Universidade Estadual de Campinas
Instituto de Computação

Juan Jesús Salamanca Guillén

Thread-Level Speculation on Hardware Transactional Memory Architectures

Especulação de Threads usando Arquiteturas de Memória Transacional em Hardware

CAMPINAS
2016

Juan Jesús Salamanca Guillén

Thread-Level Speculation on Hardware Transactional Memory Architectures

Especulação de Threads usando Arquiteturas de Memória Transacional em Hardware

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo
Co-supervisor/Coorientador: Prof. Dr. José Nelson Amaral

Este exemplar corresponde à versão final da Tese defendida por Juan Jesús Salamanca Guillén e orientada pelo Prof. Dr. Guido Costa Souza de Araújo.

CAMPINAS
2016

Agência(s) de fomento e nº(s) de processo(s): FAPESP, 2015/04285-5; CNPq

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Salamanca Guillén, Juan Jesús, 1988-
Sa31t Thread-level speculation on hardware transactional memory architectures / Juan Jesús Salamanca Guillén. – Campinas, SP : [s.n.], 2016.

Orientador: Guido Costa Souza de Araújo.
Coorientador: José Nelson Amaral.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Especulação no nível de threads. 2. Memória transacional. 3. Processamento paralelo (Computadores). 4. Threads (Programa de computador). I. Araújo, Guido Costa Souza de,1962-. II. Amaral, José Nelson,1964-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Especulação de threads usando arquiteturas de memória transacional em hardware
Palavras-chave em inglês:
Thread-level speculation
Transactional memory
Parallel processing (Electronic computers)
Threads (Computer programs)
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Alexandro José Baldassin
Márcio Bastos Castro
Sandro Rigo
Emilio de Camargo Francesquini
Data de defesa: 21-12-2016
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Juan Jesús Salamanca Guillén

Thread-Level Speculation on Hardware Transactional Memory Architectures

Especulação de Threads usando Arquiteturas de Memória Transacional em Hardware

Campinas, 21 de dezembro de 2016

Dedication

This work is dedicated to my dear mother, Flavia Guillén.

Acknowledgements

First and foremost, I would like to thank God and my beloved son Juan Pablo Matías for their love. I would also like to thank my mother, Flavia, for encouraging me to pursue my dreams and for her tremendous support in each step of my life. I would like to thank my brother Jesús and my sisters Ana Flavia and Jennifer for their friendship, affection, and confidence. I would like to thank all my family, especially my father Jesús, my grandmother Elsa, and my aunt Yemile, who were always supporting me and encouraging me with their best wishes.

I will always be grateful to my supervisor, Guido Araújo, for believing in me and for all the incentive, support, and patience throughout our project. You became more than a supervisor, you became a friend and a second father. Through a number of difficult times, you were very supportive and understanding. Thank you, Guido. I thank my co-supervisor Nelson Amaral for his support, for always helping me think outside the box, and for welcoming me in Canada. Nelson, meeting you and working with you proved to be an invaluable experience. I am very grateful to both for their guidance to learn about a research career.

I thank all my dear friends, especially Renzo, Luis, Edmanuel, Pepe, José Manuel, Lucas, Laura, Ticiana, and Martin, for being always there. Also, I want to thank all my friends of the Institute of Computing at UNICAMP as well as of the Department of Computing Science at the University of Alberta, whose friendship have made both incredible places to study, to learn, and to have a good time.

Finally, I would like to thank CNPq, Samsung, CAPES, LGE, and FAPESP (grants 15/04285-5 and 15/12077-3) for their financial support to this work.

Resumo

Especulação no nível de threads (TLS) é uma técnica em hardware/software que possibilita a execução paralela de múltiplas iterações de um laço, inclusive na presença de algumas dependências loop-carried. TLS exige mecanismos em hardware para auxiliar a detecção de conflitos, o armazenamento especulativo, os commits das transações em ordem, e o roll-back das transações. Trabalhos anteriores exploraram enfoques para implementar TLS, tanto em hardware dedicado como puramente em software, e tentaram predizer o desempenho de futuras implementações de TLS em hardware. Contudo, não existe nenhum processador comercial que forneça suporte direto para TLS. Entretanto, execução especulativa é suportada na forma de Memória Transacional em Hardware (HTM) — disponível em processadores modernos como Intel Core e IBM POWER8. HTM implementa três características essenciais para TLS: detecção de conflitos, armazenamento especulativo, e roll-back de transações.

Antes de aplicar TLS a um laço quente, é necessário determinar se o laço tem potencial para ser especulado. Um laço pode ser adequado para TLS se a probabilidade de dependências loop-carried em tempo de execução for baixa; para estimar esta probabilidade um perfilamento de dependências do laço deve ser usado. Este trabalho apresenta um verificador das dependências loop-carried integrado como uma nova extensão de OpenMP, a diretiva `parallel for check`, a qual pode ser usada para ajudar desenvolvedores a identificarem a existência destas dependências em construções `parallel for`.

Este trabalho também apresenta uma análise detalhada da aplicação de HTM para a paralelização de laços com TLS e descreve uma avaliação cuidadosa da implementação de TLS usando HTMs disponíveis em processadores modernos. Como resultado, esta tese proporciona evidências para validar várias afirmações importantes sobre o desempenho de TLS nestas arquiteturas. Os resultados experimentais mostram que TLS usando HTM produz speedups de até 3.8$\times$ para alguns laços.

Finalmente, este trabalho descreve uma nova técnica de especulação para a otimização, e execução simultânea, de múltiplos traços de regiões de código quente. Esta técnica, chamada Speculative Trace Optimization (STO), enumera, otimiza, e executa especulativamente traços de laços quentes. Isto requer o suporte em hardware disponível em sistemas HTM. Este trabalho discute as características necessárias para suportar STO: multi-versão, resolução de conflitos tardia, detecção de conflitos prematura, e sincronização das transações. Uma revisão das arquiteturas HTM existentes — Intel TSX, IBM BG/Q, e IBM POWER8 — mostra que nenhuma delas tem todas as características requeridas para implementar STO. Entretanto, este trabalho mostra que STO pode ser implementado nas arquiteturas HTM existentes através da adição de privatização e código para esperar/retomar.

Abstract

Thread-Level Speculation (TLS) is a hardware/software technique that enables the execution of multiple loop iterations in parallel, even in the presence of some loop-carried dependences. TLS requires hardware mechanisms to support conflict detection, speculative storage, in-order commit of transactions, and transaction roll-back. Prior research has investigated approaches to implement TLS, either on dedicated hardware or purely in software, and has attempted to predict the performance of future TLS hardware implementations. Nevertheless, there is no off-the-shelf processor that provides direct support for TLS. Speculative execution is supported, however, in the form of Hardware Transactional Memory (HTM) — available in recent processors such as the Intel Core and the IBM POWER8. HTM implements three key features required by TLS: conflict detection, speculative storage, and transaction roll-back.

Before applying TLS to a hot loop, it is necessary to determine if the loop has potential to be amenable. A loop could be amenable if the probability of loop-carried dependences at runtime is low; to measure this probability loop dependence profiling is used. This project presents a novel dynamic loop-carried dependence checker integrated as a new extension to OpenMP, the `parallel for check` construct, which can be used to help programmers identify the existence of loop-carried dependences in `parallel for` constructs.

This work also presents a detailed analysis of the application of HTM support for loop parallelization with TLS and describes a careful evaluation of the implementation of TLS on the HTM extensions available in such machines. As a result, it provides evidence to support several important claims about the performance of TLS over HTM in the Intel Core and the IBM POWER8 architectures. Experimental results reveal that by implementing TLS on top of HTM, speed-ups of up to 3.8× can be obtained for some loops.

Finally, this work describes a novel speculation technique for the optimization, and simultaneous execution, of multiple alternative traces of hot code regions. This technique, called Speculative Trace Optimization (STO), enumerates, optimizes, and speculatively executes traces of hot loops. It requires hardware support that can be provided in a similar fashion as that available in HTM systems. This work discusses the necessary features to support STO, namely multi-versioning, lazy conflict resolution, eager conflict detection, and transaction synchronization. A review of existing HTM architectures — Intel TSX, IBM BG/Q, and IBM POWER8 — shows that none of them has all the features required to implement STO. However, this work demonstrates that STO can be implemented on top of existing HTM architectures through the addition of privatization and wait/resume code.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Loops account for most of the execution time in programs and thus extensive research has been dedicated to parallelize loop iterations [1, 26, 45]. Unfortunately, in many cases these efforts are hindered when the compiler cannot prove that a loop is free of *loop-carried* dependences. However, sometimes when static analysis concludes that a loop has a *may* dependence — for example when the analysis cannot resolve a potential alias relation — the dependence may actually not exist or it may occur in very few executions of the program [5, 62]. *Thread-Level Speculation* (TLS) is a promising technique that can be used to enable the parallel execution of loop iterations in the presence of *may* loop-carried dependences.

Recently hardware support for speculation has been implemented in commodity off-the-shelf microprocessors [28, 29]. However, the speculation support in these architectures was designed with *Hardware Transactional Memory* (HTM) in mind and not TLS. The only implementation of hardware support for TLS to date is in the IBM Blue Gene/Q (BG/Q), but BG/Q is not a commodity machine and thus not readily available for experimentation or usage. HTM extensions, available in the Intel Core and in the IBM POWER8 architectures, allow for the speculative execution of atomic program regions [29, 67, 28, 36]. Such HTM extensions enable the implementation of three key features required by TLS: (a) conflict detection; (b) speculative storage; and (c) transaction roll-back.

Similar to HTM, TLS employs an optimistic approach to parallelism. TLS assumes that the iterations of a loop can be executed in parallel — even in the presence of potential dependences — and then relies on a mechanism to detect dependence violations and correct them. The main distinction between TLS and HTM is that in TLS speculative transactions must commit in order, a required feature when parallelizing the execution of a loop so that loop-carried dependences from one iteration transaction to another are respected. However, among all hardware implementations that support speculation, only the IBM BG/Q supports in-order transaction commit, as it was initially designed to enable TLS [23].

Until now, the majority of the attempts to estimate the performance benefits of TLS were based on simulation studies [54, 56, 55, 47]. Unfortunately, studies of TLS execution based on simulation have serious limitations. Some interesting research questions are: (1) can the existing speculation support in commodity processors, originally designed for HTM, be used to support TLS and reduce its overhead to execute loop code? and (2) if it can, what performance effects would be observed from such implementations? This thesis has a cautiously positive answer to the first question, i.e. supporting TLS on top of HTM hardware is possible. To address the second question, this work presents an in-depth evaluation of the implementation of TLS on top of the HTM extensions available in the Intel Core and in the IBM POWER8 that leads to new techniques to support TLS over HTM and to some surprising discoveries about the interaction between prefetching, false sharing, and the relevance of loop characterization to predict the potential performance of TLS. The experimental results indicate that: (1) false sharing is a very important performance-hindering effect in both architectures; (2) strip mining is an effective transformation to eliminate false sharing; (3) the selected size of the strip can be critical; (4) in some cases the strip size needed to eliminate false sharing may lead to aborts because the speculative capacity of the HTM is exceeded; (5) small loops are not amenable to be parallelized with TLS on the existing HTM hardware because of the expensive overhead of: (a) starting and finishing transactions, (b) aborting a transaction, and (c) setting up loop for TLS execution; (6) loops with potential to be successfully parallelized in both Intel Core and IBM POWER8 architectures have better performance on the POWER8 because TLS can take advantage of the ability of this architecture to suspend and resume transactions to implement *ordered transactions*; (7) the larger storage capacity for speculative state in Intel TSX can be crucial for loops that execute many read and write operations; and (8) the ability to suspend/resume a transaction is important for loops that execute for a longer time because their transactions may abort due to OS context switching.

This work also proposes Speculative Trace Optimization (STO) to speculatively optimize and execute multiple alternative *traces* of a single iteration of a hot loop. The goal is to simultaneously execute speculative traces in hot loops to uncover hidden optimizations that could not be carried out at compile time because of program-flow indeterminism. STO is not a loop parallelization technique, rather it is a technique that speeds up both sequential and parallelizable loops. The discussion in this work focuses on the exhaustive execution of inner-loop traces, but STO can be used in other code regions and it can also be used to selectively execute a subset of speculative traces. Contrary to whole-procedure traces, the number of inner-loop traces is reasonably small, making them good candidates for speculation in current HTM architectures that have limited capacity to store speculative state [29, 30, 28, 23]. In an initial exploration that applied STO to the hot inner-loops from a set of programs, we found that at most four traces were present.

The use of STO described in this work enumerates all possible traces, optimizes them, and executes each trace speculatively in a *transaction*, using a fork/join paradigm. All conditionals that select a specific trace are evaluated at the end of each transaction to determine if the trace should commit or abort. For each loop iteration, a single trace commits while the others miss-speculate and thus should be aborted.The initial assessment of STO presented in this work uses the prototype based on TSX and applies it to benchmarks from Mediabench, Parboil and SPEC2006 benchmarks. The results reveal speed-ups of up to 9% for four cores. This initial result is encouraging given that TSX lacks multi-versioning and lazy-conflict resolution, and it has a significant abort overhead [50]. To compensate for the missing features, extra code is inserted into the original program leading to additional overhead. Achieving speed-ups even in the presence of such overheads suggests that if an HTM architecture were to incorporate such features, significant speed-ups could emerge.

On the other hand, tools to support programming correctness are central in any programming model, particularly in parallel programming, in which bugs are typically very hard to detect and reproduce [66]. A fairly common source of bugs in OpenMP and many other parallel programming models shows up when programmers need to evaluate if loops can have their iterations parallelized. In order to do so, programmers have to perform a careful and complex evaluation of the dependence of the loop-body variables across iterations. If such dependences are not present, loops are called DOALL, and its iterations can be easily parallelized. Otherwise, they are called DOACROSS loops, which are harder to parallelize and to extract good speed-ups [66]. Given that loop-bodies can have complex nested function calls, and pointer aliasing, dynamic cross-iteration dependences can occur at runtime, making the work of the programmer much harder and error prone. Complex loop-bodies can easily produce intricate runtime dependences which cannot be easily detected by the typical programmer at compile time. For this reason, effectively detecting dynamic loop cross-iteration violations is a relevant tool to support parallel programming.

In this work, we also present *parallel for check* (*check*), a new construct to OpenMP, which enables the seamless integration of loop dynamic data dependence verification in OpenMP. This construct makes possible the detection of loop-carried dependences at runtime in OpenMP programs, thus helping programmers to identify potential violations resulting from hard to detect loop-carried dependences. *check* was implemented in Pin/GCC-OpenMP and LLVM/Clang-OpenMP.

This thesis makes four main contributions. First, it shows that false sharing is an important cause of performance loss in TLS on commercial HTMs and it improves the implementation of TLS using HTM through code transformations. Second, it proposes a classification of loops based on TLS performance and doing so provides guidance to developers as to what loop characteristics make them amenable to the use of TLS on the Intel Core or on the IBM POWER8 architectures. Third, it presents a novel technique to optimize and speculate exhaustive traces, called STO, that uses HTM (specifically TSX in the prototype) to execute these traces in transactions using a fork/join paradigm. STO does not parallelize loops, rather it accelerates the sequential execution of loops; and it identifies the main features that an HTM mechanism should have to enable STO. Fourth, it presents a novel OpenMP *parallel for check* construct, also named *check* or *checker*, which enables the dynamic detection of loop-carried dependences. It does on-the-fly dynamic loop-carried dependence analysis of multithreaded applications, making it possible to measure the probability of loop-carried dependences (*%lc*) and to detect patterns of loop-carried dependences which can not be detected by means of serial or per-thread analysis, as in [69, 33, 32, 31].

The remainder of this thesis is organized as follows. Chapter 2 describes the relevant aspects of our work. Chapter 3 details the related work. Chapter 4 explains the limitations of HTM to support TLS and discusses performance limitations caused by false sharing, capacity limitations and non-consecutive array accesses. Chapter 5 describes an in-depth evaluation of TLS on HTM. Chapter 6 presents Speculative Trace Optimization (STO) and describes a prototype implemented on HTM. Chapter 7 motivates and describes the implementation of *checker*. Finally, Chapter 8 concludes the work.

Some of the material used in this thesis has been published or submitted for publication in the following papers:

- Juan Salamanca, José Nelson Amaral, and Guido Araújo. Using Hardware-Transactional-Memory Support to Implement Thread-Level Speculation. *Paper submitted to IEEE Transactions on Parallel and Distributed Systems (TPDS)*

- Juan Salamanca, José Nelson Amaral, and Guido Araújo. Performance Evaluation of Thread-Level Speculation in Off-the-Shelf Hardware Transactional Memories. *Paper submitted to International European Conference on Parallel and Distributed Computing (EURO-PAR) 2017*

- Juan Salamanca, José Nelson Amaral, and Guido Araújo. Evaluating and Improving Thread-Level Speculation in Hardware Transactional Memories. *In IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2016*, May 23-27, 2016, Chicago, IL, USA

- Juan Salamanca, José Nelson Amaral, and Guido Araújo. Using Hardware Transactional Memory to Enable Speculative Trace Optimization. *In International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW) 2015*, October 18-21, 2015, Florianopolis, Brazil

- Juan Salamanca, Luis Mattos, and Guido Araújo. Loop-Carried Dependence Verification in OpenMP. *In International Workshop on OpenMP (IWOMP) 2014*, September 28-30, 2014, Salvador, Brazil

Chapter 2

Background

This chapter describes the background to introduce our work. It describes the main concepts used in Transactional Memory, Thread-Level Speculation, and Speculative Trace Optimization.

2.1 Transactional Memory

Transactional memory (TM) was proposed as architectural support to make lock-free synchronization as efficient as conventional parallelization approaches based on mutual exclusion [25]. TM simplifies parallel programming by enabling a mechanism to ensure the consistency of shared data. Transactional memory systems must provide transaction *atomicity* and *isolation*, which require the implementation of the following mechanisms: *data versioning management*, *conflict detection*, and *conflict resolution* [37, 59].

In Transactional Memory, version management decides where new (speculative) and old data are stored. Conflict detection determines whether two operations executed in separate transactions cause a conflict, *i.e.* if they access a common memory location and at least one of the operations is a write. Conflict detection can be eager (detection is done immediately when the conflict occurs) or lazy (detection is done when transactions attempt to commit) [37]. A conflict causes at least one of the transactions involved in the conflict to abort and it may re-execute. Other actions could also be carried out to support a conflict-resolution policy. Resolution can happen eagerly when the conflict occurs or lazily when the transaction attempts to commit.

TMs can be supported in hardware (HTM) [25] and software (STM) [52]. HTM systems have lower overheads because conflict detection is done in hardware but they have lower speculative-state storage capacity and may support fewer active transactions [41]. HTMs are also easier to use because programmers only need to specify the start and the end of a transaction [65]. STM systems can have a large overhead because conflict detection is performed in software. On the other hand STMs have the advantage that they can be executed on any available hardware, and in principle have no limit on the amount of speculative state that a transaction may use.

Hybrid Transactional Memory (HyTM) is a approach to implement TM in software so that it can use best-effort HTM to boost performance but it does not depend on

Table 2.1: HTM implementations of Intel Core and IBM POWER [41].

Processor type	Intel Core i7-4770	IBM POWER8
Conflict-detection granularity (cache line)	64 B	128 B
Tx Load Capacity	4 MB	8 KB
Tx Store Capacity	22 KB	8 KB
L1 Data Cache	32 KB, 8-way	64 KB
L2 Data Cache	256 KB	512 KB, 8-way
SMT level	2	8

Table 2.2: HTM Architectural Features.

Features	TLS	Intel Core	P8
Eager Conflict Detection	✓	✓	✓
Speculative Storage	✓	✓	✓
Ordered Transactions	✓		
Rollback Transactions	✓	✓	✓
Multi-versioned caches	✓		
Resolution Conflict Policy	✓		
Suspend/Resume			✓
Lazy Conflict Detection			
Data Forwarding	✓		
Word Conflict Detection	✓		

HTM. This approach exploits HTM if it is available to achieve hardware performance for transactions that do not exceed the HTM's limitations [17].

2.2 Intel Core and IBM POWER8

This section reviews HTM extensions, such as those found in Intel Core and IBM POWER8, and and the features to enable TLS.

Intel's *Transactional Synchronization Extensions* (TSX) provides an instruction-set interface to specify transactional execution [29] with two software interfaces: *Hardware Lock Elision* (HLE) and *Restricted Transactional Memory* (RTM). The RTM is an instruction-set extension that includes the instructions `xbegin`, `xend`, and `xabort`. When a transaction aborts, the state of the program immediately before the `xbegin` instruction is recovered, all speculatively written data are dismissed, and the values stored in registers are rolled back to their values prior to the transaction. The execution restarts at a program point specified by the address given as argument to the `xbegin` instruction. Data written transactionally are not visible to other transactions until the transaction commits by executing the `xend` instruction.

POWER8 provides the first implementation of HTM that is supported directly by the POWER ISA. The main difference from POWER8 with respect to Intel TSX is its ability of pausing transactions. In POWER8, through the use of suspend regions, transactions can survive interrupts and can access memory non-transactionally while the transaction is still active. Suspended regions were designed to support debugging. While in a suspended state the thread can load memory locations accessed within the transaction and store their value into memory locations that are not included in the transaction footprint.

POWER8's *Rollback-Only Transactions* (ROTs) allow store buffering without the detection of data conflicts. ROTs support single-thread speculative optimization techniques such as Trace Scheduling [36, 41].

Both Intel and IBM architectures provide instructions to begin and end a transaction, and to force a transaction to abort. To perform such operations Intel Core's *Transactional Synchronization Extensions* (TSX) implements RTM that includes `xbegin`, `xend`, and `xabort`. The corresponding instructions in the POWER8 are `tbegin`, `tend`, and `tabort`.

All data conflicts are detected at the granularity of the cache line size because both processors use cache mechanisms — based on physical addresses — and the cache coherence protocol to track transactional states. Aborts may be caused by: memory access conflicts, capacity issues due to excessively large transactional read/write sets or overflow, conflicts due to false sharing, and OS and micro-architecture events that cause aborts (*e.g.* system calls, interrupts or traps) [41, 67].

In both architectures, when a transaction aborts, the execution of the thread is rolled back to the point immediately before the transaction's begin instruction. An abort handler then determines if the transaction should retry or if a fall-back code should be executed. Data written transactionally are not visible to other transactions until the transaction commits by executing the end instruction. Table 2.1 summarizes the features of both architectures.

The main differences between POWER8 and the Intel Core HTMs are: (1) transaction capacity; (2) conflict granularity; and (3) ability to suspend/resume a transaction. The maximum amount of data that can be accessed by a transaction in the Intel Core is much larger than in the POWER8. This speculative storage capacity is limited by the resources needed both to store read and write sets, and to buffer transactional stores.

In POWER8 the execution of a transaction can be paused through the use of suspended regions — implemented with two new instructions: `tsuspend` and `tresume`. Using this mechanism, a transaction can survive interrupts and the thread can access memory non-transactionally while the transaction is suspended. The `tsuspend` instruction causes the thread to enter a suspended state where all memory accesses are non-transactional but are monitored. If any such access conflicts with the suspended-transaction working set, that transaction will abort due to a conflict after resuming (`tresume`). While in a suspended state the thread can load memory locations accessed within the transaction and store their values into memory locations that are not included in the transaction footprint. Thus, suspended regions also allow transactions to communicate with global flags without failing due to conflicts. This mechanism enables the implementation of an *ordered-transaction* feature in TLS [36].

2.3 Thread-Level Speculation

Torrellas defines Thread-Level Speculation (TLS) as an environment where execution threads operate speculatively, performing potentially unsafe operations, and temporarily buffering the state that they generate in a buffer [60]. Then, the operations of a thread are declared to be correct or incorrect. If they are correct, the thread commits; if they are

```
1   for (i = 0; i < N; i++){
2     /* Start sequential segment 1 */ /* Global scalar, glob */
3     if (cond)
4       glob++;
5     else
6       glob=i;
7     /* End sequential segment 1 */
8     A[i]= glob*i;
9     /* Start sequential segment 2 */
10    for(j = 0; j < factor; j++){
11      /* Global array, B */
12      int tmp = B[factor*(i%4) + j];
13      tmp += i*5;
14      if(tmp%2 == 0){
15        B[factor*(i%4) + j] = tmp;
16      }
17    }
18    /* End sequential segment 2 */
19  }
```

Figure 2.1: A loop with two *may* loop-carried dependences. Adapted from [40].

incorrect, the thread is rolled back and typically restarted from its beginning. The term TLS is most often associated to a scenario where the goal is to parallelize a sequential application. However, in general, TLS can be applied to any environment where speculative threads are executed and can be squashed and restarted [60].

TLS has been widely studied [54, 56, 55]. Proposed TLS hardware systems must support four primary features: (a) data conflict detection; (b) speculative storage; (c) ordered transactions; and (d) rollback when a conflict is detected. Some of these features are also supported by the HTM systems found in the Intel Core and the POWER8, and thus these architectures have the potential to be used to implement TLS. Table 2.2 shows the necessary and advanced features required to enable TLS on top of an HTM-supporting mechanism, and its availability in some modern architectures. Neither Intel TSX nor the IBM POWER8 provide all the hardware features necessary to carry out TLS effectively.

Lets examine how TLS can be applied to a simplified version of the loop example of Figure 2.1 (statement in line 8 and the inner loop are omitted) when it runs on top of an ideal HTM system containing: (a) ordered transactions in hardware; (b) multi-versioning cache; (c) eager-conflict detection; and (d) conflict-resolution policy. Figure 2.2 shows the loop after it was strip-mined and parallelized for TLS using `NUM_THREADS` threads. Assume that the `END` instruction implements: (a) ordered transactions, i.e., a transaction executing an iteration of the loop has to wait until all transactions executing older iterations have committed, and (b) a conflict-resolution policy that gives preference to the transaction that is executing the earliest iteration of the loop while rolling back later iterations. Multi-versioning allows for the removal of Write-After-Write (WAW) and Write-After-Read (WAR) loop-carried dependences on the `glob` variable. As shown in Figure 2.3, in the first four iterations `cond` evaluates false and write variable `glob` without aborts. Then, at iteration 4, the eager-conflict detection mechanism detects the RAW loop-carried dependence violation on variable `glob` between iterations 4 and 5, thus

```
d= STRIP_SIZE;
inc=(NUM_THREADS-1)*STRIP_SIZE;
i=param->i; // initial value of i for this thread

for(; i < N; i += inc ){
  prev_i=i;
Retry:
  if (!BEGIN()){
    for (; i-prev_i < d && i < N; i++){
      if(cond) glob++; else glob=i;
    }
    END();
  }
  else goto Retry;
}
```

Figure 2.2: Code of each thread to parallelize Figure 2.1's loop with TLS on ideal HTM system.

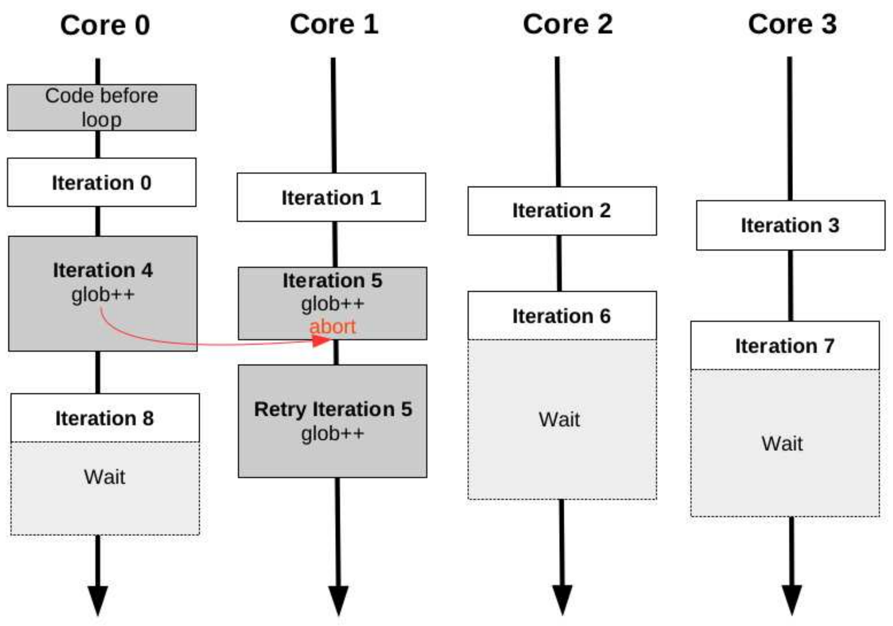


Figure 2.3: Execution flow of Figure 2.2's code with `STRIP_SIZE`=1 and `NUM_THREADS`=4.

rolling back iteration 5 because it should occur after iteration 4. Subsequent iterations wait for the previous iterations to commit.

2.4 Strip Mining

Strip Mining divides a single loop into a pair of loops (doubly-nested loop), thus the original loop is divided into strips of some size, the *strip size*. The outer loop steps between the strips and the inner loop steps through each strip. The maximum trip count of the inner loop is equal to the strip size [66].

2.5 Loop Peeling

Loop Peeling removes the first or last few iterations from a loop and performs them outside of the loop [66]. If the trip count of the loop is not constant the peeled code has to be protected with additional runtime tests.

2.6 Traces

A trace is formed by basic blocks and corresponds to a cycle-free path in a control flow graph. Part of this work focuses on generating traces for hot code regions. Such regions consist of relatively few instructions that are responsible for a large share of the program execution time. Traces are good candidates for optimization [20]. In this work, hot code regions are identified using program profilers (*e.g.* VTune [49]) and then appropriate traces for STO are found within the regions.

2.7 Optimizations using Traces

Generating larger traces creates more optimization opportunities because traces:

- Simplify the control-flow graph of a hot region of code such as a loop body. In STO control-flow statements are evaluated at the end of the execution of a trace and thus longer sections of control-flow free code are exposed to the compiler. The resulting control-flow graph is simpler and therefore more amenable to optimizations (dead-code elimination, code motion, constant propagation, *etc.*). This idea of optimization is used in this work.

- May contain function calls, thus incrementing the possibilities of optimization. Therefore *inlining* can be used to reduce the overhead of invoking and returning from methods [8].This idea of optimization is also used in this work.

- Typically contain the most frequently executed portions of a program and therefore can be used to optimize frequently executed instructions [9].

- Can be used to eliminate the cost of infrequently executed instructions from the execution of hot traces [9].

Chapter 3

Related Work

This chapter presents several works that have been proposed in the literature related to Thread-Level Speculation, Speculative Trace Optimization, and Data Dependence Profiling.

3.1 Thread-Level Speculation

Steffan *et al.* explored the potential of using *Thread-Level Data Speculation (TLDS).* TLDS is a technique that allows the compiler to safely parallelize codes in cases it cannot statically prove that dependences do not exist [55]. TLDS can offer performance improvements for applications where automatic parallelization would otherwise appear infeasible.

Steffan *et al.* presented a cache coherence that supports thread-level speculation (TLS) on a wide range of different parallel architectures because it is a straightforward extension of write-back invalidation-based cache coherence [56, 57].

The absence of hardware support for TLS led to the development of software-based implementations of TLS [43, 48]. While these implementations attempt to make the best use of existing hardware resources, the large overhead of buffering, validation, and in-order commits results in degradation of performance. Thus, hardware support appears to be essential to deliver effective performance improvement with TLS.

Although much previous research work on hardware support for TLS exists, up to now (circa 2016) most attempts to estimate the performance benefits of TLS were based on simulation studies [54, 56, 55, 47]. For example, Packirisamy *et al.* show that some of the SPEC CPU 2006 benchmarks have potentially parallel loops that can be successfully parallelized with TLS to achieve up to 78% of speed-up [47]. They also predict that parallelizing loops with infrequent RAW loop-carried dependences with TLS can result in speed-ups of up to 60%. Their performance predictions are based on a trace-driven simulator based on SimpleScalar that supports multiple cores, speculative execution, and advanced TLS features. In contrast, this work describes an evaluation of TLS running on existing HTMs from both the Intel Core and the IBM POWER8. While their study assumed the absence of false sharing and the availability of multi-version caches, ordered transactions and forwarding, this work targets actual off-the-shelf HTM hardware where none of these assumptions are true. Moreover, this work also proposes code transformations to overcome limitations of actual commercial hardware, making them amenable for efficient TLS execution.

Odaira and Nakaike study Thread-Level Speculation in the Intel TSX [44] by manually modifying parallel benchmarks from the SPEC CPU 2006 suite. Their performance evaluation indicated that up to 11% of speed-up can be achieved even without using the advanced features of HTM. However, for many benchmarks they find that TLS results in degraded performance. Their research suggests that the main reason for the performance degradation are transaction aborts due to memory conflicts. Therefore, they suggest that future HTM hardware should support not only ordered transactions but also data forwarding, multi-versioning cache, and word-level conflict detection. However, the comparison of TLS performance on Intel Core and IBM POWER8 presented in this work demonstrates that speed-ups can be achieved for some loops even on off-the-shelf hardware that does not implement their recommendations.

Nakaike *et al.* compare four HTM systems: Blue Gene/Q, zEC12, Intel TSX, and POWER8 [41]. Their experimental results indicate that the four HTM systems have implementation issues, and none has the best performance in all of the benchmarks. They claim that Intel TSX has extra transaction aborts due to adjacent cache-line prefetcher — which is corroborated by the results in this study, and that POWER8 has more capacity aborts than the other HTM systems because of its small transaction capacity — another result that is confirmed by this study. They also claim that `tsuspend`/`tresume` instructions improve TLS performance and support this claim with evaluation of two benchmarks: `milc` and `sphinx3`. They do not show the abort ratios for these two cases. In contrast, this study claims that false sharing is considerably exacerbated in TLS parallelization on top of HTM, and that although HTM features (*e.g.* suspend/resume instructions) can provide support for TLS, they do not provide all the support that is required for TLS to deliver on its promised performance.

3.2 Speculative Execution of Loops with Transient Dependences

HELIX is a compiler that has previously delivered good speed-ups for irregular programs on a six-core Intel i7 [29] machine [11]. HELIX parallelizes loops in sequential programs, distributing the iterations to available cores in a round-robin fashion. To preserve dependences between iterations or (may) loop-carried dependences, HELIX creates *sequential segments* that are subsets of iterations whose execution on cores must respect the loop-iteration order of the sequential program. These sequential segments correspond to *Strongly Connected Components (SCCs)* in a *Data-Dependence Graph (DDG)* that have at least one loop-carried dependence. An SCC formed by a single node with no loop-carried dependences is considered a *parallel segment* that does not need synchronization. A sequential segment implements the necessary synchronization to wait for the production of a loop-carried dependence variable value from a previous iteration, and to signal when the value is ready to use in a future iteration.

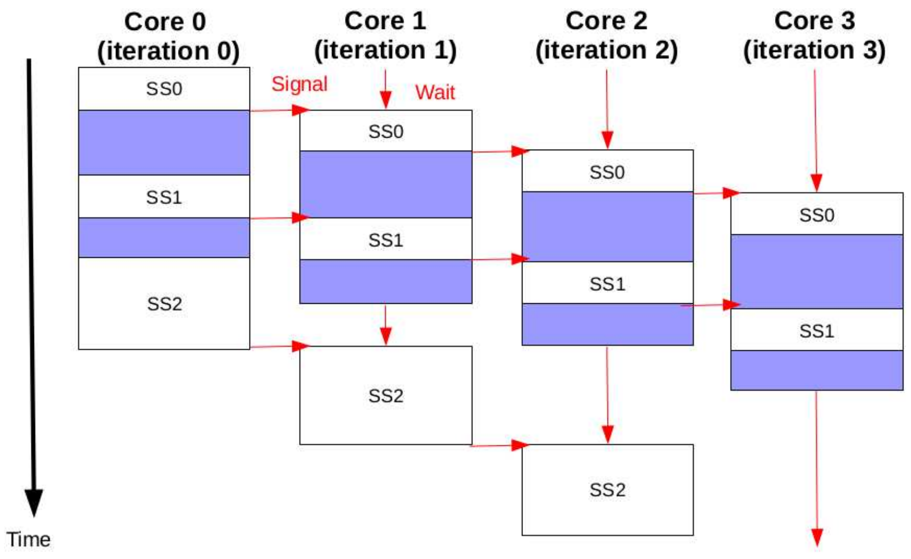


Figure 3.1: HELIX Execution flow of Figure 2.1. Sequential segments are synchronized.

To understand the HELIX approach, consider the code shown in Figure 2.1. This code contains a loop where the increment of a global scalar variable `glob` is controlled by the condition `cond`. The inner loop `for` in lines 10- 17 reads and conditionally updates a position from array `B`. HELIX creates three sequential segments: Sequential Segment 0 (SS0), also called *prologue*, is always created to control the end of the loop; Sequential Segment 1 (SS1) preserves dependences in global variable `glob` because HELIX pessimistically assumes that `cond` always evaluates true; and Sequential segment 2 (SS2) surrounds the inner loop to preserve a possible dependence in some index in the array `A`. Figure 3.1 shows the execution of sequential segments of the loop in Figure 2.1 using synchronization, blue portions represent parallel segments. Static analysis cannot prove that the loop is free of loop-carried dependences, thus it must be conservative and create these sequential segments. If these dependences actually occur at runtime but they are transient, HELIX could use TLS to avoid synchronization in those sequential segments as proposed in [40].

For instance, assume an execution of the loop of Figure 2.1 that uses an input that leads to the following dependences: SS0 never has a loop-carried dependence (the value of `N` is known at compile time); SS1 always contains a dependence because `cond` always evaluates true; and SS2 contains a transient dependence. Due to this transient dependence, HELIX must synchronize SS2 and thus will not exploit parallelism in it. However, the index `factor*(i%4) + j` evaluates to different values at each iteration of the outer loop if the number of cores is less than or equal to four — assuming that the distribution of iterations to cores follows a round-robin fashion.

Murphy *et al.* [40] propose a technique to speculatively parallelize loops that exhibit transient loop-carried dependences — a loop where only a small subset of loop iterations have actual loop-carried dependences. The code produced by their technique uses a TM hardware (TCC hardware) and software (Tiny STM) model running on top of the HELIX time emulator. They developed three approaches to predict the performance of implementing TLS on the HELIX time emulator: coarse-grained, fine-grained, and judicious. The *coarse-grained approach* speculates a whole iteration while the *fine-grained approach* speculates sequential segments and executes parallel segments without speculation. The *judicious approach* uses profile data at compile time to choose which sequential segment to speculate or synchronize so as to satisfy (may) loop-carried dependences. They conclude that TLS is not only advantageous to overcome limitations of the compiler static data-dependence analysis, but that performance might also be improved by focusing on the transient nature of dependences.

Murphy *et al.* evaluated TLS on emulated HTM hardware using `cBench` programs [15] and, surprisingly, predicted up to 15 times performance improvements with 16 cores [40]. They arose at these predictions even though they did not use strip mining to decrease the overhead of starting and finishing transactions as we suggest in this work. Particularly, fine-grained speculation without strip mining can result in large overheads due to multiple transactions (sequential segments) per iteration, even larger than coarse-grained speculation. They parallelized loops in a round-robin fashion which can result in small transactions, large number of transactions, high abort ratio, bad use of memory locality, and false sharing.

Their over-optimistic predictions are explained by the fact that their emulation study does not take into account the overhead of setting TLS up — which is specially high without strip mining. For instance, their emulation study predicted speed-ups even for small loops. However, when executing such loops in real hardware, the TLS overhead — setup, begin/end transactions, and aborts — would nullify any gain from parallel execution. In [40] the authors conclude that fine-grained speculation coupled with static dependence analysis is possibly the best way to exploit all the parallelism in loops. However, static dependence analysis can be very imprecise and report a large number of sequential segments that would prevent good speed-ups.

Odaira and Nakaike and Murphy *et al.* use coarse-grained TLS to speculate a (strip-mined) whole iteration and perform conflict detection and resolution at the end of the iteration to detect RAW dependence violations [44, 40]. To illustrate, assume an execution of the example of Figure 2.1 where `cond` always evaluates true, and thus the `glob` variable is increased at each iteration of the outer loop. With coarse-grained TLS the execution of this outer loop would be serialized for such execution. The advantages of coarse-grained TLS are: (a) it is simple to implement because it does not need an accurate data dependence analyzer. (b) the number of transactions is smaller than or equal to the fine-grained or judicious approaches; and (c) there is no synchronization in the middle of an iteration. The downside is that even a single frequent actual loop-carried dependence will cause transactions to abort and re-execute the whole iteration, thus serializing the execution.

3.3 Speculative Trace Optimization

Traces have been used for traditional optimizations. Fisher was the first to introduce the concept of traces and to use it for instruction scheduling [20]. *Trace Scheduling* is a global compaction technique in contrast with local compaction techniques whose domain is a basic block of code. The idea is to schedule the most frequently executed traces (defined by a feedback) quickly. Extra instructions, referred as *compensation code*, must be added so that other paths not optimized by the technique become valid. This work has been extended by Ellis in the Bulldog Compiler [19], Chang *et al.* [12], and Hwu *et al.* [27].

Hwu *et al.* developed a set of techniques for exploiting ILP across basic block boundaries [27]. These techniques are based on a structure called the *superblock*. A superblock is a trace that has no side entrances, thus the control may only enter from the top but may leave at one or more exit points. A copy of a portion of a trace is made from the first side entrance to the end, and all side entrances into the trace are moved to the corresponding copy. The superblock enables the optimizer and scheduler to extract more ILP along important execution paths by removing constraints due to other unimportant execution paths.

Chang *et al.* used profiling information in the Trace Selection Algorithm (the first pass of Trace Scheduling) [12]. They examined the predictability of branches within traces, concluding the use profiling information in Trace Scheduling can guide global code motion effectively with very small off-trace overhead.

Hank *et al.* introduced a technique called *region-based compilation* where the compiler is allowed to divide the program into regions of code (instead of methods) as a desirable unit of work [22]. Region-based compilation allows the compiler to control the problem size while exposing inter-procedural optimization and code motion opportunities.

Static Trace Scheduling involves selecting traces and scheduling instructions on these traces trying to increase ILP, and improving the performance on a single processor. STO differs from these approaches because it collects all traces and speculatively optimizes and executes them on an HTM system trying to improve the performance on multiple processors.

Profile information is used to identify heavily executed paths in a program (or traces). Ball and Larus described an algorithm for path profiling that determines how many times each acyclic path in a routine executes [4]. The algorithm selects and places profile instrumentation to minimize run-time overhead, and it accurately determines dynamic execution frequency of control-flow paths in a routine. This type of profiling subsumes the basic block and edge profiling that do not always correctly predict frequencies of overlapping paths. They select a number of paths and encode them so that each path has an index that can be used to access the counter of the corresponding path.

Young developed a technique to collect path profiles efficiently, and then applies the path profile to two optimizations: static correlated branch and path-based superblock scheduling [68]. The potential next paths for a given path are kept track. At runtime, the program uses this information to find what path is being followed by at a given instruction.

Data-flow analysis computes its solution over all paths of the program; however, programs execute a small fraction of all possible paths, this subset is called *hot paths*. Ammons *et al.* described an approach to analyzing and optimizing programs, which improves the precision of data-flow analysis throughout hot paths [2]. Their technique detects hot paths, creating a *hot path graph (HPG)* where these paths are isolated. They then perform flow analysis in the original CFG and the HPG, taking a subset of the HPG that contains hot paths for which the analysis differed in a favorable way from that in the CFG. This final subset is used to perform constant propagation.

Our work differs from these approaches on path profiling as they developed techniques to improve the performance of profiling, thus to use this more accurate profile information in optimizations. We profile the entire program to find the frequently executed sections of a program and then collect the traces of these sections to be speculatively executed.

Bradel *et al.* proposed and evaluated an approach for automatic parallelization based on traces as units of parallel work [7]. They described an execution model that uses traces to extract parallelism from programs. They implemented a system that shows the benefits and addresses the challenges of using traces for data-parallel programs in an off-line feedback-directed system. The results of performance compares favorably to the performance of these programs manually parallelized. Bradel *et al.* propose and evaluate an approach for automatic parallelization based on traces as units of parallel work [7]. They implement a system that takes a sequential program (binary file), identifies the traces on it, and groups them into coarse-grain units of computation (tasks). STO differs from this approach in that it uses a real HTM to speculatively optimize traces of loop iterations and we are not automatically parallelizing traces of the binary file.

Neelakantam *et al.* proposed that microprocessors provide hardware primitives for atomic execution to increase the effectiveness of speculative compiler optimizations [42]. Thus, the compiler may speculatively optimize a program's hot path in isolation as a superblock. Atomic execution guarantees that if a miss-speculation is produced, the control is transferred to a non-speculative version of the code, relieving the compiler from generating compensation code. They considered that the implementation of the proposed hardware atomicity has significant differences from TM. These optimizations result in 10-15% average speed-up. STO differs from this in that, to carry out the speculative compiler optimizations, it speculates in parallel all possible traces within a hot-loop iteration using a real HTM.

3.4 Data-Dependence Profilers

This section analyzes two techniques used to detect loop-carried dependences. The first one is the *Pairwise* method, which was used in [35]; the second technique is the *Stride-based* method, which was implemented in the SD^3 profiler [33, 32].

Static dependence analysis techniques have been extensively studied in the literature. Approaches like the GCD Test [39] and Banerjee's equality test [34] have been used, for a long time, in the design of parallelizing compilers. These techniques analyze data dependences in array-based memory accesses, and thus are not effective when used in languages which allow pointers and dynamic allocation. Besides, static analysis can become complex in situations when: (a) the bounds of the loop are not known, (b) dynamically created arrays are passed through deep procedure call chains, or (c) the loop-body has a complicated control-flow. In such cases, dynamic loop dependence analysis is an alternative as all memory addresses are resolved at runtime.

3.4.1 Pairwise Method

The Pairwise Method is still considered the state-of-the-art for loop-carried dependence testing. The basic idea of this method is to store, into a hash table (*pending table*), all memory references (*pending references*) occurring during the current iteration of a loop. When an iteration finishes, the pending table is compared against the *history table*, which stores all memory references (*history references*) of all previous iterations. This method solves nested loops dependences, by having a pending and a history table for each loop.

The Pairwise algorithm works as follows. First, memory references are stored into the pending table during an iteration. After of finishing the iteration, the pending table is checked against the history table to discover loop-carried dependences. Before of continuing with the next iteration, the pending references are copied to the history. This process is repeated until the end of all iterations for this loop. If this loop is nested within another loop, the history table of the inner loop is propagated to the pending table of the outer loop. Afterwards, this pending table is checked against the history table of the outer loop (that is initialized empty) to discover loop-carried dependences. This process for the outer loop continues until the end of all its iterations.

Loop-independent dependence does not prevent parallelization; thus, any dependence analyzer must distinguish if a dependence is loop-carried or loop-independent. The Pairwise algorithm, as described in [33, 69], detects loop-independent dependence by implementing *kill addresses* (a technique similar to the notion of *kill sets* in *data-flow analysis*), which marks a memory address as killed once it is written in an iteration. Then all memory references within the same iteration to the killed address are ignored. However, this technique could lead to incorrect results in multithreaded program executions not reporting existing violations of loop-carried dependences between threads as it only works in serial executions or per-thread analysis.

To demonstrate *kill addresses* effectiveness, SD^3 authors analyze the following code from SPEC 179.art [33].

```
void match() {
     if (condition)
            pass_flag=1;
}
void scan_recognize(...) {
     for (j = starty; j < endy; j += stride)
            for (i = startx; i < endx; i += stride){
                   ...
                   pass_flag = 0;
                   match();
                   if (pass_flag == 1)
                          do_something();
                   ...
            }

}
```

Figure 3.2: Dependence in pass_flag in `179.art`.

Assuming that $pass\_flag$ is a global variable, they argue that a loop-independent flow dependence exists on $pass\_flag$ as this variable is always initialized at line 9 before any use on every iteration, which should not prevent parallelization, and it is true in a serial execution context. However, in a multithreaded execution this code could have the following problem. Lets assume that thread X executes line 9 after the same thread executes line 3. Before this thread executes line 11, another thread Y executes line 9. Thus, when thread X reads variable $pass\_flag$ at line 11, it will be incorrect as the execution does not respect the loop-carried WAR dependence between the write reference at line 9 (executed by thread Y) and the read reference at line 11 (executed by thread X). Thread X will not execute *do_something* when it had to do so.

This problem can be solved with privatization as SD^3 authors argue in [33]. On the other hand, according to our approach, all violations of loop-carried dependences must be informed to force not omitting corrections of renaming of variables that avoid WAR and WAW loop-carried dependences.

Killed addresses technique is also used by SD^3 [33, 32] so it could lead to inaccurate results due to multithreaded program executions. Our *OpenMP checker* deals with this problem by storing the thread identifier (*thread ID*) for each memory event within the loop body.

Other problems of Pairwise Method are the time and memory overhead it requires to store all memory references within a loop. These problems can be more complicated when considering nested loops, as the Pairwise Method propagates history references of inner loops to pending tables of outer loops. We focused on the functionality of the new *check* construct and the integration with GCC and LLVM. On the other hand, we partially addressed the time overhead using pipeline-parallelization of the stages of our implementation.

3.4.2 Stride-based Method

This method was proposed in [33] and has the Pairwise Method as a baseline algorithm. It tries to solve the problem of memory overhead by means of compression, and to solve the time overhead by using data-level parallelism.

The compression is achieved by using stride formats. For example, array reference $A[d * i + b]$ generates an address stream that has a stride composed by a base (b), a distance (d), and an induction variable (i). SD^3 [33] discovers strides dynamically and uses them directly to check loop-carried dependences. Strides are detected by a detector assigned to each PC. If a memory reference is not part of a stride, it is called a *point*.

Stride-based method is implemented using an extension of the Pairwise algorithm defining pending and history *stride* tables. To detect dependences in strides they first do an interval test employing interval trees based on *red-black trees* [14]. They then perform Dynamic-GCD test, as described in [32]. Notice that SD^3 focuses on reducing the memory overhead due to deep nested loops, contrary to this work, which considers an inner loop as serial code within the loop body as it is more focused on the integration with OpenMP, and to solve the problems with multithreaded executions.

SD^3 solves the problem of time overhead by exploiting data-level parallelism contrary to the task-level parallelism approaches adopted in previous works [38]. It distributes memory references into tasks that perform data-dependence checking with a subset of the entire input. The address space is divided at every 2^k bytes and the subsets are mapped to M tasks on N cores.

As in the previous Pairwise algorithm, this method maintains killed addresses to distinguish between loop-carried and independent dependences. However, as discussed in the previous Section 3.4.1, this technique could lead to incorrect results for multithreaded application executions. Therefore, SD^3 method can ignore some violations of loop-carried dependences for multithreaded executions as SD^3 analysis is performed sequentially or on a per-thread basis. As explained before, *checker* deals with this problem.

Another problem with SD^3 is that it is more effective for profiling inner loops than outer loops. As data-dependence analysis proceeds to outer loops, irregular strides are more frequent (the compression method will not work), making the cost of detecting dependences extremely expensive. Also, this method requires additional static analysis to recover control flows and loop structures from a binary executable, which is complicated to implement [33]; thus, the selection of loops to analyze is also complicated.

Our solution to these problems is to limit the analyzed loops according to the programmer instructions, while storing memory references in a memory/time efficient data structure as Multilevel Hash Table [10].

Chapter 4

Evaluating and Improving TLS in HTMs

This chapter presents a detailed analysis of the application of Hardware Transactional Memory (HTM) support for loop parallelization with Thread-Level Speculation (TLS).

4.1 Loop-Carried Dependences and False Sharing

Consider the simple `for` loop in Figure 4.1 (ommiting the inner loop in lines 10-17) that has two alternative paths of execution. Executing line 4 in iteration `i` creates a loop-carried dependence with iteration `i-1`. The alternative path does not create a dependence. When the condition evaluated by the `if` statement is unknown at compilation time, this loop cannot be parallelized. However, if the loop-carried dependence is rare for a given execution of the program, the loop could profitably be executed in parallel speculatively using TLS.

TLS works best when the compiler has some information about the likelihood of dependences occurring at runtime. For instance, assume that a compiler has information — perhaps through profiling — that there is a high probability that the condition in line 3 of the code in Figure 4.1 is false when the loop index is a multiple of $N/4$ and that it is true for all other iterations. When the condition is false, the write to `glob` at line 6 kills the value written into the previous iteration and therefore there is no loop-carried dependence. Figure 4.2a shows a possible parallelization for this loop with the arrows indicating loop-carried dependences. Applying TLS in this fashion would not be productive because in three out of four speculative executions the loop-carried dependence would cause an access conflict leading to an abort followed by a retry. An alternative parallelization is shown in Figure 4.2b. Here the compiler has grouped the likely dependent iterations into a single thread using strip mining and privatizing variable `glob` for each thread, thus the expectation is that there will be no aborts and retries due to dependence when executing the loop in this fashion. However, if the compiler is using probabilities rather than proofs of independence, DOALL parallelization cannot be applied [26]. The performance of TLS will depend on the amount of computation that is successfully speculated at runtime and on the probability of loop-carried dependence occurring [47].

```
1   for (i = 0; i < N; i++){
2     /* Start sequential segment 1 */ /* Global scalar, glob */
3     if (cond)
4       glob++;
5     else
6       glob=i;
7     /* End sequential segment 1 */
8     A[i]= glob*i;
9     /* Start sequential segment 2 */
10    for(j = 0; j < factor; j++){
11      /* Global array, B */
12      int tmp = B[factor*(i%4) + j];
13      tmp += i*5;
14      if(tmp%2 == 0){
15        B[factor*(i%4) + j] = tmp;
16      }
17    }
18    /* End sequential segment 2 */
19  }
```

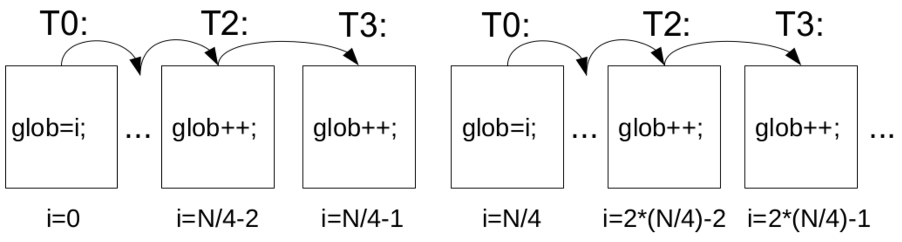
Figure 4.1: Figure 2.1. A loop with two *may* loop-carried dependences.

False sharing due to the cache coherence protocol also limits the performance of TLS. Figure 4.3 revisits the round-robin parallel execution of the example loop into four threads, which appeared in Figure 4.2a, but now also showing the accesses to the array `A`. Assume that thread $T0$ accesses (line 8) the position `A[i]` of the vector and thread $T1$ accesses position `A[i+1]` of the vector. Also assume that positions `A[i]` and `A[i+1]` map to the same cache line. The flow of execution could be as follows: (1) T_0 writes `A[0]` to its cache line; (2) T_1 reads the cache line; (3) T_1 writes `A[1]` to its cache line and thus issues a write-invalidate command; (4) T_0 reads the cache line; (5) T_0 writes `A[4]` to its cache line issuing a write-invalidate command. As illustrated in Figure 4.3, multiple threads will be accessing the same cache line, thus producing a large number of write-invalidates and cache misses and reducing the parallelization performance.

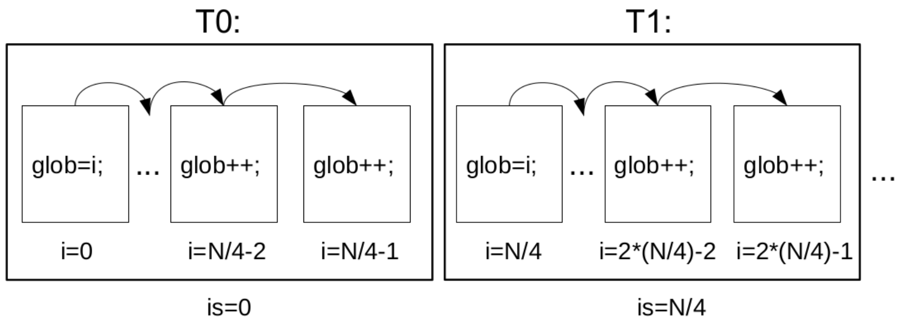
It is well known that false-sharing can have a considerable impact in loop performance even for embarrassingly parallel DOALL loops [61]. Techniques, such as loop strip mining, have been used to eliminate such overhead [13] by forcing writes to the same cache line to occur within the same thread. For example, the code of Figure 4.1 could be re-written (Figure 4.4) in such a way that all positions of the vector `A` that map to the same cache line are accessed by the same thread, thus eliminating false-sharing (Figure 4.5). Given that `A[i]` is a vector of `doubles`, in an architecture with cache line that stores eight doubles (64 bytes), setting `STRIP_SIZE` to eight eliminates false sharing. When targeting a specific architecture, a compiler has access to the size of the cache line and thus can set an appropriate `STRIP_SIZE` to optimize performance.

The effect of false sharing on the performance of TLS has not been thoroughly evaluated. Also, no research has been conducted aiming at understanding how false sharing impacts program performance when HTM is used to support speculative techniques such as TLS.

The experimental results in this work indicate that false sharing can considerably

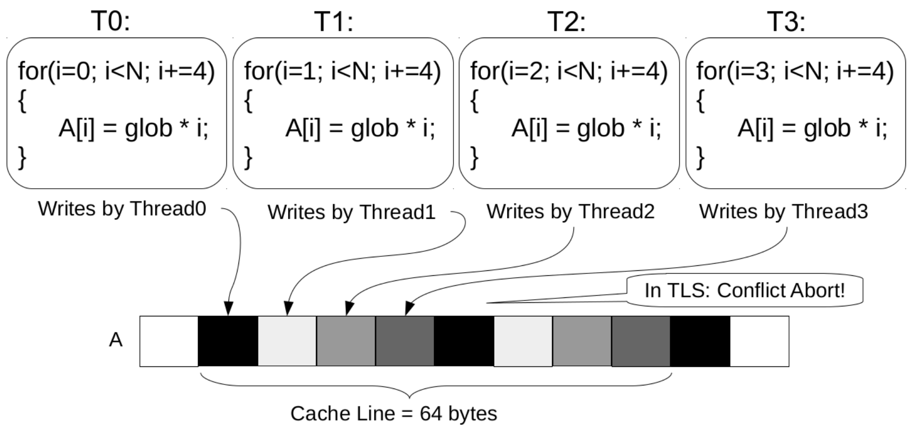


(a) Round-robin fashion parallelization.



(b) Strip mining parallelization.

Figure 4.2: Parallelization of Figure 4.1 with four threads.



```
for(i=0; i<N; i+=4)
{
     A[i] = glob * i;
}
```

```
for(i=1; i<N; i+=4)
{
     A[i] = glob * i;
}
```

```
for(i=2; i<N; i+=4)
{
     A[i] = glob * i;
}
```

```
for(i=3; i<N; i+=4)
{
     A[i] = glob * i;
}
```

Figure 4.3: False sharing in the loop of Figure 4.1.

downgrade loop performance in the presence of HTM and TLS — much more so than in a regular DOALL parallelization. For instance, Table 4.1 shows the times (in seconds) for the execution of the hottest loop in the `sphinx3` benchmark program [24] when: (a) its serial version is executed; (b) the loop is parallelized using OpenMP `parallel for`; and (c) the loop is parallelized using TLS on HTM. In this loop the compiler reports a *may* dependence, but this dependence never occurs during runtime with the input provided for the benchmark. Therefore, while it is safe to parallelize the loop with OpenMP for the benchmark, for correctness TLS must be used for an execution of the program with an unknown input. Table 4.1 provides the execution time with four threads with and without strip-mining on the Haswell processor described in Section 2.2. Applying strip mining to eliminate false sharing in the `parallel for` version yields a fairly small speed-up in comparison with the parallelization without strip mining (2%). However, for the

```
    d = STRIP_SIZE;

    for(is = 0 ; is < N; is += d){
      for (i=is ; i - is < d && i < N; i + =1 ){
        if(...)
          glob++;
        else
          glob=i;
        A[i] = glob * i;
      }
    }
```

Figure 4.4: Loop of Figure 4.1 after applying strip mining.

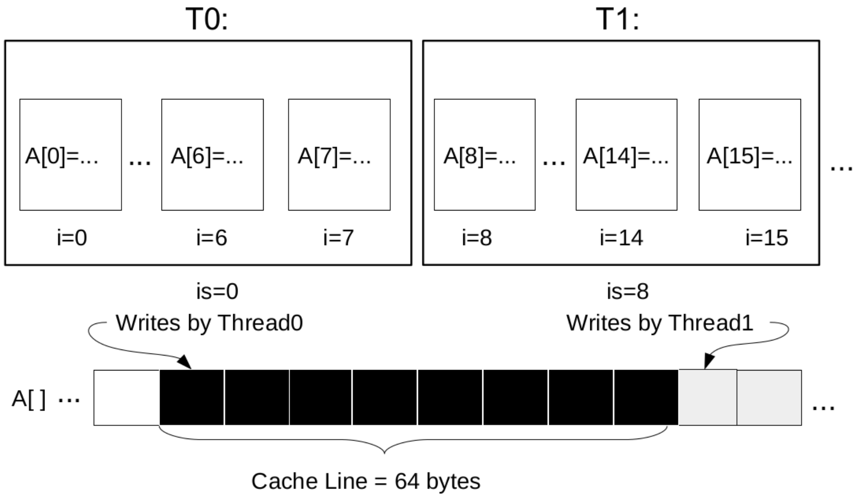


Figure 4.5: Strip-mining (`STRIP_SIZE=8`) the loop of Figure 4.1.

HTM-TLS version the whole-program speed-up is 11%. This suggests that, although HTM support reduces the overhead of maintaining speculative storage, it suffers from a large number of false-sharing induced aborts. Understanding this impact is one of the main goals of the remaining of this chapter.

4.2 TLS on top of HTM

Hardware support for Thread-Level Speculation (TLS) must have four features: (a) data conflict detection; (b) speculative storage; (c) ordered transactions; and (d) rollback when a conflict is detected [54, 56, 55]. Three of these features are also supported by the HTM

Table 4.1: Impact of false sharing on `sphinx3`.

	Without strip mining	**With strip mining**	**% Improvement**
Serial	390	390	
OpenMP	353	347	2%
TLS-HTM	409	370	11%

systems found in the Intel Core and POWER8, and thus these architectures could be used to support TLS. Odaira *et al.* evaluates TLS on Intel TSX; their work yielded some speed-ups but poor performance in most cases [44]. They claim that their performance result is explained by Intel TSX lacking some advanced TLS features such as ordered transactions, multi-versioning caches, data forwarding, or word-level conflict detection.

Table 2.2 shows the necessary features required to enable TLS on top of an HTM-supporting mechanism, and its availability in some modern architectures. Neither Intel TSX nor the IBM POWER8 provide all the features necessary to support TLS effectively. Blue Gene/Q is the architecture that implements almost all the features required for TLS. For this study we did not have access to a Blue Gene/Q machine and therefore we have not evaluated TLS in that machine. Bhattacharyya *et al.* found that the cost of starting a TLS region in BG/Q is high and observed only modest speed improvements due to TLS [5]. Given an architecture with HTM support that misses features required to implement TLS, one might ask if it could be used to implement TLS. The experimental results in this work indicate that such an implementation is possible provided that software emulate the required behaviour.

Implementing coarse-grained TLS on top of the POWER8 TM requires the software to emulate multi-versioning, a conflict-resolution policy, and ordered transactions. The code shown in Figure 4.6 is a TLS version of the loop example of Figure 4.1 (statement at line 8 and inner loop are omitted) with versions for TSX and POWER8 managed by a `switch-case` statement. In POWER8 the `tsuspend`/`tresume` instructions are used to non-speculatively wait for the iteration commit counter `next` to reach the value of the index variable for the current transaction before committing (`while` loop in line 26). The Intel TSX does not provide the ability to suspend transactions and to execute instructions non-speculatively and therefore a transaction cannot wait for its turn to commit as in the POWER8. The solution is to roll back a transaction that completes execution out of order using an explicit abort instruction (`xabort`) as shown in line 29 of Figure 4.6. This kind of abort is called an *order-inversion* abort.

To emulate multi-versioning the software privatizes all global variables written within a transaction by creating local copies. For example, `globL` in Figure 4.6 is a private local version of `glob` in the code of Figure 4.1. Besides eliminating WAW and WAR loop-carried conflicts, the privatization of global variables also simulates a conflict-resolution policy. For instance, in the data conflict between iterations 4 and 5 shown in Figure 4.7 iteration 4 could be aborted and rolled back. By privatizing variable `glob`, through its replacement with `globL` within the transaction, iteration 4 will not be aborted by data conflicts in `glob`; when it commits, it non-speculatively writes variable `glob` as shown in line 32 of Figure 4.6. This non-speculative write causes the abortion of any other iterations that read variable `glob` thus enforcing Read-After-Write (RAW) dependences.

The code in Figure 4.6 omits some details for clarity. If the transaction aborts, the program control jumps back to the `tbegin` instruction. We assume that each software thread is bounded to one hardware thread and executes a determined number of pre-assigned iterations. Strip mining is used to enable a single transaction to execute multiple iterations, however, the `STRIP_SIZE` must be limited to avoid exceeding the speculative storage capacity of the HTM system. False sharing can be exacerbated on a HTM as

```
1  d= STRIP_SIZE;
2  inc=(NUM_THREADS-1)*STRIP_SIZE;
3  i=param->i; // initial value of i for this thread
4
5  for(; i < N; i += inc ){
6     prev_i=i;
7     flag=0;
8  Retry:
9     if (next!=i){
10       if (!tbegin()){
11          for (; i-prev_i < d && i < N; i++){
12             if(cond){
13                if (!flag) {
14                   flag=1
15                   globL=glob;
16                }
17                globL++;
18             }
19             else
20                globL=i;
21          }
22
23          switch(ARCHITECTURE){
24             case IBM_POWER8:
25                tsuspend();
26                while(next!=prev_i);
27                tresume();
28             case INTEL_CORE:
29                if (next!=prev_i) tabort();
30          }
31          tend();
32          glob=globL;
33       }
34       else goto Retry;
35    }
36    else{
37       if (cond)
38          glob++;
39       else
40          glob=i;
41    }
42    next=prev_i+d;
43 }
```

Figure 4.6: Code of each thread to parallelize Figure 4.1's loop with TLS on POWER8 and TSX HTM systems.

explained in 4.1 thus our implementation uses code transformation techniques as strip mining and/or privatization. Moreover, if at the start of a retry the value of the iteration counter `i` is equal to the value of the commit counter `next` (see line 9) then that iteration executes non-speculatively.

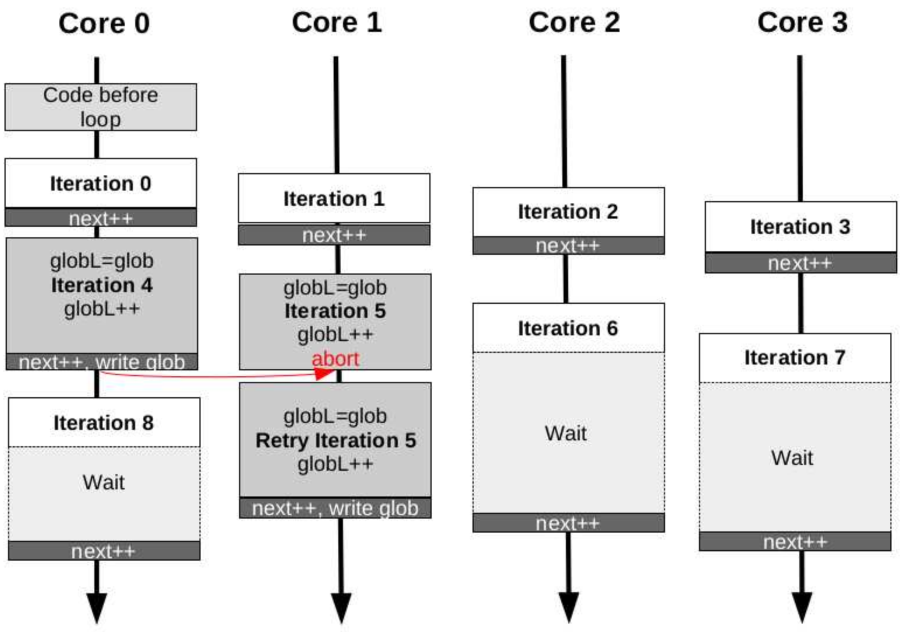


Figure 4.7: Execution flow of Figure 4.6's code with `STRIP_SIZE`=1 and `NUM_THREADS`=4.

4.3 False Sharing Effects on TLS

This section describes the problems due to false sharing and capacity overflow in a TLS parallelization on top of HTM and provides solutions to reduce or eliminate false sharing.

4.3.1 Capacity Overflow of Transactions

False sharing can be overcome with strip mining in some cases. For instance, in the example shown in Figure 4.1, setting `STRIP_SIZE` to eight removes the false sharing in the TLS parallelization and avoids conflict aborts as shown in Figure 4.5. However, in loops where each iteration performs transactional writes to multiple locations, there is a limit to the `STRIP_SIZE` that can be used before the speculative storage capacity of the HTM is exhausted.

```
SWEEP_START( 0, 0, 0, 0, 0, SIZE_Z ) // beginning of the for loop
  ...
  DST_C ( dstGrid ) = (1.0-OMEGA)*SRC_C ( srcGrid ) + DFL1*OMEGA*rho*(1.0- u2);  //
      write to dstGrid array
  DST_N ( dstGrid ) = (1.0-OMEGA)*SRC_N ( srcGrid ) + DFL2*OMEGA*rho*(1.0 +
      uy*(4.5*uy   + 3.0) - u2);
  ...
SWEEP_END
```

Figure 4.8: `lbm` hot loop.

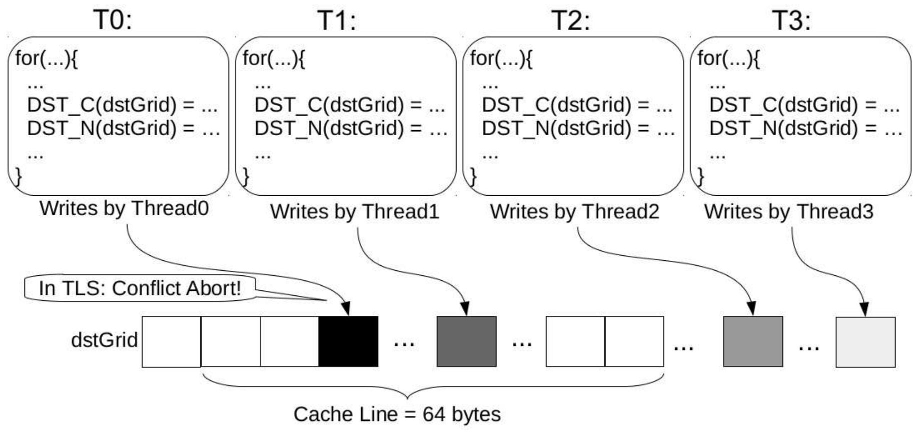


Figure 4.9: False sharing due to non-consecutive writes in the array `dstGrid`.

4.3.2 Non-consecutive Writes Within Transactions

False sharing cannot be overcome with strip mining in all cases. For instance, consider the code of Figure 4.8, which is a fragment of the hottest loop in the `lbm` benchmark. In this fragment each loop iteration performs transactional writes to the `dstGrid` array through macros (*e.g.* `DST_C`). However, consecutive iterations perform transactional writes to non-consecutive elements of the `dstGrid` array. Therefore, in a naive parallelization of this loop these non-consecutive accesses lead to false sharing. Figure 4.9 illustrates this case with a round-robin TLS parallelization running on an HTM system where the transaction executed by each thread aborts because of data conflicts induced by false sharing. False sharing cannot be overcome by using strip mining because the writes of consecutive iterations are not adjacent in the array `dstGrid`. One way to solve this problem is to implement word-level conflict detection in future architectures [44]. However, we can overcome the false sharing caused by the non-consecutive writes through the use of thread-local arrays to perform writes within the transaction and then copying back to the original `dstGrid` array. The writing into this array is a small fraction of the execution of this loop. Thus this TLS solution yields performance improvement even with the additional copy. With this code transformation, the conflict abort ratio decreases from 95% to 8% in Intel TSX. In POWER8 the TLS parallelization of this loop yields a speed-up of 30%.

4.3.3 TSX Cache-line-prefetcher Issues

In principle, prefetching should not affect the operation of any transaction because locations in a prefetched line should not be deemed as read or written by any thread. However, experimental evaluation clearly indicates that in TSX the cache-line prefetcher can be a source of conflict aborts due to false sharing. This evaluation confirms a hypothesis by Odaira *et al.* and a finding by Nakaike *et al.* [44, 41]. Figure 4.10 shows the parallelization of the same loop of Figure 4.1 with strip mining to overcome false sharing. On Intel TSX with the prefetcher enabled, when thread $T0$ writes eight consecutive positions of array `A` (64 byte cache line), adjacent memory locations are fetched by the cache-line prefetcher *and tracked as reads.* Therefore a conflict is generated between the two transactions executed by the respective threads because the thread $T1$ is writing to these adjacent

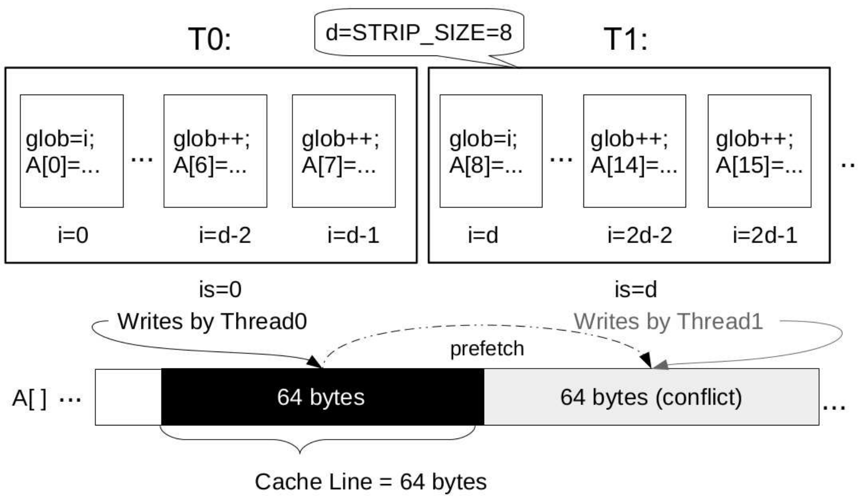


Figure 4.10: False sharing due to prefetching in TSX.

locations. This conflict causes a transaction to be aborted and rolled back.

4.4 Experiments

This section presents a performance assessment (speed-ups and abort ratios) of the TLS parallelization of loops from the SPEC CPU2006 benchmark suite running on Intel TSX and IBM POWER8. For all experiments the ref input is used for the SPEC benchmarks. The baseline for speed comparisons is the serial execution of the same benchmark program compiled at the same optimization level (cache-line prefetcher enabled) . Whole-program times are compared and not only the execution time of the region of the code to which TLS is applied. Each benchmark was run thirty times and the average time is used. Runtime variations were negligible and are not presented.

4.4.1 Benchmarks and Settings

Benchmarks were selected because there is potential to improve their performance through TLS [47, 46]. Table 4.2 shows: the file/line of the target loop in the source code; $\%Cov$, the fraction of the total execution time ran by the loop; N, the average number of loop iterations; AIS, the average iteration size measured in bytes [46]; $\%lc$, the percentage of iterations that have loop-carried dependence for the ref input. TLS makes most sense when the compiler cannot prove that iterations are independent, but dependences do not occur at runtime — thus most benchmarks that are amenable for TLS have an $\%lc$ of zero for the SPEC reference inputs. In Table 4.2, ss is the *strip size* used for the experimental evaluation in each architecture; and N' is the number of iterations after using strip mining for the respective loop.

This study uses an Intel Core i7-4770 processor with 4 cores with 2-way SMT, running at 3.4 GHz, with 16 GB of memory on Ubuntu 12.04.3 LTS (GNU/Linux 3.8.0-29-generic x86_64). Each core has a 32 KB L1 data cache and a 256 KB L2 unified cache. The four cores share an 8 MB L3 cache. The study also presents results from the same experiments

Table 4.2: Loop Characterization in Benchmarks.

Benchmark Loop	General					Intel TSX				POWER8			
	Location	%*Cov*	*N*	Average Iteration Size (AIS)	%*lc*	*ss*	*N′*	Speed-up	Problems	*ss*	*N′*	Speed-up	Problems
`mcf`	`pbeampp.c`, 165	40%	300	335	3%	20	15	1.12	RAW dependence and f. sharing	48	6	0.82	RAW, f. sharing and capacity
`sphinx3`	`vector.c`, 513	37%	2048	1108	0%	8	256	1.12	–	16	128	1.22	–
`h264ref`	`mv-search.c`, 394	36%	1089	6837	0%	16	68	1.20	–	32	34	0.93	capacity
`lbm`	`lbm.c`,186	99%	1300000	525	0%	15	86666	0.63	false sharing	19	68421	0.72	capacity and f. sharing
`milc`	`quark_stuff.c`, 1523	20%	160000	1972	0%	4	40000	1.07	–	4	40000	1.07	f. sharing
`libquantum`	`gates.c`, 89	62%	2097152	43	0%	1024	2048	1.07	f. sharing and small AIS	324	5462	0.83	f. sharing and small AIS
`astar`	`Way2_.cpp`, 100	60%	1234	1548	20%	128	10	0.92	RAW dependence	256	5	0.77	RAW dependence

```
1   for( ; arc < stop_arcs; arc += nr_group )
2   {
3       if( arc->ident > BASIC )
4       {
5           red_cost = arc->cost - arc->tail->potential + arc->head->potential;
6           if( bea_is_dual_infeasible( arc, red_cost ) )
7           {
8               basket_size++;
9               perm[basket_size]->a = arc;
10              perm[basket_size]->cost = red_cost;
11              perm[basket_size]->abs_cost = ABS(red_cost);
12          }
13      }
14  }
```

Figure 4.11: `mcf`'s hottest loop.

with an Intel Core i7-6700HQ processor with *TSX New Instructions (TSX-NI)*, 4 cores with 2-way SMT, running at 2.6 GHZ, with 16 GB of memory on Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-29-generic x86_64). This version of TSX has fixed the well-known bug that had led to Intel disabling TSX in the Intel Core i7-4770. The benchmarks are compiled with GCC 4.9.2 at optimization level `-O3` and with the set of flags specified in the SPEC2006 configuration file. The IBM processor used is a 4-core POWER8 with 8-way SMT running at 3 GHz, with 16 GB of memory on Ubuntu 14.10 (GNU/Linux 3.16.0-44-generic ppc64le). Each core has a 64 KB L1 data cache, a 32 KB L1 instruction cache, a 512 KB L2 unified cache, and a 8192 KB L3 unified cache. The benchmarks are compiled with the XL 13.1.1 compiler at optimization level `-O2`.

4.4.2 Results

This section presents results and analysis.

Trade-off Between Conflicts and Capacity

For some benchmarks, there is a trade-off between the elimination of conflicts through strip mining and the speculative capacity of the HTM. For instance, the hottest loop of the `mcf` benchmark is shown in Figure 4.11. When line 8 is executed this loop has a RAW loop-carried dependence that causes most of the aborts due to conflicts. For TLS, the `basket_size` variable is privatized within the transaction to reduce conflicts caused by WAW or WAR dependences. For the reference input 3% of the iterations have loop-carried dependences and there are no speed-ups in POWER8 — even using `tsuspend`/`tresume` instructions — because the *ss* that is necessary to reduce conflicts caused by RAW dependences and false sharing (aligning to 128-byte cache line size) results in capacity aborts.

Figure 4.13 shows the distribution of aborts and commits. In `mcf` order inversion is not a major problem in POWER8 but is a significant source of aborts in TSX. In spite of these aborts, `mcf` sees speed-ups of up to 12% with four threads on TSX. To achieve

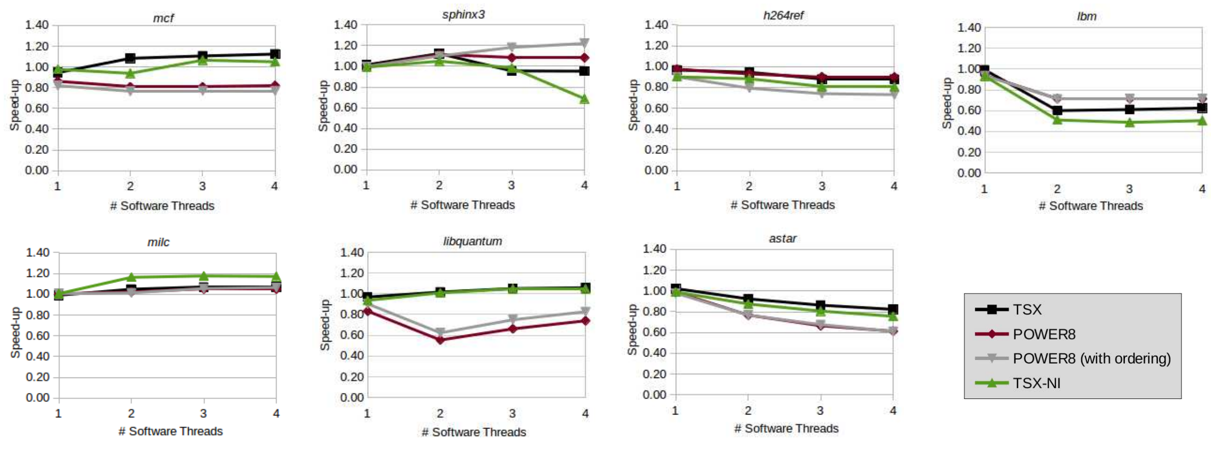


Figure 4.12: Speed-ups for TLS execution on TSX, POWER8, POWER8 with ordering (`tsuspend`/`tresume`), and TSX-NI.

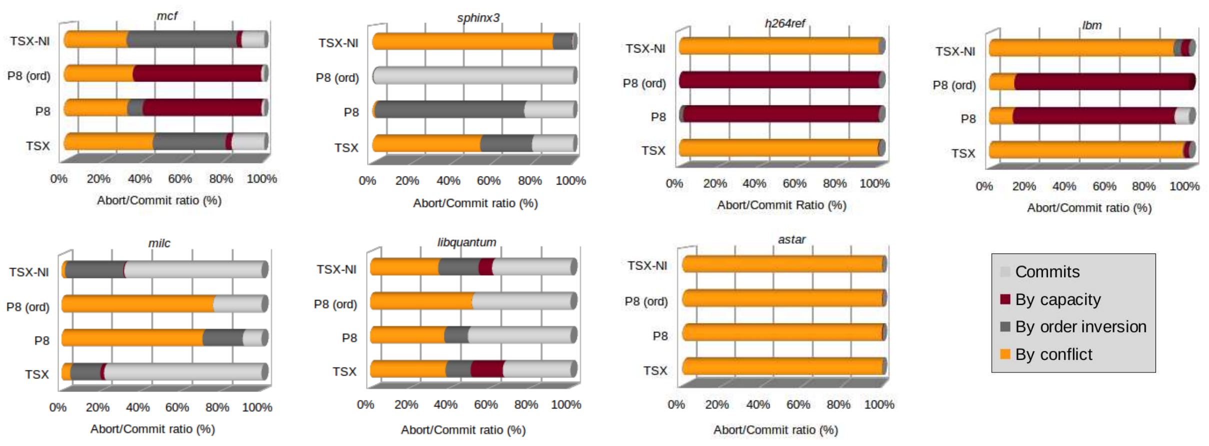


Figure 4.13: Abort/Commit ratio by TLS execution with 4 threads on TSX, POWER8, POWER8 with ordering (`tsuspend`/`tresume`), and TSX-NI. Abort reasons are shown.

this speed-up, both the false sharing caused by `BASKET` structures pointed by elements of the `perm` array and the occurrence of RAW loop-carried dependence violations must be mitigated with an appropriate ss.

The hottest loop of `h264ref`, shown in Figure 4.15, is another example of the tradeoff between conflicts and capacity. False sharing, caused by writes to the `block_sad` array in line 16, can be eliminated by setting ss to 16 in TSX and to 32 in POWER8: in each iteration of the `pos` loop four bytes are written into the `block_sad` array and the cache line sizes of TSX and POWER8 respectively have 64 and 128 bytes. Loop peeling is used to align the accesses to the start of cache lines.

In POWER8 an $ss = 32$ leads to a large amount of writes in each iteration and results in capacity aborts, see Figure 4.13 (the elimination of false sharing in this loop requires a strip size of $32 \times k$, where k is a positive integer). The use of `tsuspend`/`tresume` to ensure ordering leads to worse performance, see Figure 4.12, because these instructions are only beneficial to eliminate order-inversion aborts.

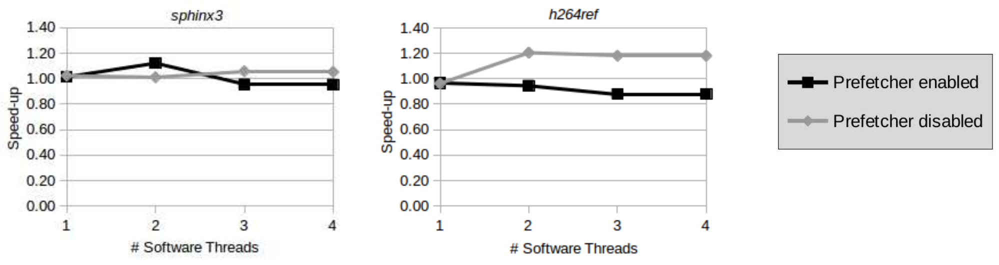


Figure 4.14: Speed-ups for `sphinx3` and `h264ref` by TLS execution on TSX with prefetcher enabled/disabled.

```
      for (pos = 0; pos < max_pos; pos++) {
         ...
         if (range_partly_outside){
            if (...)
               PelYline_11 = FastLine16Y_11;
            else
               PelYline_11 = UMVLine16Y_11;
         }
         bindex = 0;
         for (blky = 0; blky < 4; blky++){
            for (y = 0; y < 4; y++){
               refptr = PelYline_11 (ref_pic, abs_y++, abs_x, img_height, img_width);
               LineSadBlk0 += byte_abs [*refptr++ - *orgptr++];
               ...
            }
            block_sad[bindex++][pos] = LineSadBlk0;
            ...
         }
      }
```

Figure 4.15: `h264ref`'s hottest loop.

Eliminating False Sharing with Strip Mining

Strip mining with an appropriate strip size can eliminate false sharing. For example, the hottest loop of the `sphinx3` benchmark is shown in Figure 4.16. Although this loop has no conflicts with the reference input ($\%lc = 0\%$ in Table 4.2), the writes to the `score` array in lines 10 and 11 cause conflict because of false sharing. Eight bytes are written in each iteration of the loop. Therefore, setting the strip size (ss) to 8 in TSX and to 16 in POWER8 eliminates the false sharing because the cache line sizes of TSX and POWER8 are, respectively, 64 and 128 bytes. Accesses must be aligned to start of cache lines through loop peeling. The speed-ups of up to 22% with four threads on POWER8, see Figure 4.12, require the use of `tsuspend`/`tresume` instructions to spin-wait for ordering outside of the transaction thus eliminating aborts due to ordering as shown in Figure 4.13.

```
    for (r = offset; r < end-1; r += 2)
    {
      m1 = gautbl->mean[r];
      m2 = gautbl->mean[r+1];
      v1 = gautbl->var[r];
      v2 = gautbl->var[r+1];
      dval1 = gautbl->lrd[r];
      dval2 = gautbl->lrd[r+1];
      ...
      score[r] = (int32)(f * dval1);
      score[r+1] = (int32)(f * dval2);
    }
```

Figure 4.16: `sphinx3`'s hottest loop.

False Sharing Caused by Prefetching

The performance of `sphinx3` on TSX with and without the prefetcher enabled, shown in Figure 4.14, illustrates the occurrence of false sharing due to prefetching [44] (prefetcher is always enabled in the serial executions). Conflict aborts caused by false sharing, see Figure 4.13, leads to slowdowns for three and four threads. Disabling the prefetcher hurts performance because of additional cache misses, but helps performance because of the elimination of the false sharing. For two threads the benefits of prefetching outweigh the cost of the additional aborts. Another example of the problem with prefetching is the hottest loop of the `h264ref` benchmark. In TSX, this benchmark sees a slowdown with the prefetcher enabled. Figure 4.14 shows the speed-ups of `h264ref` after disabling the prefetcher. It achieves speed-ups of 20% with two, three and four threads.

Variable Privatization to Eliminate False dependences

Aborts caused by Write-After-Write (WAW) and Write-After-Read (WAR) dependences can be eliminated when private copies of the global variables that cause these dependences are created for accesses within transactions. For instance, the `h264ref`'s loop, shown in Figure 4.15, has WAW and WAR loop-carried dependences on `PelYline_11`.

If the compiler can prove that the live range of this variable is contained within an iteration, it can create private copies to eliminate the dependence. A WAR loop-carried dependence between the read through the `refptr` pointer (line 13 of Figure 4.15) and a write to the same memory address inside of the `PelYline_11` function call (line 12) is also removed through privatization.

Even though these transformations improve the performance of `h264ref` with TLS, there are no speed-ups with either architecture. POWER8 suffers from limited speculative storage capacity and TSX from conflicts caused by the prefetcher.

False Sharing due to Non-Consecutive Writes

For some benchmarks the major source of conflict aborts is the false sharing because of non-consecutive writes. For instance, the hottest loop of the `lbm` benchmark is shown in

Table 4.3: Privatization Results.

Benchmark Loop	Arch	ss	N'	Speed-up with priv	Problems
`h264ref`	POWER8	6	182	1.10	capacity
`lbm`	TSX	33	39394	0.72	order inversion
`lbm`	POWER8	17	76471	1.30	–

Figure 4.8. This loop has no true dependences for the ref input. However the writes to the `dstGrid` array (as shown in line 3 of Figure 4.8) cause false sharing, and this issue results in conflict aborts in the TLS version. In this case, strip mining cannot remove false sharing because the writes performed are non-consecutive.

The `lbm` benchmark yields slowdowns in all machines as shown Figure 4.12. The abort ratio due to capacity on POWER8 is very large as shown in Figure 4.13. There is also a trade-off between the new trip count (N' in Table 4.2) and the speculative capacity of the transactions because the large value of $N(1300000)$ results in an overhead by HTM instructions (`tbegin`/`tend`). Reducing this overhead requires an increase of ss, but such increase results in a larger pressure in the HTM speculative capacity thus increasing the number of aborts due to capacity. In TSX, false sharing leads to a high conflict abort ratio and prevents it from achieving any speed-up.

Privatization to Remove False Sharing

When array accesses follow a pattern it is possible to use strip mining to remove false sharing. However, a large strip size can result in many capacity aborts. An alternative would be a word-level conflict detection mechanism in HTM [44], which is not supported by current HTMs. Our solution is to privatize arrays within the TLS transaction, and to write non-speculatively to these arrays after committing. For instance, the TLS parallelization of the `h264ref`'s hot loop on POWER8 with a strip size of 32 results in a slowdown. However, after removing false sharing with privatization a smaller strip size ($ss = 6$) can be used to reduce capacity aborts, see Figure 4.18, leading to a speed-up of up to 10% on POWER8 as shown in Figure 4.17. Further reducing the capacity aborts in POWER8 would require a strip size lower than 6, but this results in an increment of N' and thus in a larger overhead due to HTM instructions.

In `lbm`, a hot loop writes to non-consecutive elements of an array. This benchmark's

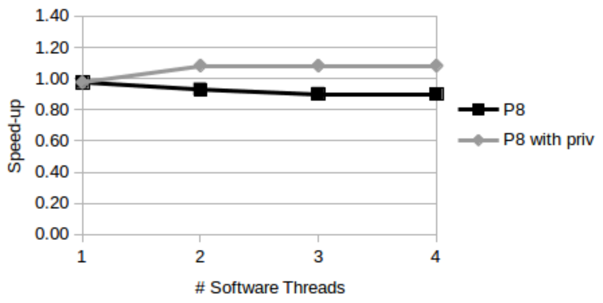


Figure 4.17: Speed-ups for `h264ref` by TLS execution on POWER8 with and without privatization.

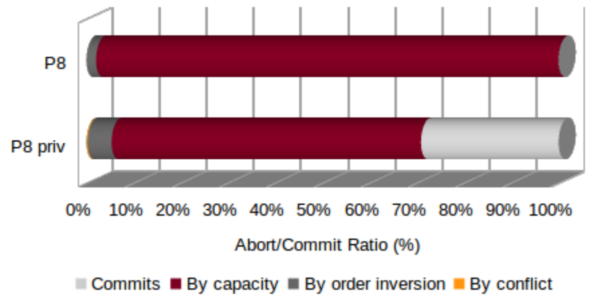


Figure 4.18: Abort/Commit ratio for `h264ref` by TLS execution with 4 threads on POWER8 with and without privatization.

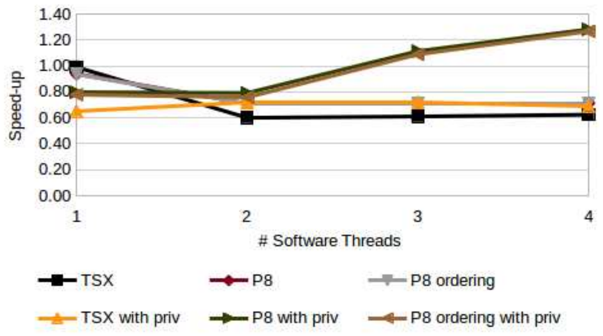


Figure 4.19: Speed-ups for `lbm` by TLS execution on TSX, POWER8, and POWER8 with ordering, with and without privatization.

performance also improves through the removal of false sharing, see Figure 4.20, with privatization using thread-local arrays. As shown in Figure 4.19, `lbm` sees speed-ups of up to 30% using privatization on POWER8 (with and without ordering), but no speed-ups on TSX. Decreasing the strip size from 19 to 17 in POWER8 is beneficial because it reduces the iteration size and thus the capacity aborts.

The abort ratio for `lbm` due to conflicts on POWER8 is reduced as we remove false sharing with privatization. Besides, some aborts due to order inversion appear in POWER8, but they can be removed by using `tsuspend` and `tresume` instructions as shown in Figure 4.20. In TSX, the conflict-abort ratio decreases; however, aborts due to order inversion are still present.

Reducing HTM Overhead

A small iteration size and a large trip count leads to overhead due to the instructions required to start and commit a transaction. For instance, a hot loop in `libquantum` has a trip count of 2097152 and a very small average iteration size (AIS) — see Table 4.2. For TLS to be performant in such cases, it is necessary to use strip mining to decrease the trip count and increase the AIS. In `libquantum`, the hot loop also has false sharing that is not eliminated by neither strip mining nor privatization. This false sharing leads to a large conflict-abort ratio on both architectures as shown in Figure 4.13. There is limit to the increase in the strip size to reduce HTM overhead: each strip should not exceed the

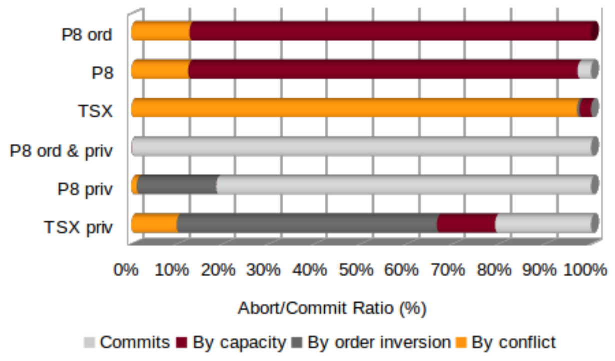


Figure 4.20: Abort/Commit ratio for `lbm` by TLS execution with 4 threads on TSX, POWER8, and POWER8 with ordering, with and without privatization.

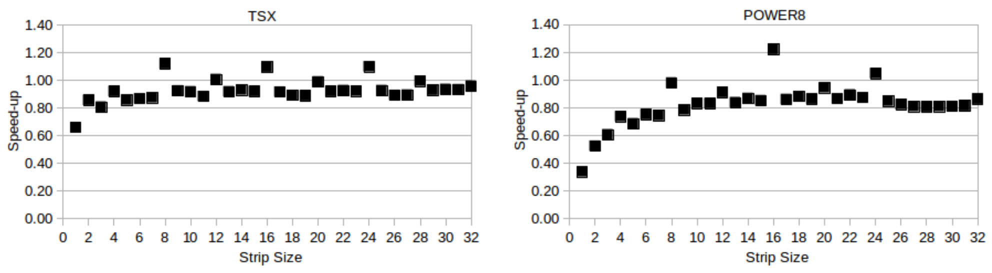


Figure 4.21: Speed-ups for `sphinx3` by TLS execution on TSX and POWER8 for different strip sizes with 2 and 4 threads respectively.

speculative-state capacity of the HTM system. For instance, in TSX, the large strip size used results in some capacity aborts. Nonetheless, `libquantum` still achieves speed-ups of up to 7% on TSX. On POWER8 there is no TLS speed-up because the abort ratio due to conflicts is large, see Figure 4.13.

Actual dependences Lead to Poor Performance

Even a small ratio of actual dependences occurring at runtime prevents TLS from delivering performance improvements. One example is the `astar` benchmark where 10% of the loop iterations have actual dependence for the reference input. These true dependences lead to a large conflict-abort ratio (99%), see Figure 4.13. The consequence is that there is no speed-up on either architecture. We studied the use of different values of strip sizes but none resulted in improved performance for this benchmark. This example underscores the need for a precise dependence-prediction mechanism to be used by a compiler that incorporates TLS in the code generator[5].

Sensitivity to Strip Size

An interesting question is how sensitive the performance results are to the selection of strip sizes. Varying the strip size in several benchmarks indicates that it can have some non-trivial effect in performance. For instance, Figure 4.21 shows the performance variation for `sphinx3` when the strip size is varied in both TSX and POWER8. In this case, strip sizes that are multiples of eight perform best because they mitigate or remove false sharing. However, when the strip size leads a transaction to exceed speculative storage capacity, performance is degraded.[1]

[1]Two threads are used for this sensitivity study with TSX because with a larger number of threads the false-sharing issue caused by prefetching would obfuscate the sensitivity to strip size in this benchmark.

Chapter 5

In-depth Evaluation of TLS in off-the-shelf HTMs

Section 4.2 describes how speculation support designed for HTM can also be used to implement TLS [51]. It also provides a detailed description of the additional software support that is necessary in both the Intel Core and the IBM POWER8 architectures to support TLS. Our work uses software solutions to provide: multi-versioning, a conflict-resolution policy, and ordered transactions. In our solution, ordered transactions are supported in the POWER8 using the `tsuspend`/`tresume` instructions to non-speculatively wait for the iteration commit counter to reach the value of the index variable of the current transaction before committing. The Intel Core does not provide the ability to suspend transactions and to execute instructions non-speculatively and therefore a transaction cannot wait for its turn to commit as in the POWER8. Our solution is to roll back a transaction that completes execution out of order using an explicit abort instruction (`xabort`). This kind of abort is called an *order-inversion* abort [51].

The performance evaluation presented in Section 5.2 uses the method described in Section 4.2 to implement TLS on top of HTM. However, the previous chapter focused on the impact of false sharing and the importance of judicious strip mining to achieve performance. In contrast, this chapter carefully evaluates the performance of TLS on Intel Core and POWER8 using 22 benchmarks from the `cBench` suite focusing on the characterization of the loops. This loop characterization could be used in the future to decide if TLS should be used for a given loop.

5.1 Benchmarks, Methodology and Experimental Setup

The performance assessment reports speed-ups and abort ratios for the coarse-grained TLS parallelization of loops from the Collective Benchmark (`cBench`) benchmark suite [15] running on Intel Core and IBM POWER8. For all experiments the default input is used for the `cBench` benchmarks. The baseline for speed-up comparisons is the serial execution of the same benchmark program compiled at the same optimization level. Loop times are compared to calculate speed-ups. Each software thread is bounded to one hardware thread (core) and executes a determined number of pre-assigned iterations. Each benchmark was

Table 5.1: Loops extracted from cBench applications.

Class	Loop ID	Previous ID	Benchmark	Location	Function	%*Cov*	Invocations
I	A	14	`automotive_bitcount`	`bitcnts.c`,65	`main1`	100%	560
	B	18	`automotive_susan_c`	`susan.c`,1458	`susan_corners`	83%	344080
	C	22	`automotive_susan_e`	`susan.c`,1118	`susan_edges`	18%	165308
	D	24	`automotive_susan_e`	`susan.c`,1057	`susan_edges`	56%	166056
	E	28	`automotive_susan_s`	`susan.c`,725	`susan_smoothing`	100%	22050
	F	15	`automotive_bitcount`	`bitcnts.c`,59	`main1`	100%	80
II	G	19	`automotive_susan_c`	`susan.c`,1457	`susan_corners`	83%	782
	H	23	`automotive_susan_e`	`susan.c`,1117	`susan_edges`	18%	374
	I	25	`automotive_susan_e`	`susan.c`,1056	`susan_edges`	56%	374
	J	29	`automotive_susan_s`	`susan.c`,723	`susan_smoothing`	100%	49
III	K	1	`consumer_jpeg_c`	`jfdcint.c`,154	`jpeg_fdct_islow`	5%	1758848
	L	2	`consumer_jpeg_c`	`jfdcint.c`,219	`jpeg_fdct_islow`	5%	1758848
	M	4	`consumer_jpeg_c`	`jcphuff.c`,488	`encode_mcu_AC_first`	10%	5826184
	N	6	`consumer_jpeg_d`	`jidcint.c`,171	`jpeg_idct_islow`	14%	7280000
	O	7	`consumer_jpeg_d`	`jidcint.c`,276	`jpeg_idct_islow`	15%	7280000
	P	13	`automotive_bitcount`	`bitcnts.c`,96	`bit_shifter`	35%	90000000
	Q	16	`automotive_susan_c`	`susan.c`,1615	`susan_corners`	7%	344080
	R	26	`automotive_susan_s`	`susan.c`,735	`susan_smoothing`	96%	198450000
	S	34	`security_rijndael_d`	`aesxam.c`,209	`decfile`	7%	31864729
	T	3	`consumer_jpeg_c`	`jccolor.c`,148	`rgb_ycc_convert`	10%	439712
	U	5	`consumer_jpeg_c`	`jcphuff.c`,662	`encode_mcu_AC_refine`	17%	5826184
Others	V	17	`automotive_susan_c`	`susan.c`,1614	`susan_corners`	7%	782

run twenty times and the average time is used. Runtime variations were negligible and are not presented.

Loops from `cBench` were instrumented with the necessary code to implement TLS, following the techniques described in Section 4.2. They were then executed using an Intel Core i7-4770 and the IBM POWER8 machines, and their speed-ups measured with respect to sequential execution. Based on the experimental results, the loops studied are placed in four classes that will be explained later. Table 5.1 lists the twenty two loops from `cBench` used in the study. The table shows (1) the loop class (explained later); (2) the ID of the loop in this study; (3) the ID of the loop in the previous study [40]; (4) the benchmark of the loop; (5) the file/line of the target loop in the source code; (6) the function where the loop is located; (7) %*Cov*, the fraction of the total execution time spent in this loop; and (8) the number of invocations of the loop in the whole program.

This study uses an Intel Core i7-4770 processor with 4 cores with 2-way SMT, running at 3.4 GHz, with 16 GB of memory on Ubuntu 14.04.3 LTS (GNU/Linux 3.8.0-29-generic x86_64). The cache-line prefetcher is enabled (by default). Each core has a 32 KB L1 data cache and a 256 KB L2 unified cache. The four cores share an 8 MB L3 cache. The benchmarks are compiled with GCC 4.9.2 at optimization level `-O3` and with the set of flags specified in each benchmark program.

The IBM processor used is a 4-core POWER8 with 8-way SMT running at 3 GHz, with 16 GB of memory on Ubuntu 14.04.5 (GNU/Linux 3.16.0-77-generic ppc64le). Each core has a 64 KB L1 data cache, a 32 KB L1 instruction cache, a 512 KB L2 unified cache, and a 8192 KB L3 unified cache. The benchmarks are compiled with the XL 13.1.1 compiler at optimization level `-O2`.

Table 5.2: Characterization and TLS Execution of Classes.

Class	Loop ID	Loop Characterization								TLS Execution								
		N	Intel's *Tbody*	Intel's *Tloop*	%*lc*	Read Size		Write Size		Privatization	Intel Core			IBM POWER8		Speed-ups in [40]		
			(*ns*)	(*ns*)		avg	max	avg	max		*ss*	Duration (*ns*)	Speed-up	*ss*	Speed-up	C	F	J
I	A	1125000	5.0	5680000	0%	12 B	24 B	0 B	20 B	Reduction	502	2600.0	2.20	502	3.80	14.0	14.3	14.3
	B	590	12.7	7500	0%	48 B	176 B	0 B	36 B	No	59	749.0	1.20	59	1.59	10.2	12.0	12.0
	C	592	8.1	4810	0%	14 B	192 B	0 B	32 B	Array	72	584.0	1.20	68	1.21	7.5	8.0	8.0
	D	594	14.1	8420	0%	76 B	176 B	0 B	28 B	Array	88	1240.0	1.28	72	2.22	13.0	15.0	15.0
	E	600	198.0	118000	0%	14 B	192 B	0 B	32 B	Array	15	2970.0	1.60	15	3.18	14.0	15.0	15.0
	F	7	5840000.0	40800000	0%	48 B	268 B	155 B	604 B	Array	1	5840000.0	0.98	2	2.40	1.0	2.5	2.5
II	G	440	7710.0	3390000	0%	2 KB	3 KB	29 B	328 B	No	1	7710.0	1.23	1	1.15	13.0	15.0	15.0
	H	442	4790.0	2120000	0%	3 KB	8 KB	37 B	260 B	Array	1	4790.0	2.09	2	0.84	12.0	13.8	13.8
	I	444	8680.0	3850000	0%	4 KB	4 KB	206 B	1 KB	Array	2	17300.0	1.76	1	1.05	13.0	15.0	15.0
	J	450	117000.0	52900000	0%	3 KB	8 KB	37 B	260 B	Array	1	117000.0	1.89	1	0.73	0.5	1.0	1.0
III	K	8	8.7	69	0%	16 B	32 B	16 B	32 B	Array	1	8.7	0.07	1	0.03	5.5	6.0	6.0
	L	8	8.5	68	0%	16 B	32 B	16 B	32 B	Array	1	8.5	0.06	1	0.03	5.5	6.0	6.0
	M	38	5.4	205	100%	12 B	68 B	4 B	36 B	Scalar	1	5.4	0.07	1	0.02	0.5	1.0	0.5
	N	8	8.1	65	0%	23 B	64 B	16 B	32 B	Array	1	8.1	0.05	1	0.05	4.0	4.2	4.2
	O	8	9.4	75	0%	24 B	68 B	5 B	16 B	Array	1	9.4	0.07	1	0.05	5.8	6.0	6.0
	P	23	1.1	26	0%	4 B	12 B	4 B	16 B	Reduction	3	3.4	0.02	3	0.02	1.0	2.3	2.3
	Q	590	1.0	567	0.14%	4 B	212 B	0 B	36 B	Scalar	118	113.0	0.46	95	0.49	9.0	8.5	8.5
	R	15	1.8	27	0%	12 B	68 B	4 B	56 B	Reduction	10	18.2	0.05	10	0.04	4.0	4.0	4.0
	S	16	1.3	21	0%	7 B	8 B	4 B	16 B	Array	2	2.6	0.02	2	0.01	1.0	3.0	3.0
	T	162	2.5	404	0%	40 B	44 B	12 B	24 B	Array & Scalar	8	19.9	0.15	30	0.33	11.0	11.0	2.0
	U	63	4.6	289	30%	7 B	8 B	4 B	20 B	Scalar	9	41.4	0.20	10	0.16	10.0	11.0	11.0
Others	V	440	511.0	225000	34%	1 KB	4 KB	20 B	196 B	Scalar	1	511.0	1.25	1	1.34	2.5	2.5	1.0

```
for (i = 0; i < FUNCS; i++) { ///loopF
  for (j = n = 0, seed = 1; j < iterations; j++, seed += 13) //loopA
    n += pBitCntFunc[i](seed);
  if (print)
    printf("%-38s> Bits: %ld\n", text[i], n);
}
```

Figure 5.1: `loopA` and `loopF`.

5.2 Classification of Loops Based on TLS Performance

To understand and explain the experimental results, the `cbench` loops are separated into four classes according to their performance when executing TLS on top of HTM in the POWER8 and in the Intel Core architectures. Loops in each class were then scrutinized to identify common features that may explain their performance characteristics.

The features used to characterize the loops are shown in the first part of Table 5.2: (1) N, the average number of loop iterations; (2) *Tbody*, the average time in nanoseconds of a single iteration of the loop on Intel Core; (3) *Tloop*, $Tbody \times N$; (4) $\%lc$, the percentage of iterations that have loop-carried dependences for the default input; (5) the average (and maximum) size in bytes read/written by an iteration.

TLS has been applied to the loops in each class. The parameters in the right side of Table 5.2 describe TLS execution: (1) the type of privatization within the transaction used in TLS implementation;[1] (2) *ss*, the *strip size* used for the experimental evaluation in Intel Core; (3) Transaction Duration in the Intel Core, which is the product $ss \times Tbody$; (4) the average speed-ups with four threads for Intel Core after applying TLS; (5) the *ss* for POWER8; (6) the speed-ups for POWER8; and (7) the predicted speed-up from TLS emulation reported in [40] for coarse-grained (C), fine-grained (F), and judicious (J) speculation using 16 cores.

For all the loops included in this study $N > 4$, thus they all have enough iterations to be distributed to the four cores in each architecture. When the duration of a loop, *Tloop*, is too short there is not enough work to parallelize and the performance of TLS is low — in the worst case, LoopS, TLS can be 100 times slower than the sequential version. Even a small percentage of loop-carried dependences, $\%lc$, materializing at runtime may have a significant effect on performance depending on the distribution of the loop-carried dependences throughout loop iterations at runtime; thus TLS performance for those loops is difficult to predict. The size of the read/write set in each transaction can also lead to performance degradation because of capacity aborts. For the Intel Core the duration of each transaction is important: rapidly executing many small transactions leads to an increase of order-inversion aborts. The number of such aborts is lowest for medium-sized transactions that have balanced transactions — when the duration of different iterations of the loop varies the number of order-inversion aborts also increases. Finally, long transactions in both architectures may cause aborts due to traps caused by the end of the OS quantum.

[1]A Reduction privatization is a scalar privatization of a reduction operation.

```
for (is = 0; is < FUNCS; is+=STRIP_SIZE) { //loopF
  for (i=is; i-is < STRIP_SIZE && i< FUNCS; i++)
    for (j = n_arr[i] = 0, seed = 1; j < iterations; j++, seed += 13) //loopA
      n_arr[i] += pBitCntFunc[i](seed);

  if (print)
    for (i=is; i-is < STRIP_SIZE && i< FUNCS; i++)
      printf("%-38s> Bits: %ld\n", text[i], n_arr[i]);

}
```

Figure 5.2: `loopF` after applying strip mining and dividing into two components.

5.2.1 Class I: Low speculative demand and better performance in POWER8

The speculative storage requirement of loops in this class is below 2 KB and thus they are amenable for TLS, and see speed-ups, in both architectures. A sufficiently small speculative-storage requirement is more relevant for POWER8 which has smaller speculative-storage capacity (see Table 2.1). These loops also result in better scaling in POWER8, when compared to Intel Core, because they can take advantage of the `suspend` and `resume` instructions of POWER8 to implement ordered transactions in software. They do not scale much beyond two threads on Intel Core due to the lack of ordered transactions support.

Table 5.2 shows the characterization of Class I. These loops typically provide a sufficient number of iterations to enable their distribution among the threads. They also have a relatively moderate duration, as shown by the *Tloop* values, and thus they have enough work to be parallelized. TLS makes most sense when the compiler cannot prove that iterations are independent, but dependences do not occur at runtime, therefore most loops that are amenable for TLS (loops in Class I and II) have $\%lc$ of zero.

A typical example of a loop in Class I is `loopA`, shown in Figure 5.1. This loop achieves speed-ups of up to 3.8× with four threads. This loop calls the same bit-counting function with different inputs for each iteration `j`. The loopA is the inner `j` loop, which calls the same `pBitCntFunc[i]`, with different input, in every iteration. Even though `loopA` has *may* loop-carried dependences inside the functions called, none of these dependences materialize at runtime. Thus, a successful technique to parallelize this loop consists in privatizing variable `n` within the transaction and adding the partial result to a global variable after the transaction commits. The successful parallelization of `loopA` stems from a moderate duration (*Tloop*), no actual runtime dependences, and a read/write set size that is supported by the HTM speculative-storage capacity. The large number of iterations of this loop allows increasing the strip size (*ss*), and thus the new *Tbody* (after strip mining) — $ss \times Tbody$ — is longer; after that, order-inversion aborts decrease (`loopB` has more order-inversion aborts than `loopA`, although its *Tbody* is longer).

For most of the loops in this class — `LoopF` is an exception discussed later — the performance is directly related to the effective work to be parallelized, represented by *Tloop*. In the Intel Core the proportion of order-inversion aborts is inversely related to the transaction duration because very short transactions may reach the commit point even

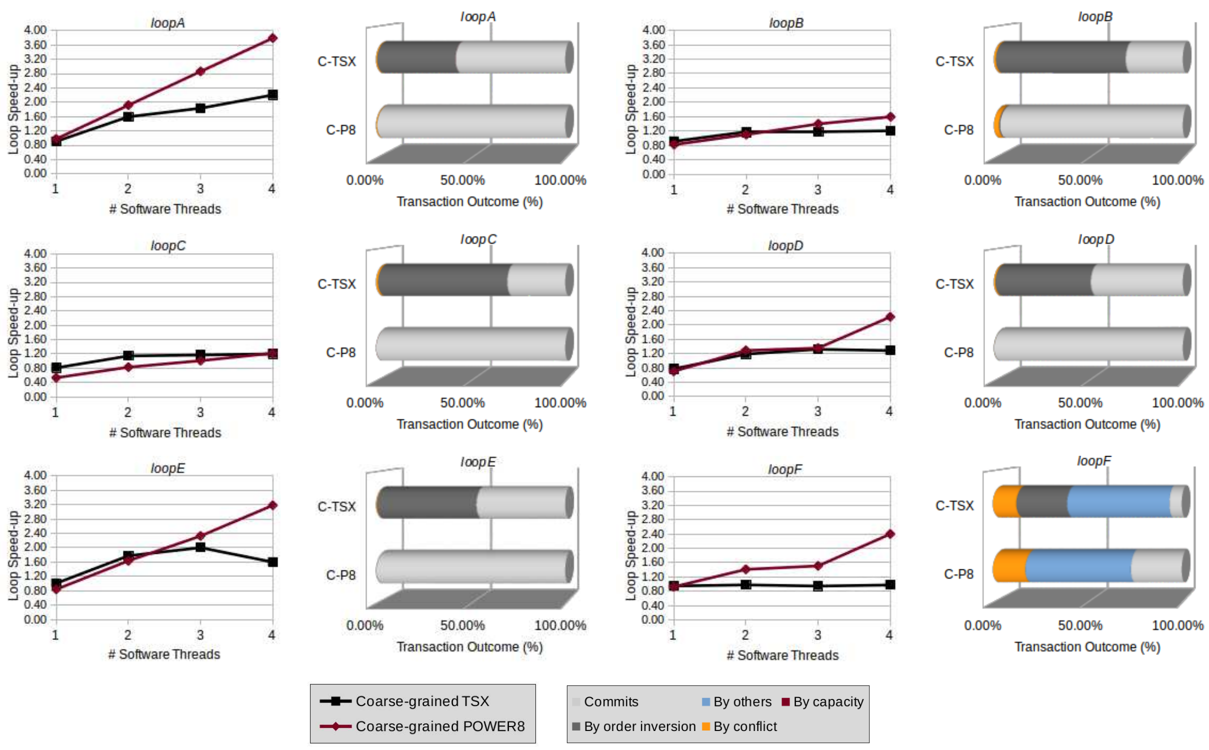


Figure 5.3: Class I. Speed-ups and Abort ratios for coarse-grained TLS execution on TSX and POWER8.

before previous iterations could commit. Another issue is that very long transactions may abort due to traps caused by the end of OS quantum.

The performance of `loopC` from one to three threads is higher on Intel Core than on POWER8 because the larger speculative store capacity in the Intel Core allows for the use of a larger strip size. With four threads, there is a small improvement in POWER8 due to the reduction of order-inversion aborts. The increment in the number of threads intensifies the effect of order inversion in performance. Therefore, for machines with a higher number of cores, better speed-ups should be achieved in POWER8 than in Intel Core.

In `loopC`, `loopD`, and `loopE` consecutive iterations write to consecutive memory positions leading to false sharing when these iterations are executed in parallel in a round-robin fashion. For instance, `loopE`, shown in Figure 5.4, writes to `*out++` (consecutive memory positions) in consecutive iterations generating false sharing in a round-robin parallelization. The solution is privatization: write instead into local arrays during all the transaction and copy the values back to the original arrays after commit [51].

Each iteration `i` of `loopF` (shown in Figure 5.1) executes `loopA` invoking a different bit-counting function for each `i` with various inputs. The sum of the return values is accumulated in `n`. No loop-carried dependences materialize for the standard input ($\%lc = 0$), there is enough work in all iterations of the loop ($Tloop > 4\mu s$) and its read/write size does not exceed the speculative storage capacity of POWER8. The inner loop of `loopF`, `loopA`, executes 1125000 iterations. Hence, `loopF` has the longest $ss \times Tbody$ among all loops evaluated and thus many transactions abort due to traps caused by the end of

```
for (i=mask_size;i<y_size-mask_size;i++){ //loopJ
  for (j=mask_size;j<x_size-mask_size;j++){ //loopE
    area = 0;
    total = 0;
    dpt = dp;
    ip = in + ((i-mask_size)*x_size) + j - mask_size;
    centre = in[i*x_size+j];
    cp = bp + centre;
    for(y=-mask_size; y<=mask_size; y++){
      for(x=-mask_size; x<=mask_size; x++){ //loopR
        brightness = *ip++;
        tmp = *dpt++ * *(cp-brightness);
        area += tmp;
        total += tmp * brightness;
      }
      ip += increment;
    }
    tmp = area-10000;
    if (tmp==0)
      *out++=median(in,i,j,x_size);
    else
      *out++=((total-(centre*10000))/tmp);
  }
}
```

Figure 5.4: `loopE`, `loopJ`, and `loopR`.

the OS quantum, which explains this loop showing a high abort ratio by *other causes* in Figure 5.3.

Whole Coarse-grained TLS parallelization of `loopF` is not possible because each iteration has a `printf` statement that is not allowed within a transaction in either architecture. Therefore, each iteration of `loopF` must be divided into two components: `loopA` and the `printf` (as shown in Figure 5.2), before applying TLS only to the first component. The second component is always executed non-speculatively. Only POWER8 can deliver speed-ups for this loop because aborts by order inversion are eliminated through the use of suspend/resume.

5.2.2 Class II: High speculative demand and better performance in Intel Core

These loops can scale better in the Intel Core compared to the POWER8 because of the larger transaction capacity of the Intel Core: the read/write sizes of these loops overflow the transaction capacity of the POWER8 (see Table 2.1) leading to a high number of capacity aborts.

Table 5.2 shows the characterization of loops in Class II. With more than 400 iterations and a loop execution time *Tloop* larger than 2 *ms* these loops have enough work to be parallelized. Also, no dependences materialize at runtime for the default inputs ($\%lc = 0$).

The smaller write size in `loopG` means that 50% of its transactions do not overflow the POWER8 speculative-storage capacity resulting in this loop showing speed-ups of

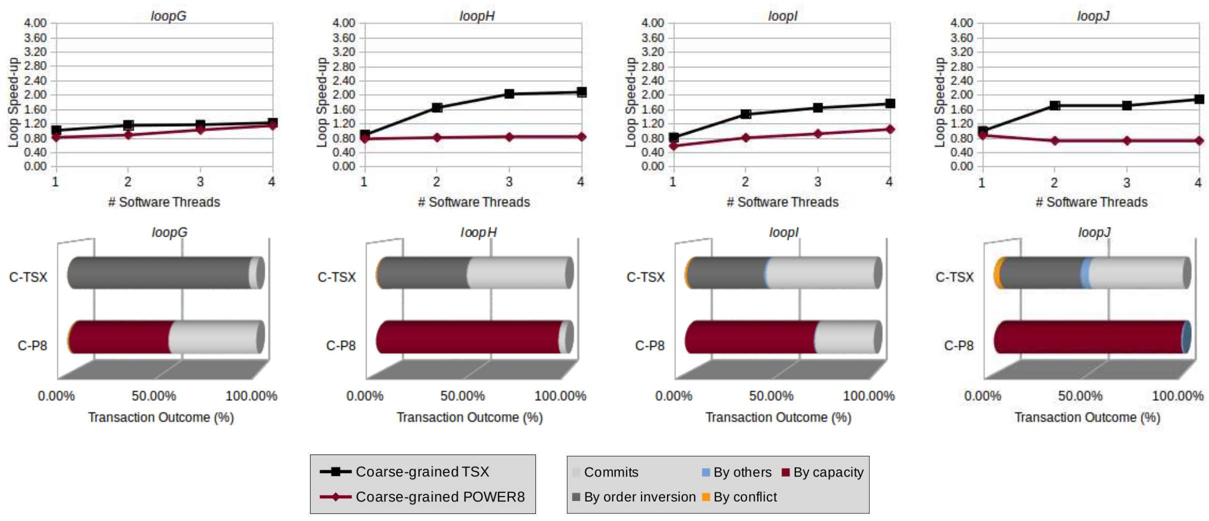


Figure 5.5: Class II. Speed-ups and Abort ratios for coarse-grained TLS execution on TSX and POWER8.

up to 15% with four threads on POWER8. With $ss = 1$, this loop's transaction takes 7.7 μs (medium-duration) in the Intel Core. However, it has an elevated proportion of order-inversion aborts. As explained in [40], `loopG` has significant imbalance between its iterations and this aggravates order inversion in the Intel Core. A contrast is `loopH` that has better performance in the Intel Core, as shown in Table 5.2, even though its transactions are even shorter, lasting almost 5 μs. `loopH` results in much fewer order-inversion aborts because the durations of its transactions are moderate and balanced across the iterations.

In class II, loops `loopJ` and `loopH` suffer from false sharing and require array privatization. For instance, in `loopJ`, shown in Figure 5.4, the auto-increment of the pointer `out` in lines 20 and 22 leads to false sharing. This loop has the second longest transaction duration between all loops evaluated, thus some aborts due to OS traps appear. False sharing can also be removed from `loopI` through privatization. With many aborts due to capacity overflow, `loopI` speed-up in POWER8 is limited to 1.05×. In the Intel Core this loop achieves speed-ups of up to 1.76× because, spending less than 20 μs executing each transaction (medium transaction duration), `loopI` suffers fewer aborts due to order inversion.

5.2.3 Class III: Not enough work to be parallelized with TLS

These are loops where TLS implementation does not have enough work to be distributed among the available threads resulting in poor performance in any architecture. The overhead of setting up TLS for these loops is too high in comparison to the benefits of parallelization. Murphy *et al.* [40] reported speed-ups in these loops because their emulation of TLS hardware did not take into consideration these costs. The experiments in this section reveal that their emulated numbers overestimate the potential benefit of TLS for these loops. As shown in Table 5.2 the available work to be parallelized, *Tloop* in

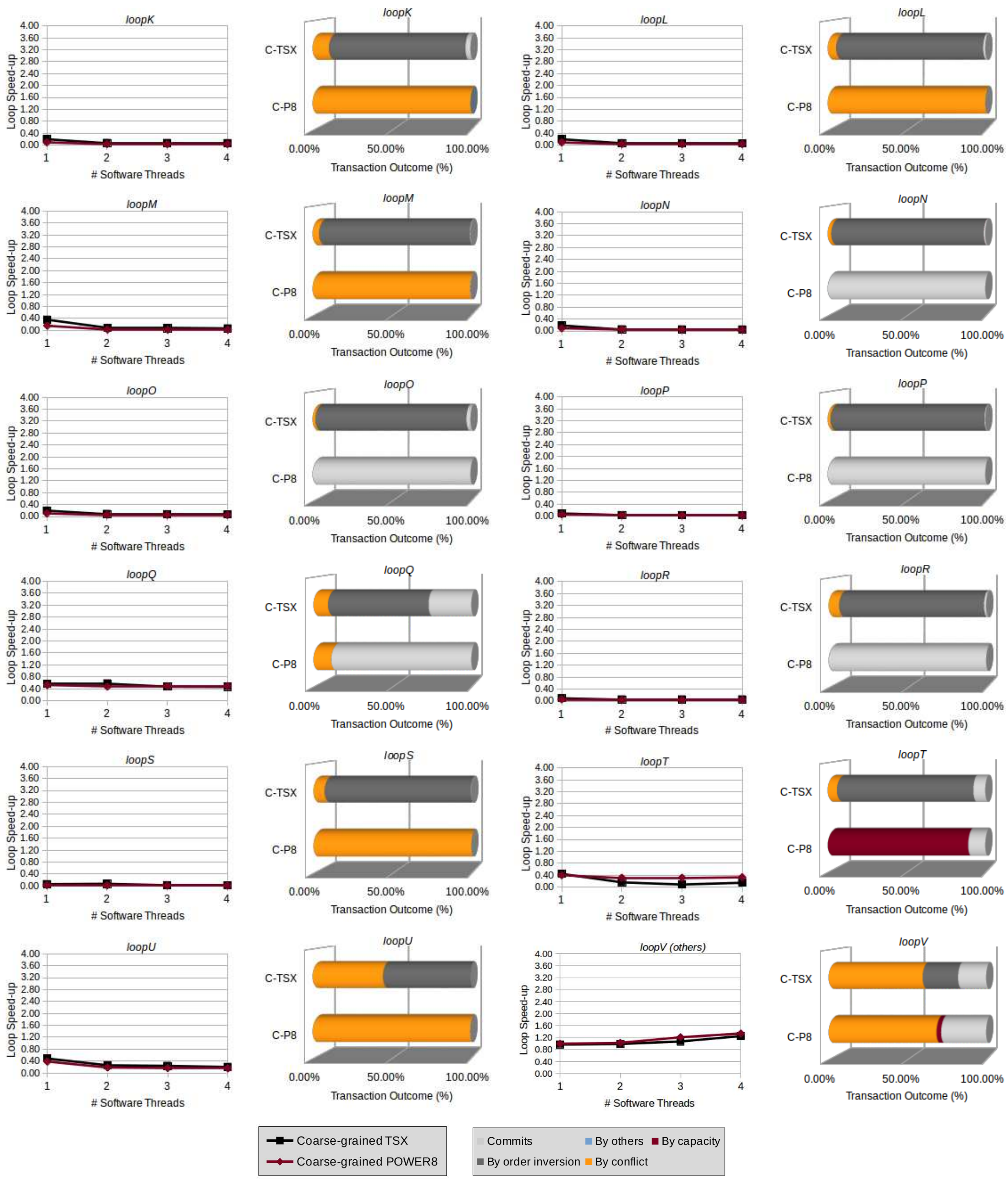


Figure 5.6: Class III and Others. Speed-ups and abort ratios for coarse-grained TLS execution on TSX and POWER8.

```
n=0;
for (i=5;i<y_size-5;i++){ /* Loop V */
  for (j=5;j<x_size-5;j++) { /* Loop Q */
    x = r[i][j];
    if (x>0) {
      if (/* Abbreviated: compare x to each pixel in window*/){
        corner_list[n].info=0;
        corner_list[n].x=j;
        corner_list[n].y=i;
        corner_list[n].dx=cgx[i][j];
        corner_list[n].dy=cgy[i][j];
        corner_list[n].I=in[i][j];
        n++;
        if(n==MAX_CORNERS){
          fprintf(stderr,"Too many corners.\n");
          exit(1);
        }
      }
    }
  }
}
```

Figure 5.7: `loopQ` and `loopV`.

```
for (i = n = 0; x && (i < (sizeof(long) * CHAR_BIT)); ++i, x >>= 1)
  n += (int)(x & 1L);
```

```
for(i = 0; i < 16; ++i) /*xor it with previous input block */
  outbuf[i] ^= bp2[i];
```

Figure 5.8: `loopP` and `loopS`.

```c
for (col = 0; col < num_cols; col++) {
  r = GETJSAMPLE(inptr[RGB_RED]);
  g = GETJSAMPLE(inptr[RGB_GREEN]);
  b = GETJSAMPLE(inptr[RGB_BLUE]);
  inptr += RGB_PIXELSIZE;
  outptr0[col] = ... ;
  outptr1[col] = ... ;
  outptr2[col] = ... ;
}
```

```c
for (k = cinfo->Ss; k <= Se; k++) {
  if ((temp = absvalues[k]) == 0) {
    r++;
    continue;
  }
  while (r > 15 && k <= EOB) {
    ...
    emit_buffered_bits(entropy, BR_buffer, BR);
    BR_buffer = entropy->bit_buffer;
    BR = 0;
  }

  if (temp > 1) {
    BR_buffer[BR++] = (char) (temp & 1);
    continue;
  }
  ...
  emit_buffered_bits(entropy, BR_buffer, BR);
  BR_buffer = entropy->bit_buffer;
  BR = 0;
  r = 0;
}
```

Figure 5.9: `loopT` and `loopU`.

all the loops in this class is below 0.6 μs, which is too small to benefit from parallelization.

For instance, `loopP`, shown in Figure 5.8, has a loop-carried dependence in variable `n` thus coarse-grained TLS implementation would lead to a high conflict abort ratio. The privatization of variable `n` and its initialization to zero for each transaction decreases conflicts. A similar situation occurs for variables `area` and `tmp` in `loopR`. Both loops and `loopO` have no aborts in POWER8, but their performance is poor because of the overhead of setting TLS up. Conflicts due to actual loop-carried dependences in `loopQ` and `loopM` cannot be removed by privatization.

Most of the loops in this category have many order-inversion aborts in Intel Core because their transaction duration is below 120 *ns* leading to a fast end of the transactions/iterations probably even before previous iterations could commit.

Use of TLS in some of the small loops in `cBench` is constrained by several factors. For instance, Figure 5.8 shows `loopS`, which presents false sharing in an array of chars (`outbuf`). This false sharing cannot be overcome with strip mining because the loop executes only 16 iterations and it would be necessary to group 64 consecutive iterations in the same thread to avoid false sharing in TSX. The position `i` of array `outbuf` is read at each iteration; thus privatization is not a solution because there will always be conflicts between the transactional reads and non-transactional writes located in the same cache line; besides, its short *Tbody* increases the conflict ratio and the small number of iterations does not permit *ss* to be larger. Conflict abort ratio is higher in POWER8 due to its cache-line size. `loopK` and `loopL` present the same issues.

Figure 5.9 shows `loopT` which dereferences three pointers to arrays (`outptr0`, `outptr1`, and `outptr2`). These pointers always point to different parts of a dynamically-allocated memory region. The false sharing in the access to these three arrays can be removed with privatization to reduce conflict aborts. The scalar variable `intptr` also must be privatized to avoid conflict aborts. This loop presents a high order-inversion abort rate in Intel Core because its transactions last less than 20 *ns*. In POWER8, this kind of abort disappears; however, the strip size needed to increase the loop body and the privatization of three arrays lead to aborts because the speculative capacity of the HTM is exceeded.

In `loopU` (shown in Figure 5.9), privatization is used to mitigate the impact of loop-carried dependences in variables `BR`, `r`, and `BR_buffer`; however, the performance is still poor due to false sharing. Privatization of the array `BR_buffer` would be impractical because it would require the creation of a local copy of the array for each thread and for each transaction. The high percentage of conflict abortions shown in Figure 5.6 is due to this false sharing.

5.2.4 Others

`loopV` could belong to Class I due to its *Tloop* and read/write size, but it has a substantial *%lc*. This loop is a special case because although it has 34% of probability of loop-carried dependences, TLS can still deliver some performance improvement. As explained in [40], this loop finds local maxima in a sliding window, with each maximum being added to a list of corners, each iteration of `loopQ` processes a single pixel whereas a complete row is processed by each iteration of `loopV`. The input of this loop is a sparse image with

Table 5.3: Characterization of 6 loops from SPEC CPU 2006.

Loop ID	Benchmark	Location	%*Cov*	*N*	*Tbody* (*ns*)	*Tloop* (*ns*)	%*lc*	Iteration Size	Class
`mcf`	`429.mcf`	`pbeampp.c`,165	40%	300	20	6000	3%	300 B	Others
`milc`	`433.milc`	`quark_stuff.c`,1523	20%	160000	94	15000000	0%	1 KB	I
`h264ref`	`464.h264ref`	`mv-search.c`,394	36%	1089	156	170000	0%	6 KB	II
`sphinx3`	`482.sphinx3`	`vector.c`,513	37%	2048	29	60000	0%	1 KB	I
`astar`	`473.astar`	`Way2_.cpp`,100	60%	1234	41	50000	20%	1 KB	Others
`lbm`	`470.lbm`	`lbm.c`,186	99%	1300000	55	71000000	0%	500 B	I

Table 5.4: TLS Execution for 6 loops from SPEC CPU 2006.

Loop	*ss*		Intel	Speed-up	
ID	Intel	P8	Tx Duration (*ns*)	Intel	P8
`mcf`	20	48	400	1.45	0.60
`milc`	4	4	375	1.44	1.50
`h264ref`	16	6	2490	1.74	1.27
`sphinx3`	8	16	234	1.16	1.95
`astar`	128	256	5180	0.74	0.49
`lbm`	33	17	1800	0.69	1.30

most of the pixels set to zero, and the suspected corners (iterations with loop-carried dependences) are processed close to each other.

5.2.5 Predicting the TLS Performance for Other Loops

Besides providing a detailed analysis for the implementation of TLS over current commodity HTM implementations for loops in the `cBench` suite, the characterization of the loops given in Table 5.2 and the performance evaluation presented in the various graphs could also be used to predict the potential benefit of applying TLS for new loops that were not included in this study. For loops with short *Tloop*, such as those in class III, TLS is very unlikely to result in performance improvements in either architecture. For loops with small read/write sets and no dependences materializing at runtime, such as those in class I, TLS is likely to result in modest improvement for the Intel Core and more significant improvements for the POWER8. Loops that have sufficient work to be parallelized and no actual dependences but have larger read/write sets, such as those in Class II, are likely to deliver speed improvements in the Intel Core but will result in little or no performance gains in the POWER8 because of the more limited speculative capacity in this architecture. Finally, loops that have sufficient work to be parallelized but whose dependences materialize at runtime are difficult to predict — such as `loopV`. The distribution of loop-carried dependences among the iterations of such loops must be studied.

Six loops from the SPEC CPU 2006 suite are characterized to determine to which class they belong according to the classification resulting from this experimental evaluation (as shown in Table 5.3). Loops `milc`, `sphinx3`, and `lbm` are classified as Class I; `h264ref` as Class II; and `mcf` and `astar` as Others. Based on this classification a prediction can be made about the relative performance of the loops on TLS over HTM for both architectures. Results of coarse-grained TLS parallelization of these loops shown in Table 5.4 and Figure 5.10 confirm the predictions.

5.3 Fine-grained TLS on top of HTM

Murphy *et al.* proposed fine-grained TLS and described an implementation on emulated hardware for speculative execution [40]. They conclude that the fine-grained approach is most profitable and close to the limit of thread-level parallelism, and that judicious speculation can be unpractical because it relies on complex profilers.

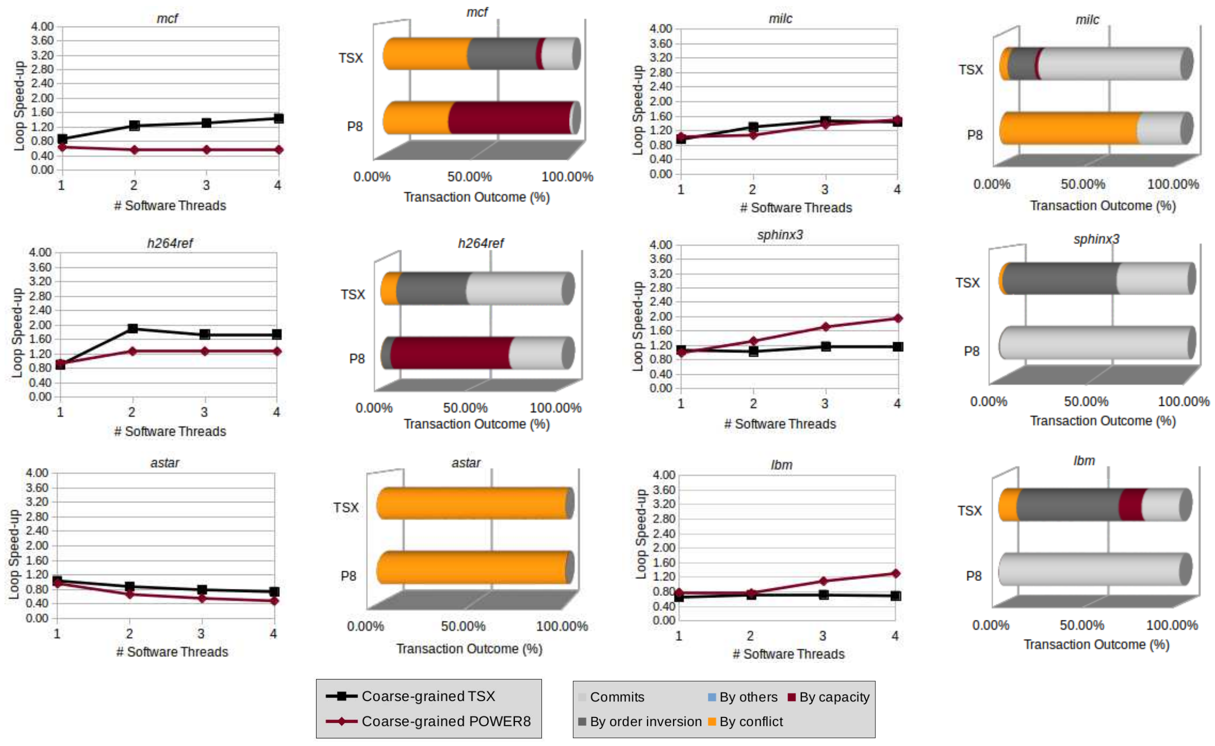


Figure 5.10: SPEC2006 Loops. Speed-ups and abort ratios for coarse-grained TLS execution on TSX and POWER8.

```
for (i = 0; i < N; i++){
  /* Start sequential segment 1 */ /* Global scalar, glob */
  if (cond)
    glob++;
  else
    glob=i;
  /* End sequential segment 1 */
  A[i]= glob*i;
  /* Start sequential segment 2 */
  for(j = 0; j < factor; j++){
    /* Global array, B */
    int tmp = B[factor*(i%4) + j];
    tmp += i*5;
    if(tmp%2 == 0){
      B[factor*(i%4) + j] = tmp;
    }
  }
  /* End sequential segment 2 */
}
```

Figure 5.11: Figure 2.1. A loop with two *may* loop-carried dependences.

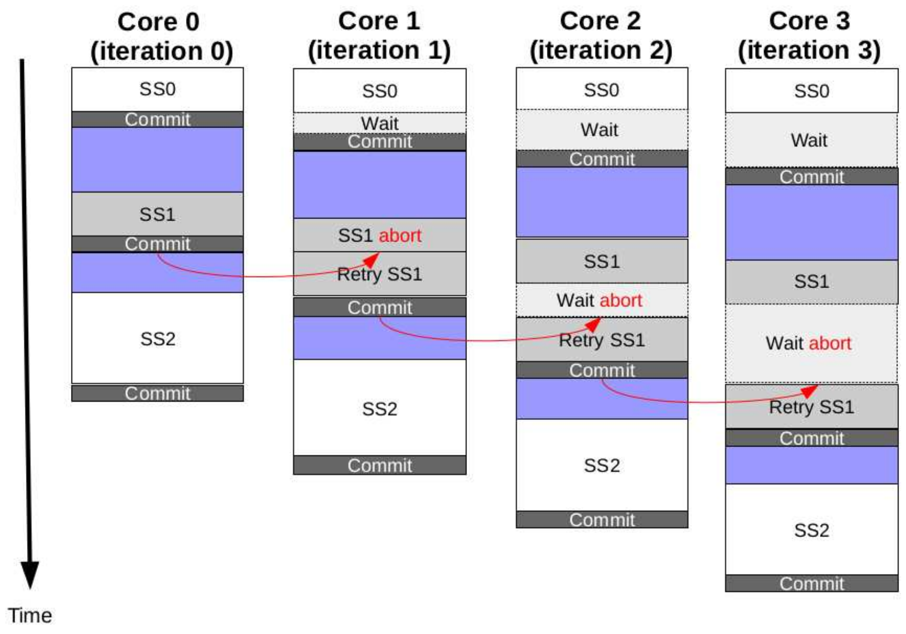


Figure 5.12: Fine-grained Speculation Execution flow of Figure 5.11.

```
    arc = arcs + group_pos;
    for( ; arc < stop_arcs; arc += nr_group) ){
1)       if( arc->ident > BASIC ){
2)           red_cost = arc->cost - arc->tail->potential
             + arc->head->potential;
3)           if( bea_is_dual_infeasible( arc, red_cost )){
4)               basket_size++;
5)               perm[basket_size]->a = arc;
6)               perm[basket_size]->cost = red_cost;
7)               perm[basket_size]->abs_cost = ABS(red_cost);
             }
         }
     }
```

Figure 5.13: `mcf`'s hottest loop.

```
   for(... ){
A)          cond=arc->ident > BASIC
            if (cond)
B)             red_cost = arc->cost-arc->tail->potential + arc->head->potential;
            if (cond)
C)             cond2= bea_is_dual_infeasible(arc,red_cost);
            tbegin();
            if (cond && cond2)
D)             basket_size++;
            tsuspend(); while (arc!=next_iter_commit); tresume();
            tend();
            if (cond && cond2 ){
E)             perm[basket_size]->a = arc;
F)             perm[basket_size]->cost = red_cost;
G)             perm[basket_size]->abs_cost = ABS(red_cost);
            }
   }
```

Figure 5.14: Fine-grained TLS without strip mining for `mcf`'s hottest loop.

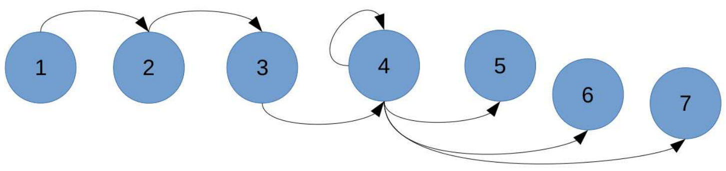


Figure 5.15: DDG of `mcf`'s hottest loop.

The goal of their fine-grained approach is to create transactions that surround only segments of a loop iteration instead of a whole iteration. To accomplish that they use sequential segments of HELIX to define the beginning and the end of transactions. Fine-grained TLS decreases the overhead of speculating a whole iteration in comparison with coarse-grained speculation and avoids capacity aborts because not all reads and writes of an iteration are performed within the same transaction. Besides, in the case of a conflict only a sequential segment is rolled-back and retried (not the whole iteration). However, the HTM overhead may increase because, with multiple transactions per iteration, more transactions are started and finished. The flow of execution of the code of Figure 5.11 for fine-grained speculation is shown in Figure 5.12: each transaction commits in order, and waits for the younger iteration/transaction if that is not ready. SS0 has no conflicts. Assuming that `cond` evaluates to true in all iterations, SS1 always conflicts, rolls back and retries. Finally, SS2 has no conflicts because only four cores are used.

Murphy *et al.*'s implementation of this approach surrounds sequential segments within transactions and distributes iterations to cores in a round-robin fashion. Hence they do not use techniques — such as strip mining or loop-unrolling — to group iterations. This work shows that strip mining is a code transformation that allows decreasing overhead of starting/finishing transactions, aborts, and false sharing when coarse-grained TLS is used with off-the-shelf speculative support. Thus, the implementation of fine-grained TLS on top of existing HTMs discussed in this section uses strip mining.

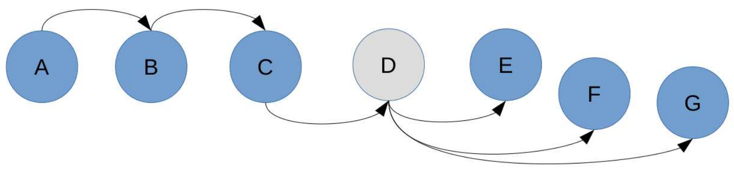


Figure 5.16: SCCs of the DDG of Figure 5.15.

Figure 5.13 shows the serial code for the hottest loop of `mcf`. As explained earlier, to implement fine-grained TLS, it is necessary to build the Data Dependence Graph (DDG) of the code and to find the Strongly Connected Components (SCCs) of the graph (Figure 5.16). Each SCC with (may) loop-carried dependences is considered a sequential segment, whereas each SCC without loop-carried dependences is considered a parallel segment. Only sequential segments are speculated using TLS. For example, in the case of `mcf` hottest loop, the component D is a sequential segment. Figure 5.14 shows the code of fine-grained TLS implementation (like in [40]) of the loop in Figure 5.13 (some details are omitted). Contrary to the result in [40], this loop has a poor performance when is executed on a commercial architecture with support for HTM. As discussed before, this poor performance is due to the lack of strip mining.

To compensate for that, we propose using strip mining to implement fine-grained TLS. If fine-grained TLS is tried in a loop that is not restructured after strip mining, it is not possible to use TLS in small segments because the whole inner loop, that resulted of applying strip mining, defines only one sequential segment. To implement fine-grained TLS with strip mining successfully, it is necessary to restructure the loop using well-known code-transformation techniques as loop fission and scalar expansion.

Loop fission is used to separate each SCC in a loop iterating $STRIP\_SIZE$ times. Each one of these loops can be considered a *stage*. If scalar variables need be communicated between stages, scalar expansion is used. Thus, thread-local buffers are created to store dependence variables for each iteration of a producer stage. The result of this implementation is shown in Fig. 5.17. Stages A, B, and C (corresponding to the same name of SCC respectively) are merged because they are parallel segments and do not need to be speculated. Stage D is speculated and (may) loop-carried dependences of the same stage in different threads are detected and resolved by HTM conflict detection and resolution as explained in Section 4.2. Ordered transactions has to be implemented for each speculative stage because different sequential segments can be executed at the same time. Speed-ups achieved by this technique, coarse-grained TLS, DOACROSS parallelization [16], and TLP-limit in the `mcf`'s hottest loop (SPEC 2006 suite) and `loopV` (`cBench` suite) on Intel Core using four cores are shown in Fig. 5.18. TLP-limit is the implementation of DOACROSS with strip mining but only synchronizing when it is necessary (perfect synchronization) and not at all iterations.

Fine-grained speculation can have good performance when the number of parallel segments is large with respect to the number of sequential segments, and there are a few sequential segments, thus this technique completely depends on the accuracy of static dependence analyzer of the compiler. For instance, `loopE` and `loopJ` have only one

```
  for(...){
     prev_arc=arc;
     for(i=0;i<STRIP_SIZE;i++,arc+=nr_group){
A)        cond_arr[i]=arc->ident > BASIC
          if (cond_arr[i]){
B)            red_cost[i] = arc->cost - arc->tail->potential + arc->head->potential;
C)            cond2_arr[i]= bea_is_dual_infeasible( arc, red_cost[i] ) ;
          }
     }
     tbegin();
     for(i=0,arc=prev_arc;i<STRIP_SIZE;i++,arc+=nr_group){
          if( cond_arr[i] && cond2_arr[i] )
              if (!flag){ basket_sizeL=basket_size; flag=1; }
D)            basket_arr[i]=++basket_sizeL;
     }
     tsuspend(); while (arc!=next_iter_commit); tresume();
     tend();
     if (flag) basket_size=basket_sizeL;
     next_iter_commit=prev_arc+nr_group*STRIP_SIZE;
     for(i=0,arc=prev_arc;i<STRIP_SIZE;i++,arc++){
          if( cond_arr[i] && cond2_arr[i] ){
E)            perm[basket_arr[i]]->a = arc;
F)            perm[basket_arr[i]]->cost = red_cost[i];
G)            perm[basket_arr[i]]->abs_cost = ABS(red_cost[i])
          }
     }
  }
```

Figure 5.17: Fine-grained TLS with strip mining in restructured `mcf`'s hottest loop. Privatization in `basket_size` is shown.

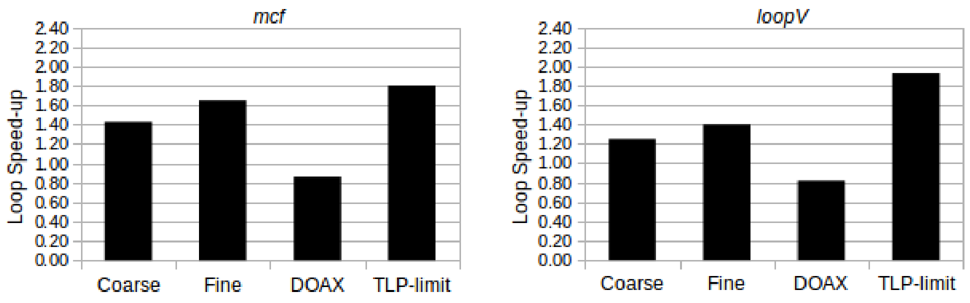


Figure 5.18: Speed-ups in `mcf`'s hottest loop and `loopV` using fine-grained TLS on Intel Core.

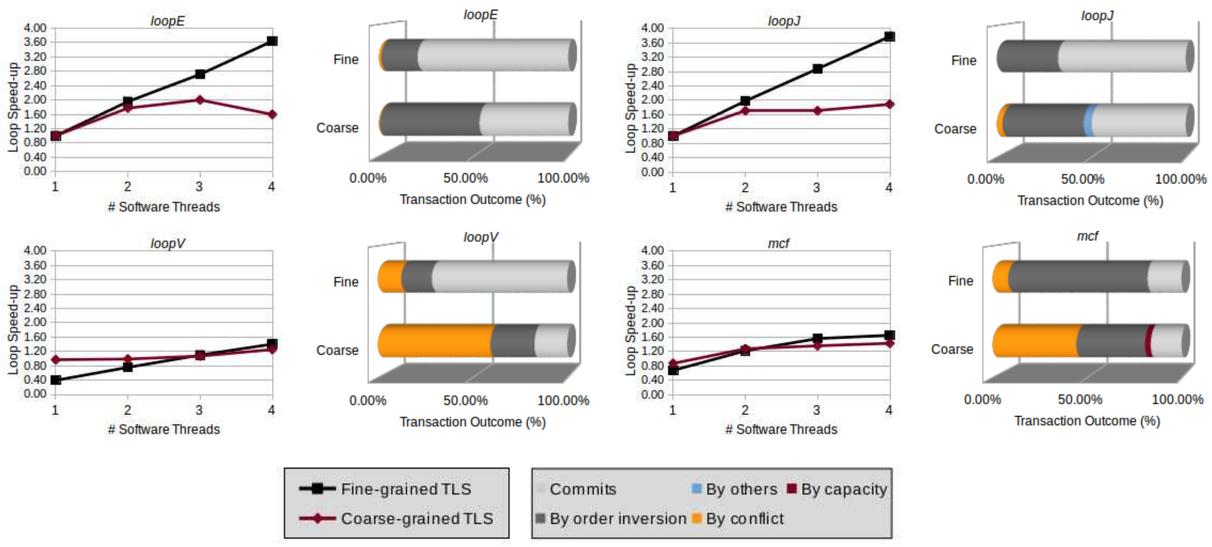


Figure 5.19: Speed-ups and Abort ratios for fine-grained TLS execution on Intel Core.

sequential segment induced by non-actual loop-carried dependences, thus both loops are performant with four threads as shown in Fig. 5.19. The other two loops have actual loop-carried dependences that are transient, thus some improvements are still achieved by fine-grained TLS. For all cases the percentage of aborts decreases.

Chapter 6

Using HTM to Enable STO

To explain Speculative Trace Optimization (STO), consider the code of a simple `for` loop in Figure 6.1. Figure 6.2 shows the four possible traces that could be executed at each iteration of the loop, namely A, B, C and D. The loop has two consecutive `if` statements with conditional expressions that cannot be resolved until runtime and may depend on calculations performed earlier in the traces. In that case, it is not possible to select one amongst multiple traces before executing the traces. The instructions in lines 3-5 are dead code and can be eliminated when *Trace A* is executed because of the redefinition of `z` without prior use in line 18. Similarly, when *Trace B* is executed, the instructions in lines 9-11 are dead code because of the definition of `r` in line 24. Although some compilers may attempt to apply partial dead-code elimination [6] to this code, in general such techniques are not successful or come at high cost. The creation of longer traces of execution with STO enables many known compiler optimizations [27]. The simple program in this example is meant as a motivation for the ideas behind STO.

The central idea of this chapter is that hardware support for speculation, created originally to support HTM and Thread-Level Speculation, can also support STO. One current implementation of HTM support is provided through Intel *Transactional Synchronization Extensions* (TSX). In this prototype implementation of STO, a *pool of threads* is used, and each trace is executed by a thread for each loop iteration. To implement STO, the source code of the benchmarks was modified to insert TSX code, to enable the speculative execution of traces as transactions, and to insert additional features, such as waiting and variable privatization, that compensate for the absence of multi-versioning and lazy-conflict resolution (for trace synchronization) in TSX.

6.1 Speculative Trace Optimization Supported by HTM

This section describes the main ideas behind STO and how to implement it on top of an HTM architecture. The unit of speculation is a *trace.* When the speculation in STO is supported by HTM, each trace is executed as a *transaction.*

```
1        for (i=0; i<n; i++) {
2               if (cond1(z,r)){//condition calculated with z and r
3                      z=z+y;
4                      z=z*2;
5                      z=y+z;
6                      p=y*2;
7               }
8               else{
9                      r=r+q;
10                     r=r*2;
11                     r=q+r;
12                     x=q*2;
13              }
14              if (cond2(x,p)){//condition calculated with x and p
15                     q=r-p;  //z is not used here
16                     p=p*q;
17                     p=p+1;
18                     z=p;
19              }
20              else {
21                     y=z-x;  //r is not used here
22                     x=x*y;
23                     x=x+1;
24                     r=x;
25              }
26       }
```

Figure 6.1: Example of code to optimize.

6.1.1 STO on Ideal HTM

This prototype evaluation focuses on the use of STO to speculate traces found in frequently executed loops. However, STO can be also applied to other hot code regions such as frequently executed functions.

Figure 6.2 shows the possible traces of execution in the body of the `for` loop shown in Figure 6.1. In this example, if Trace A is executed, lines 3-5 of the code shown in Figure 6.1 are dead and can be eliminated. Similarly, if Trace B is executed, lines 9-11 are dead and can be also eliminated. However, without executing these traces, the value of the conditions are unknown and a compiler must preserve the full path in both cases.

The algorithm that leads to STO is described in Algorithm 6.1, using the code in Figure 6.1 as a guideline.

Figure 6.3 shows each trace of Figure 6.2 as a transaction enclosed by the `begin` and `end` instructions of an ideal HTM system. Figure 6.5 shows an example of an execution sequence of the loop of Figure 6.1 using STO. In this example, each trace of each loop iteration is executed in a single transaction by a thread. This ideal system has four hardware threads and an ideal HTM, which has a negligible abort overhead and the following features: eager conflict detection and lazy conflict resolution (Eager-Lazy HTM [59, 53]), multi-versioned cache memory addresses, ability to *pause* a transaction to *wait* for another commiting one (`wait` instruction), and large speculative capacity. For the sake of the this example, assume that there is no false sharing.

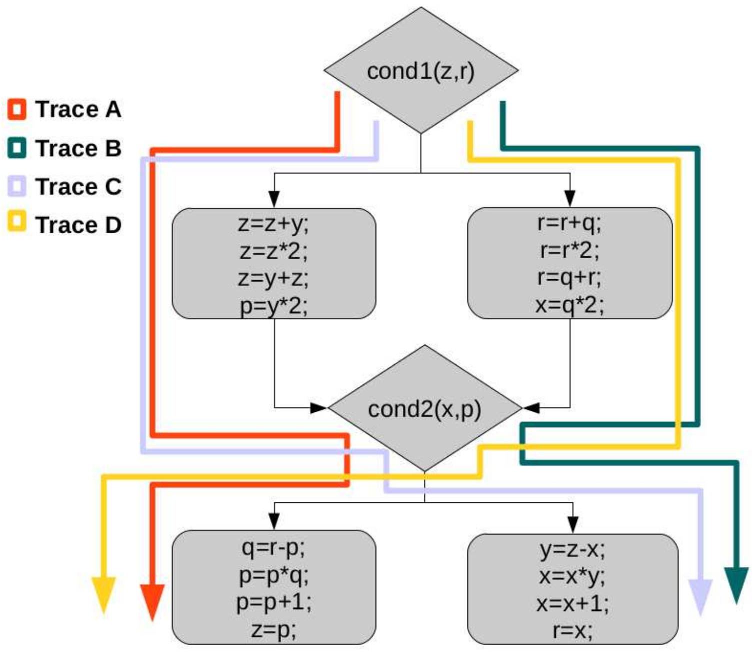
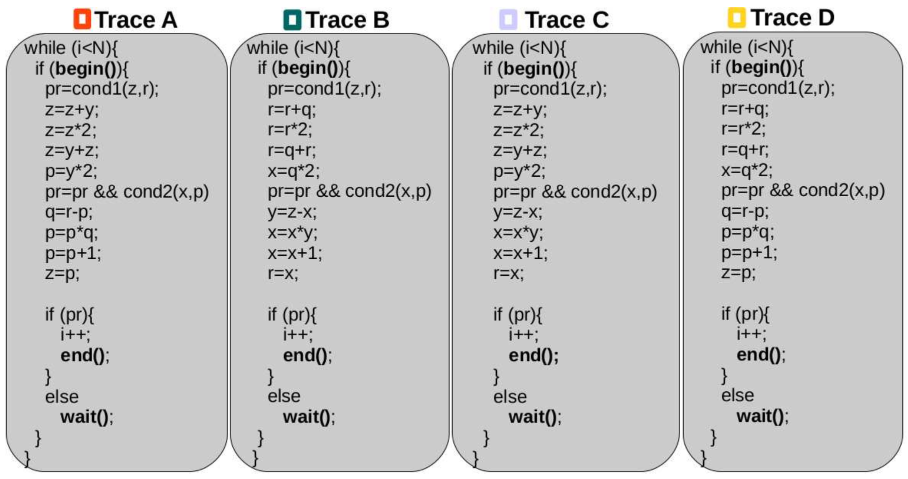


Figure 6.2: Possible traces of execution.

```
1. Profile the program to identify the hottest loops.

2. Collect the source code for all traces (when exhaustively
   speculating) of the selected loops identified in Pass 1. In the
   example, there are four traces --- A, B, C and D --- shown in Figure
   6.2.

3. Create a thread pool with sufficient threads to execute all traces.
   In the example, four threads will be created.

4. Use the thread pool to dispatch a task for each trace. Each task
   executes all iterations of their corresponding trace as shown in
   Figure 6.3.

5. In the source code transform each trace into a transaction enclosed
   by the begin and end instructions. At each iteration of the loop,
   traces must evaluate all their conditions at the end of the
   transaction (Figure 6.3). If all conditions are true the trace must
   commit (and update the induction variable), otherwise it must wait
   for the correct transactional trace to commit.

6. Activate compiler optimizations to be applied to speculative traces.
   Figure 6.4 shows the traces of Figure 6.3 optimized by the compiler.
   As shown, Trace A and Trace B were optimized using a classic
   dead-code elimination. Other traces (C and D) were not optimized.
```

Algorithm 6.1: *STO* Algorithm



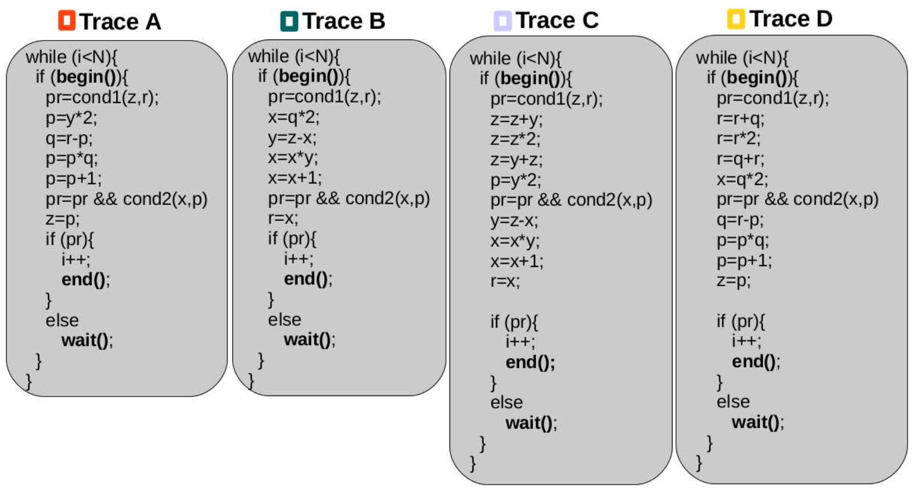
Trace A
```
while (i<N){
  if (begin()){
    pr=cond1(z,r);
    z=z+y;
    z=z*2;
    z=y+z;
    p=y*2;
    pr=pr && cond2(x,p)
    q=r-p;
    p=p*q;
    p=p+1;
    z=p;

    if (pr){
       i++;
       end();
    }
    else
       wait();
  }
}
```

Trace B
```
while (i<N){
  if (begin()){
    pr=cond1(z,r);
    r=r+q;
    r=r*2;
    r=q+r;
    x=q*2;
    pr=pr && cond2(x,p)
    y=z-x;
    x=x*y;
    x=x+1;
    r=x;

    if (pr){
       i++;
       end();
    }
    else
       wait();
  }
}
```

Trace C
```
while (i<N){
  if (begin()){
    pr=cond1(z,r);
    z=z+y;
    z=z*2;
    z=y+z;
    p=y*2;
    pr=pr && cond2(x,p)
    y=z-x;
    x=x*y;
    x=x+1;
    r=x;

    if (pr){
       i++;
       end();
    }
    else
       wait();
  }
}
```

Trace D
```
while (i<N){
  if (begin()){
    pr=cond1(z,r);
    r=r+q;
    r=r*2;
    r=q+r;
    x=q*2;
    pr=pr && cond2(x,p)
    q=r-p;
    p=p*q;
    p=p+1;
    z=p;

    if (pr){
       i++;
       end();
    }
    else
       wait();
  }
}
```

Figure 6.3: Traces as transactions.



Trace A
```
while (i<N){
  if (begin()){
    pr=cond1(z,r);
    p=y*2;
    q=r-p;
    p=p*q;
    p=p+1;
    pr=pr && cond2(x,p)
    z=p;
    if (pr){
       i++;
       end();
    }
    else
       wait();
  }
}
```

Trace B
```
while (i<N){
  if (begin()){
    pr=cond1(z,r);
    x=q*2;
    y=z-x;
    x=x*y;
    x=x+1;
    pr=pr && cond2(x,p)
    r=x;
    if (pr){
       i++;
       end();
    }
    else
       wait();
  }
}
```

Trace C
```
while (i<N){
  if (begin()){
    pr=cond1(z,r);
    z=z+y;
    z=z*2;
    z=y+z;
    p=y*2;
    pr=pr && cond2(x,p)
    y=z-x;
    x=x*y;
    x=x+1;
    r=x;

    if (pr){
       i++;
       end();
    }
    else
       wait();
  }
}
```

Trace D
```
while (i<N){
  if (begin()){
    pr=cond1(z,r);
    r=r+q;
    r=r*2;
    r=q+r;
    x=q*2;
    pr=pr && cond2(x,p)
    q=r-p;
    p=p*q;
    p=p+1;
    z=p;

    if (pr){
       i++;
       end();
    }
    else
       wait();
  }
}
```

Figure 6.4: Optimized traces of execution.

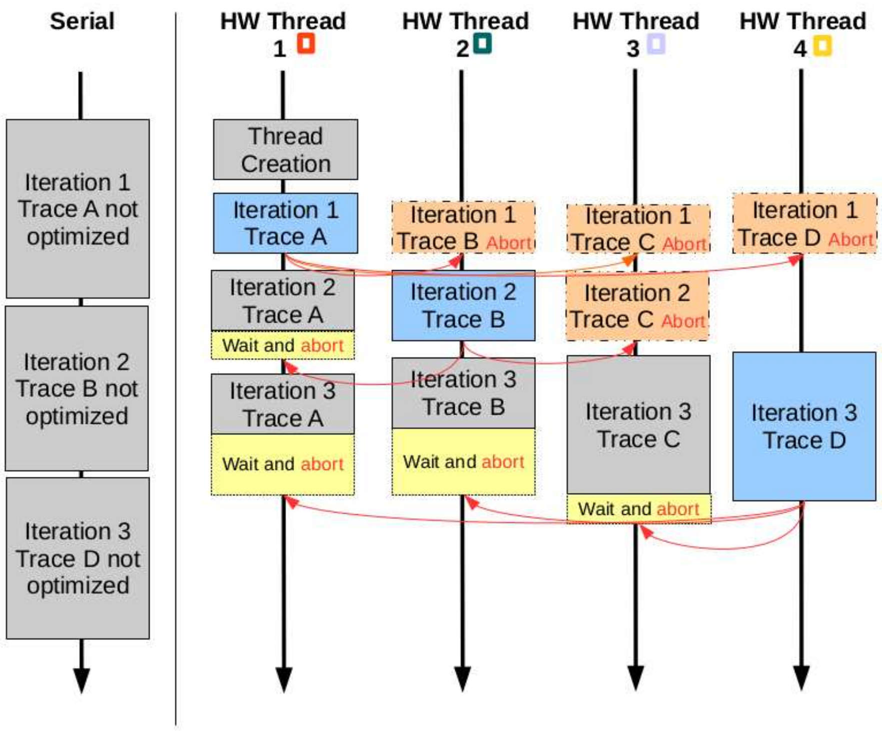


Figure 6.5: Possible execution flow of STO Traces shown in Figure 6.4 on ideal HTM.

The conditionals of all `if` statements in each trace are converted into *predicates* for the execution of the trace and are evaluated at the end of the trace. Only one trace evaluates all its predicates to true and must commit when the `end` instruction is executed. The other traces either: (a) wait for the correct trace to commit and abort afterwards, or (b) will be aborted due to a conflict, by the transaction that commits, without having finished its execution. Figure 6.5 shows, in blue, the transactions (traces) that commit, in yellow the time spent by a transaction waiting for another one to commit and, in orange, the transactions that abort without finishing. Transactions that never complete are shown with dashed borders.

As shown in Figure 6.5, all threads are created before the execution of loop iterations starts (Pass 3 of Algorithm 6.1). This preamble is executed by the hardware thread 1. In Figure 6.5, hardware threads 1, 2, 3, and 4 execute traces A, B, C, and D, respectively. At the first iteration, suppose that the if-conditions of Lines 2 and 14 of Figure 6.1 are both true. Thus Trace A evaluates its predicates to true and commits. The other three traces B, C, and D — which read or write variables `p`, `q`, and `z` written by Trace A — abort without finishing.

In the second iteration, suppose that the conditions in Lines 2 and 14 of Figure 6.1 are both false. This way Trace B evaluates all its predicates to true and commits writing to variables `x`, `y`, and `r`. On the other hand, traces A and C have to abort because they read or write these variables. Trace A finishes before Trace B commits, and thus it has to wait and then abort. Trace C, on the other hand, aborts before completion. In some iterations not all traces are executed. Assume that, when the value of the induction variable is k, the trace that evaluates all its predicates to true finishes execution and updates the induction variable to $k + 1$ before one of the traces with false predicates starts. Then the thread responsible for this later trace will skip the execution of a transaction for iteration k. In the example, Trace D skips iteration 2 entirely and resume execution in the iteration 3.

Finally, in the third iteration, Trace D evaluates all its conditions to true and commits writing to variables `p`, `q`, `r`, `x`, and `z`. The other three traces A, B, and C read or write these variables and finish before Trace D, thus they have to wait and then abort due to the commit of Trace D.

The left side of Figure 6.5 shows the serial execution of the above three iterations.

6.1.2 STO Prototype on Real-world HTM

The previous discussion assumed that STO would be executed using an ideal HTM. This sections explains how STO works in a real HTM (*i.e.* Intel TSX) and discusses the main features, which are lacking in TSX, that need to be addressed to enable speculative trace optimization.

Privatization to Simulate Multi-versioning and Lazy Conflict Resolution

Eager conflict resolution is a problem for STO because multiple traces write to the same variable, and thus each of those conflicting writes would cause a conflict abort. The aborted transaction could be the one that had to commit causing a significant retry overhead. To overcome this limitation, STO prototype privatizes all variables that have to be written within a transaction (trace) and the induction variable of the loop. In the example of Figure 6.6, variable `i` is the induction variable and variables `p`, `q`, and `z` are written within Trace A. Variable `i` is copied at the beginning of the transaction executing Trace A. Variables that are written are also copied into their thread-local copies (*e.g.* `zL` is a local copy of `z`). Conflicts are detected and resolved after the commit (`_xend`), when the local copies are non-speculatively written back to the original variables by the trace with all predicates true. Thus, privatization implements multi-versioning and lazy-conflict resolution, necessary features for speculative trace optimization.

Non-speculative Writes to Simulate Conflict-Resolution Policy

The conflict-resolution policy used by TSX can interfere with the implementation of STO. Let T be the trace that evaluates all its predicates to true, and let $F_0, \ldots, F_k$ be the traces for which at least one of the predicates is false. When T writes to the variables that it modifies and attempts to commit, it is likely that T has a conflict with a transaction that is executing one of the other threads. If the conflict-resolution policy were allowed to abort T and allow the survival of some F_i trace, the intent of STO would be defeated. To overcome this limitation, once T commits, T non-speculatively writes the modified variables, including the loop-induction variable. Some F_i may be executing or have finished. If F_i finishes, STO forces F_i to spin on an infinite loop while only T proceeds to the commit phase. These non-speculative writes lead to the abortion of all F_i (spinning or not) because each trace has to read the induction variable. This mechanism is used to create the effect of a Conflict-Resolution Policy — a necessary feature for speculative trace optimization — in an HTM that does not have this feature.

```
1  i=&(param->i);
2  while ((*i) < n){
3      status = _xbegin();
4      if (status == _XBEGIN_STARTED){
5          iL=*i;
6          pr=cond1(z,r);
7          zL=z+y;
8          zL=zL*2;
9          zL=y+zL;
10         pL=y*2;
11         pr=pr && cond2(x,pL);
12         qL = r - pL;
13         pL = pL*qL;
14         pL = pL + 1;
15         zL = pL;
16         if (pr && (iL < n)){
17             _xend();
18             param->i= param->i + 1;
19             z=zL;
20             p=pL;
21             q=qL;
22         }
23         else if (iL < n)
24             while(1);
25     }
26 }
```

Figure 6.6: Modified Source Code of Trace A.

Pausing to Simulate Trace Synchronization

A trace that evaluates any of its predicates to false is a miss-speculation and should abort. One way to abort such a trace, would be to issue an `_xabort` instruction whenever a predicate fails and to retry. An alternative is to keep the miss-speculation trace executing an idle loop until it is aborted **only once** because of a detection of conflict with the correct trace. A non-speculative write of the value of the induction variable — which is read at the start of all traces — causes all incorrect traces to abort. These two approaches have been tested on Intel TSX for the traces of Figure 6.1 and their impact on performance was measured. The use of the `_xabort` instruction to interrupt incorrect traces resulted in a 1.19× slowdown when compared to waiting for the correct trace to commit. This slowdown is due to the cost of recovery from `_xabort` in TSX, which is high (*150 cycles*) [50], resulting in a large performance penalty for issuing this instruction **many times** by retrying. Therefore, with the current implementation of TSX, waiting at the end of the transaction for the commit of the correct trace is a better approach to build a STO prototype. Lower-cost aborts in future architecture would change this trade-off. This prototype implementation of STO on top of TSX uses an infinite loop `while (1);` statement, as shown in the Line 24 of Figure 6.6. A trace that evaluates any of its predicates to false will wait until the thread executing the correct trace commits and then writes, non-speculatively, the new value of the induction variable (Lines 19 - 18 of Figure 6.6). This non-speculative write will lead to the intended eager conflict detection

Table 6.1: HTM Architectures.

Features/HTMs	TSX	BG/Q	P8
Multi-version		✓	
Eager Detection	✓	✓	✓
Lazy Resolution		✓	
Ordered Txs		✓	
Suspend/Resume			✓
Lazy Detection		✓	
Data Forwarding		✓	
ROTs			✓

and conflict resolution mechanisms of TSX to abort all the incorrect traces. In the current implementation of Intel TSX a transaction may also abort due to other reasons such as traps when the limit of OS quantum has been reached, interrupts, temporary capacity overflow, *etc.* Thus, the STO prototype retries the transaction when such spurious aborts occur to ensure that the correct trace is eventually executed to completion.

The above analysis suggests that multi-versioning, eager conflict detection, lazy conflict resolution, and transaction synchronization are central features to enable trace speculation. Unfortunately, as shown in Table 6.1, none of the current HTM architectures (Intel TSX, IBM BG/Q and POWER8) have all the features required to implement trace speculation strategies like STO. Intel TSX does not allow for multi-versioning nor lazy-conflict resolution. On the other hand, although POWER8 has suspend/resume instructions, which could eventually implement transaction synchronization, it does not allow multi-versioning nor lazy conflict resolution. Moreover, in the current version of POWER8 the cost of suspend/resume is comparable to the cost of starting a transaction. Blue Gene/Q [23] is the architecture that is closest to implement all the features required for trace speculation. BG/Q features multi-versioning cache, ordered transactions in hardware (for transaction synchronization), and lazy conflict resolution; features that are useful to enable STO. However, the runtime system implemented on top of the best-effort HTM in BG/Q provides forward-progress guarantees that assume that each started transaction must eventually commit [64, 65]. This assumption does not fit well with the concept of speculation in STO, where all but one trace should abort.

Dice *et al.* described several pitfalls that show that lazy subscription is not safe for Transactional Lock Elision (TLE) [18]. Those problems are not present for STO because the Trace T (that evaluates all its predicates to true) writes the induction variable i non-speculatively. Thus, if another trace F starts its transactional execution, it will read the value of i. Either it reads that value before trace T commits — and thus have the same i value as trace T —, or it reads the new value of i after trace T commits. If it reads the value before Trace T commits, the committing (and update of i) of trace T will cause a conflict leading Trace F to abort. If it reads i after trace T commits (and updates i), then Trace F is executing a different iteration than Trace T did.

6.1.3 Running STO on Intel TSX

Figure 6.6 shows Trace A, after the code in Figure 6.3 is modified using Intel TSX; the other traces were modified accordingly. Algorithm 6.2 explains the implementation of the STO strategy when using TSX, again considering the example of Figure 6.1.

```
1-5 Same as Passes 1-5 in Algorithm 6.1.

6. At each trace, make a local copy of each variable that is written
   within the transaction (for example variables $z$, $p$, and $q$ whose
   corresponding copies are $zL$, $pL$, and $qL$ in Figure 6.6) and replace
   the original variable by their private copies in the transaction.

7. At each trace, after committing, write the value of the private
   variables to the original ones as in the Lines 19-21 of Figure 6.6.

8. For each trace, read the induction variable at the beginning of the
   transaction into a local variable as shown in the Line 5 of Figure
   6.6; replace the original induction variable by the private copy in
   the whole transaction. After committing, update the induction
   variable as shown in the Line 18 of Figure 6.6.

9. At each trace, simulate waiting for the correct trace commit by
   putting a while(1) for the case when the predicates of the trace are
   false as shown in Line 24 of Figure 6.6.

10. Same as Pass 6 in Algorithm 6.1.
```

Algorithm 6.2: *STO* Strategy Using TSX

Figure 6.7 is an example of an execution sequence of the loop in Figure 6.1 using STO on Intel TSX in a system with four hardware threads. As detailed below, the performance of this execution is worse than the execution shown in Figure 6.5 because Intel TSX lacks many features of an ideal HTM for STO, as mentioned in Section 6.1.2.

As before, in Figure 6.7 transactions (traces) that commit are colored in blue, the time spent by a transaction waiting for another one to commit — in this case in the `while(1);` statement — is colored in yellow, and transactions that abort without having finished are shown in orange. The overhead to update (after commit) the induction variable and the written variables of the transaction (as in Passes 7 and 8 of Algorithm 6.2) are shown in red. The overhead to read the induction variable is shown at the beginning of each trace (transaction). The main differences between the execution of Figure 6.7 and the execution of Figure 6.5 are: (a) The insertion of the induction variable read in each transaction. (b) The insertion of the induction variable non-speculative write after committing the correct trace.

Hardware threads 1, 2, 3, and 4 execute traces A, B, C, and D, respectively. In the first iteration, suppose that all the predicates of Trace A evaluate to true and thus its

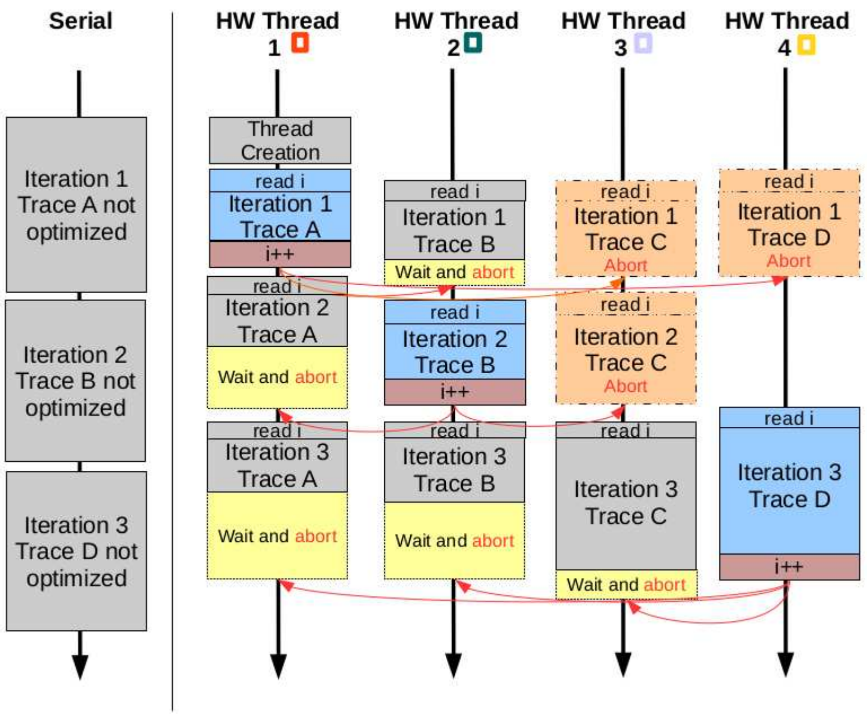


Figure 6.7: Possible execution flow of STO Traces shown in Figure 6.4 on Intel TSX.

transaction commits after testing the predicates. Immediately after committing, Trace A non-speculatively copies the values from the thread-local copies `pL`, `qL`, `zL`, and `iL` into variables `p`, `q`, `z`, and `i`, respectively. The other three traces — B, C, and D — read these variables in their respective transactions and have to abort due to a conflict. Every trace has the loop induction variable in its read set. Thus when the correct trace — Trace A in this example — non-speculatively writes to that variable, the eager-conflict detection and resolution in TSX forces all other traces to abort. Assume, in this example, that Trace B finishes before the non-speculative writes by the thread of Trace A. Thus Trace B has to wait (in the `while(1);` statement) until it is aborted by the eager conflict mechanism.

Finally, during the third iteration, all predicates of Trace D evaluate to true, it commits and then writes non speculatively to variables `p`, `q`, `r`, `x`, `z`, and the induction variable `i`. The other three traces — A, B, and C — read these variables in their transactions, and finish before Trace D writes non speculatively, they have to wait and then abort due to the conflict with the non-speculative writes of Trace D.

The left side of Figure 6.7 shows the serial execution of the program.

6.2 Performance Assessment of Proof-of-Concept Prototype

This section presents performance assessment of the prototype of STO using TSX for benchmarks from Parboil, SPEC CPU2006, and Mediabench-II.

6.2.1 Benchmarks, Implementation, Settings, and Environment

To select programs for this initial experimental evaluation we searched for hot regions of code in the Parboil, SPEC CPU2006, and Mediabench benchmark suites to uncover code that could have potential to improve performance through STO. Profiling of **all benchmarks** from these three suites revealed hot loops that were then analyzed to determine if these loops are amenable to STO optimization. This analysis evaluates the number of traces in the loop and measures the trace hotness, execution probability and optimization potential. Finally, we select hot loops according to our analysis and we modify them using TSX.

Table 6.2 shows the selected benchmarks that contain loops for which STO is applicable at the moment. Except for `h263dec` that contains two STO loops, in all other benchmarks STO was applied to a single loop. The second column of the Table 6.2 indicates the Benchmark Suite that the program came from. The fourth column of Table 6.2 shows the locations/lines of the target regions in the source code. The fifth column shows the fraction of the total execution time ran by the hot code regions.

This implementation of STO on Intel TSX uses Pthreads. Creating a *pthread* incurs a significant overhead in this experimental platform (Linux). Therefore, a pool of Pthreads, with one Pthread for each hardware thread, is created once, just before the execution of the hot region of code. This pool of threads is then reused at each iteration.

This initial evaluation of the prototype uses an Intel Core i7-4770 processor, running at 3.4 GHz, with 16 GB of memory on Ubuntu 12.04.3 LTS (GNU/Linux 3.8.0-29-generic x86_64). The Intel Core i7-4770 has 4 cores with 2-way SMT. Each core has a 32 KB L1 data cache and a 256 KB L2 unified cache. The four cores share an 8 MB L3 cache. The benchmarks are compiled with GCC 4.9.2 at optimization level `-O3`.

The following section discusses the performance evaluation comparing the STO execution with the serial execution of the same benchmark also compiled at level `-O3`. STO accelerates loops that may contain data dependences that prevent parallelization, thus comparison with the sequential code is appropriate. Whole-program executions are compared and not only the execution time of the region of the code to which STO is applied. Each benchmark was run 100 times.

6.2.2 Benchmark Results

Figure 6.8 shows the speed-up of the selected benchmarks with respect to the sequential execution. The average performance improvement over 100 runs due to STO on TSX varies between 1% (for `458.sjeng`) and 9% (for `Sad`). Performance variability with a 95% confidence is also shown in Figure 6.8. The modest speed-ups are due to the overhead of privatization, padding, copying variables after commit (to simulate lazy conflict resolution), and mainly due to the expensive cost of aborts in Intel TSX (an abort costs 150 cycles).

At each iteration only one transaction (trace) should commit and the others must abort, thereby the number of aborts by conflict at each iteration should be *number_ of_ traces-1*. Moreover, the cause of an abort should be, in most cases *memory conflicts*. Aborts

Table 6.2: Amenable loops to STO.

Benchmark	Origin	Description	Location in source code	% Coverage
`Lbm`	Parboil	A fluid dynamics simulation of an enclosed, lid-driven cavity.	`lbm.c`, 186	93%
`Mri-g`	Parboil	Computes a regular grid of data representing an Magnteic Resonance scan.	`CPU_kernels.c`, 174	71%
`Sad`	Parboil	Sum of absolute differences kernel, used in MPEG video encoders.	`sad_cpu.c`, 81	83%
`458.sjeng`	SPEC CPU 2006	Based on Sjeng 11.2, which is a program that plays chess and several chess variants.	`neval.c`, 493	23%
`h263dec`	Mediabench-II	A video decoder (h263dec) based on the ITU H.263 standard targeting video compression.	`store.c`, 400	45%
			`store.c`, 442	35%

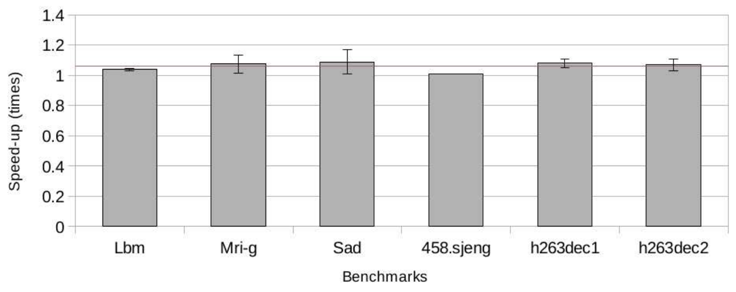


Figure 6.8: Speed-ups of benchmarks with respect to serial execution.

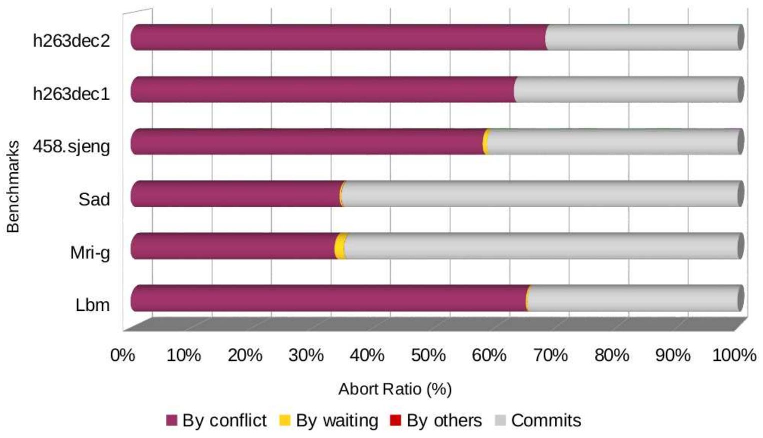


Figure 6.9: Abort Ratio (%) of the benchmarks.

due to other reasons, such as capacity and interruptions, occur occasionally. In such cases the transaction has to retry.

Figure 6.9 shows the abort ratio, computed as the number of aborted transactions divided by the number of started transactions, for the benchmarks. This ratio fluctuate between 35% and 68%. It also shows the causes of a abort: conflict, waiting and others. Aborts due to *waiting* (while the transaction is executing `while(1)`) are due to limit of the OS quantum allocated to the thread and not due to memory conflicts. Almost all aborts are caused by memory conflicts, the other reasons are almost insignificant.

An interesting question is whether most conflict aborts are due to the commit of a transaction (lazy conflict resolution) or are they due to other causes (*e.g.*, false sharing). To provide some insight, Table 6.3 shows the number of commits and the number of conflict aborts for each trace in `Lbm`.

Table 6.3: Conflict aborts and commits of `Lbm` traces.

Conflict Aborts	**Trace A (commits)**	**Trace B (commits)**	**Trace C (commits)**	**Total Commits**	**Factor**
Trace A (1806K)		26K	1789K	26K+1789K =1815K	*1806/1815 =0.995*
Trace B (1995K)	343K		1789K	2132K	*0.936*
Trace C (258K)	343K	26K		369K	*0.700*

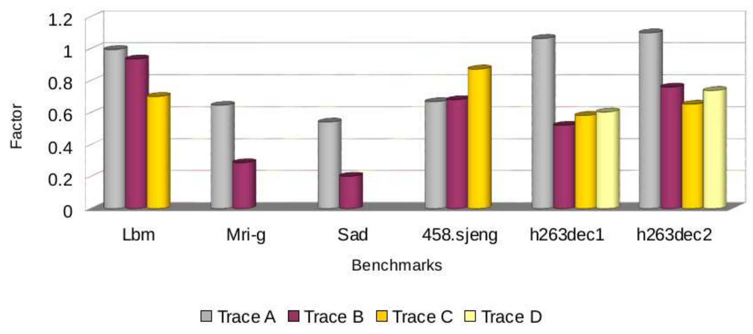


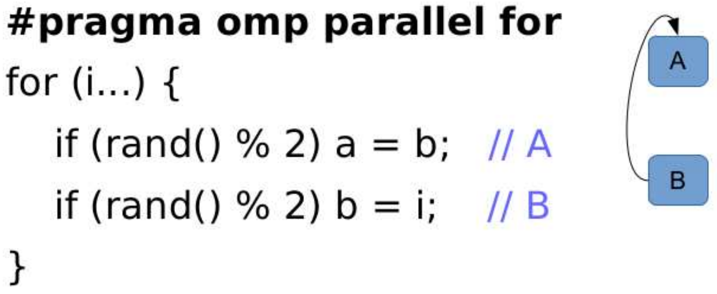
Figure 6.10: Factor for each trace in the benchmarks.

The number of conflict aborts of a given trace should be almost equal to the sum of commits of the other traces because each time that a trace with all predicates true commits, all other traces must abort. For example, consider the case of Trace A in `Lbm` shown in Table 6.3. It should have aborted 1815K times (sum of commits of traces B and C) but it aborted 1806K times by conflict, resulting in a (real/expected) ratio of 0.995. This small difference is explained by the fact that some threads occasionally skip an iteration because they do not start before the correct thread commits and updates the induction variable, as explained in Section 6.1.1. Table 6.3 shows similar results when considering the other traces of `Lbm`. Measurements for the other benchmarks reveal similar results as in the `Lbm` case. The ratios for each trace of each benchmark are shown in Figure 6.10. Most ratios are closer to or less than one, meaning that the number of conflict aborts for each trace is closer to or less than the number of commits of the other traces and thus the impact of other conflict abort causes is imperceptible, as expected.

Chapter 7

`parallel for check` Directive

A loop (as shown in Figure 7.1) has a *loop-carried dependence* if there is a statement A dependent on B and both statements are executed in different iterations. As mentioned before, loop-carried dependences limit loop iteration parallelization.



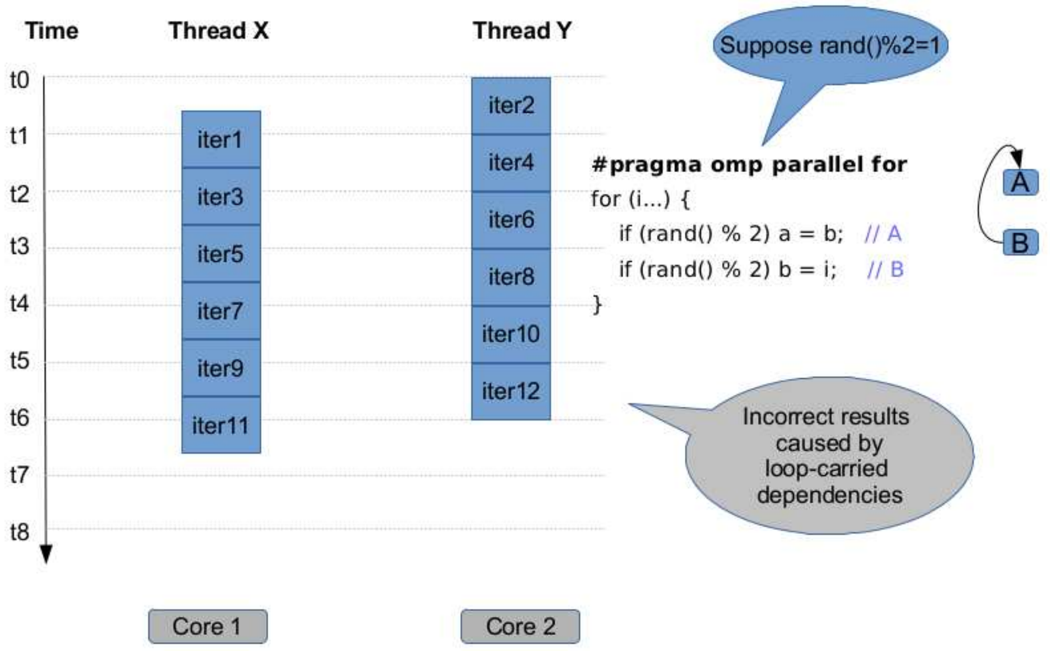
```
#pragma omp parallel for
for (i...) {
    if (rand() % 2) a = b;   // A
    if (rand() % 2) b = i;   // B
}
```

Figure 7.1: Loop-carried dependence example.

Data-dependence analysis is an important technique to detect loop-carried dependences and to exploit parallelism in programs. It works by detecting if two instructions access the same memory location, and at least one of them is a write operation. A *loop-carried dependence* occurs when these instructions execute in different iterations; otherwise they are called *loop-independent* [66]. As discussed previously, if two instructions are loop-independent, the iterations can be safely executed in parallel without the need of synchronization. Otherwise, if they define a loop-carried dependence, this can not be achieved.

For example, Figure 7.2 shows an incorrect execution of the previous loop (Figure 7.1), as iteration 2 is executed before iteration 1, so it does not respect the loop-carried dependence between instruction A and B. Specifically, the read of variable b in iteration 2 is incorrect as variable b has a loop-carried dependence to the execution of statement B in the previous iteration.

One potential source of bugs, while programming in OpenMP, shows up if a programmer incorrectly evaluates this as a *DOALL* loop, and thus parallelizes it using a *parallel for* construct. By using the *parallel for check* construct, proposed herein, this error could be detected at runtime.



```
#pragma omp parallel for
for (i...) {
    if (rand() % 2) a = b;   // A
    if (rand() % 2) b = i;   // B
}
```

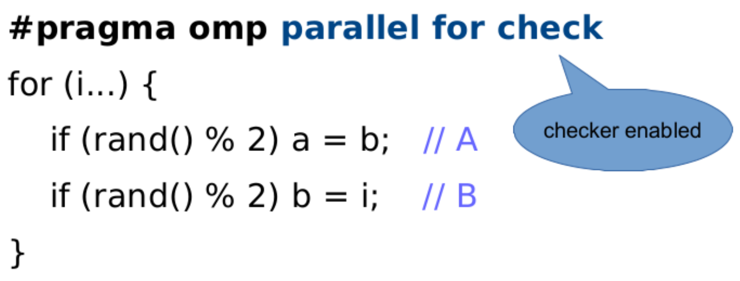
Figure 7.2: Possible execution flow of the loop of Figure 7.1.

7.1 *Check* Construct in OpenMP

This section presents the new *check* construct, a novel OpenMP construct which can detect loop-carried dependences in OpenMP. It capitalizes on some advantages of both, Pairwise and Stride-based methods; while it tries to minimize their deficiencies.

7.1.1 Overview of the Algorithm

Pairwise and Stride methods use, for each instrumented loop of the program, one pending table that is flushed at each new iteration of the loop, and one big history table to store all dynamic memory references seen so far along the loop execution. In the case of Stride method, it duplicates the number of tables for managing strides and points. By contrast, our approach uses a memory efficient data structure per-loop.



```
#pragma omp parallel for check
for (i...) {
    if (rand() % 2) a = b;   // A
    if (rand() % 2) b = i;   // B
}
```

Figure 7.3: Usage of *check* construct in the program of Figure 7.1.

To store memory references in *check*, we use two (read and write) *Multilevel Hash Tables (MHT)* [10] which maps references to two `int` numbers: `maxIter` and `minIter` —

the maximum and minimum number of iteration stored due to a memory reference by the corresponding thread at that moment — as shown in Figure 7.4. By doing so, the size of the memory footprint required to store iteration addresses is considerably reduced. MHT has a two-level key composed by the memory address and the thread ID, mapping two numbers that indicates the maximum (or minimum) iteration where the corresponding address was written or read by that thread. On the average case, search time in this kind of structure is $O(k)$ where k is the number of levels. In our case, $k = 2$ and thus search time is $O(1)$ on average.

The detailed algorithm is described in Listing 7.3 as follows.

```
1. When a loop with $check$ directive, $L$, starts, the checker is
   activated.

2. On a memory address, $R$, of $L$'s $i$-th iteration done by thread $X$,
   store min($i$,$minIter$) and max($i$,$maxIter$) into the corresponding numbers
   of the key composed by $R$ and $X$ on the Multilevel Hash Table.

3. If the memory reference in $R$ is a read instruction, the checker
   looks for if there is a memory write on this address $R$, in another
   thread different from $X$. If this memory write exists and its
   $maxIter > i$ , the checker reports a violation of WAR loop-carried
   dependence. If warning_option is activated, the checker also looks
   for if there is a memory write on this address $R$, in any thread. If
   this memory write exists and $minIter \leq i \leq maxIter$, the checker
   reports a warning of WAR loop-carried dependence.

4. If the memory reference in $R$ is a write instruction, the checker
   looks for if there is a memory write or read on this address $R$, in
   another thread different from $X$. If a memory write exists and its
   $maxIter > i$, the checker reports a violation of WAW loop-carried
   dependence. If a memory read exists and its $maxIter > i$, it reports a
   violation of RAW loop-carried dependence. If warning_option is
   activated, the checker also looks for if there is a memory write or
   read on this address $R$, in any thread. If a memory write exists and
   $minIter \leq i \leq maxIter$, the checker reports a warning of WAW
   loop-carried dependence. If a memory read exists and
   $minIter \leq i \leq maxIter$, it reports a warning of RAW loop-carried
   dependence.

5. When $L$ finishes, we flush the Multilevel Hash Table.
```

Algorithm 7.3: OpenMP *checker* Algorithm

The algorithm focuses on detecting violations of loop-carried dependences. Some loop-carried dependences do not cause violations, as the order of execution is respected, and thus *checker* does not report such errors. Nevertheless, in some specific cases the programmer might want to be informed of all existing loop-carried dependences as this information

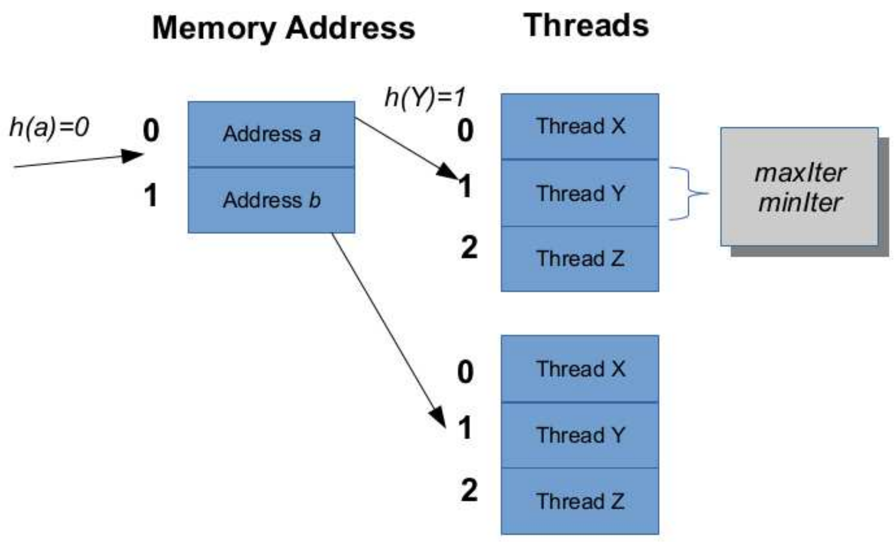


Figure 7.4: Multilevel Hash Table mapping to a memory reference in *address a* stored by *thread Y*.

could be useful to understand the causes of violations in future program runs. In order to enable the detection of all loop-carried dependences in *checker*, the programmer should activate an optional parameter called *warning_ option*.

Our approach does not need a sophisticated compression algorithm as described in [33, 32], given that we perform dependence verification for single loops. By combining this with the possibility of selecting the specific loop to analyze and the MHT data structure, we managed to reduce the memory and time overheads of the methods described in Section 3.4. Moreover, we merge identical dependences to reduce the memory overhead and the time of the algorithm by using auxiliary structures that store all the (violation of) dependences found for any two instruction pointers, thus avoiding to detect the same dependence several times.

As explained in Sections 3.4.1 and 3.4.2, previous solutions could have problems with multithreaded executions as they mark an address as killed once the memory address is written in an iteration; besides, they only report dependences per-thread. Thus, they can omit possible violations of loop-carried dependences. Our approach identifies these ignored violations given the analysis is not done on a thread basis but for the whole program. As explained before, we do not use the killed addresses method, as in our approach all violations of loop-carried dependences must be informed to force not omitting corrections of renaming of variables (that avoids WAR and WAW loop-carried dependences). These corrections can generate *privatized variables.* We inform, in multithreaded executions, loop-carried patterns by using the *thread ID* to do the verification of dependences.

7.1.2 Parallelization of the Algorithm

Instrumentation is a very time-consuming task because all memory writes and reads are instrumented for each loop as proposed by the Pairwise and Stride methods. SD^3 [33, 32] uses data-level parallelism and pipelining to reduce the time overhead. In contrast, our approach uses only pipeline-level parallelism. *Check* is composed by the following stages as shown in the Figure 7.5:

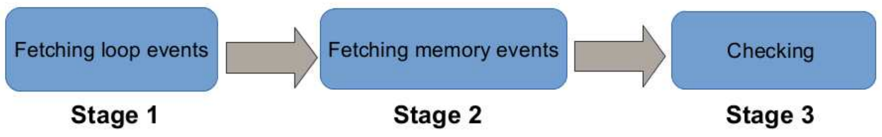


Figure 7.5: *OpenMP checker* exploits pipeline level parallelism (*3 stages*).

- *Fetching loop events.* This stage provides information about the beginning and termination of a loop and corresponds to the *Pass 1* of the algorithm shown in Listing 7.3.
- *Fetching memory events and storing memory references.* At this stage information about memory addresses, thread ID, number of iteration, and program counter is collected and stored into the MHT. This stage corresponds to the *Pass 2* of the Listing 7.3.
- *Checking loop-carried dependences.* Here dependence violations are verified as described in *Passes 3, 4 and 5* of the Listing 7.3.

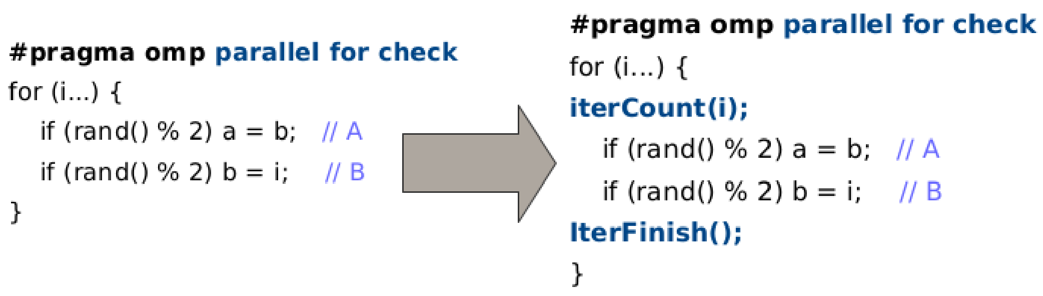
With pipeline parallelism, we parallelize a single task by dividing it into a series of sequential stages as shown in Figure 7.5 [21]. Parallelism is achieved by pushing succeeding data elements through a consumer-producer pipeline, where stages run simultaneously on different cores [21]. This approach has considerably reduced latency compared to *data-level parallelism.* However, it introduces extra synchronization, because producers and consumers must be tightly coupled; also, it is limited by inter-stage dependences and the duration of the longest stage. In our case, the *third stage* is the most time consuming stage, and thus it will determine the overall speed-up of the pipeline; however, we can still hide the latencies of *stages 1 and 2* from pipelining.

7.2 Implementation

In this section we describe both implementations of *checker* using GCC/Pin and LLVM. First, we present the basic structure of our *checker* and then we detail each implementation.

7.2.1 Basic Structure

The basic structure of *checker* consists of two modules, a *tracer* (stages 1 and 2 of the pipeline shown in Figure 7.5) and an *analyzer* (stage 3 of the pipeline). The first stage instruments the program, fetches loop and memory events at runtime, and stores memory references in a shared memory MHT. The second verifies, on-the-fly, the existence of loop-carried dependences. As *checker* is an online construct, it cannot afford to have large costs



```
#pragma omp parallel for check
for (i...) {
   if (rand() % 2) a = b;   // A
   if (rand() % 2) b = i;   // B
}
```

```
#pragma omp parallel for check
for (i...) {
iterCount(i);
   if (rand() % 2) a = b;   // A
   if (rand() % 2) b = i;   // B
IterFinish();
}
```

Figure 7.6: Imaginary source file with the *check* construct.

of instrumenting all loads and stores of program thus it is very useful an implementation where the programmer chooses which loop wants to verify.

7.2.2 GCC/Pin

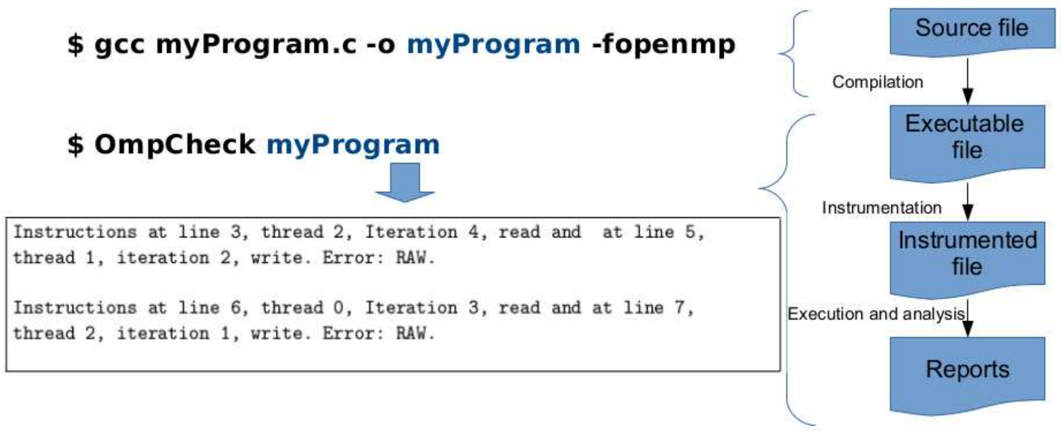
This section describes how the *check* construct was integrated into the GCC compiler. First, we adapted the GCC source code to recognize the *parallel for check* directive into the *#pragma* annotation. This implementation was very challenging as we had to adjust some very critical source files of GCC compiler (e.g. *c-parser.c*) to allow it to accept the new directive and also to delimit which loop will be analyzed.

Basically, when the check directive is inserted, the compiler recognizes it as a token and as part of a correct grammar expression, then inserts two function calls into the IR code: (a) *iterCount*, at the beginning of the chosen loop, which receives the number of the current iteration as parameter and is responsible for marking the beginning of an iteration annotation; and (b) *iterFinish*, at the end of the loop body, which marks the end of the instrumentation region. Finally, the compiler produces an executable file with the identified loops to be analyzed.

The Figure 7.6 shows the modifications inserted by the compiler when reflected into the source code.

Tracer module was implemented on top of Pin [3], which is a dynamic instrumentation framework that enables the creation of dynamic program analysis tools. The advantage of using Pin to implement our tracer is that it does not require recompilation for doing the verification, and could be applied to executable files from different compilers. The disadvantages of the Pin tracer, as explained in [32], are: (a) the need of the static analysis to recover control flow graphs and loop structures, and (b) the difficulty of filtering useless loads and stores. In our case, we discriminate loads and stores within a loop by inserting function calls (*iterCount* and *iterFinish*).

The instrumentation is performed at runtime on the compiled binary files. Pin allows a tool to insert code in arbitrary places of the executable, the code is added dynamically while the executable is running. Thus, our tracer walks through the executable files, when it finds an *iterCount* function call, it inserts instrumentation code to store the current iteration. Also, it inserts instrumentation code after every memory reference, be it a read or write, until finding an *iterFinish* function call, after which the instrumentation finishes. At runtime, for every memory reference, the tracer fetches the memory address, the



```
$ gcc myProgram.c -o myProgram -fopenmp

$ OmpCheck myProgram
```

```
Instructions at line 3, thread 2, Iteration 4, read and  at line 5,
thread 1, iteration 2, write. Error: RAW.

Instructions at line 6, thread 0, Iteration 3, read and at line 7,
thread 2, iteration 1, write. Error: RAW.
```

Figure 7.7: Flow overview of the *OpenMP checker* with GCC/Pin.

number of the current iteration, the instruction pointer and the ID of the thread making the memory reference. Finally, it stores the memory reference (`maxIter` and `minIter`) into the MHT indexed by memory address and thread ID; the instruction pointer and the source line are also stored in auxiliary maps.

Analyzer module implements the *passes 3, 4, and 5* of Listing 7.3 and could be used by different tracers (e.g. Pin and LLVM). During its implementation, it was necessary to use many efficient programming techniques and customized data structures to improve the efficiency of the analysis of *checker*. Figure 7.7 shows the execution flow of *checker* when implemented using Pin and GCC.

7.2.3 LLVM

As in the previous implementation of *checker* using GCC, we had to adapt the Clang front-end to accept the new *check* directive. The main ideas involved in this implementation are analogous to those used in GCC. We modify the Lexer and Parser files to insert function calls *iterCount* and *iterFinish*. Afterwards, the main issues involved in the LLVM tracer implementation are similar to those used in the Pin Tracer. The GCC/Pin analyzer can be used in LLVM as well.

We implemented the tracer in LLVM by creating an LLVM pass, which provides a very good static-analysis infrastructure. In contrast to Pin, LLVM provided an infrastructure which simplified the task of building control flow and loop structures. Besides, previous LLVM static-analysis passes can considerably decrease instrumentation and analysis overhead by identifying loop-carried dependences, at compile-time, and then ignoring them in the dynamic loop-carried verification, as is described in [63]. On the other hand, the main disadvantage of using an LLVM tracer is the recompilation for each analysis.

7.3 Experimental Results

This section evaluates the performance of *checker*, when compared to serial and OpenMP executions, using groups of experiments. Our experimental results were obtained on machines with Ubuntu 13.10 (64-bit), Intel i7 4-core with hyper-threading technology, and 8 GB main memory. We use 8 *Parboil benchmarks* [58] to report time and memory overheads by running the most executed loops (hottest loops) one at a time with *check*. [1] [2] [3]

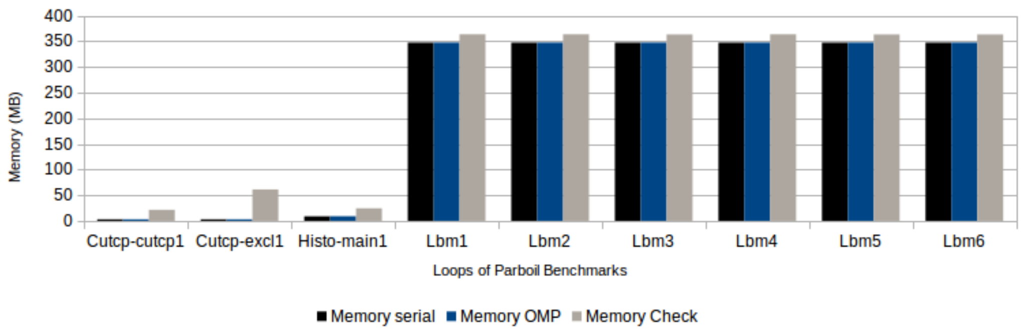


Figure 7.8: Memory footprint of three Parboil Benchmarks (*Cutcp*, *Histo* and *Lbm*) executed *serially*, with *OpenMP*, and with *check* modifying different hottest loops.

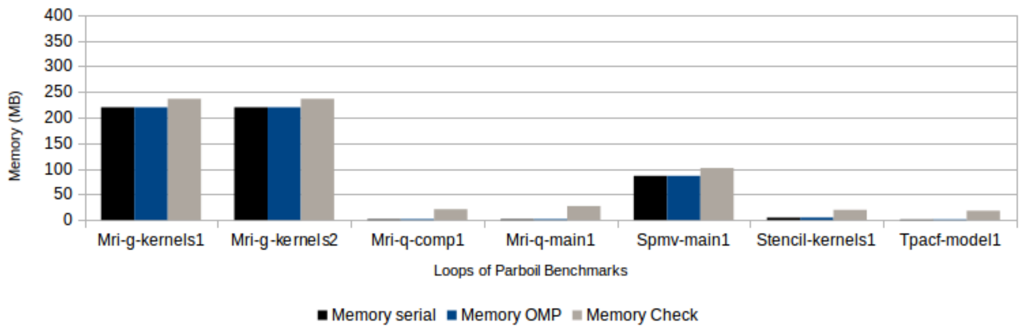


Figure 7.9: Memory footprint of five Parboil Benchmarks (*Mri-gridding*, *Mri-q*, *Spmv*, *Stencil* and *Tpacf*) executed *serially*, with *OpenMP*, and with *check* modifying different hottest loops.

Figure 7.8 and Figure 7.9 show the memory footprint for serial, OpenMP, and *checker* executions of the hottest loops of 8 Parboil benchmarks using the Parboil Datasets. As shown in Figure 7.8 and Figure 7.9, the memory overhead of *checker* is considerably smaller for most selected programs. The *OpenMP checker* verified all the benchmarks successfully as shown in Table 7.1, requiring not more than 400 MB of memory. Thus, selection of loops by the programmer and the data structure used in *checker* are effective techniques to avoid large memory overheads.

Table 7.1 shows the verification results of executing 8 Parboil benchmarks with *checker*. The checker reports 4 loops with violations of loop-carried dependences, and the column

[1]Our results are from GCC/Pin, but LLVM shows a similar performance.

[2]The remaining three benchmarks of Parboil were ignored as they do not have OpenMP *parallel for* constructs, or they were not programmed in C.

[3]The charts shown below are for different loops for each benchmark. For example, *lbm1* is the lbm parboil benchmark but with its *loop1* modified to use the *checker*. Thus, all variants of *lbm* have the same serial and OpenMP time/memory overhead.

Table 7.1: Verification of 8 Parboil executed with *check* modifying different hottest loops.

Benchmark	Program	Loop	Violation	Verified
`Cutcp`	`cutcp.c`	1	✓	✓+ is commutative on `pg`.
	`excl.c`	1	✓	✓All threads writing the same value on `pg`.
`Histo`	`main.c`	1	✓	✓+ is commutative on `histo`.
`Lbm`	`lbm.c`	1		✓
		2		✓
		3		✓
		4		✓
		5		✓
		6		✓
`Mri-gridding`	`CPU_kernels.c`	1		✓
		2		✓
`Mri-q`	`ComputeQ.c`	1		✓
	`main.c`	1		✓
`Spmv`	`main.c`	1		✓
`Stencil`	`kernels.c`	1		✓
`Tpacf`	`model_compute_cpu.c`	1	✓	✓+ is commutative on `data_bins`.

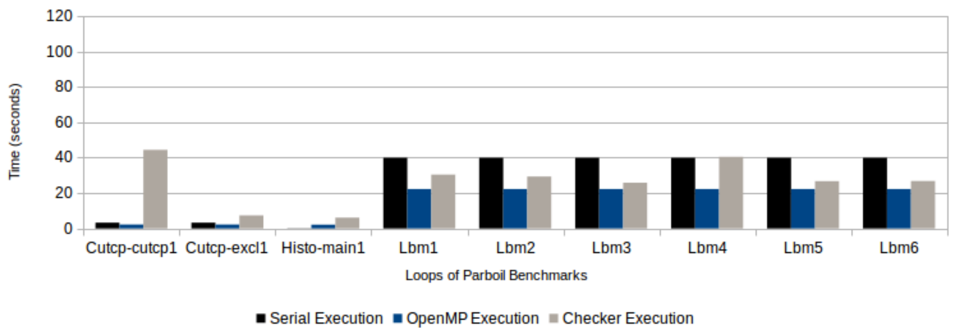


Figure 7.10: Execution time of loops of three Parboil benchmarks.

Verified explains the reasons. Notice that, if the operation involves updating a shared variable by means of a commutative operation the violation does not correspond to an error.

The time overhead results are presented in the Figure 7.10 and Figure 7.11. As shown, some executions with *check* are still faster than serial, with speed-ups of about 1.6×. This indicates that, although *check* adds instrumentation overhead it can, for some cases, still keep part of the performance resulting to the *parallel for* parallelization.

The largest slowdowns against *OpenMP execution* are about 20× and 8× as shown in Figure 7.12, corresponding respectively to the *cutcp*1 *loop* of *Cutcp* benchmark and the *kernels*1 *loop* of *Stencil* benchmark. The largest slowdowns against *serial execution* are about 56× and 36×, corresponding to the *main*1 *loop* of the *Histo* benchmark and the *model*1 *loop* of the *Tpacf* benchmark respectively. However, the *OpenMP* time executions are larger than the *serial* executions, as these two benchmarks have been poorly parallelized in the original distribution. Thus, only the slowdowns against *OpenMP* are valid (*OpenMP* execution times would be smaller than *serial* using methods as *privatization*). We can conclude that *check* offers a reasonably smaller overhead when compared to the serial and OpenMP executions. This has been achieved due to the pipeline parallelization and the *OpenMP checker* algorithm described in Section 7.1.

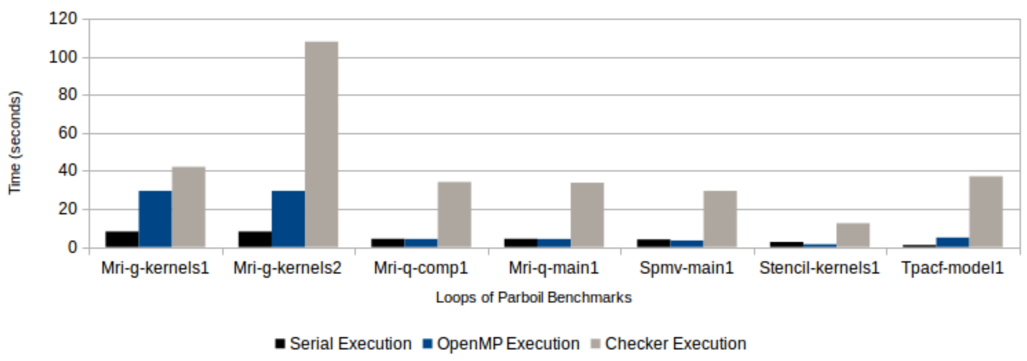


Figure 7.11: Execution time of loops of five Parboil benchmarks.

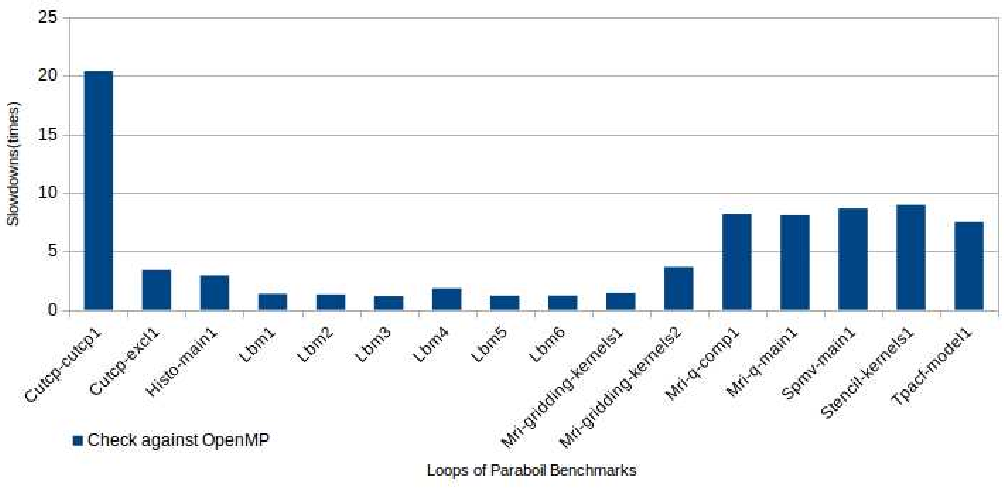


Figure 7.12: Slowdowns of loops of Parboil benchmarks using *check* respect to OpenMP execution.

Chapter 8

Conclusions

This work described how TLS can be supported on top of HTM support available in commodity off-the-shelf processors. The performance evaluation of an implementation of compiler-supported TLS over two existing commodity HTM-enabled processors provided interesting new insights on the issues that limit performance. The main findings are that false sharing is an important cause of performance loss and that false sharing may originate from different aspects of the program execution: too fine a distribution of iterations per executing thread; non-consecutive accesses to arrays; and the incorrect tracking of locations prefetched automatically in TSX. The results of this performance evaluation indicate that a more careful distribution of loop iterations per thread through a criterious selection of the strip size in strip mining, along with the alignment of accesses to the start of cache lines, can recover some of the performance lost to false sharing. The results also indicate that privatization of memory writes within transactions can successfully eliminate false dependencies and enable performance gains with TLS. Surprisingly, in some cases even when there is a substantial amount of additional copies introduced by privatization, TLS can still produce performance improvements. The study showed that even a crude mechanism for ordering the commit of iterations — such as the one in POWER8 — can be helpful to improve performance in loops with low speculative demand. On the other hand, loops with high speculative demand take advantage of the larger storage capacity in Intel Core. The results indicate that loops with short duration are not amenable to be parallelized with TLS on the existing HTMs.

Earlier work, based on the emulation of hardware support for TLS, had predicted surprisingly high performance improvements with this technique [40]. This work presents a detailed performance study of an implementation of TLS on top of existing commodity HTM of two architectures. Based on the performance results it classified the studied `cBench` loops and provided guidance to developers as to what loop characteristics make them amenable to the use of TLS on the Intel Core or on the IBM POWER8 architectures. Future design of hardware support for TLS may also benefit from the observations derived from this performance study. This work indicates that not all the requirements recommended by previous research [44] are necessary to deliver performance with TLS over HTM. But it does point out that multi-version speculative storage and ordered commit of transactions would be desirable in future hardware support for TLS.

This work also introduces STO, a technique to enable speculative optimization and execution of traces from hot loops. It describes the creation of a prototype for an initial evaluation of STO using Intel TSX. This evaluation uses six benchmarks, which were modified to enable STO under TSX. The initial performance experiments produced speed improvements varying from 1% to 9%. Another contribution of this work is a discussion of the features that would be necessary in hardware to enable STO, such as multi-versioning cache, eager conflict detection, lazy conflict resolution, and pausing transaction. An HTM with such features would lead to significantly higher speed gains due to STO. One of the focus of this thesis is to present the idea of STO. Automatic transformation of code to use STO is considered future work. Impacts on power consumption, memory traffic, and chip area will require a detailed architectural simulation of the features required for STO.

Finally, this work proposes the *check* OpenMP extension (i.e. *parallel for check* construct), a novel implementation of a dynamic loop-carried dependence *checker* in OpenMP which was used in the experimental evaluation of the techniques discussed in this thesis. It enables on-the-fly dynamic loop-carried dependence analysis of multithreaded applications, making it possible to detect hidden loop-carried dependences which can result in hard to detect parallel execution bugs. Some of these bugs can not be detected even by means of serial analysis or per-thread analysis as in previous works [33] described in Section 3.4. In order to reduce memory overhead, *OpenMP checker* analyzes only the loops that the programmer wants and uses a memory/time efficient data structure (Multilevel Hash Table). To reduce the time overhead, we used a three-stage Pipeline: (1) fetching loop events; (2) fetching memory events and storing memory references; and (3) checking loop-carried dependences. Furthermore, we showed how to integrate the *check* construct into GCC/Pin and LLVM.

Bibliography

[1] Alexander Aiken and Alexandru Nicolau. Perfect pipelining: A new loop parallelization technique. In *European Symposium on Programming (ESOP)*, pages 221–235, Nancy, France, March 1988. Springer.

[2] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *Programming Language Design and Implementation (PLDI)*, pages 72–84, Montreal, Quebec, Canada, 1998. ACM.

[3] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C. K. Luk, G. Lyons, H. Patil, and A. Tal. Analyzing parallel programs with PIN. *Computer*, 43(3):34–41, March 2010.

[4] Thomas Ball and James R. Larus. Efficient path profiling. In *Intern. Symp. on Microarchitecture (MICRO)*, pages 46–57, Paris, France, 1996. IEEE Computer Society.

[5] Arnamoy Bhattacharyya, Jose Nelson Amaral, and Hal Finkel. Data-dependence profiling to enable safe thread level speculation. In *Conference of the Center for Advanced Studies on Collaborative Research*, Markham, ON, Canada, November 2015.

[6] Rastislav Bodík and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *Programming Language Design and Implementation (PLDI)*, pages 159–170, Las Vegas, Nevada, USA, 1997.

[7] B.J. Bradel and T.S. Abdelrahman. Automatic trace-based parallelization of Java programs. In *Intern. Conf. on Parallel Processing (ICPP)*, pages 26–26, XiAn, China, Sept 2007.

[8] Borys J. Bradel and Tarek S. Abdelrahman. The use of traces for inlining in Java programs. In *Languages and Compilers for High Performance Computing (LCPC)*, pages 179–193, West Lafayette, IN, 2005. Springer-Verlag.

[9] Borys Jan Bradel. The use of traces in optimization. Master's thesis, Univ. of Toronto, Toronto, Canada, 2004.

[10] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 43–53, San Francisco, California, USA, 1990. Society for Industrial and Applied Mathematics.

[11] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. Helix: Automatic parallelization of irregular programs for chip multiprocessing. In *Code Generation and Optimization (CGO)*, pages 84–93, San Jose, USA, 2012.

[12] P. P. Chang and W. W. Hwu. Trace selection for compiling large C application programs to microcode. In *Workshop on Microprogramming and Microarchitecture*, pages 21–29, Los Alamitos, CA, USA, 1988.

[13] J.-H. Chow and V. Sarkar. False sharing elimination by selection of runtime scheduling parameters. In *Intern. Conf. on Parallel Processing*, pages 396–403, Bloomington, IL, Aug 1997.

[14] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[15] cTuning Foundation. cbench: Collective benchmarks, http://ctuning.org/cbench, 2016.

[16] Ron Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing (ICPP)*, pages 836–844, 1986.

[17] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, San Jose, California, USA, 2006. ACM.

[18] Dave Dice, Timothy L. Harris, Alex Kogan, Yossi Lev, and Mark Moir. Hardware extensions to make lazy subscription safe. *CoRR*, abs/1407.6968, 2014.

[19] John R Ellis. Bulldog: A compiler for VLIW architectures. Technical report, Yale Univ., New Haven, CT (USA), 1985.

[20] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[21] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, San Jose, California, USA, 2006. ACM.

[22] Richard E. Hank, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: An introduction and motivation. In *Intern. Symp. on Microarchitecture (MICRO)*, pages 158–168, Ann Arbor, Michigan, USA, 1995. IEEE Computer Society Press.

[23] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The IBM Blue Gene/Q compute chip. *IEEE Micro*, 32(2):48–60, March-April 2012.

[24] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[25] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Intern. Conf. on Computer Architecture (ISCA)*, pages 289–300, San Diego, CA, USA, May 1993.

[26] Ali R Hurson, Joford T Lim, Krishna M Kavi, and Ben Lee. Parallelization of doall and doacross loops—a survey. *Advances in computers*, 45:53–103, 1997.

[27] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, 1993.

[28] IBM. *Power ISA Transactional Memory*, 2012.

[29] Intel Corporation. *Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions*, 2012.

[30] Intel Corporation. *Intel Xeon Processor E3-1200 v3 Product Family Specification Update August 2014 Revision 008*, 2014.

[31] Alain Ketterlin and Philippe Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *Intern. Symp. on Microarchitecture (MICRO)*, pages 437–448, Vancouver, B.C., CANADA, 2012. IEEE Computer Society.

[32] M. Kim, N. B. Lakshminarayana, H. Kim, and C. K. Luk. Sd3: An efficient dynamic data-dependence profiling mechanism. *IEEE Transactions on Computers*, 62(12):2516–2530, Dec 2013.

[33] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *Intern. Symp. on Microarchitecture (MICRO)*, pages 535–546, Atlanta, USA, 2010. IEEE Computer Society.

[34] X. Kong, D. Klappholz, and K. Psarris. The I test: an improved dependence test for automatic parallelization and vectorization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2(3):342–349, Jul 1991.

[35] J. R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 4(7):812–826, Jul 1993.

[36] H.Q. Le, G.L. Guthrie, D.E. Williams, M.M. Michael, B.G. Frey, W.J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, Jan 2015.

[37] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *High Performance Computer Architecture (HPCA)*, pages 254–265, 2006.

[38] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Code Generation and Optimization (CGO)*, pages 198–208, San Jose, California, USA, March 2007.

[39] Steven S. Muchnick. *Advanced compiler design implementation.* Morgan Kaufmann, 1997.

[40] Niall Murphy, Timothy Jones, Robert Mullins, and Simone Campanoni. Performance implications of transient loop-carried data dependences in automatically parallelized loops. In *Intern. Conf. on Compiler Construction (CC)*, pages 23–33, Barcelona, Spain, 2016.

[41] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Intern. Conf. on Computer Architecture (ISCA)*, pages 144–157, Portland, OR, 2015.

[42] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware atomicity for reliable software speculation. In *Intern. Conf. on Computer Architecture (ISCA)*, pages 174–185, San Diego, California, USA, 2007.

[43] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. A lightweight in-place implementation for software thread-level speculation. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 223–232, Calgary, AB, Canada, 2009.

[44] R. Odaira and T. Nakaike. Thread-level speculation on off-the-shelf hardware transactional memory. In *Intern. Symp. on Workload Characterization (IISWC)*, pages 212–221, Atlanta, Georgia, USA, Oct 2014.

[45] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I August. Automatic thread extraction with decoupled software pipelining. In *Intern. Symp. on Microarchitecture (MICRO)*, page 12 pp, November 2005.

[46] V. Packirisamy, A. Zhai, and Wei-Chung Hsu. Exploring speculative parallelism in SPEC2006. Technical report, Department of Computer Science and Engineering, University of Minnesota, 2008.

[47] V. Packirisamy, A. Zhai, Wei-Chung Hsu, Pen-Chung Yew, and Tin-Fook Ngai. Exploring speculative parallelism in SPEC2006. In *Intern. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 77–88, Boston, Massachusetts, USA, April 2009.

[48] Christopher JF Pickett and Clark Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *International Workshop on*

Languages and Compilers for Parallel Computing, pages 304–318, New York, USA, 2005.

[49] James Reinders. *VTune performance analyzer essentials.* Intel Press, 2005.

[50] Carl G Ritson and Frederick RM Barnes. An evaluation of Intel's restricted transactional memory for CPAs. *Communicating Process Architectures 2013*, pages 271–291, 2013.

[51] J. Salamanca, J. N. Amaral, and G. Araujo. Evaluating and improving thread-level speculation in hardware transactional memories. In *IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 586–595, Chicago, USA, 2016.

[52] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[53] A. Shriraman, S. Dwarkadas, and M.L. Scott. Flexible decoupled transactional memory support. In *Intern. Conf. on Computer Architecture (ISCA)*, pages 139–150, Beijing, China, June 2008.

[54] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Intern. Conf. on Computer Architecture (ISCA)*, pages 414–425, S. Margherita Ligure, Italy, 1995.

[55] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *High Performance Computer Architecture (HPCA)*, pages 2–, Washington, DC, USA, 1998.

[56] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Intern. Conf. on Computer Architecture (ISCA)*, pages 1–12, Vancouver, BC, Canada, 2000.

[57] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, August 2005.

[58] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[59] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M Valero. EazyHTM: EAger-LaZY hardware transactional memory. In *Intern. Symp. on Microarchitecture (MICRO)*, pages 145–155, New York, NY, USA, 2009.

[60] J. Torrellas. *Speculation, Thread-Level*, pages 1894–1900. Springer US, Boston, MA, 2011.

[61] J. Torrellas, M.S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, Jun 1994.

[62] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F.P. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Programming Language Design and Implementation (PLDI)*, pages 177–187, Dublin, Ireland, 2009. ACM.

[63] Rajeshwar Vanka and James Tuck. Efficient and accurate data dependence profiling using software signatures. In *Code Generation and Optimization (CGO)*, pages 186–195, San Jose, California, 2012. ACM.

[64] A. Wang, M. Gaudet, P. Wu, M. Ohmacht, J. N. Amaral, C. Barton, R. Silvera, and M. M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 127–136, Minneapolis, MN, USA, September 2012.

[65] A. Wang, M. Gaudet, P. Wu, M. Ohmacht, J. N. Amaral, C. Barton, R. Silvera, and M. M. Michael. Software support and evaluation of hardware transaction memory on Blue Gene/Q. *IEEE Transactions on Computers*, 64(1):233–346, January 2015.

[66] Michael Joseph Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, 1996.

[67] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Intern. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 19:1–19:11, Denver, CO, USA, 2013.

[68] Reginald Clifford Young. *Path-based compilation.* PhD thesis, Harvard University, 1998.

[69] Hongtao Yu and Zhiyuan Li. Fast loop-level data dependence profiling. In *ACM Int. Conf. on Supercomputing (ÌSC)*, pages 37–46, San Servolo Island, Venice, Italy, 2012. ACM.