 227-238.


