

**SOFTWARE SIMULTANEOUS MULTITHREADING THROUGH  
COMPILATION**

by

Yuanfang Chen

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Winter 2018

© 2018 Yuanfang Chen  
All Rights Reserved

**SOFTWARE SIMULTANEOUS MULTITHREADING THROUGH  
COMPILATION**

by

Yuanfang Chen

Approved: \_\_\_\_\_  
Mark Mirotznik, Ph.D.  
Acting Chair of the Department of Electrical and Computer Engineering

Approved: \_\_\_\_\_  
Babatunde A. Ogunnaike, Ph.D.  
Dean of the College of Engineering

Approved: \_\_\_\_\_  
Ann L. Ardis, Ph.D.  
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_  
Xiaoming Li, Ph.D.  
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_  
Chengmo Yang, Ph.D.  
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_  
Hui Fang, Ph.D.  
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: \_\_\_\_\_  
Haitao Wei, Ph.D.  
Member of dissertation committee

## ACKNOWLEDGEMENTS

First I want to express my deepest appreciation for my advisor Professor Xiaoming Li. This dissertation would not be possible without his guidance, patience, encouragement and vital suggestions throughout the years. He is always there to help and point me to the right directions when I'm getting lost or stuck on problems.

I would like to thank the dissertation committee members: Professor Hui Fang, Professor Chengmo Yang, Dr. Haitao Wei for their suggestions and comments. A lot of improvements to this dissertation come from their valuable inputs.

I also want to thank my lab colleagues: Sha Li and Shuo Chen for their discussion and comments. It is always joyful and helpful talking to them.

Furthermore, I want to thank my parents for their love and support. Their encouragement carry me this far.

At last, I want to dedicate this dissertation to my daughter Emma.

## TABLE OF CONTENTS

<b>LIST OF TABLES</b> . . . . .	<b>viii</b>
<b>LIST OF FIGURES</b> . . . . .	<b>ix</b>
<b>ABSTRACT</b> . . . . .	<b>xi</b>
 <b>Chapter</b>	
<b>1 INTRODUCTION</b> . . . . .	<b>1</b>
1.1 CPU Inefficiency . . . . .	1
1.2 Toolchain Inefficiency . . . . .	3
1.3 Workload Characteristics . . . . .	3
1.4 Organization of the Dissertation . . . . .	5
<b>2 HARDWARE BASED SMT</b> . . . . .	<b>6</b>
2.1 ILP vs TLP . . . . .	6
2.2 Multithreading Types . . . . .	8
2.3 Simultaneous Multithreading (SMT) Implementations . . . . .	9
2.3.1 IBM POWER . . . . .	11
2.3.2 Intel Hyper-threading . . . . .	12
2.4 SMT Drawbacks . . . . .	13
2.4.1 Big Design Space . . . . .	13
2.4.2 Performance Variation . . . . .	13
2.4.3 Complex Scheduling . . . . .	14
2.4.4 Availability . . . . .	15
<b>3 COMPILER-BASED SOFTWARE LEVEL SMT (CSSMT)</b>	
<b>ORGANIZATION</b> . . . . .	<b>16</b>
3.1 Low Level Virtual Machine (LLVM) . . . . .	19

3.2	Link Time Optimization (LTO)	20
3.3	Cost Model	21
3.4	Limit	22
<b>4</b>	<b>CSSMT PROFILING</b>	<b>23</b>
4.1	Early Abortion	24
4.2	Hot Spot Identification	26
4.3	Implementation	30
4.4	Discussion	33
<b>5</b>	<b>CSSMT RUNTIME</b>	<b>34</b>
5.1	Coroutine	34
5.1.1	Ucontext	39
5.1.2	Fcontext	39
5.2	Input/Output Redirection	40
<b>6</b>	<b>CSSMT COARSE-GRAINED MERGE</b>	<b>43</b>
6.1	The Two Methods	43
6.2	Code Extraction	43
6.3	The Algorithm	44
6.3.1	Ordered Loop Merge	51
6.3.2	Unordered Loop Merge	52
6.3.3	Trip Count Mismatch	52
6.4	OpenMP Support	54
6.4.1	OpenMP Parallel Loop Construct Information Extraction	55
6.4.2	Loop Nest Bypassing	57
<b>7</b>	<b>CSSMT FINE-GRAINED MERGE</b>	<b>58</b>
7.1	Granularity	59
<b>8</b>	<b>CSSMT BUILD SYSTEM INTEGRATION</b>	<b>61</b>
8.1	Profiling Compile Phase (cssmt-prof)	62
8.2	Final Compile/Merge Phase (cssmt <string>)	62

<b>9</b>	<b>EXPERIMENTAL RESULTS . . . . .</b>	<b>64</b>
9.1	X86-64 . . . . .	65
9.1.1	Top-Down Analysis . . . . .	66
9.1.2	NAS Parallel Benchmarks . . . . .	70
9.1.3	SPEC2006 . . . . .	70
9.2	AArch64 . . . . .	72
9.2.1	NAS Parallel Benchmarks . . . . .	74
9.2.2	SPEC2006 . . . . .	74
9.3	Mix SPEC2006 And NPB On X86-64 . . . . .	74
9.4	Conclusion . . . . .	77
<b>10</b>	<b>DISCUSSION . . . . .</b>	<b>78</b>
10.1	Employ Source-to-Source Instead Of IR Transformation? . . . . .	78
10.2	Micro-architecture profiling information Helps Performance? . . . . .	78
10.3	Hot Spot In the Precompiled Library Such As Glibc? . . . . .	79
10.4	How To Merge Recursive Function? . . . . .	79
10.5	Execution Time Imbalance Between Two Threads . . . . .	79
<b>11</b>	<b>RELATED WORK . . . . .</b>	<b>81</b>
<b>12</b>	<b>CONCLUSION AND FUTURE WORK . . . . .</b>	<b>84</b>
	<b>BIBLIOGRAPHY . . . . .</b>	<b>86</b>

## LIST OF TABLES

4.1	X86 Haswell Events Indicating Bottleneck . . . . .	24
5.1	Libc functions accessing standard file and the replacement functions	41
6.1	Important Acronyms . . . . .	54
9.1	Experiment Machine Configuration (X64) . . . . .	66
9.2	This is the caption; This is the second line . . . . .	67
9.3	This is the caption; This is the second line . . . . .	69
9.4	This is the caption; This is the second line . . . . .	71
9.5	Experiment Machine Configuration (AArch64) . . . . .	72
9.6	This is the caption; This is the second line . . . . .	73
9.7	This is the caption; This is the second line . . . . .	75
9.8	Experiment Results on SPEC2006 and NAS Benchmark . . . . .	76

## LIST OF FIGURES

2.1	Comparison: TLP, ILP and ILP+SMT . . . . .	7
2.2	Multithreading Types . . . . .	9
3.1	SMT vs CSSMT . . . . .	17
3.2	CSSMT Workflow (Two-way SMT) . . . . .	18
3.3	LLVM Components [52] . . . . .	19
3.4	LLVM Link Time Optimization [7] . . . . .	21
4.1	Intel TopDown Analysis [78] . . . . .	25
4.2	CSSMT Profiling Workflow . . . . .	27
4.3	<code>perf report</code> with call frames . . . . .	28
4.4	Summarized Binary Profiling Data . . . . .	29
4.5	Summarized Source Profiling Data (Compiler Input/Output) . . . . .	32
5.1	Purpose of Coroutine. Black represents hot spots. Stripe represents cold spots. (a) Original two threads. (b) Both hot&code spots are merged. (c) Hot spots are merged using coroutine. Code spots are intact. . . . .	35
5.2	Coroutine Library Implementation (ucontext version) . . . . .	36
5.3	Coroutine Library Implementation (fcontext version) . . . . .	38
6.1	Example of Code Extraction . . . . .	43
6.2	Illustration of INSERT-BARRIER and CLONE-CALL-PATH . . . . .	45

6.3	Pseudo Code of MERGE . . . . .	46
6.4	Illustration of Lines 2–5 in MERGE-CALL-GRAPH-PATH . . . . .	47
6.5	Example of MERGE-CALL-GRAPH-PATH. Call graph path of host is for1:h1:for2:h2:for3. Call graph path of guest is guest:for4:g1:for5:g2.	47
6.6	Transformation Steps to Merge Outmost Loop of Host and Guest. After these steps, call graph path of host is h1:for2:h2:for3, call graph path of guest is g1:for5:g2. Recursively do these steps for host and guest call graph path to merge for2 with for5, etc.. . . . .	49
6.7	Illustration of Loop Merge: line 9 and line 11 in MERGE-CALL-GRAPH-PATH . . . . .	50
6.8	Coarse-grained Merge OpenMP Workflow . . . . .	53
6.9	Algorithm for Merging two Merge Places (Nomenclature in Table 6.1)	56

## ABSTRACT

With the Dennard Scaling law break for a long time, the computer architecture design progress towards the wider rather than deeper organization. There are three ways to design wider architecture: 1. Putting more cores on the die to utilize thread level parallelism(TLP); 2. Putting more execution ports in the pipeline to utilize instruction level parallelism(ILP); 3. Making vector register wider to utilize data level parallelism(DLP). To speed up a wide spectrum of applications, modern CPU processors usually have all these characteristics at the same time. However, not all applications could make effective use of these characteristics simultaneously. To efficiently use any of these is still a challenging problem in the optimizing compiler research community even though these problems are not new. Processor architect designed simultaneous multithreading (SMT) to alleviate the problem.

Simultaneous multithreading is an essential technique for improving pipeline resource utilization and the overall power efficiency of chips especially when the processor is either wide or comprised of an in-order pipeline. For a wide-issue superscalar processor, there are two kinds of wasted issue slots: vertical waste where all issue slots in a cycle are empty; and horizontal waste where the issue slots in a cycle are partially empty [74]. Simultaneous multithreading, contrary to its other two counterparts: fine-grained multithreading and coarse-grained multithreading, can fill both vertical and horizontal waste, hence enhancing the overall efficiency. From the user applications point of view, there are two ways to improve the speed or the throughput: thread level parallelism (TLP) and instruction level parallelism (ILP). Simultaneous multithreading can exploit both TLP and ILP in the same cycle whereas fine-grained or coarse-grained multithreading can only exploit either TLP or ILP in a single cycle.

Despite all the benefits brought by simultaneous multithreading (SMT), it's adopted by semiconductor chip makers at a slow pace. AMD most recent Zen processor is its first CPU product featuring SMT. The only other well-known chip makers that offer SMT enabled processors are Intel and IBM. The reason for this is that SMT is very complex to implement. Many of the pipeline stages and memory system need hardware logic to have an efficient SMT implementation. For embedded chips, SMT is not even an affordable choice.

To harvest the benefits provided by SMT with incurring significant hardware costs, we propose a Compiler Based SMT implementation framework called CSSMT that achieves comparable performance to hardware-based SMT. With the help of advanced profiling techniques enabled by precise PMU counters in modern CPU, CSSMT can identify those applications that could potentially benefit from SMT and guide our LLVM based compiler to merge the hot spots in respective threads co-running in the same pipeline. CSSMT is orthogonal to the effect of hardware SMT and can bail out when the merging is not profitable based on its cost model derived from profiling data.

# Chapter 1

## INTRODUCTION

Application speedup is achieved by the joint effort of both processors and toolchain software (or toolchain for simplicity in the rest of the thesis) innovations. From the processor side, the state of the art architecture usually has multi-core (instead of many core), SIMD instructions and related wide registers and multi-issue out-of-order superscalar pipeline. Each aspect of the CPU is designed to utilize the specific characteristics of the applications, namely ILP, TLP, DLP. From the toolchain side, speedup can only be realized with the help of the compiler to generate appropriate code for the target machine.

However, both the processors and the toolchain are not perfect.

### 1.1 CPU Inefficiency

For the processors, the biggest inefficiency comes from the imbalance between CPU speed and the DRAM memory speed [76].

While Moore's law is still being kept alive thanks to the technological innovations, more and more cores and various bandwidth-critical components have been integrated into the chip. Limited off-chip memory bandwidth gradually replaces the memory latency or the operation latency as the major performance bottleneck [15, 72]. There are several reasons for that. First, the core is getting more powerful. Faster instruction fetching & decoding and larger out of order execution window such as reorder buffer(ROB) and scheduler demand for more instructions and matching operands being fed to the chip adding memory pressure. Moreover, any wrong branch prediction will lead to pipeline flush and unnecessary off-chip traffic. Similarly, prefetching unwanted

data wastes memory bandwidth and possibly create cache pollution, which in turn, exacerbate bandwidth limitation even more.

There are substantial amount of research effort devoted to solve the bandwidth problem [80, 73, 60, 77, 30]. The problem is tackled from two perspectives: thread level parallelism(TLP) and instruction level parallelism(ILP). On the TLP level, contention among co-scheduled threads for the shared bandwidth affects the execution efficiency of TLP [60, 59]. The solution is to find a way to control the timing of sharing and the partition of the resource. On the ILP level, the bandwidth bottleneck degrades the core performance if there is not enough exposed ILP to hide the extra memory latency due to bandwidth limitation. The solution is to enhance ILP to hide latency and improve data reuse to reduce memory requests.

[Thread Level Parallelism (TLP)] Considering memory bandwidth as a shared resource in the system, the vast majority of previous work [80, 73] have been concentrating on using operating system scheduler to orchestra the memory requests from all threads to achieve QoS, fairness and/or throughput. Usually, the scheduler makes the decision based on three factors: 1) the resource usage pattern of threads, such as memory bandwidth requirement, working set size, cache locality, etc.; 2) resource configuration of the system such as NUMA vs Non-NUMA, SMT or not, if in SMT, is shared cache and TLB statically or dynamically partitioned, etc.; and 3) scheduling goal, such as fairness, QoS, throughput, etc. However, the scheduler is by design a thread level mechanism. While it can be tuned to remove contention among threads, it is impossible for a scheduler to compensate for the loss of ILP due to long latency. The fundamental problem lies in that scheduler can schedule two threads to avoid resource contention but cannot make them collaborate even if they have no resource sharing. Imagine one thread keeps sending memory requests, having all ALU ports idle and the other retains all of its execution ports busy, having perfect cache locality and hardly any last- level cache miss. To achieve optimal workload performance, its best to pick one instruction from each thread and run instructions concurrently on the same core. However, this kind of collaboration is too fine-grained for scheduling methods.

[Instruction Level Parallelism (ILP)] There are two ILP ways to mitigate bandwidth problem. One way is to eliminate unnecessary memory accesses by enhancing data reuse. [29] uses loop-fusion to expose more data-reuse across loop boundary to save memory bandwidth consumption. [30] proposes to extend the data reuse scope to the whole program(single thread) by compiler optimization. The other way is to increase ILP to hide latency. [19] uses unroll-and-jam to transform irregular control-flow and data flow structure into ILP. [17, 18] use various loop transformations to break loop-carry data dependency, increasing ILP scope.

However, as mentioned in [75], there is a limit on how much ILP we could extract from the whole program. Also, the amount of memory space used by a program depends on the characteristics of the program itself. Even if every memory unit is accessed only once during the entire program execution, the bandwidth could still be saturated if the memory space accessed is large and the access frequency is high.

## 1.2 Toolchain Inefficiency

For the toolchain, specifically the compiler, it usually takes many man years to develop the necessary optimization and code generation to support the hardware features that are already available a few years ago. One primary example is that Intel introduced MMX technology at the year of 1995 as the first generation of SIMD instruction on mainstream chips, and later AMD, IBM, etc. all phased out the similar instruction set to accelerate SIMD heavy workloads. However the corresponding compiler optimization: automatic vectorization is not mature enough until this day [58].

## 1.3 Workload Characteristics

Another inhibiting issue is that even if there is a perfect compiler that could extract all the available ILP, TLP and DLP from the source program and the processors are perfect at executing these parallelism to the appropriate part of chips, then the

available ILP, TLP and DLP inherently inside of the source program becomes another limit of how much speedup could be achieved.

The processor's speed imbalance among core and memory, slow toolchain development and the limited ILP, TLP, DLP of workloads require some degree of flexibility to transfer hardware resources among ILP, TLP and DLP to achieve better resource utilization.

Simultaneous multithreading (SMT) was the answer from hardware side to solve the problem of converting TLP to ILP[57]. It allows multiple threads of execution to share the hardware resources in the same core. When the workload has long data-dependency chain, large working set or frequent mis-prediction, it means it has little ILP and there are many gaps/bubbles in the pipeline. Hardware SMT allows other threads to run on the same pipeline (TLP) to make use of the wasted cycles (ILP). (The other direction of SMT is converting ILP to TLP is parallelization. In hardware, it is achieved by putting more cores on the same die(CMP). In software, it requires advanced compiler technology auto-parallelization, which could mean extracting task level parallelism from ILP, then spreading them on each core.)

Although SMT is a very effective way to increase throughput and energy efficiency, SMT costs precious die area, adds nontrivial complexity to the core design and more importantly it does not work equally well for all applications. For some application, it may even cause performance degradation. A flexible way is desired to compose program performance pattern to accommodate hardware configuration. Specifically, we propose a compiler-based software level SMT implementation called CSSMT as the software answer to the problem of converting TLP to ILP. It addresses the limits of the hardware SMT implementation. For machines without SMT support, CSSMT works as the replacement; for machines with SMT support, CSSMT keeps orthogonal to the effects of hardware SMT. More importantly, the implementation techniques of CSSMT is immediately useful in the direction of composable software performance, which we believe is equally important in the future as hardware innovation. The testing results of applying CSSMT to NAS [8] and SPEC2006 benchmarks [41] on both Intel X64 and

ARMv8 architecture are very promising considering the profiling information is still constrained by the inaccuracy of compiler generated debug information [2].

## 1.4 Organization of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 introduces the theoretical background of simultaneous multithreading and several mainstream implementations from major chip vendors such as Intel, IBM and AMD. Also, the drawbacks of the hardware based SMT are explained. Chapter 3 gives an overview of the CSSMT framework. Since CSSMT is implemented in an industrial strength compiler infrastructure LLVM, chapter 3 also gives a brief introduction of LLVM components that influence the CSSMT implementation. Then it lists the comprising modules of CSSMT and their respective functionality. Chapter 4, 5,6 and 7 present the profiling steps, the runtime module, the coarse-grained merge and the fine-grained merge steps of CSSMT. Related algorithms are explained. Chapter 8 describe the method to integrate CSSMT into build systems such as Makefile or CMake. Chapter 9 presents the result and correlates it with our expectation. Chapter 10 describes design decisions that have been made for CSSMT and their pros and cons. Chapter 11 discusses related work and then chapter 12 concludes the dissertation.

## Chapter 2

### HARDWARE BASED SMT

The arise of the simultaneous multithreading (SMT) is attributed to the abundance of TLP and the limit of ILP [75]. It is one of the earliest successful attempt to reconfigurable and flexible architecture due to the enormous differences among modern workloads [33]. At the beginning of this chapter, a comparison and contrast of ILP and TLP are presented. Then three kinds of multithreading organization are discussed and compared. At the last, real-world SMT implementations from IBM, Intel and Oracle are described. Their drawbacks are noted, which lead to the core of this thesis: a software compiler assisted SMT implementation (CSSMT).

#### 2.1 ILP vs TLP

Instruction level parallelism (ILP) and thread level parallelism (TLP) are the primary forms of parallelism computer architecture expect to extract. A quick comparison of TLP and ILP is shown in figure 2.1. In TLP, the workload is separated into two independent thread of work and scheduled separately onto different cores; on the other hand, ILP issue dependency free instructions in the cycle to efficiently utilize the wide-issue pipeline. They each represent distinct types of architectural organization and each only satisfy the kinds of workloads they are suitable for. For the workloads showing unfitting performance characteristics, both TLP and ILP structure are not productive. In figure 2.1, there are ample amount of stalled cycles could have been exploited in both TLP and ILP. However, the hardware restricts the kind of parallelism it chooses to implement. There is no way to either converting ILP to TLP or converting TLP to ILP.

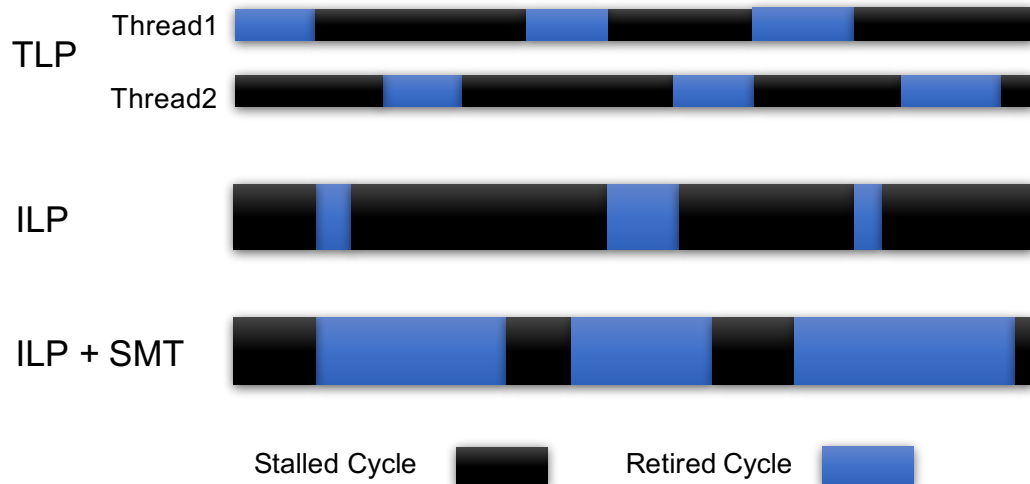


Figure 2.1: Comparison: TLP, ILP and ILP+SMT

To increase the TLP performance chip-multiprocessing (CMP) were proposed. It also maintains the same ILP performance. Multi-core, being one kind of CMP, gave birth to a plethora of traditional workloads employing new programming model such as Cilk [11], OpenMP [25] and Chapel [16] etc.. On the other hand, emerging workloads in areas such as graph analytics, web search and media streaming etc. pose new challenge for the computer architecture to cater to their distinct performance pattern. [33] found that some applications contain up to a factor of 929X more ILP than what is currently being extracted from real machines. Further [75],[33] have shown that there are limited ILP and TLP that can be extracted from programs. And different workloads exhibit different ILP and TLP ratio and interaction, which calls for flexible architectural design to accommodate this requirement. Thus multi-core is not sufficient regarding performance and energy ratio for these new workloads. When a workload is heavily dominated by TLP, then each core is severely underutilized causing inefficiency. The more cores there are on die, the more multi-core may suffer from TLP and ILP imbalance of workloads.

## 2.2 Multithreading Types

Multithreading was developed as a way to increase processor energy efficiency and resource utilization. In a traditional wide super-scalar processor pipeline, there are two kinds of waste contributing the inefficiency of the pipeline: vertical cycle waste and horizontal cycle waste [74]. Horizontal waste is the empty issue slots in a partially occupied cycle. Vertical waste is the cycle where all issuing slots are empty. As shown in figure 2.2, multithreading could only fill the vertical waste. The horizontal waste is still under-utilized. Fine-grained multithreading architectures such as Tera [4] and MASA [40] could issue instructions from different threads or processors in each cycle. Since instructions from different threads are inherently free of control and data dependence, the wide pipeline is more likely to be employed. Coarse-grained multithreading architecture such as APRIL [3] is less flexible where it could only switch threads in multiple cycles.

Simultaneous multithreading [74] is different from coarse-grained and fine-grained multithreading in that it could fill both vertical and horizontal waste cycles in a wide super-scalar processor pipeline whereas coarse-grained and fine-grained multithreading can only tackle vertical waste [74]. When a processor is running in simultaneous multithreading mode, if one of the co-running thread incurs excessive memory or pipeline latency, the other simultaneous threads can still issue instructions to the otherwise empty issue slots. The overall processor utilization is thus improved as shown in the rightmost sub-graph in figure 2.2.

Simultaneous multithreading also distinguishes itself from coarse-grained and fine-grained multithreading in the way it issues instructions to the wide-issue pipeline. In coarse-grained multithreading and fine-grained multithreading, when the pipeline finds out that one thread is stalling due long latency operations, it stops the currently running thread and then schedules another thread to issue instructions to fill the otherwise empty issue cycles. The drawback of this design is that there is thread switching event that happens at a cost. Also when the thread switching has to happen, the pipeline needs to save the currently running context of one thread and restore another

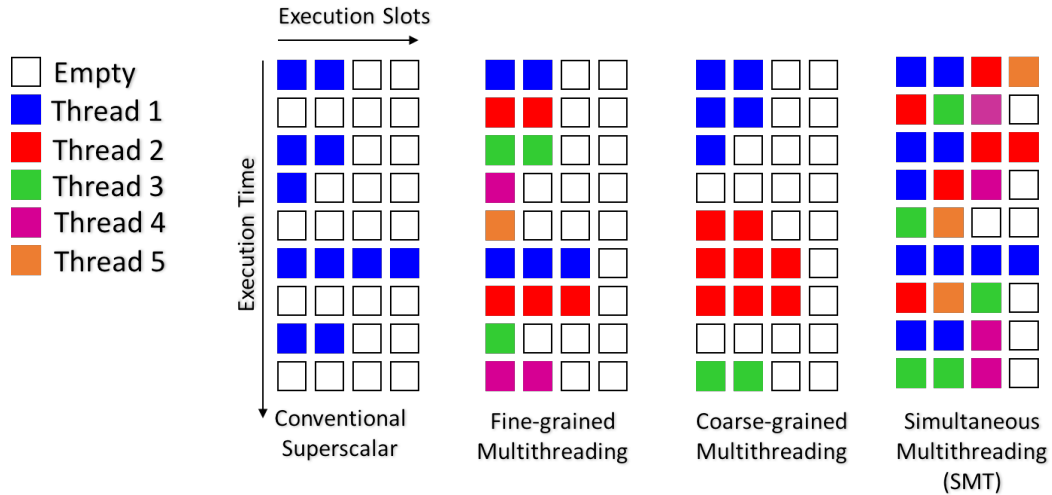


Figure 2.2: Multithreading Types

thread. However, the thread state saving is very tricky for out of order pipeline. Often time the instruction later in program order may have already finished execution, but earlier instructions are still on the fly. On the contrary, there is no thread switching event and cost in a simultaneous multithreading implementation. All selected threads could issue instructions to the same cycle at the same time [61].

### 2.3 Simultaneous Multithreading (SMT) Implementations

Simultaneous multithreading (SMT) [74] was proposed to boost the pipeline utilization significantly. It allows more than one threads to issue instructions to the same pipeline at the same time. Thus when one of the thread is waiting on long latency operations, the other thread could still make progress with the unused resources. The idea is to bring some degree of flexibility in the hardware that can speed up both heavily single-thread application and light multi-threaded application. All existing SMT implementations are implemented in a chip multiprocessor to take advantage of the applications thread-level parallelism.

Intel claims up to 30% performance boost observed for their two way SMT implementation called Hyper-Threading [50]. IBM goes a step further. Their recent

server processor offering POWER8 implements an eight-way SMT where they see more than twice performance increase over their Intel counterpart [69, 36].

There are many elements to a real-world SMT implementations.

First, the number of sharing threads (also called the way of SMT) could be static or dynamic. For the static implementation, the hardware decides how many threads can issue to pipeline in the design time. Neither firmware nor software options is available to the change this attribute (Intel, AMD, IBM). Whereas for the dynamic implementation, the way of SMT could be changed at machine running time (IBM).

Second, the number of SMT ways are either a fixed number or some range. Intel and AMD use two-way SMT predominately in their chips. One well-known exception is Xeon Phi, which features four-way SMT. IBM initially uses two-way SMT in POWER5 and POWER6, later change to use four-way on POWER7, eight-way on POWER8 and POWER9. And on POWER processor, eight-way SMT can run in four-way SMT mode where each thread has twice the amount of hardware resource than eight-way or two-way SMT where each thread has four times the hardware resource of eight-way SMT thread. On POWER, the mode used could be controlled by software, which allows a lot of software flexibility to tune the performance depending on the specific workloads.

Third, the resource allocation policy of SMT thread can be either static or dynamic. Static resources sharing hard wire the amount of resources dedicated to each SMT thread, whereas dynamic resources sharing allows software to tune the amount of resources dedicated to each SMT thread to achieve better performance. Static resources allocation is relatively easy to implement but picky on the combination of workloads that could benefit from SMT. Dynamic resources allocation allows a wider range of workloads to achieve speedup since SMT threads have more chance of possessing proper amount of resources to make progress. However dynamic resource allocation requires complicated design and verification.

Last but not least, depending on the attributes of the specific architecture, the thread resource allocation of the core must make sure that any stalled thread could not block other threads to make forward progress. In all these real-world SMT

implementations, there are fare sharing or monopoly detection mechanism that prevent thread resource starvation from happening [50, 68, 53, 67].

Intel X86 and IBM POWER are two different ways to implement SMT. Intel, being a major CPU vendor for both consumer and business market, need a cost-effective design of SMT to accommodate both regular user application and warehouse-scale server workload. A balanced two-way SMT setting is the result of such trade-off. Because IBM POWER processors only need to accelerate commercial workload, it could adopt more aggressive and more expensive SMT design. Here gives a gentle description of both to show the design choice of SMT and the resulting performance difference.

### 2.3.1 IBM POWER

POWER processors [68, 53, 67, 69, 64] have been adopting SMT since POWER5. POWER5 have a few key SMT features that have been improved over the time until the most recent POWER9. POWER's SMT implementation allows dynamically switching between single-thread mode and SMT mode. It is initiated by the software to solve the resource sharing/contention problem caused by SMT.

There are three resources related modifications to support two-way SMT on POWER5 [68, 47, 63].

1. **Increasing resource availability.** Many resources are adjusted to make resource fairly available to both threads in a minimal budget. L1D and L1I have their associativity doubled. L1 TLB, L1 SLB and LMQ are tagged with thread id. The LRQ, SRQ and BIQ are partitioned into half of their original size.
2. **Dynamic Fair sharing.** The core is equipped with advanced monitoring facility to watch the progress of each thread. If any thread is hogging too much resource such that other threads are not able to make forward progress, the pipeline will apply throttling automatically.

3. **Software Thread Priority.** Both operating system and the user-level application could adjust the eight level priority of threads. Some of the priority levels are privileged for the operating system. For example, the operating system may choose to use thread priority to throttle specific thread to achieve fair sharing. Operating system scheduler is another potential user of thread priority [32, 12, 34]. Applications may use thread priority to implement quality of service (QoS).

POWER6's SMT is roughly the same as that of POWER5. POWER7 [67] settles on a more flexible way of arranging SMT threads. Its core is physically four-way SMT. However, it supports switching among single-thread, two-way and four-way SMT modes. POWER7 core supports two thread by one register file, and there are two register files. In two-way SMT mode, two register files have the same content. The best model to use depends on the workload's resource utilization patterns.

POWER8 [69] pushes the SMT implementation to a new level. With its doubled L1D, L2 and L3 cache, POWER8 can support up to eight-way SMT. Like POWER7, there are also single-thread, two-way and four-way modes to use depending on the scenario. One new SMT feature POWER8 introduced is to dynamically change SMT mode among ST, SMT2, SMT4 and SMT8. For example, as long as there is only one thread on running on the pipeline, no matter which thread context it is using out of the total eight contexts, the pipeline can put the pipeline into single-thread mode allowing the thread to exploit all resources in the pipeline. This flexibility frees the application from manually detecting and switching SMT modes, gaining better performance.

### 2.3.2 Intel Hyper-threading

Intel's proprietary SMT implementation is called Hyper-threading [42]. While IBM is aggressive on its SMT implementation on each generation of POWER processor, Intel adopted a conservative approach. Since Intel begin to support hyper-threading on Northwood-based Pentium 4, Hyper-threading is mostly two-way SMT. No dynamic SMT mode switching. One highlight of hyper-threading is that it only added %5 more

chip size and power requirement but achieved much more performance increase than that. IBM POWER's cost is much higher. But its benefits are also much more than hyper-threading [36].

## **2.4 SMT Drawbacks**

While SMT could improve whole system throughput significantly in a lot of situations, SMT makes the chip design much more complicated. A lot of pipeline components need to competitively shared by co-running applications, sometimes with priority and some other metrics that decides with thread should be allocated more hardware resource. To achieve a design that has well-balanced performance, throughput and power consumption is a challenge for chip designer. On the software side, application and programmer need to be SMT-aware to utilize its potential benefits successfully.

### **2.4.1 Big Design Space**

As shown in section 2.3.1, the design space of advanced SMT implementation can be huge. That not only cost significant engineering effort, also it makes the design trade-offs hard if the targeted workloads showing a wide spectrum of resource utilization patterns. Non trivial SMT implementation also put substantial pressure on chip validation effort [14]. Hardware SMT bugs are hard to fix. Sometimes a correctness fix entails performance degradation.

### **2.4.2 Performance Variation**

The performance boost is not guaranteed under SMT. Each chip vendor has their proprietary SMT design and the target workload the design is for. The same application or application mix working well in processors of one vendor does not imply the similar performance boost in processors of another vendor. Depending on the SMT configuration, the same argument goes true even for different generations of processors from the same vendor. Vendors like IBM make radical changes to their SMT implementation for each processor generation. It is still user's burden to tune the performance

for the new processor. The operating systems also have different scheduling policies for allocating SMT threads on the same core. Often time the scheduler needs to consult a performance monitoring unit (PMU) about the resource utilization pattern of workload to make scheduling decisions. So the precision of the performance data collected by PMU also affects the SMT workload indirectly. Furthermore, compiler optimization is the direct producer of workload binary executable. The optimizations it applies dictate the code generation and hence the resource utilization patterns. Last but not least, the workload performance characteristics are decided by the problem domain. Artificial intelligence workloads inherently use a tremendously different set of resources that the high-performance computing workload may need.

### 2.4.3 Complex Scheduling

Scheduling of SMT threads to achieve optimal fairness or throughput can be done either in processors or the operating system scheduler. Either way, the scheduler needs to understand the performance results of thread mixes. Then a resource reallocation method is applied to fulfill the purpose. The reallocation methods include starving threads occupying disproportionate amount of resource or binding the thread to another core to prevent if taking resources from other threads. The scheduling purpose (fairness, throughput or both), performance feedback mechanism, the interplay of SMT hardware the co-scheduled SMT threads and phase-behavior of workload, all are the challenging aspects of the SMT threads scheduling. There are many works in this front ([20, 71, 23, 1, 32, 34]).

Among others, [20] makes use of cache behavior to classify threads as slow or fast and use several processor resources to classify each thread as active or inactive. It decides if throttling SMT thread instruction fetch by looking at the dynamic classification of threads. [71] use performance monitoring unit (PMU) to indirectly estimate the dynamic performance patterns of workload, then feed the information back to the scheduler to make scheduling decisions. [23] use instructions per cycle (IPC) as a direct indicator of thread performance. Through an online learning process to predict

runtime threads colocation. [34] is similar to [71] whereas the technique is specifically applied for IBM POWER8.

#### **2.4.4 Availability**

Intel is widely available, but its design is restrictive. It is only two-way SMT. For threads with severe resource under-utilization issue, more ways are needed. There is no thread priority. Thread starvation or performance degradation could easily happen without the careful choice of co-scheduled threads. And single-thread and two-way SMT mode are not dynamically changeable.

IBM POWER processors target high-end business level workload and services. They are not commonly accessible for regular users.

## Chapter 3

### COMPILER-BASED SOFTWARE LEVEL SMT (CSSMT) ORGANIZATION

Observing the inefficiency of hardware-based SMT discussed in chapter 2, we propose Compiler-based Software level SMT (CSSMT) as an alternative to help throughput computing. As shown in figure 3.1, under CSSMT, the pipeline only see one mixed and optimized thread instead of two independent threads. This help to achieve the same effect of SMT without the actual hardware support.

Figure 3.2 gives the CSSMT workflow for two way SMT. Higher ways of CSSMT workflow are implemented as repetitive applications of two-way CSSMT.

To materialize the effect of SMT, instructions from both threads need to be very close to each other in the binary format so that in the running time, the processor core dispatches these independent instructions from both threads in either same cycle or consecutive cycles. The exact binary generated in turn depends on the compiler code generation, to be specific, instruction selection and instruction scheduling, etc.. The idea of CSSMT is first to transfer the hot spot of one thread to the level of a loop where hot spot of the other thread resides. We call this *Coarse-grained Merge* (up right green box in figure 3.2) since after which, instruction scheduling may still not be able to schedule instructions of both threads at the same time. Then the compiler transforms the coarsely fused code into a fine-grained format in a way that is explicitly specified by the programmer through compiler option. We call this *Fine-grained Merge* (lower left green box in figure 3.2). The purpose is to accommodate the particular instruction scheduling strategy used by the compiler. If the compiler uses local scheduling (for example, LLVM), the transformation must make sure that hot spots from both threads reside in the same basic blocks for the pipeline to issue

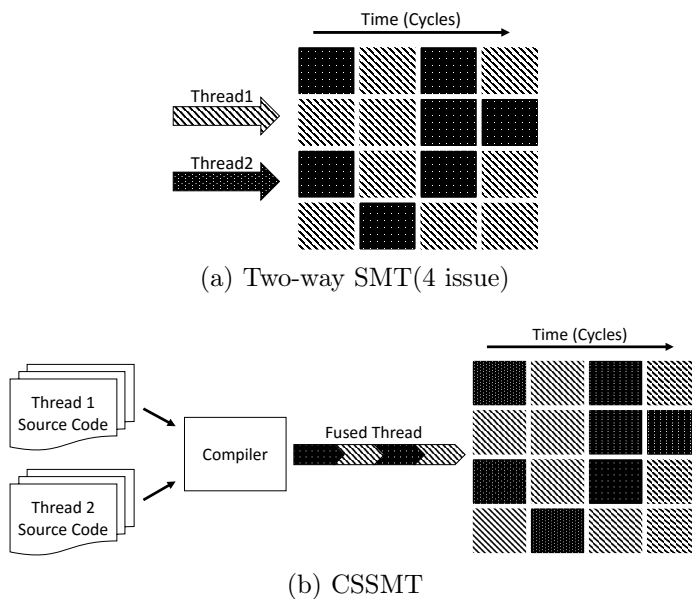


Figure 3.1: SMT vs CSSMT

them together. If the compiler uses scheduling techniques that have a larger scope than a basic block, such as a function(global scheduling) or a trace(trace scheduling), the transformation could simply merge hot spots along the hot code path of one thread. Then it is the processor’s job to issue instructions to same or consecutive cycles by using out-of-order execution and branch prediction etc.. Hot spots information is gathered using sampling based profiling provided by Linux Perf tool [27].

Since CSSMT merging algorithm needs to have the global view of the program, CSSMT can be considered an inter-module analysis and optimization. The CSSMT merging mechanism is invoked at the link time with the help of LLVM LTO [56] to have the whole program level view. In LTO, sources of both programs are first compiled into object files with a special section that contains the LLVM IR for that module. And then in the linking step, the linker extract the LLVM IR from the object file, calling the LLVM LTO component to report symbols and invoking target code generation. When CSSMT is enabled for LLVM, the LTO process will dump the whole program IR, and the linker would save to a file the linked static and dynamic libraries specified as positional arguments to the linker. We need to specify these libraries as input to

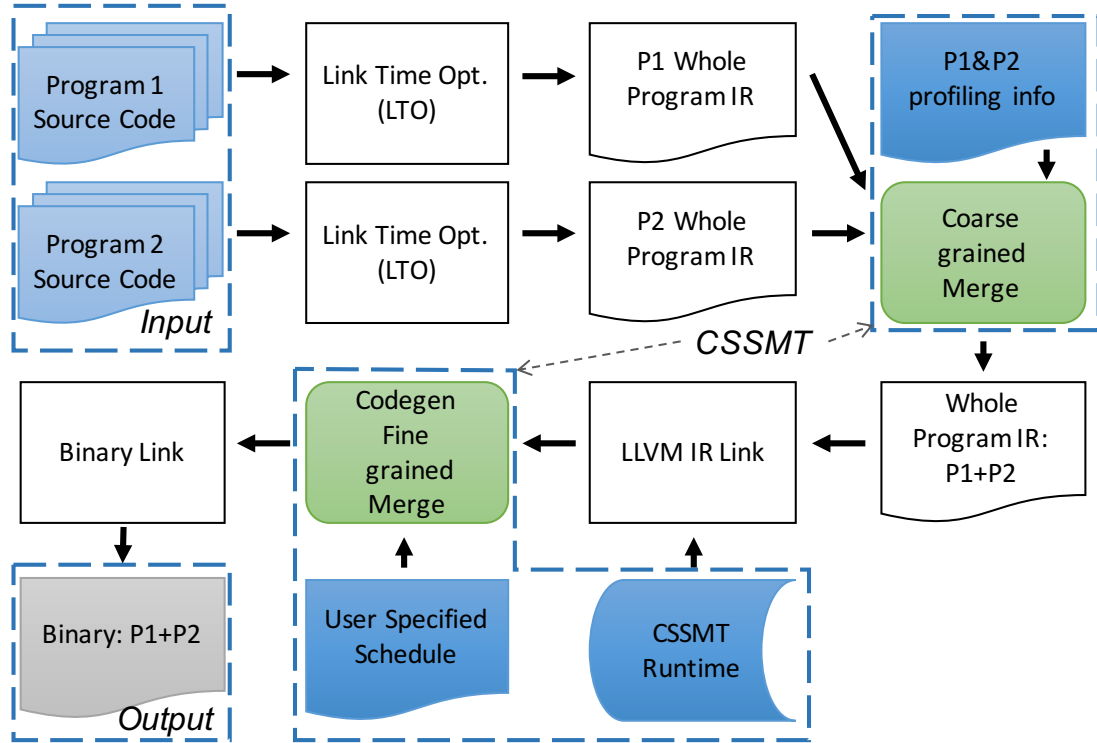


Figure 3.2: CSSMT Workflow (Two-way SMT)

the "Binary Link" step in figure 3.2 to correctly generate SMT'd executable.

CSSMT is implemented around the LLVM compiler infrastructure [51]. Workflow steps that happen in LLVM are represented as white boxes in figure 3.2. The green boxes are CSSMT components implemented as LLVM IR/MIR passes. The intense blue boxes are CSSMT components as input to the CSSMT compilation process. The "profiling info" input is generated by sampling based profiler offline using training input (Chapter 4). It is used to guide the "Coarse-grained Merge" step to merge only hot spot source location that contributes to the majority of the running time. The "Runtime" is set of library functions called by CSSMT during SMT'd executable is running (Chapter 5). "User Specified Schedule" allows the programmer to control fine-grained merge mode with regard to the cache behavior since there are two general ways to merge two sequential codes. One way is to perfectly mix them such that processors are more likely to schedule them together considering they are free of dependency. The other way is to mix instructions from both programs to the extent that their respective

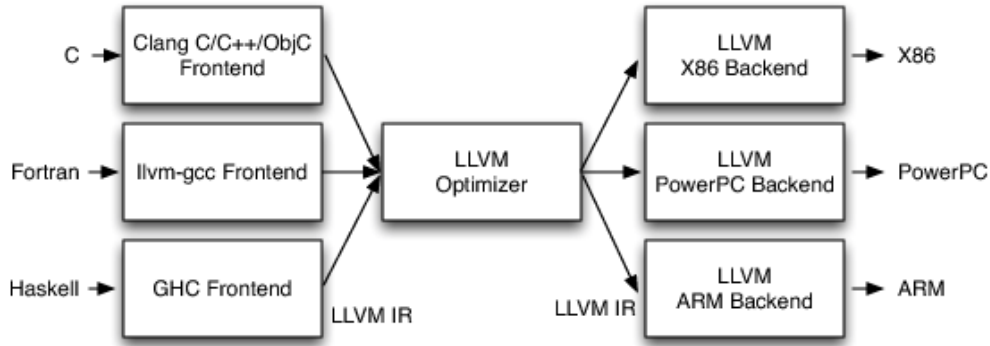


Figure 3.3: LLVM Components [52]

cache locality is preserved.

This chapter is organized as follows. Since CSSMT is heavily dependent on LLVM for its implementation, this chapter begins with an overview of LLVM structure. Then it describes how LLVM LTO interacts with CSSMT. After that, CSSMT cost model is presented. Unlike hardware SMT, CSSMT can make better analysis and transformation decisions with both static and dynamic program behavior information. The end of the chapter discusses the restrictions of CSSMT and the reason behind it.

### 3.1 Low Level Virtual Machine (LLVM)

LLVM features a expressive intermediate representations (IR) such that all target independent optimizations could be intuitively applied (Enable CSSMT to be useful for all the programming languages that could target LLVM IR). And the same IR could be lowered to different target platforms (CSSMT fine-grained merge step need to be target-aware to generate performant code). The IR is also the representation data for each object file when LTO is enabled at link time. The IR is also the used as input to the LLVM just-in-time compilation (JIT) infrastructure (makes CSSMT also applicable to dynamic programming languages). The LLVM IR is in static single assignment form (SSA) form, meaning that every value is defined once and used several times. SSA helps optimization passes to reason about the program semantics and benefits CSSMT equally. The other nice feature of LLVM IR is that it is a complete representation of

source program, rather than the GCC's GIMPLE IR. LLVM IR makes CSSMT have a monolithic and reliable implementation.

### 3.2 Link Time Optimization (LTO)

The linker has access to all comprising object files and libraries during the program building process. The other phases of building cycle only have the view of a single source file. LTO makes link time the only opportunity to apply optimizations that require inter-module information such as call graph etc.. So link time optimization is essential for the generated code quality.

LLVM LTO module is a shared library called libLTO. libLTO defines a set of public APIs that would be called from a system linker during several critical linking stages. On Linux, the system linker gold provides interfaces for LTO, so there is a thin layer in LLVM to bridge the gold interfaces and the interfaces LLVM defined.

To use LLVM link time optimization, the user only needs to specify `-flto` at both compile and link time (figure 3.4). Specifying `-flto` flag at compile time means there will be special ELF section in output binary to store the LLVM IR of the source module. Specifying `-flto` flag at link time implies linker will be able to query libLTO if a file is an object file or of the format only the libLTO understands. If libLTO understands the file format, the linker will collect the symbols and associated attributes from libLTO to perform symbol resolution. Then the linker asks libLTO to do native code generation and return the generated object to link with the rest of the program object files.

CSSMT integrates naturally with the LLVM LTO framework. CSSMT is a whole program transformation. It needs access to each function in a program to merge its usually interprocedural loop nest. Coroutines(section 5.1) may help to relax this constraint, but not always. For example, in Figure 6.2, hot spot is in  $F5$ , but merging needs to start from  $F3$ . LTO support in LLVM could generate whole program IR without changing the existing build system of the program. Enabling LTO is as simple as adding compiling and linking flag to the build system.

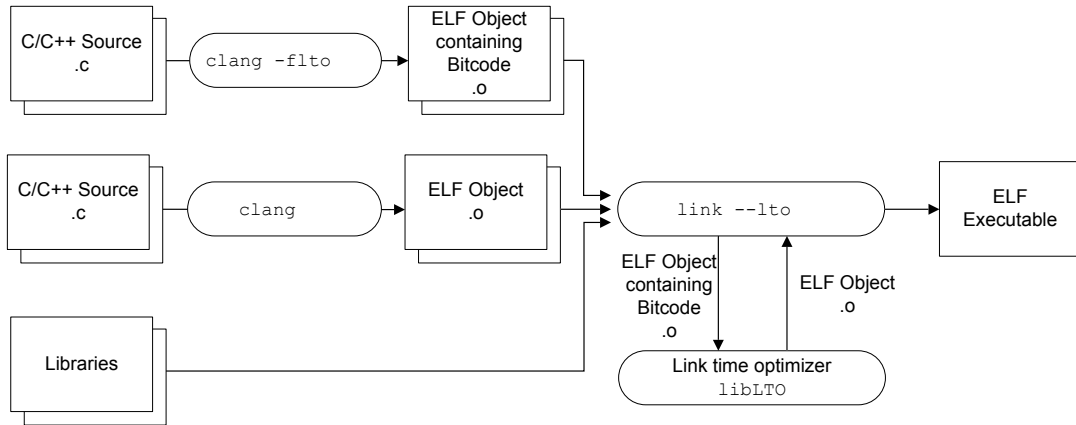


Figure 3.4: LLVM Link Time Optimization [7]

The whole program scope requirement of CSSMT also excludes the use of a source to source transformation tool, since they lack linking capability.

### 3.3 Cost Model

There are two reasons for the necessity of a cost model. First CSSMT is a compiler transformation causing extra code instrumented into the original program. If we straightforwardly apply the CSSMT merging algorithm without considering the overhead, it maybe the case that SMT'd executable performs worse than the original executable. How much overhead is needed to use CSSMT is measured at compile time after analyzing the original program, which is static information.

The other reason is that CSSMT is a feedback-directed compiler optimization. It could make better transformation decision with the workload running time information. Since a significant part of the SMT's effectiveness is decided by the running time characteristics the workloads. For workload that is not suitable to run under SMT, the running time performance would be worse if applying SMT. Thus, a cost model could help CSSMT determine if it is profitable to employ CSSMT or not. If not, bailing out is a better choice rather than making the performance worse.

The detailed cost model metrics are described in chapter 4 and chapter 6.

### 3.4 Limit

The technique described in this chapter only applies to two-way SMT. However, SMT of more than two-way can be implemented by applying this method repetitively. In this two-way SMT scenario, the thread that keeps its `main` function is called host thread(HT); the other thread is called guest thread(GT). Guest thread has its `main` function renamed to `main2` during transformation since each program can only have one main entry function.

In hardware SMT, instructions from both threads are issued into same cycles by the processor core. In contrast, in CSSMT, co-issuing is realized by compiler code generation mixing instructions from both threads close enough that processor core could issue them together. So the crucial part of this work is to bring instructions from both threads as close as possible such that processor co-issuing is possible. At the same time, care must be taken to avoid breaking their respective intra-thread data and control dependence, and also the transformation incurred running time overhead should be at little as possible. Otherwise, either the correctness of the original programs are violated, and the benefits of CSSMT may be contradicted by its overhead. For the above reasons, CSSMT is implementing a limited scope SMT in that only code from hot spot are considered. At this moment, CSSMT is useful for programs with stable regions of hot spots. To handle programs with graph-based algorithms which usually possesses no obvious hot spots, a full scope SMT is required. To the best of our knowledge, implementing a full scope SMT is impractical in that the actual code path in running time is unpredictable. To make full scope SMT work, for each possible code path of one thread, there must be instructions from the other thread to go with it. And this would cause massive overhead and code duplication. Considering these constraints, CSSMT limit the SMT scope to the dominating hot spots of both threads.

## Chapter 4

### CSSMT PROFILING

There are two commonly used methods to profile an workload.

**Instrumentation.** The first is instrumentation based profiling [9, 10]. It works by statically inserting counters at carefully chosen places along paths on the control flow graph to capture the path execution frequency for later use by profile guided optimization (PGO) or software coverage test and so forth. Instrumentation profiling is precise regarding the counters it collected. So it is very suitable for software coverage test. However, it is less than ideal for PGO since the instrumentation overhead may cause imprecise hot spot location to be reported. Moreover, counters in general are not good indicators of where a program spends most of its running time.

**Sampling.** The other method to profile a workload is sampling. Sampling-based profiling requires the processor supporting performance monitoring unit (PMU). The PMU usually are comprised of a set of hardware event counters and a mechanism either to sample those counters at a fixed time interval or to have the counters to trigger an interrupt for a fixed number of counter increase. Either way, the program counter is recorded to correlate the interrupt event with the exact source location where the event happens. Depending on the hardware support, the call stack or last few branch instruction can also be recorded to give more context information for afterward analysis.

Sampling-based profiling has zero static time overhead and negligible running time overhead. Moreover, it requires no recompilation of the program, reducing the deploy costs. The downside of sampling based profiling is that it is not precise at both instruction and basic block level although prior work has mitigated the precision issue to great extent [22].

Table 4.1: X86 Haswell Events Indicating Bottleneck

Event Name	Description
UOPS_EXECUTED_PORT.PORT_(0-8)	Cycles per thread when uops are executed in port
RESOURCE_STALLS.ANY	Resource-related stall cycles
RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available
RESOURCE_STALLS.SB	This event counts cycles during which no instructions were allocated because no Store Buffers were available
RESOURCE_STALLS.ROB	Cycles stalled due to reorder buffer full

**CSSMT uses sampling based profiling.** CSSMT needs profiling information to serve two purposes. The first purpose is to decide at compile time if it is beneficial to apply CSSMT. If the profiling information shows that either host or guest program have apparent architectural bottlenecks, CSSMT will choose to abort the operation since when two programs compete for the same fully utilized resources, neither can make forward progress to make SMT beneficial. Due to the lack of relevant PMU support for resource stall related hardware events on ARMv8 processors, CSSMT supports early abortion on X86 Haswell architecture and later. The other purpose to use profiling information is to identify profitable source locations in host and guest program to apply coarse-grained and fine-grained merge. Since the merge step incurs instrumentation overhead, the source locations must be potentially rewarding for SMT.

In conclusion, only sampling based profiling has the two properties CSSMT looking for: **early abortion** and **precise hot spot identification**. Instrumentation based profiling may hold the second property to a certain extent but not the first one since it does not have access to the hardware events counters.

#### 4.1 Early Abortion

**Early abortion** is required for the fact that even hardware-based SMT is not always beneficial for performance provided the mixed threads are not chosen properly.

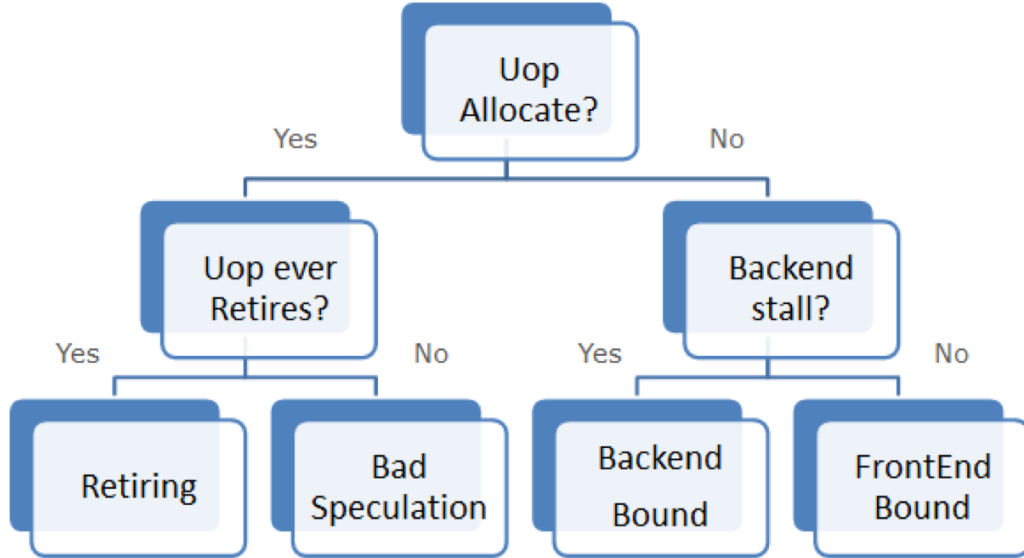


Figure 4.1: Intel TopDown Analysis [78]

There are numerous prior works on this front [32, 34]. However, most of these works are used in the program execution time through the operating system scheduler, and advanced hardware support is needed to make intelligent decisions.

The PGO based CSSMT implementation prevent the use of any of these existing methods. Also, the CSSMT is designed to be useful in a wide variety of platforms where there may lack advanced online monitoring facility in the operating system.

To that end, CSSMT adopts two methods in the order of reliability for **early abortion**. The first method used is Intel top-down analysis (figure 4.1). It is an Intel officially supported performance modeling method to identify bottlenecks. In this analysis, there are four major areas where cycles are spent: 1. **Frontend Bound**; 2. **Bad Speculation**; 3. **Retiring** (good metric); 4. **Backend Bound**. Each of these four metrics is represented as the percentage of the total number of running time cycles. The combined integer value of these metrics is %100. If either **Frontend Bound** or **Backend Bound** has a dominating value (for instance, %70), CSSMT early abortion is invoked. The **Bad Speculation** being not considered bottleneck for SMT is rather counterintuitive. It is because bad speculated instruction stream tends to have more

bubbles in the pipeline. Thus more likely to gain pipeline throughput increase. The `Retiring` is a good sign that issue uops are predominately successfully retired. If another thread of execution was scheduled on the same core, performance degradation is highly likely.

In situations where these metrics are well balanced, CSSMT would resort to the second method: looking at performance events directly. If any of the events listed in table 4.1 is identified as a bottleneck in sampling-based profiling, CSSMT makes the early abortion decision.

The first entry in table 4.1 counters the number of cycles uops in ALU port. A high percentage of this number out of the total number of cycles shows that the ALU is fully utilized. If we still apply CSSMT in this situation, the benefit would be minimal since there exist no extra cycles in ALU ports for the guest program to make any progress. The guest program would compete with the host program for ALU throughout the execution time, making the performance worse. The rest of the entries in table 4.1 are associated with resource related stall cycles. If any of these resources is already a bottleneck in the host program, the SMT'd program is less likely to benefit from CSSMT.

## 4.2 Hot Spot Identification

If CSSMT decide in the early abortion phase that applying transformation is profitable, it begins to identify where in the host and guest program the merging algorithm should be applied to achieve expected speedup. This phase is called hot spot identification.

In hot spot identification, CSSMT needs to know where to apply the coarse-grained and fine-grained merge before actually doing the transformation. So profiling the workload beforehand with training inputs to find the hot spots is very important. Furthermore, the precision of the exact location of hot spots dramatically affects how much benefit we could gain by using CSSMT.

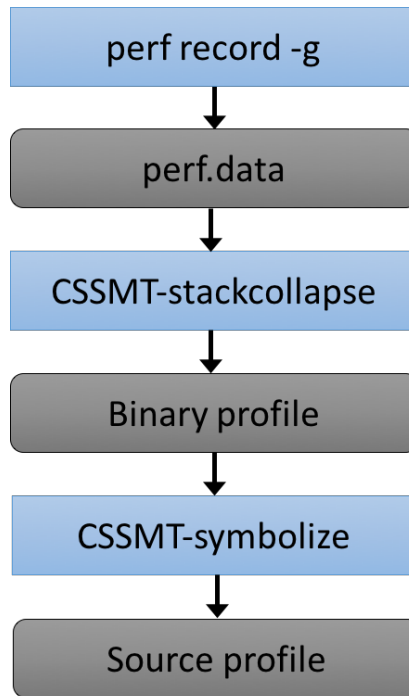


Figure 4.2: CSSMT Profiling Workflow

There are several challenges to make the sampling based profiling have precise source location to be useful for PGO based compilation [22].

1. State-of-the-art PMU lacks a reliable and sufficient mechanism for correlating hardware events with the program counter triggering the event. Commonly, there will be skid between the time hardware event happens and the time program counter being recorded. If the skipped instructions contain terminating instructions such as 'return', 'jump' etc., the symbolized source location could be skewed since the hardware skid cross the basic block boundary in terms of the source program.
2. The system software such as **compilers** and **profilers** are not quite ready for the PGO based compilation that requires precise source location. There are still many bugs in production compiler [2] and profiling tool [27]. For **compilers**, optimizations manipulating stack related registers often use frame pointers or leaf frame pointers as general purpose registers, which causes intermittent call

```

$ perf report --call-graph --stdio -G
# Samples: 40 of event 'cycles:ppp'
# Event count (approx.): 62981600819507
#
# Children      Self  Command  Shared Object      Symbol
# .....
#
100.00%      0.00%  ft.A.x   ft.A.x             [.] _start
|
---_start
  __libc_start_main
  main
  appft
  |
  |--69.83%--compute_initial_conditions
  |          |
  |          --3.49%--vranlc
  |
  |--23.62%--fftXYZ
  |          |
  |          --23.62%--Swarztrauber
  |
  |--3.49%--exp@plt
  |
  --3.06%--__GI___exp
             @plt

100.00%      0.00%  ft.A.x   libc-2.23.so      [.] __libc_start_main
|
---__libc_start_main
  main
  appft
  |
  |--69.83%--compute_initial_conditions
  |          |
  |          --3.49%--vranlc
  |
  |--23.62%--fftXYZ
  |          |
  |          --23.62%--Swarztrauber
  |

```

Figure 4.3: perf report with call frames

```

callchain:
4202244:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x
4198139:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x
4204828:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x
samples:
4203648:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x 14737563
4204477:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x 2199003840771
4204455:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x 2199003839065

callchain:
4201519:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x
4198139:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x
4204828:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x
samples:
4204465:/home/user/NPB3.3-SER-C/bin-profile/ft.A.x 2199003846493

```

Figure 4.4: Summarized Binary Profiling Data

frame missing in the final profile. The misrepresented call stack information in the profile further dilute the performance gain otherwise possible with the compiler optimizations. The debug information maintained by compilers is also a source of inaccuracy. Especially some aggressive optimizations are detrimental to the debug information. Currently, all existing solutions are still ad-hoc rather than systematic. For **profilers**, the profilers usually have very obscure interfaces for PGO. Existing PGO based optimizations commonly use customized boilerplate codes to collect and summaries profiling data. The compilers must be extended to read and understand each customized format the specific PGO optimization is using.

- Existing PGO infrastructure in production compilers is not extensible. For example, AutoFDO [21] implementations in LLVM and GCC serve only to optimize binary code placement in the final binary or indirect call inlining. The first version of CSSMT profiling module implementation finally grows into a complete standalone implementation rather than a proper extension of the existing PGO framework such as AutoFDO.

### 4.3 Implementation

An overview of how CSSMT collect host and guest program profiling data is shown in figure 4.2. To increase the precision of CSSMT profiling [5] and reduce the transformation overhead incurred in coarse-grained merge (chapter 6) and fine-grained merge (chapter 7), CSSMT profiling collects calling context-sensitive information, meaning for each sample recorded, the corresponding full call stack is also saved. Thus during the merging steps, the hot spot functions are cloned to make sure other call graph paths leading to the same function are not affected by CSSMT. This also decreases the running time of CSSMT since less coroutine context switches are incurred for this situation.

**Perf Profile Collection.** Since CSSMT is evaluated in Linux, it uses Linux Perf [27] to do sampling based profiling. Perf profiling corresponds to the top box in figure 4.2. The `-g` flag turns on call chain recording. The generated file is in a special format that can only be parsed by Linux Perf tools. The text output of the binary sampling data is shown in figure 4.3.

**Binary Profile Summarization.** Then the binary sampling data are grouped according to their call stack signature, which is an ordered list of program counters. As shown in figure 4.4, entries are separated by one empty line. For each entry, there are two kinds of data. One is `callchain` data which consists of consecutive lines where each line is in the `<IP>:<MODULE>` format. Each line represents one binary frame, which not necessarily corresponding to one source code level frame since the compiler may inline functions for performance reason. Those inlined call frames would be rediscovered in the following symbolization step. Those binary frames are recorded by using Intel Last Branch Record (LBR) feature. The LBR feature record all branch instruction including ‘jump’, ‘call’ etc.. The Linux Perf tool can filter out all other kinds of instructions except ‘call’. So the final profiling data contains call frames only. The `<IP>:<MODULE>` are two pieces of information as input to symbolizer. For each `callchain`, there is an unordered list of leaf locations where each one is decorated with the total number of cycles have been recorded for that sample location. These samples

are not sorted by their recorded total cycles since the exact source level call stack may be the same as another `callchain` entry. The sorting would be performed in following source profile summarization/symbolization step.

**Source Profile Summarization.** An exemplary output for NAS FT benchmark is shown in figure 4.5. It is human readable which is convenient for programmer inspection. Also, it is the format used by CSSMT passes in LLVM to perform the transformation. Like binary profile, source profile also contains two kinds of data. One is `stack`; the other is `top of stack samples`. The difference is that in source profile, all call frames and top of stack samples are symbolized and un-inlined. The call frame and sample are using the same set of fields to represent it symbolized information:

1. **function.** The function where PC is sampled. Using unmangled name since LLVM debug information keeps the unmangled name.
2. **file.** The full path to the file containing the sampled function.
3. **start\_line.** The starting source line number for the sampled function in the source file.
4. **line.** The absolute line number in the source file of the sample.
5. **column.** The absolute column number in the source file of the sample.
6. **discriminator.** Discriminator is DWARF specific information to help debugger or profiler to distinguish among multiple paths of execution on a single source line since one could write programs with arbitrary size into single source line for a lot of the programming languages, most notably C/C++. CSSMT is not DWARF specific. However, using this field increase the PC mapping precision significantly.

```

- stack:
  - function:      main
    file:         /home/user/NPB3.3-SER-C/FT/mainft.c
    start_line:   50
    line:         74
    column:       3
    discriminator: 0
  - function:      appft
    file:         /home/user/NPB3.3-SER-C/FT/appft.c
    start_line:   53
    line:         106
    column:       5
    discriminator: 0
top of stack samples:
  - source location:
    function:      Swarztrauber
    file:         /home/user/NPB3.3-SER-C/FT/fft3d.c
    start_line:   59
    line:         130
    column:       27
    discriminator: 0
    source location counter: 52
  - source location:
    function:      Swarztrauber
    file:         /home/user/NPB3.3-SER-C/FT/fft3d.c
    start_line:   59
    line:         114
    column:       33
    discriminator: 4
    source location counter: 1
  - source location:
    function:      Swarztrauber
    file:         /home/user/NPB3.3-SER-C/FT/fft3d.c
    start_line:   59
    line:         95
    column:       17
    discriminator: 0
    source location counter: 170

```

Figure 4.5: Summarized Source Profiling Data (Compiler Input/Output)

## 4.4 Discussion

There is no existing profiling guided optimization framework suitable for CSSMT. In CSSMT, profiling uses the commonly available PMU in modern processors to sample the program counters of measured workload. Comparing the instrumentation based profiling, it has considerable less running overhead and does not skew the real hot spots of tested workload. The hardware, toolchain and debug information related specification are not mature enough to implement CSSMT profiling in a well-supported way. For this reason, when CSSMT mapping program counters to source level abstractions, it needs various ad-hoc fixes to tame some imprecision originated from above areas. One good example is sample skid problem mentioned in [22]. On Intel X86, for Nehalem based or later processors, the Processor Event Based Sampling (PEBS) facility could mitigate the sample skid issue pretty well. However, on the ARMv8 processor, CSSMT has to search in a range of instructions for real sampled instructions in the source code. It is not reliable and more importantly could potentially impact the effectiveness of CSSMT. However, there is no mature solution from either the hardware or the software.

## Chapter 5

### CSSMT RUNTIME

There are two major components in CSSMT runtime: **coroutine support** and **input/output redirection**. Since constraining the merge overhead to be minimal is critical, the coroutine is used to skip call graph paths that do not benefit from coarse-grained (chapter 6) and fine-grained (chapter 7) merge. On the other hand, IO redirection solves the problem of merged SMT threads can only have one set of file descriptors. There must be a systematic way to redirect guest SMT thread's input and output file to other places to prevent conflict on file descriptors.

#### 5.1 Coroutine

The purpose of using coroutine in CSSMT is illustrated in figure 5.1 with runtime view. Suppose we have two standalone threads 5.1a, the most natural approach of merging is by inserting entrance (`main2`) of the guest thread as the first instruction of `main` of host thread, then perform legal transformations afterward. In this way, we have a runtime view of merging as shown in figure 5.1b. The drawback of this approach is that it merges both hot spots and cold spots (figure 5.1b). The cold spots are merged only for the correctness of the final program. However, these merging may cause much overhead depending on the control flow complexity of the program. Moreover, those overhead incurs without any benefits for trade-off.

Another approach is to have one thread for each program in a single process. Each does its own things in cold code paths. Also, barriers are used as the synchronization point to only merge hot code paths (figure 5.1c). Depending on the granularity of hot code paths, the OS-level thread may cause a lot of synchronization overhead, because each entry into a hot code path incurs one synchronization. Fine-grained hot

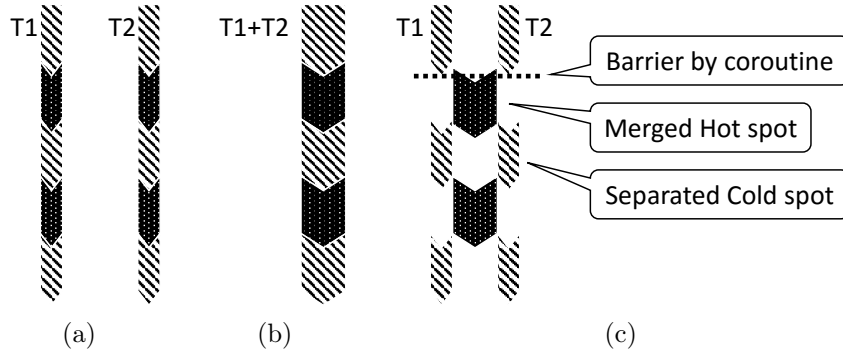


Figure 5.1: Purpose of Coroutine. Black represents hot spots. Stripe represents cold spots. (a) Original two threads. (b) Both hot&code spots are merged. (c) Hot spots are merged using coroutine. Code spots are intact.

spot would cause a huge number of synchronizations. Besides we do not need the time sharing of the single pipeline on the cold code paths. It causes OS scheduling overhead without benefits since cold code paths consume little time.

In another perspective, hardware-based SMT conducts instruction level combination of ISA streams of SMT threads. It does not require the high-level abstraction information such as call graph, basic block, loop, and so forth. These high level information are represented directly or indirectly by the control flow instructions such as ‘call’, ‘jump’ and ‘ret’. On the contrary, CSSMT works on the intermediate representation of source programs. After pinpointing the suitable merging source location with the profiling information (chapter 4), the merging algorithm would have to figure out which call graph path leads to that source location. Before the actual hot spot merging could happen, the full call graph path starting from the root node (usually main) have to be merged first and most importantly with very low compile-time and running time overhead. The reason is that very call frame along the call graph path except the leaf frame should already be proved by profiling information to be cold spot rather than the hot spot. There is no benefit of merging those considering the coarse-grained merge algorithm is not overhead free (The coarse-grained merge algorithm sometimes need to clone functions to isolate the path).

Coroutine [24] is the perfect tool to solve this call graph path skipping need

```

1 // Use mpm_prefix to disambiguate global variables
2 #include <ucontext.h>
3 bool mpm_over = false;
4
5 ucontext_t uc[2];
6
7 void mpm_ucontext_init (int (*main2)(int, char**),
8                       int argc2, char **argv2) {
9     // 'ulimit -s' gives 8192Kb.
10    size_t sz = 8*1024*1024;
11
12    int s;
13    if ((s = getcontext (&uc[1]) == -1))
14        handle_error_en(s, "getcontext");
15
16    uc[1].uc_link = &uc[0];
17    uc[1].uc_stack.ss_sp = mmap(0, sz,
18                               PROT_READ | PROT_WRITE | PROT_EXEC,
19                               MAP_PRIVATE | MAP_ANON, -1, 0);
20    uc[1].uc_stack.ss_size = sz;
21    makecontext (&uc[1], (void) (*) (void) main2, 2, argc2, argv2);
22 }
23
24 void mpm_ucontext_destroy() {
25     if (!mpm_over) {
26         mpm_over = true;
27         swapcontext (&uc[0], &uc[1]);
28     }
29 }
30
31 void mpm_ucontext_host_yield() {
32     int s;
33     if ((s = swapcontext (&uc[0], &uc[1])))
34         handle_error_en(s, "mpm_ucontext_host_yield");
35 }
36
37 void mpm_ucontext_guest_yield() {
38     int s;
39     if ((s = swapcontext (&uc[1], &uc[0])))
40         handle_error_en(s, "mpm_ucontext_guest_yield");
41 }

```

Figure 5.2: Coroutine Library Implementation (ucontext version)

of CSSMT. In CSSMT context, the host and guest program are separate coroutines. They voluntarily yield to each other to implicitly skip cold spots along call graph paths as required by CSSMT. Coroutines are well-suited for implementing cooperative tasks (without unwanted OS scheduling). Coroutine differs itself from the subroutine in state keeping among invocations. Each subroutine invocation is a clean slate. The state is created on the entry of subroutine and destructed on the exiting. Coroutine could enter its invocation as many times as it desires (In CSSMT context, we want to enter the host and the guest main function as many as it needs to). Host and guest SMT thread take turns to execute their cold code paths. Just before one thread enters into its hot spots, it checks if the other thread stopped at the beginning of its hot spots. If so, control flow jumps or falls through to a code section consisting of instructions from both threads. In this way, we could skip the merging of cold code paths to reduce merging overhead. Another attractive property of coroutine is that it has minimal overhead comparing to OS thread since it does not carry operating system attributes such as scheduling, uid and gid etc..

In CSSMT there are two versions of coroutine libraries: `ucontext` based or `ucontext` based. `ucontext` [37] is used to create user thread context in System V environment. It keeps the thread state including the signal mask. The `ucontext` based coroutine library in CSSMT exclude the signal mask during context saving before yielding to other thread. The benefits are much faster context switching than the `ucontext` version. `ucontext` based and `ucontext` based coroutine are used in different situations. Only when the compiler finds out either host or guest invoking signal related Libc function, CSSMT uses `ucontext` based coroutine. For all other scenarios, CSSMT uses `fcontext` based coroutine to reduce overhead as much as possible.

`ucontext` and `fcontext` coroutine implementation have the same API which consists of four methods: *init*, *destroy*, *host\_yield* and *guest\_yield*. By their name, *init* initializes a coroutine context for the guest thread and *destroy* destroys the created coroutine context before SMT'd process finish execution. *host\_yield* causes the control flow stop at the host's call stack and then jump to the call stack of the guest thread.

```

1  #include <boost/context/all.hpp>
2  bool mpm_over = false;
3
4  // host thread continuation
5  ctx::continuation gsink;
6
7  // guest thread continuation
8  ctx::continuation gsource;
9
10 // main2 is the function pointer to the guest main function.
11 void mpm_fcontext_init(int (*main2)(int, char**),
12                       int argc2, char **argv2) {
13     ctx::continuation source=ctx::callcc(
14         [main2,argc2,argv2] (ctx::continuation && sink) {
15         gsink = std::move(sink);
16         main2(argc2, argv2);
17         mpm_over = true;
18
19         return std::move(gsink); // continue on host thread
20     });
21
22     gsource = std::move(source);
23 }
24
25 void mpm_fcontext_destroy() {
26     if (!mpm_over) {
27         mpm_over = true;
28         gsource = gsource.resume();
29     }
30 }
31
32 void mpm_fcontext_host_yield() {
33     if (!mpm_over)
34         gsource = gsource.resume();
35 }
36
37 void mpm_fcontext_guest_yield() {
38     if (!mpm_over)
39         gsink = gsink.resume();
40 }

```

Figure 5.3: Coroutine Library Implementation (fcontext version)

*guest\_yield* performs similar operation. In the next two subsections, the CSSMT coroutine implementation details are described. The exact way of using these functions to implement barrier is shown in the coarse-grained merging algorithm introduced in chapter 6.

### 5.1.1 Ucontext

The functions comprising `ucontext` based coroutine library are listed in figure 5.2. The `ucontext` header is exposed by the kernel to support user-level context switching among threads within a single process. `ucontext` statically create two user thread contexts for host and guest thread. When the SMT'd program begins execution, the *mpm\_ucontext\_init* function fill the proper fields of the guest context with the guest main function pointers and main function input arguments. On the exit of combined process, either host or guest thread call *mpm\_ucontext\_destroy* to check if their counterpart has finished execution. If not, the control flow is redirected to the unfinished thread.

### 5.1.2 Fcontext

The functions comprising `fcontext` based coroutine library are listed in figure 5.3. It uses Boost context library [13] to implement its functions. Like `ucontext`, two global contexts are created for threads. However in `fcontext`, the initial *continuation* are both empty. The name *continuation* shows that it is often used in functional language environment. The *callcc* function takes a function/lambda as input, executable it in a newly created continuation and then return the newly created continuation. In this case, the returned continuation represents guest thread context. In figure 5.3, the lambda created from line 14 to line 19 are the root frame of the guest thread. It begins by saving the continuation of host thread to the global continuation variable for host thread so CSSMT could use it jump/yield back to host thread when it needs to. The second thing it does is to invoke the guest main function. The important point to notice here is that the guest main function here is called inside

a coroutine context (continuation). This call site to the guest main function would not execute through the end and return without any host thread instruction passing by in between. Intuitively, descendants of the guest main function in guest call graph may call *mpm\_fcontext\_guest\_yield* to jump back to the host thread coroutine context (gsink). The returning of *mpm\_fcontext\_init* is after guest thread first call of *mpm\_fcontext\_guest\_yield* and the host thread saved the continuation of guest thread (line 22). The tricky part is the root of guest call frame (the lambda function to *calloc* function does not return until guest thread finish execution. That is also when guest thread return the continuation of host thread so that host thread could wind up the rest of its instructions.

As another measure to reduce overhead and expose the compiler optimization opportunities, CSSMT links `fcontext` in the compiler intermediate representation level but not in binary level as shown in figure 3.2. It gives the compiler chance to inline *yield* functions which suppose to be called many times even if its absolute cost is small ( 20 cycles). Any kind of call stack setup would easily cost much more than that. For `ucontext`, it is less of a problem since `ucontext` itself is very expensive ( 1000 cycles). The extra cycles consumed by call frame setup instruction would not perturb the overall CSSMT performance much. Another reason is that Libc source code is platform dependent. Linking its source in IR level could incur correctness bugs that are difficult to find.

## 5.2 Input/Output Redirection

Operating system processes have exclusive platform-dependent attributes that are not shared with other processes. These include call stack, execution context, process id, scheduling related properties and input/output (IO). The Input/output are special for CSSMT in that, CSSMT only wants to merge the computation process but not the input data set or the output of the computation the user wants to acquire. Unfortunately, the coroutine context does not save an array of opened files (Here file is a general term, it could include the terminal, networking, in-memory, on-disk or

Original	Replaced With*
scanf(...)	vscanf(in_file, ...)
printf(...)	fprintf(out_file, ...)
vprintf(...)	vfprintf(out_file, ...)
getchar(...)	fgetc(in_file, ...)
puts(...)	fputs(..., out_file)
putchar(...)	fputc(..., out_file)
perror(...)	fprintf(err_file, "%s\n", strerror(errno))

\*in\_file,out\_file,err\_file substitute stdin,stdout,stderr.

\* n.b. this list is incomplete.

Table 5.1: Libc functions accessing standard file and the replacement functions

special file such as `/proc/*` in Linux machine). Provided CSSMT does not conduct special treatment to SMT threads's IO, implicitly, there are conflict on assigned file descriptor ID for both threads. For example, on Linux machine, the standard IO files include `stdin`, `stdout` and `stderr`. Each process has their own standard IO files before merging. The final program after merge can only have one set of standard IO files. By redirection of the standard IO of the merged program to files, each thread still shares the same redirected files, which obviously would not work. Merged process could not take correct input from `stdin` and could not print garbled output to `stdout` and `stderr`.

There are two solutions: 1. redirect standard IO of either thread to file. The other thread takes over the standard IO of the merged program. 2. redirect standard IO of both threads to the respective set of files. CSSMT uses the latter for implementation simplicity. For each coroutine (call graph tree rooted at `main` and `main2`), a set of three files are created to substitute `stdin`, `stdout` and `stderr` respectively. Then each function call using standard IO explicitly(such as `fprintf`) or implicitly(such as `printf`), is replaced by a semantically equivalent function. A list of libc functions and their replacing functions that are used to implement the equivalent semantic are shown in table 5.1.

Not only standard IO requires redirection for CSSMT to function correctly as discussed above, the regular disk file based IO also need to be redirected. The Libc file

IO related functions that need redirection are `access`, `chdir`, `chmod`, `chown`, `create`, `create64`, `execl`, `execle`, `execlp`, `execv`, `execve`, `execvp`, `fopen`, `fopen64`, `freopen`, `freopen64`, `ftw`, `ftw64`, `lstat`, `lstat64`, `lutimes`, `mkdir`, `mkfifo` and `mknod` (This list may not be exhaustive for all platforms).

## Chapter 6

### CSSMT COARSE-GRAINED MERGE

#### 6.1 The Two Methods

CSSMT coarse-grained merge uses two methods to perform context-sensitive hot spot function combine. One is a coroutine-based method where the context is implicitly stored in the compiler inserted runtime library, namely `ucontext` or `fcontext`. The merge effect is achieved by yielding from the context from either thread to that of the other thread. Coroutine based merge is applied to the cold segment of call graph path. For hot segments that are close to the leaf function, the method described in the rest of this chapter is used since it could expose more fine-grained merge opportunities due to its intrusive nature.

#### 6.2 Code Extraction

To be able to move the guest code over to where the host code is, the guest code is outlined into a new function, and a call to this new function is inserted. The

```
int foo(int x){
    int t = 0;
    for(int i = 0; i < x; ++i) {
        if (i%3 == 2)
            break;
        t += i*i;
    }
    return t;
}

int foo(int x){
    int t = 0;
    extracted_func(x, t);
    return t;
}

void extracted_func(int x, int &t){
    for(int i = 0; i < x; ++i){
        if (i%3 == 2)
            break;
        t += i*i;
    }
}
```

Figure 6.1: Example of Code Extraction

input to the guest code is passed by value; the output is passed by reference. “Store” instructions to the matching output arguments are inserted. Also, the inputs and outputs are promoted to global variables to make the communication easy between the context of guest code before merging and after merging.

The coarse-grained merge algorithm uses code extraction to outline code sections along call graph path when necessary. The important benefits of code extraction are greatly reduced code duplication and simplified merge algorithm. One use of code extraction is in step 2 of figure 6.6 where the loop is extracted into a standalone function. In the OpenMP support of coarse-grained merge, code extraction is often used to outline *parallel for* pragma guarded loops into standalone functions.

### 6.3 The Algorithm

In CSSMT, call graph path is represented by consecutive call instructions and loop nest identifier(may use loop header block id in IR also). The caption of figure 6.5 and figure 6.6 contain the example. If the first element of this sequence is a call instruction, we call it *top instruction*. If the first element of this sequence is a loop, we call it *top loop*. The length of a call graph path is the number of loop levels in it.

The merging algorithm is listed in figure 6.3. First, the host and guest thread are preprocessed. (lines 1–2). The preprocessing includes coroutine setup and IO function call sites replacement etc.. Then their whole program IR are linked for further transformation (line 3). They can be linked since after preprocessing host thread has a reference to `main2` defined in the guest thread. There is no global name conflict between two threads since in LTO stage, all global functions and variables with a definition except `main` are modified to have internal linkage. During linking, any internal name conflict can be resolved automatically by the compiler. Next, the program entrance of guest thread `main2` are initialized as coroutine using library defined in figure 5.2 or figure 5.3 depending on the requirement of the workload. The coroutine cleanup code is inserted at the end of host thread.

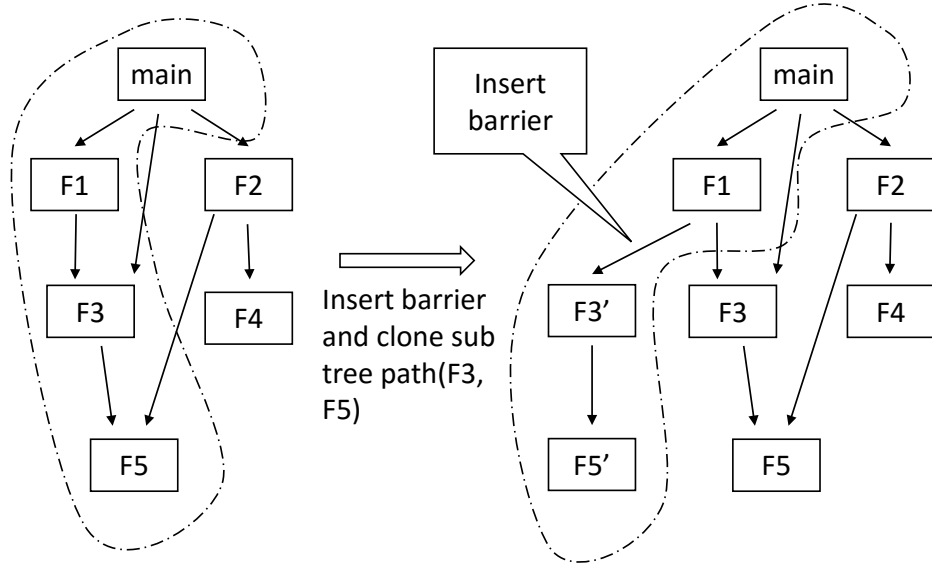


Figure 6.2: Illustration of INSERT-BARRIER and CLONE-CALL-PATH

The way of inserting barriers are illustrated in figure 6.2. It corresponds to line 4 in figure 6.3. First hot spots are located from profiling information ( $F5$ ). Then the call graph path is calculated as  $main \rightarrow F1 \rightarrow F3 \rightarrow F5$ . Barrier is inserted at edge  $F1 \rightarrow F3$  so that merging of  $main$  and  $F1$  is skipped. Call graph path consisting of  $F3'$  and  $F5'$  is then merged by MERGE-CALL-GRAPH-PATH. The edge  $F1 \rightarrow F3$  is chosen to insert barrier because they consume a proper amount of running time. The running time of  $F3$  and  $F5$  is not too small so that `swapcontext` not causing too much overhead, and not too large so that some loop level ( $main, F1$ ) can be relieved from merging by using `swapcontext`. It is a trade-off between merging overhead and the gained performance by merging. The final step in merging algorithm (line 9) is to do the merging of two call graph paths ( $F3'$  and  $F5'$  is one of two paths) leading to the hot spot.

A running example of how MERGE-CALL-GRAPH-PATH work is listed in figure 6.5 and figure 6.6. Figure 6.5 shows the original source code of two call graph paths in which host hot spot `hs1` and guest hot spot `gs1` need to be finally merged. After merging they reside in the same level of loop body (they are not necessarily in the same or consecutive basic block, which is the goal of fine-grained merging in

```

MERGE( $T1, T2, P1, P2$ )
1  PREPROCESS-HOST-THREAD( $T1$ )
2  PREPROCESS-GUEST-THREAD( $T2$ )
3  link modules of  $T1$  and  $T2$ 
4  install code to construct and destruct coroutine
5   $CP1 = \text{INSERT-BARRIER}(T1, P1)$ 
6   $CP2 = \text{INSERT-BARRIER}(T2, P2)$ 
7   $NCP1 = \text{CLONE-CALL-PATH}(CP1)$ 
8   $NCP2 = \text{CLONE-CALL-PATH}(CP2)$ 
9  MERGE-CALL-GRAPH-PATH( $NCP1, NCP2$ )

PREPROCESS-HOST-THREAD( $T$ )
1  replace IO operations in  $T$ 
2  insert main function of guest thread

PREPROCESS-GUEST-THREAD( $T$ )
1  replace IO operations in  $T$ 
2  rename main function to main2

MERGE-CALL-GRAPH-PATH( $P, Q$ )
1  while  $P.length > 0$  and  $Q.length > 0$ 
2      if  $P.topinst$  not in the preheader of  $Q.toploop$ 
3          sink  $P.topinst$  to the preheader of  $Q.toploop$ 
4      if  $Q.topinst$  not in the preheader of  $P.toploop$ 
5          sink  $Q.topinst$  to the preheader of  $P.toploop$ 
6
7      if  $P.length == 1$  or  $Q.length == 1$ 
8          unroll either  $P.bottomloop$  or  $Q.bottomloop$ 
9          merge loop  $P.bottomloop$  and  $Q.bottomloop$ 
10     else strip mine  $P.toploop$  or  $Q.toploop$ 
11         merge loop  $P.toploop$  and  $Q.toploop$ 
12
13      $P.length = P.length - 1$ 
14      $Q.length = Q.length - 1$ 

```

Figure 6.3: Pseudo Code of MERGE

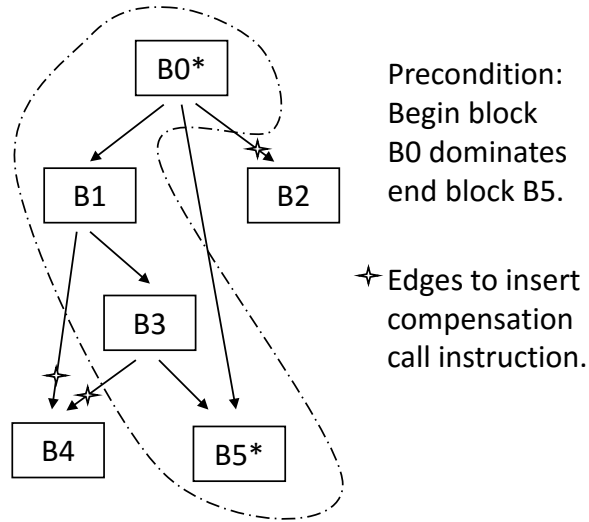


Figure 6.4: Illustration of Lines 2–5 in MERGE-CALL-GRAPH-PATH

chapter 7). All transformation steps leading to the merging of for1 and for4 loops are shown in figure 6.6. Step 0 naively merge host and guest. In step 1 and step 4, *top instruction* of one thread is sunk to just before the *top loop* of the other thread. In step 2, if *top loop* is the first element of host call graph path, it is extracted into a standalone function so that there is a *top instruction* to easily move around. In step 3, host *top instruction* are sunk into guest function. Finally, in step 5, for1 and for5 can be merged since they are now consecutive in the control flow of function `guest_ext1`. The loop merging algorithm (invoked in line 9 and line 11 of merging algorithm) is discussed in section 6.3.1.

```

void host()          void h1()          void guest()        void g1()
{
  if(1) {
    for(1)
      h1()
  } else {
    ...
  }
}

void h1()           void h2()           void g1()           void g2()
{
  for(2)
    h2()
}
void h2()           {
  for(3)
    hs1
}

void guest()        void g1()           void g2()
{
  if(2) {
    for(4)
      g1()
  } else {
    ...
  }
}

void g1()           void g2()
{
  for(5)
    g2()
}
void g2()
{
  gs1
}

```

Figure 6.5: Example of MERGE-CALL-GRAPH-PATH. Call graph path of host is for1:h1:for2:h2:for3. Call graph path of guest is guest:for4:g1:for5:g2.

Step 1 and step 4 in figure 6.6 sink *top instruction* of one thread to be the last instruction (before branching instruction of the block) of the preheader of *top loop* of the other thread. It is called **merging across selection branching**. The idea is illustrated in figure 6.4. *Top instruction* is in block  $B0$ . Target block is  $B5$ . First, the set of the direct and indirect control dependence blocks of  $B5$  are recognized( $B0, B1, B3$ ). Then the blocks not dominated by beginning block ( $B0$ ) are removed from the set. Then all successor edges of the blocks in the set are examined. If an edge goes to a block that is not in the set, then the edge ( $B0 \rightarrow B2, B1 \rightarrow B4, B3 \rightarrow B4$ ) is picked to insert compensation call instruction. Notice that target block of the edge is not the right place to insert compensation code since the to block of the edge may have predecessors that are not in the set.

Two facts make this legitimate. One fact is that the block containing sinking *top instruction* must dominate the target block. So in the sinking process, no need to consider the case that control flow jumps into the set mentioned above. The other fact is that the *top instruction* represents code from the other thread. No data dependence need to be maintained. This algorithm is similar to the repairing code insertion algorithm in trace scheduling[35]. The difference is that there are no edges that jump into the trace that the from block of the edge is not dominated by our beginning block. It only needs to consider the case that blocks in the trace jump out of trace.

Step 3 sinks *top instruction* of one thread to be the first instruction of entry block of callee of the *top instruction* of the other thread. This is called **merging across function call**. The complicated case is when either or both call instructions return value. The method used here is changing both function types to return void. The return variable is passed as the reference in the parameter list. And the receiving function(*guest*) parameter list is augmented with the parameter list of sinking function(*ext1*). Inside receiving function, the parameter list of sinking function are forwarded(call to *ext1* in *guest\_ext1*).

Step 5 is called **merging across loop**.

```

// Step 0
// 1 in if and for
// statement are only
// for occupying space.
void host(){
    guest()
    if(1) {
        for(1)
            h1()
    } else {
        ...
    }
}

// Step 1
void host(){
    if(1) {
        guest()
        for(1)
            h1()
    } else {
        guest()
        ...
    }
}

// Step 2
void host(){
    if(1) {
        guest()
        ext1()
    } else {
        guest()
        ...
    }
}
void ext1(){
    for(1)
        h1()
}

// Step 3
void host(){
    if(1) {
        guest_ext1()
    } else {
        guest()
        ...
    }
}
void guest_ext1(){
    ext1()
    if(2)
        for(4)
            g1()
}

// Step 4
void guest_ext1(){
    if(2) {
        ext1()
        for(4)
            g1()
    } else {
        ext1()
        ...
    }
}

// Step 5
void guest_ext1(){
    if(2) {
        for(1,4)
            h1()
            g1()
    } else {
        ext1()
        ...
    }
}

```

Figure 6.6: Transformation Steps to Merge Outmost Loop of Host and Guest. After these steps, call graph path of host is h1:for2:h2:for3, call graph path of guest is g1:for5:g2. Recursively do these steps for host and guest call graph path to merge for2 with for5, etc..

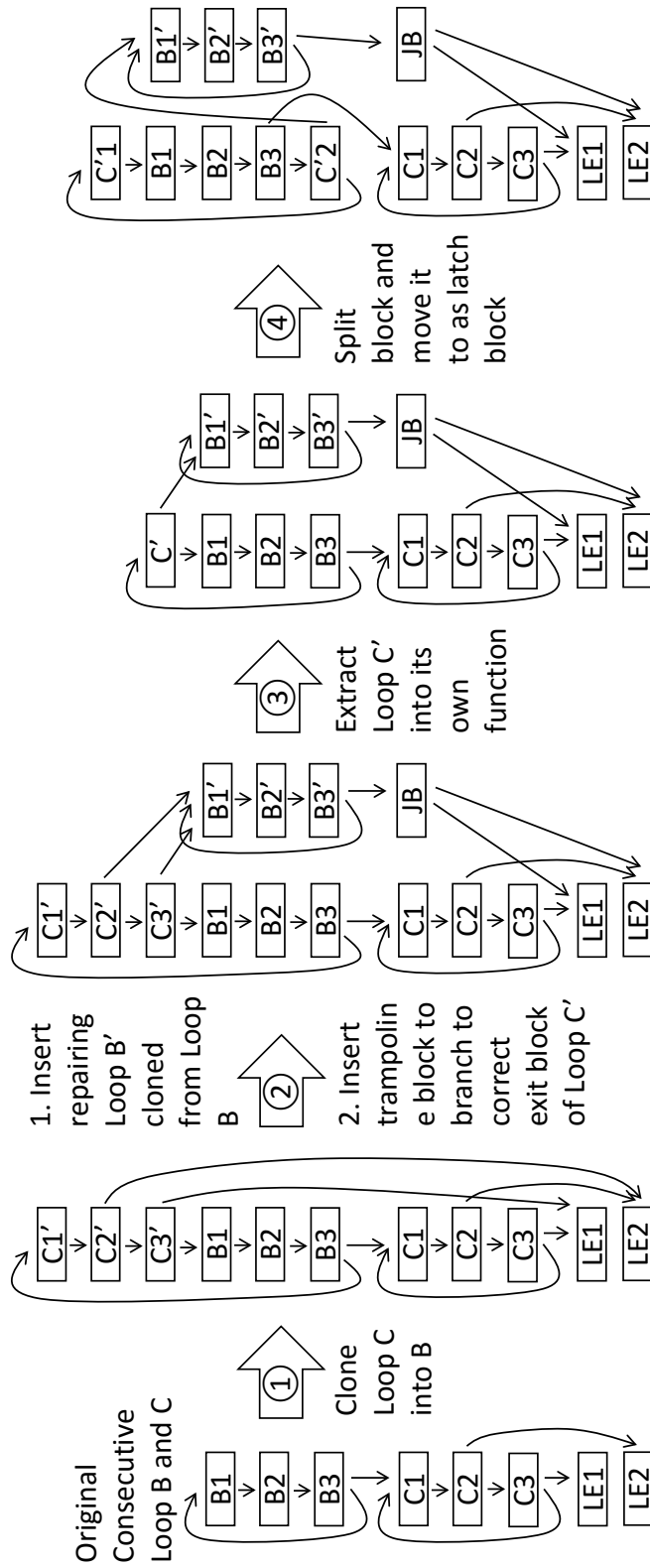


Figure 6.7: Illustration of Loop Merge: line 9 and line 11 in MERGE-CALL-GRAPH-PATH

### 6.3.1 Ordered Loop Merge

The steps to merge two consecutive loops are illustrated in figure 6.7. The four steps showed do not necessarily match actual implementation. It is for the convenience of illustration.

In the first step, all basic blocks of loop  $C$  are cloned into the body of loop  $B$ . This clone is legitimate because all loop  $C$  output variables are already demoted to stack. The cloned loop header  $C1'$  becomes the new header of the fused loop, and the back-edge  $C3' \rightarrow C1'$  is removed. So the essence of the loop merging is to share the back-edge ( $B3 \rightarrow C1'$ ) while maintaining correctness. The exit blocks of loop  $C$  now have two predecessors. One is from the original loop; the other is from the cloned loop. After step one, the IR becomes semantically invalid until step four is finished.

If the cloned loop branches out of the fused loop, then the left iterations of loop  $B$  are nowhere accounted for. Thus, in step two, compensation code for loop  $B$ , cloned loop  $B'$ , are inserted at the converging exit block of loop  $C'$ . However, loop  $C'$  may have more than one exiting blocks. After loop  $B'$  finishes execution, control flow should go to the correct exit blocks of loop  $C'$ . It is achieved by storing different integer value to a local variable at the end of each exiting block of loop  $C'$  and switching on this local variable in the newly created trampoline block  $JB$ .

There are usually more levels of loop inside loop  $C'$  and loop  $B$ . So it needs to satisfy the precondition of the loop in MERGE-CALL-GRAPH-PATH to apply loop merging recursively. The precondition is that at the beginning of merging algorithm and after each loop merging, the *top instruction* of one thread dominates *top instruction* or *top loop* preheader of the other thread. Thus in step three, loop  $C'$  is extracted to a standalone function and a replacement block  $C'$  is inserted instead. Block  $C'$  contains, among other instructions, the call to the extracted function. Block  $C'$  must be an exiting block of the fused loop. Suppose the *top instruction* in block  $C'$  is sunk into the body of loop  $B$  in next iteration of loop merge, if the control flow exits the loop from block  $C'$ , the body of loop  $B$  is partially executed, which would generate an incorrect program. Thus, in step four, block  $C'$  is split in to  $C'1$  and  $C'2$ . Block  $C'2$

contains only a branching instruction. Block  $C'2$  also becomes the new latch block of the fused loop.

The relative position of exiting blocks of loop  $B$  to block  $C'1$  is not important. Even if the *top instruction* in block  $C'1$  needs to be sunk down below any of exiting blocks of loop  $B$ , the transformation correctness is still maintained. The reason is that when the MERGE-CALL-GRAPH-PATH algorithm goes to next iteration of loop merge, the algorithm illustrated in figure 6.4 will insert compensation code on control flow branching out of call graph path (in this case, probably  $B3 \rightarrow C1$ ).

### 6.3.2 Unordered Loop Merge

In section 6.3.1, loop levels from each thread are merged from top-level loop to the inner loops until one of the innermost loop is merged. However, for reasons stated in section 6.3.3, sometimes a method is required to merge *top loop* of one thread with an inner loop of the other thread. The implementation of this unordered loop merge in CSSMT is similar to figure 6.7 with a minor difference. The difference is that in step one, loop  $C$  blocks are copied to the inner loop of  $B$  instead of loop  $B$  itself. If the target level of loop of in  $B$  is not in the same function as loop  $B$  and  $C$ , then loop input and output of  $C$  are communicated through global variables to the copied loop  $C'$ .

### 6.3.3 Trip Count Mismatch

Let's first define the term "merging window" to be the time window at execution time where instruction stream of one thread is fused with that of the other thread. The loop merging scheme discussed in section 6.3.1 does not consider the balancing of the trip count of the loops. If two loops take the same running time where loop A has a trip count of 10 and loop B 1000, then only 10 iterations of each can be merged considering both loops share the back-edge in the fused loop. The merging window for loop B is only 1%. This imbalance would be transferred to the inner loop level where the hot spot lives, which may cause greatly reduced merging opportunity. Thus

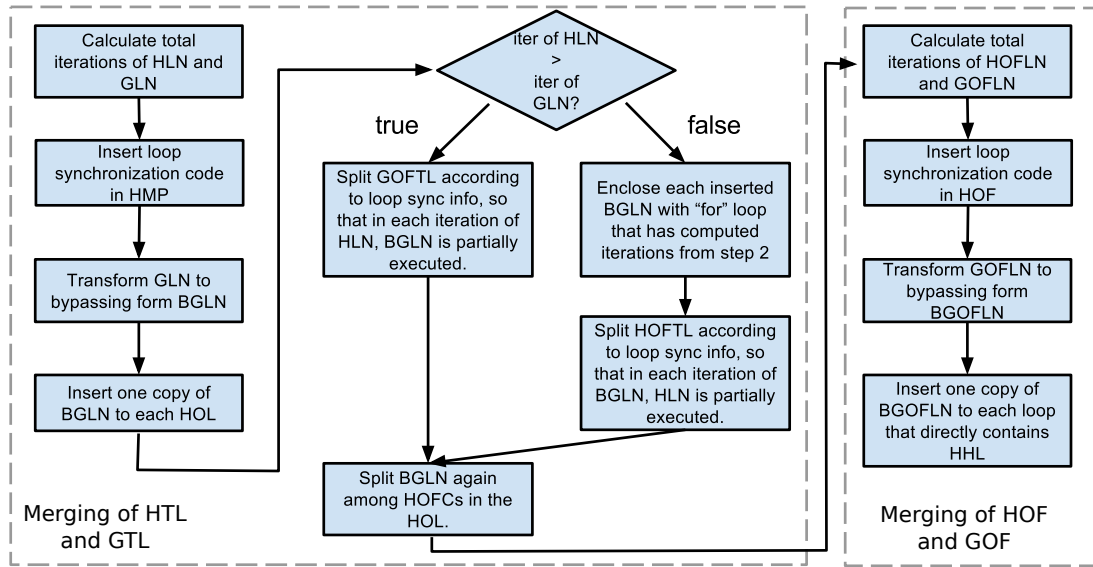


Figure 6.8: Coarse-grained Merge OpenMP Workflow

a greedy algorithm is developed to solve this problem. The purpose is to use minimal numbers of loop merge to achieve the largest merging window.

Suppose a call graph path is represented by an array. Array index represents the loop depth. Each array element stores the accumulated strip count for the loop at that depth (trip count data is estimated by profiling). An example:

$$T1 : [10, 100, 1000] \quad T2 : [15, 150, 150, 200]$$

Comparing the first element of  $T1, T2$  to get the bigger one  $T2 : 15$ . Then  $T1$  is iterated to find the first element that is bigger than  $T2 : 15$ .  $T1 : 100$  is found. Skipping loops in this iteration process is allowed since we could merge unordered loop with method described in section 6.3.2. Since  $T1 : 100$  is much larger than  $T2 : 15$ , naively merge them would cause reduced merging window. Thus we first strip mine  $T1 : 100$  to  $T1 : 15, T1 : 100$ , then merge loops represented by  $T1 : 15$  and  $T2 : 15$ . Then we have:

$$T1 : [100, 1000] \quad T2 : [150, 150, 200]$$

By redoing last step, pair  $T2 : 150, T1 : 1000$  is to be merged. Again  $T1 : 1000$  needs to be stripmined to  $T1 : 150, T1 : 1000$  to enlarge merging window. Afterwards  $T1 : 150, T2 : 150$  are merged. Then we have:

$$T1 : [1000] \quad T2 : [150, 200]$$

Now since  $T1$  has only one level loop left, the loop merge need to be conducted on the last loop level of both threads ( $T1 : 1000, T2 : 200$ ). Instead of strip-mining  $T1$  like previous steps,  $T1$  is unrolled to enlarge its loop body before loop merge. In this example, the unroll factor is 5, which is still reasonable. Sometimes this trip count imbalance of innermost loops is so large that choosing appropriate unroll factor is tricky. This is one of the aspects that programmer intervention is needed.

## 6.4 OpenMP Support

Table 6.1: Important Acronyms

Acronym	Explanation
ompfunc	a outlined function, whose address is passed to OpenMP runtime function <code>GOMP_parallel_start()</code>
H(G)HL	host(guest) hot source lines in omp func.
H(G)OF	host(guest) hot ompfunc which contains at least one H(G)HL
H(G)OFC	call instruction to H(G)OF
H(G)OFTL	the only top loop in H(G)OF
H(G)OL	host(guest) omp loop, which directly contains at least one H(G)OFC
H(G)TL	host(guest) top loop, outmost loop in H(G)MP that contains all H(G)OFC
H(G)MP	host(guest) merge place, which contains the H(G)TL
H(G)LN	host(guest) loop nests, all loop nest that contains a H(G)OFC, m,n is the loop nest for the first two “parallel for” in host; m,n,p is the loop nest for the last “parallel for” in host
H(G)OFLN	host(guest) ompfunc loop nests, all loop nest in H(G)OF that contains a hot line
BGLN	bypassing form of GLN
BGOFLN	bypassing form of GOFLN

This section presents the preliminary support for OpenMP in CSSMT. The overall workflow is shown in figure 6.8. Algorithm 6.9 illustrates the high-level merging

algorithm. The acronyms used in Algorithm 6.9 and this section are shown in Table 6.1. It aims to evenly distribute one hot “ompfunc” in guest merge place into several hot “ompfunc”s in host merge place. It is assumed that all loop nests containing ompfuncs have loop invariant loop start, range and stride. Note that the OpenMP standard requires the “parallel loop” in canonical form [25], which means that “parallel for” loops also have loop invariant loop start, range and stride.

Merging takes place in two kinds of functions in the host program. In HMP, GLN is transformed to “bypassing” form(BGLN). Then one copy of BGLN is inserted into each HOL. The loop iteration coordination depends on the loop iteration calculation(Line 5 and Line 7 in Algorithm 6.9) with result V9 and V10 when “HI  $\geq$  GI” (The case where HI < GI is not discussed here, but is similar.) (Line 9), which means it takes V9 iteration of HLN to execute one iteration of GLN and currently it is at the V10’tth iteration of the V9 iterations. V9 and V10 decide the loop base(GB) and range(GR) of guest ompfunc that needs to be merged into one HOL, which is GB’ and GR’(Line 14), which are further distributed in each HOFc of the HOL as GB” and GR”.

In the HOF’s, GB” and GR” iteration of BGOFLN are distributed among a list of insertion places that are decided by profiling information(Line 22). Then the total iterations of all insertion places AllIter are computed(Line 25). The last step is merging GR” loop iterations in guest ompfunc into AllIter loop iterations in host ompfunc. A “for” loop with range “GR / AllIter” is inserted at every insertion place whose loop body is a copy of the BGOFLN.

#### 6.4.1 OpenMP Parallel Loop Construct Information Extraction

OpenMP is mainly composed of compiler directives and runtime libraries. The compiler fronted translates the OpenMP directive to proper runtime libraries calls and some extra code. But some useful information for merging are lost during this translation, such as loop start, range and step of “parallel loop” construct. This information could be recovered from IR according to two facts: 1. The OpenMP standard requires

```

MERGE-TWO-MERGEPLACES(HMP, GMP)
1  for each HOL in HMP // HI = 0
2    HI = HI + (Number of times a HOFC is executed)
3    GI = (Number of times the only GOF in GMP is executed)
4    Initialize HIX as a index variable has the range from 0 to HI
5    In HMP, insert the following code just before the entry into HTL:
    [V0 = GI/HI; V1 = GI%HI; V2 = HI/V1; V3 = HI%V1;
     V4 = V2 + 1; V5 = V4 * V3; HIX = 0]
6  for each HOL in HMP
7    Insert below code at the beginning of the header basic block of HOL:
    [V6 = HIX < V5; V7 = V6?0 : V3; V8 = HIX - V7; V9 = V6?V4 : V2;
     V10 = V8%V9; V11 = V10 == 0; Iter = V0 + V11; HIX + = 1]
8  Transform GLN into bypassing form of BGLN
9  if HI >= GI // HI < GI case is similar
10  Insert a copy of BGLN after HIX + = 1 with the enclosing
    "for loop" having loop range 1
11  Add "If" statement to skip execution of non-GOF code when V9!=V10
12  For GOF, recognize the total loop base(GB) and range(GR)
13  for each HOL in HMP
14    Before GOF, use V9, V10 to divide GOF into GB' and GR'
15    for each HOFC in HOL // NewGB = 0
16      GR'' = GR' * (run time of HOFC) / (run time of HOL)
17      GB'' = NewGB
18      NewGB = GB'' + GR''
19      Combine struct ompdata type of GOF and HOF
20      Replace GB and GR of GOF in the combined ompdata
        with GB'' and GR''
21      Move entry BB and preheader BB of loop construct in GOF
        after that of HOF
22      Choose a number of insertion places(IP) in HOF that contains
        one or more HHL (using profiling info)
23      AllIter = 0
24      for each IP chosen above
25        AllIter = AllIter + (total loop iterations of IP)
26        LR = GR'' / AllIter
27        Transform the loop in GOFLN into bypassing form BGOFLN
28      for each IP chosen above
29        Insert a copy of BGOFLN with the enclosing "for loop"
        having range from 0 to LR

```

Figure 6.9: Algorithm for Merging two Merge Places (Nomenclature in Table 6.1)

the “parallel loop” in canonical form, so the loop start, range and step are all loop invariant that must be a constant or some variable passed into the ompfunc as part of ompdata. 2. Even if different compilers have different translation code for the same “parallel for” construct, some key properties are the same. There must be a all to “omp\_get\_num\_threads()” to know how many threads to split loop iterations into, which is in turn divided by the loop range to get the range for one thread. Thus extracting the dividend of the division gives us the loop range. Similarly, the loop start and step could be inferred from the entry basic block of outlined ompfunc. This step is needed in Line 25 in Algorithm 6.9.

#### 6.4.2 Loop Nest Bypassing

The unique property of “bypassing” form of loop nest is that 1. each iteration of this loop nest enters through the top loop. 2. Depending on the iteration number, hot loops(GOL and the loop in GOF that directly contains the hot line.) are always entered, but some non-hot loops are skipped. These properties enable us to insert several copies of GLN into HLN or GOFLN into HOFLN without violating correctness. To implement the bypassing form of loop nest, in each loop body of the loop nest except the innermost one, the code precedes and follows its child loop is enclosed in a if statement respectively. If the child index variable is equal to the loop end value, the loop body is entered. Otherwise it is skipped.

## Chapter 7

### CSSMT FINE-GRAINED MERGE

The overall structure of CSSMT are largely three steps: 1. profiling 2. coarse-grained merge 3. fine-grained merge. This chapter describes the last step: fine-grained merge. Fine-grained merge is a radically different problem from coarse-grained merge in that it is a highly target dependent optimization. In the common situation, target-dependent information usually includes stable parameters such as instruction cycles, memory consumption, explicit and implicit architectural states, and so forth. These information could be collected from instruction reference manual or optimization manual published by the chip vendors or be measured using micro-benchmarking in a systematic way, last but not least, measured by PMU based sampling and then inferred from sampling information indirectly. These target information share the commonality that they are almost always stable or fluctuate in a predictable and systematic way.

On the contrary, the information CSSMT fine-grained merge need is the exploitable empty issue slots that are resulted from a plethora of dynamic or architectural implementation factors such as dynamic resource availability, instruction fetching and reorder buffer policy, and so forth. Part of this information could be estimated by profiling. However, a lot of these factors are really hard to know.

Above implies two methods to achieve SMT effect in fine-grained merge step. The first is to exhaustively search for a mix of hot spots that are showing optimal performance. However, either host of guest hot spot contains at least tens or hundreds of native instructions. There would be hundreds or even millions of different mixes to test, which is far from practical. To remedy this problem, it requires effective search space pruning technique to reduce search space dramatically. The other method is to

use heuristic and programmer’s knowledge of the targeted workload to make the fine-grained merging decision. For example, if the programmer knows in prior that the hot spot code does intensive vectorized float point operations, it makes sense to skip the merging of this hot spot to avoid performance degradation as the float point port in the pipeline is a bottleneck preventing any performance boost from applying CSSMT.

The CSSMT fine-grained merge step uses the second method: with heuristics and provided programmer intervention option. The heuristic is mixing host and guest hot spots in bundle unit containing instructions of either host or guest thread. The bundle size depends on the target architecture. This effectively looks like using the first method with greatly pruned search space. On the other hand, the programmer could override the bundle size to control the mixing granularity. For high-performance code sections, coarse granularity brings better performance.

## 7.1 Granularity

In CSSMT, the boundary between coarse-grained and fine-grained merge and deliberately flexible with the help of coroutine and path-sensitive coarse-merge algorithm described in chapter 6. The rationale is to reduce CSSMT overhead. CSSMT incurs both static and dynamic merging overhead due to the extra code inserted to coordinate control flows of two programs and the coroutine logic embedded to simulate intra-thread multi-tasking. The major transformation actions in CSSMT include **merging across selection branching**, **merging across function call** and **merging across loop**. **Merging across selection branching** only inserts single function calls as compensation code. For non-critical paths, these inserted calls would not be inlined causing unnecessary code bloat. Also, these calls do not cause performance issue since they are on the cold path. **Merging across function call** incurs no code size overhead and dynamic overhead. For each level of the merged loop, **merging across loop** duplicate both host and guest loop body and inserts one jump block. Coordinate implementation gives rise to small code size overhead but potentially considerable dynamic overhead due to its position on the hot path. Thus INSERT-BARRIER function

illustrated in Figure 6.2 needs to make the critical decision about where to put the synchronization point for both threads.

The boundary closing to coarse-grained merge means the call graph path being applied coarse-grained merge is relatively short. It makes coarse-grained merge inserts less code to maintain CSSMT correctness but leaves fine-grained merge a broader code region to make mixing decisions. On the other hand, the boundary closing to fine-grained merge means the code section need target dependent mixing is relatively large.

## Chapter 8

### CSSMT BUILD SYSTEM INTEGRATION

CSSMT is a kind of profile guided optimization (PGO) in terms of build workflow. In PGO, the workload is first going through the training run. Then the running time metrics are saved to a disk file where the file format depends on the profiling method and compiler input constraint. For CSSMT, the file format is a custom-defined text format that is both readable by programmers and parsable by the compiler (LLVM). The format is generated using methods described in chapter 4.

The second stage of the PGO compilation is to feed the profile data file back into the same compiler that generates the training build of the workload. With the workload running time behavior data, the compiler could apply more powerful optimizations. For CSSMT, not only the host and guest profile file need to be fed back to the compiler, but also the whole program intermediate representation of guest thread must be specified.

Considering the workflow of PGO compilation, to easily integrate CSSMT into the compilation tool and environments that are familiar to most programmers, CSSMT only adds two flags to the compiler driver to invoke its functionality. The first flag is called `--cssmt-prof` which intuitively means CSSMT profiling flag. It is used in both the profiling compile phase and final compile phase. The second flag is simply called `--cssmt` which intuitively translates to ‘turning on CSSMT for this build’. This flag is for the linking stage of final optimized compile phase. In next section, the internal functionality and the usage of these two flags are described.

## 8.1 Profiling Compile Phase (`cssmt-prof`)

`--cssmt-prof` is a boolean flag in that it does not take any input. `--cssmt-prof` embodies two sets of flags serving particular purposes. One set of flags are for generating the extra whole program IR of the program. At compile time, it is `-flto`, which tells the compiler to emit IR to a special section of the object file; at link time, it is `-fuse-ld=gold -flto -Wl,-plugin-opt=save-temps`. `-fuse-ld=gold` tells the compiler driver to use Linux gold as the linker to link the object files. The gold linker has a plugin to enable it to interact with LLVM LTO module to perform linking, optimization and code generation. `-Wl,-plugin-opt=save-temps` instructs the linker to emit the whole program IR corresponding to the final executable. This extra IR file is named after the normal executable but with an extra suffix `‘.bc’` which matches what is contained in the file: the LLVM binary IR.

The other set of flags are to preserve debug information in the final executable in order to translate back to source level structure in the CSSMT merge step. They are those in `CFLAGS` except `-lto`. `-g` enables full debug information generation. It is arguably overreach for profiling purpose. However it could help increasing the precision of mapping program counter back to source code. `-fno-omit-frame-pointer` and `-mno-omit-leaf-frame-pointer` prevent the compiler to use frame register for other purposes. This is needed since CSSMT sampling based profiling rely on program counters in these registers to restore the call chain in the source code level. In conclusion, `--cssmt-prof` internally breaks down to

```
CFLAGS = -flto -g -fno-omit-frame-pointer -mno-omit-leaf-frame-pointer
CLINKFLAGS = -fuse-ld=gold -flto -Wl,-plugin-opt=save-temps
```

## 8.2 Final Compile/Merge Phase (`cssmt <string>`)

In this phase, `--cssmt-prof` is still used as one of compile flags, the reason being the same as discussed in section 8.1.

Linker flag `--cssmt` takes a comma-separated string as input. Using this flags triggers the CSSMT compilation. The flag string specifies three full file paths: `host_prof`

is host profile data file; *guest\_ir* is guest thread whole program IR file. *guest\_prof* is guest profile data file. `--cssmt` internally breaks down to

```
CLINKFLAGS = -fuse-ld=gold -flto
             -cssmt-input <host_prof;guest_ir;guest_prof;runtime>
             libboost\_context.a libboost\_thread.a libboost\_system.a
```

Notice the runtime is preinstalled with the compiler. So there is no need to specify it by the user. Furthermore, the linking of the Boost Context library is implicitly turned on by the compiler when the linker flag `--cssmt` is specified.

## Chapter 9

### EXPERIMENTAL RESULTS

CSSMT is a target-dependent compiler optimization. It makes sense to test the effectiveness of CSSMT on multiple target computer architecture. This thesis chooses two most popular computer architectures: X86-64 and AArch64. In recent years, interesting and popular workloads such as those from artificial intelligence, machine learning, computer graphics and big data domain require much more computing power and data set. Most desktops, warehouse servers and even hand-held devices like cell-phone are adopting 64-bit computing. So in this work, only 64-bit architectures are tested.

X86-64 being the most dominant architecture have an immense presence and both client and server computing scenario which represents two sets of divergent workloads. One is latency sensitive whereas the other is throughput sensitive. Also, X86-64 have a full two-way SMT implementation called Hyper-threading. Even if Hyper-threading is much less complicated SMT implementation than those on IBM POWER5 processors and later, it still a good baseline for CSSMT to compare with it considering both being a two-way SMT. Furthermore, X86-64 is an out-of-order architecture which means that it is pretty capable of identifying empty issue slots in the pipeline and issuing dependence free instructions. The dominate amount of empty issue slots come from memory system, either level one cache or long latency main memory load store. It is interesting to see how CSSMT effect in terms of filling these empty slots (comparing hyper-threading).

Another tested architecture is AArch64. AArch64 is the ARM's 64-bit architecture. It is widely used in modern mainstream cellphones and even in energy efficient

servers. It represents another totally different computing trend. That is energy efficiency computing. For that reason, SMT is not available in ARM right now and in the foreseeable future. Testing CSSMT on AArch64 reveals its effect in situations where CSSMT is used as the replacement of SMT. Also, AArch64 used in this thesis is A53 which is an in-order processor. The hardware does not assist in exploiting instruction level parallelism (ILP) to improve pipeline efficiency. In this regard, CSSMT is tested for its effect to converting thread level parallelism (TLP) to instruction level parallelism (ILP) without any hardware support.

The benchmark programs are chosen from SPEC2006 and single-threaded version of NAS Parallel Benchmarks (NPB). Benchmarks written in Fortran are excluded since LLVM lacks official Fortran frontend at this point. Benchmarks written in C++ are also avoided due to the insufficient support of debugging information mapping in CSSMT. The C version of NPB [66] is used. It will be supported in future work. For SPEC2006, the profiling phase uses the ‘train’ input size for all selected benchmarks whereas the running phase also uses the ‘train’ input size. For NPB benchmark, both the profiling run and the final run use the same input size. The input size is picked so that the running time is not too small to get stable test results, at the same time, not too big to save running time. For both SPEC2006 and NPB results, not every pair of benchmarks is displayed due to the space limit. However, the same conclusion still holds.

The compilation environment is set up in Ubuntu 14.04 OS. CSSMT is implemented in Android NDK [38]. Although Android NDK is supposed to be compiling for mobile devices only, we also use it X86-64 compilation for simplicity. Due to unresolved toolchain bugs, the results ruled out miscompiled benchmark pairs.

The rest of the chapter presents X86-64 and AArch64 results separately.

## 9.1 X86-64

The testing environment for X86-64 is shown in figure 9.1. We choose ‘train’ input size for all benchmarks so that the running time is not too long. For benchmarks

Processor	Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz, quadcore with two-way static SMT, 8M L3 cache, Turbo Boost disabled
Memory	16GB DDR3-1600, Max Memory Bandwidth 25.6 GB/s
OS	Ubuntu 14.04.4 LTS 64bit
Benchmarks	<b>SPEC2006:</b> 401.bzip2, 403.gcc, 429.mcf, 445.gobmk, 456.hmmer, 458.sjeng, 462.libquantum, 464.h264ref, 433.milc, 470.lbm, 482.sphinx3 <i>Profiling input size:</i> train <i>Benchmarking input size:</i> train <b>NAS Parallel Benchmarks:</b> ep.A.x, is.B.x, cg.A.x, mg.B.x, ft.B.x, sp.A.x, bt.A.x, lu.A.x, dc.W.x, ua.A.x <i>Class W:</i> 90's workstation size (small) <i>Classes A, B, C:</i> standard test problems; ~4x size increase going from one class to the next
Compiler	Clang + LLVM svn4393122 + CSSMT

Table 9.1: Experiment Machine Configuration (X64)

with more than sub-inputs such as `401.bzip2`, the one with a running time less than 30 seconds is chosen. The same goes for the NPB benchmark. For each benchmark, the input size is chosen so that the running time is practical. The specific LLVM version `svn4393122` is picked since it shares the code with the LLVM version used by released Android NDK r16b. It is for implementation simplicity.

### 9.1.1 Top-Down Analysis

For both SPEC2006 and NPB, we use the Intel-specific Top-Down analysis [78] to demonstrate workload performance patterns and CSSMT effect on the architectural level. Performance counter data are collected using Linux Perf to support Top-Down analysis. The result of the Top-Down analysis is neither the profiling output of CSSMT nor the CSSMT transformation step input. It gives the idea of where are the architectural bottlenecks. However, it does not guarantee CSSMT effect, and it can not predict if CSSMT is beneficial or not. Contrary to the intuition, the Top-Down analysis is quite good at explaining cases where CSSMT does not work well. However, it is relatively not reliable as an absolute indicator of CSSMT will improve performance. Usually, it gives more information when considering several cases at the same time.

Table 9.2: SPEC2006 and NAS Parallel Benchmarks Top-Down Results on Intel(R) Xeon(R) CPU E3-1231 v3  
(Highlighted cells represent the potential architectural bottleneck identified by Top-Down)

	(FB) Frontend Bound	(BS) Bad Speculation	Backend Bound	Retiring	FB Latency	FB Bandwidth	BS Mispredicts	BS Clears	Backend Memory	Backend Core	Retiring Base	Retiring Sequencer
433.milc	0.56	0.27	80.01	19.09	0.61	-0.15	0.08	0.19	72.40	7.62	18.86	0.23
401.bzip2	7.30	26.32	21.81	44.69	0.02	7.15	26.11	0.17	9.55	12.26	44.65	0.04
403.gcc	22.91	12.28	24.59	39.89	0.02	1.59	12.03	0.07	13.62	10.97	37.83	2.06
429.mcf	3.00	9.18	77.08	10.37	0.44	0.16	8.95	0.02	65.40	11.68	10.36	0.02
433.milc	0.56	0.27	80.01	19.09	0.61	-0.15	0.08	0.19	72.40	7.62	18.86	0.23
445.gobmk	33.75	26.74	12.15	27.42	0.02	17.72	26.64	0.04	4.43	7.73	27.16	0.26
456.hmmer	1.97	1.79	36.61	59.97	1.18	0.58	1.71	0.05	3.79	32.82	59.96	0.01
458.sjeng	26.10	14.92	18.97	39.97	12.77	8.26	14.57	0.03	11.49	7.48	39.95	0.02
462.libquantum	10.15	3.00	8.14	76.95	4.48	8.40	2.94	0.00	0.70	7.43	76.94	0.01
464.h264ref	6.08	5.07	26.28	61.49	0.01	0.37	5.01	0.10	9.63	16.65	61.28	0.20
470.lbm	0.39	0.33	69.06	30.48	0.35	-0.11	0.31	0.02	32.37	36.69	30.45	0.03
482.sphinx3	3.19	9.22	34.00	53.13	4.35	2.78	8.48	0.59	15.88	18.12	52.92	0.21
ep.A	14.99	18.68	50.95	15.40	28.11	2.42	4.07	14.62	6.41	44.54	14.67	0.73
is.B	0.12	0.06	79.37	20.40	0.20	0.02	0.00	0.06	18.20	61.17	20.36	0.04
cg.A	7.01	5.45	49.66	37.68	5.69	1.60	5.37	0.05	21.52	28.14	37.63	0.04
mg.B	0.94	0.80	47.51	50.67	1.52	0.40	0.77	0.03	17.28	30.23	50.61	0.06
ft.B	2.32	2.24	53.15	42.53	4.80	-1.28	0.85	1.39	29.83	23.32	42.33	0.20
sp.A	0.83	0.65	55.51	42.87	0.56	0.40	0.62	0.03	35.81	19.70	42.84	0.03
bt.A	1.01	0.11	39.82	59.00	0.29	0.71	0.10	0.01	21.73	18.09	58.99	0.01
lu.A	0.66	0.21	61.38	37.79	0.01	0.43	0.19	0.02	39.43	21.95	37.78	0.01
dc.W	16.32	44.10	17.68	21.33	18.31	-1.79	37.34	5.67	5.81	11.87	20.99	0.33
ua.A	2.03	8.29	50.14	39.78	12.18	2.01	2.99	5.34	28.85	21.29	39.21	0.57

In the Top-Down analysis, bottlenecks are put into four levels of categories (The first four columns in table 9.2. The rest of the columns corresponds to the second level bottleneck categories.). Each bottleneck is associated with a weight, which represents the percentage of empty slots caused by this bottleneck in all pipeline slots during execution. For four issue pipeline, there are four pipeline slots in each cycle. Theoretically, combined weights of each level should be 100%. However, if the weight of a bottleneck is larger than that of any of its ancestor bottlenecks, its weight carries no meaning [78]. For example, in the analysis data we collected, MEM\_Bandwidth of IS is significantly larger than MEM\_Bound. But it does not mean that bandwidth is a bottleneck for IS. On the contrary, MG suffers bandwidth problem by its high MEM\_Bound and MEM\_Bandwidth. MG is known to be memory intensive. Retiring is not a bottleneck category. Instead, it reflects slots that eventually get retired. The high number of Retiring means less bottleneck in frontend and execution stages.

In table 9.2, 20 of 22 tests are backend bounded (445.gobmk and 458.sjeng being the exception). Out of 20, roughly half are memory bound whereas the other half is core bound. The memory bound could be cache-bound, main memory bandwidth or main memory latency bound. Cache bound and main memory bandwidth mean distinct things for CSSMT. When a workload is under-utilizing the pipeline due to cache bound reasons, it is highly likely that scheduling another thread in the pipeline through either hardware-based SMT or CSSMT is beneficial for the performance. However, when the workload is limited by the total memory bandwidth, it will be degrading performance since the other thread could make forward progress with no memory traffic. The newly added memory traffic only adds extra time to their original combined running time.

For backend bound cases where Top-Down analysis failed to infer the second level bottleneck (401.bzip2, 470.lbm, 482.sphinx3, bt.A), it means memory and core are equally bounded. The bounding pattern (interleaved memory and core bound, or consecutive memory or core bound), and the severeness of bounding all decide if the benchmark could reap benefits from using another SMT thread. Core bound cases could also be SMT friendly. When one SMT thread is heavily using integer ports (or

Table 9.3: NAS Parallel Benchmarks Results on Intel(R) Xeon(R) CPU E3-1231 v3  
(L: host thread; R: guest thread; CPI: Cycle Per Instruction)

L Orig	R Orig	L Orig CPI	R Orig CPI	L Orig Time(s)	R Orig Time(s)	CSSMT Time(s)	Speedup (%)
bt.A	dc.W	169.88	5.11	42.7	26.33	78.01	-13.02
bt.A	is.B	169.88	20.07	42.7	5.24	49.57	-3.42
bt.A	mg.B	169.88	20.35	42.7	6.93	52.32	-5.43
bt.A	sp.A	169.88	38.84	42.7	24.2	69.16	-3.39
bt.A	ua.A	169.88	22.87	42.7	18.91	64.63	-4.9
cg.A	bt.A	5.92	169.88	1.18	42.7	46.08	-5.03
cg.A	ft.B	5.92	18.94	1.18	45.57	46.9	-0.33
cg.A	is.B	5.92	20.07	1.18	5.24	6.41	0.13
cg.A	mg.B	5.92	20.35	1.18	6.93	8.09	0.23
ep.A	dc.W	13.36	5.11	36.22	26.33	64.32	-2.83
ep.A	ep.A	13.36	13.36	36.22	36.22	72.64	-0.27
ep.A	is.B	13.36	20.07	36.22	5.24	41.44	0.06
ep.A	sp.A	13.36	38.84	36.22	24.2	60.43	-0.01
ft.B	bt.A	18.94	169.88	45.57	42.7	91.07	-3.18
ft.B	ep.A	18.94	13.36	45.57	36.22	82.01	-0.27
ft.B	is.B	18.94	20.07	45.57	5.24	50.86	-0.11
ft.B	sp.A	18.94	38.84	45.57	24.2	69.98	-0.31
is.B	cg.A	20.07	5.92	5.24	1.18	6.41	0.19
is.B	dc.W	20.07	5.11	5.24	26.33	33.17	-5.08
is.B	ep.A	20.07	13.36	5.24	36.22	41.46	0.01
is.B	is.B	20.07	20.07	5.24	5.24	10.42	0.61
is.B	mg.B	20.07	20.35	5.24	6.93	12.11	0.49
is.B	sp.A	20.07	38.84	5.24	24.2	29.34	0.33
lu.A	bt.A	75.96	169.88	31.76	42.7	76.84	-3.2
lu.A	cg.A	75.96	5.92	31.76	1.18	33.34	-1.21
lu.A	dc.W	75.96	5.11	31.76	26.33	54.82	5.63
lu.A	ep.A	75.96	13.36	31.76	36.22	68.74	-1.11
lu.A	ua.A	75.96	22.87	31.76	18.91	49.45	2.41
mg.B	dc.W	20.35	5.11	6.93	26.33	30.78	7.46
mg.B	ep.A	20.35	13.36	6.93	36.22	43.13	0.07
mg.B	ft.B	20.35	18.94	6.93	45.57	52.41	0.17
mg.B	is.B	20.35	20.07	6.93	5.24	12.11	0.55
mg.B	mg.B	20.35	20.35	6.93	6.93	14.03	-1.17
mg.B	ua.A	20.35	22.87	6.93	18.91	23.2	10.24
sp.A	dc.W	38.84	5.11	24.2	26.33	47.73	5.54
sp.A	ep.A	38.84	13.36	24.2	36.22	60.49	-0.12
sp.A	is.B	38.84	20.07	24.2	5.24	29.34	0.35
sp.A	mg.B	38.84	20.35	24.2	6.93	31.08	0.17
sp.A	sp.A	38.84	38.84	24.2	24.2	48.96	-1.15
sp.A	ua.A	38.84	22.87	24.2	18.91	40.86	5.23
ua.A	bt.A	22.87	169.88	18.91	42.7	64.56	-4.79
ua.A	dc.W	22.87	5.11	18.91	26.33	48.08	-6.27
ua.A	ft.B	22.87	18.94	18.91	45.57	65.22	-1.15
ua.A	is.B	22.87	20.07	18.91	5.24	24.14	0.04
ua.A	mg.B	22.87	20.35	18.91	6.93	25.85	-0.01
ua.A	sp.A	22.87	38.84	18.91	24.2	43.08	0.06

SIMD INT ports), the other thread could make use of the available float point ports (or SIMD FP ports).

### 9.1.2 NAS Parallel Benchmarks

The CSSMT results for NPB on X86-64 are shown in table 9.3. Overall, for the **Speedup** column, the deviation from the 0% is not very big. This is in part due to the profiling information mapping are still not precise enough. So some merging opportunities are missed (either good or bad for performance). `bt.A` does not play well with other benchmarks as expected since it is both memory and core bound at the same time. The other reason is that `bt.A` is already pretty fit for the pipeline by having %59 retiring. There are not many effective empty slots in the pipeline to be used by another thread. On the contrary, with `dc.W`, the combine effects of mispredicts and backend bound cause %13 performance drop. `mg.B`, `is.B` and `sp.A` are generally nicer to other threads. `mg.B` and `is.B` are more core bound oriented. This leaves plenty space in the pipeline for other thread to take steps. `sp.A` is mainly memory bound, which, on the other hand, gives core bound threads chances to execute. The biggest improvement comes from `mg.B` and `ua.A` mix. They are bounded by the resources that their peer has to offer. So they complement each other pretty well regarding pipeline resource usage patterns. Overall, the CSSMT results for NPB on X86-64 are not as good as their AArch64 counterpart since the out-of-order nature of X86 make them already excellent at filling empty slots in the pipeline.

### 9.1.3 SPEC2006

SPEC2006 results on X86-64 are listed in table 9.4. Overall, `456.hammer` and `470.1bm` have relatively better speedup than other benchmarks. The Top-Down profile of `470.1bm` shows that although the backend bound is almost %70, the bounded cycles equally distribute between core and memory. This is an exemplary candidate for CSSMT since the thread is not blocking any single point of the pipeline and the thread itself has trouble retiring cycles.

Table 9.4: SPEC2006 Results on Intel(R) Xeon(R) CPU E3-1231 v3  
(L: host thread; R: guest thread; CPI: Cycle Per Instruction)

L Orig	R Orig	L Orig CPI	R Orig CPI	L Orig Time(s)	R Orig Time(s)	CSSMT Time(s)	Speedup (%)
401.bzip2	401.bzip2	6.78	6.78	3.51	3.51	7.6	-8.37
401.bzip2	429.mcf	6.78	5.19	3.51	12.14	17.08	-9.16
401.bzip2	433.milc	6.78	51.9	3.51	10.82	14.84	-3.57
401.bzip2	456.hmmer	6.78	26.49	3.51	39.74	41.83	3.27
401.bzip2	462.libquantum	6.78	6.18	3.51	1.25	5.31	-11.7
401.bzip2	470.lbm	6.78	48.6	3.51	23.66	26.84	1.2
429.mcf	401.bzip2	5.19	6.78	12.14	3.51	17.51	-11.93
429.mcf	429.mcf	5.19	5.19	12.14	12.14	31.13	-28.22
429.mcf	433.milc	5.19	51.9	12.14	10.82	24.58	-7.05
429.mcf	456.hmmer	5.19	26.49	12.14	39.74	50.79	2.1
429.mcf	462.libquantum	5.19	6.18	12.14	1.25	14.08	-5.2
429.mcf	470.lbm	5.19	48.6	12.14	23.66	36.36	-1.58
433.milc	401.bzip2	51.9	6.78	10.82	3.51	14.62	-2.04
433.milc	429.mcf	51.9	5.19	10.82	12.14	23.72	-3.29
433.milc	433.milc	51.9	51.9	10.82	10.82	27.68	-27.84
433.milc	456.hmmer	51.9	26.49	10.82	39.74	53.14	-5.1
433.milc	462.libquantum	51.9	6.18	10.82	1.25	12.59	-4.34
433.milc	470.lbm	51.9	48.6	10.82	23.66	33.67	2.35
445.gobmk	401.bzip2	5.29	6.78	4.48	3.51	8.78	-9.87
445.gobmk	429.mcf	5.29	5.19	4.48	12.14	17.73	-6.69
445.gobmk	433.milc	5.29	51.9	4.48	10.82	18.19	-18.81
445.gobmk	456.hmmer	5.29	26.49	4.48	39.74	47.21	-6.77
445.gobmk	462.libquantum	5.29	6.18	4.48	1.25	6.78	-18.3
445.gobmk	470.lbm	5.29	48.6	4.48	23.66	27.84	1.06
456.hmmer	401.bzip2	26.49	6.78	39.74	3.51	42.31	2.16
456.hmmer	429.mcf	26.49	5.19	39.74	12.14	50.78	2.12
456.hmmer	433.milc	26.49	51.9	39.74	10.82	53.16	-5.14
456.hmmer	456.hmmer	26.49	26.49	39.74	39.74	79.74	-0.33
456.hmmer	470.lbm	26.49	48.6	39.74	23.66	61.4	3.15
458.sjeng	401.bzip2	5.46	6.78	89.53	3.51	95.78	-2.95
458.sjeng	429.mcf	5.46	5.19	89.53	12.14	104.62	-2.9
458.sjeng	445.gobmk	5.46	5.29	89.53	4.48	93.18	0.88
458.sjeng	456.hmmer	5.46	26.49	89.53	39.74	138.29	-6.98
458.sjeng	462.libquantum	5.46	6.18	89.53	1.25	91.77	-1.1
458.sjeng	470.lbm	5.46	48.6	89.53	23.66	115.57	-2.1
462.libquantum	401.bzip2	6.18	6.78	1.25	3.51	5.43	-14.19
462.libquantum	429.mcf	6.18	5.19	1.25	12.14	14.06	-5.02
462.libquantum	433.milc	6.18	51.9	1.25	10.82	12.41	-2.83
462.libquantum	462.libquantum	6.18	6.18	1.25	1.25	3.15	-26.19
462.libquantum	470.lbm	6.18	48.6	1.25	23.66	24.2	2.83
470.lbm	401.bzip2	48.6	6.78	23.66	3.51	26.78	1.38
470.lbm	429.mcf	48.6	5.19	23.66	12.14	34.67	3.14
470.lbm	433.milc	48.6	51.9	23.66	10.82	33.73	2.18
470.lbm	456.hmmer	48.6	26.49	23.66	39.74	60.93	3.89
470.lbm	462.libquantum	48.6	6.18	23.66	1.25	24.34	2.28
470.lbm	470.lbm	48.6	48.6	23.66	23.66	46.86	0.95
482.sphinx3	401.bzip2	11.23	6.78	5.88	3.51	10.02	-6.8
482.sphinx3	429.mcf	11.23	5.19	5.88	12.14	20.5	-13.74
482.sphinx3	433.milc	11.23	51.9	5.88	10.82	18.09	-8.26
482.sphinx3	456.hmmer	11.23	26.49	5.88	39.74	46.55	-2.03
482.sphinx3	462.libquantum	11.23	6.18	5.88	1.25	7.75	-8.73
482.sphinx3	470.lbm	11.23	48.6	5.88	23.66	28.93	2.04

Processor	4x1.55 GHz Cortex-A53 64-bit ARMv8-A
Memory	3 GB Low Power DDR4 1600MHz RAM
OS	Ubuntu 14.04.4 LTS 64bit (Host) Android 7.1.2 Nougat (Target)
Benchmarks	<b>SPEC2006:</b> 401.bzip2, 403.gcc, 429.mcf, 445.gobmk, 456.hmmmer, 458.sjeng, 462.libquantum, 464.h264ref, 433.milc, 470.lbm, 482.sphinx3 <i>Profiling input size:</i> test <i>Benchmarking input size:</i> test <b>NAS Parallel Benchmarks:</b> ep.S.x, ep.W.x, ep.A.x, is.A.x, is.B.x, cg.A.x, mg.A.x, mg.B.x, ft.A.x, sp.W.x, bt.W.x, lu.W.x, ua.S.x, ua.W.x <i>Class W:</i> 90’s workstation size (small) <i>Classes A, B, C:</i> standard test problems; ~4x size increase going from one class to the next
Compiler	Android NDK r16b with clang-svn4393122 + CSSMT

Table 9.5: Experiment Machine Configuration (AArch64)

The biggest slowdown come from the self merge of 429.mcf, 433.milc and 462.libquantum. 429.mcf, 433.milc are predominately restricted by memory. Self-merging simply makes the pipeline even more sluggish. 462.libquantum has a %77 retiring rate. Any extra SMT thread will disrupt its microops flow.

## 9.2 AArch64

We cross-compile SPEC2006 and NPB benchmark to target a Cortex-A53 powered Nexus 6P phone. The officially supported native development toolchain for Android platform (Android NDK) is used for the cross-compilation. The LLVM-based compiler inside Android NDK is modified to add CSSMT functionality. The profiling step uses NDK Simpleperf [39]. Although NDK Simpleperf is also able to collect various hardware counters, the counters on ARM are much more limited than X86-64. Also, there is no well-supported tool such Top-Down and Vtune [43]. We use cycles and execution time to profile the workload. Again, the problem size is chosen to stress the benchmark in manageable amount time.

Table 9.6: NAS Parallel Benchmarks Results on ARM Cortex-A53  
(**L**: host thread; **R**: guest thread; **CPI**: Cycle Per Instruction)

L Orig	R Orig	L Orig CPI	R Orig CPI	L Orig Time(s)	R Orig Time(s)	CSSMT Time(s)	Speedup (%)
bt.W	bt.W	3532.58	3532.58	22.86	22.86	54.74	-19.74
bt.W	cg.A	3532.58	467.36	22.86	36.04	60.44	-2.61
bt.W	ep.S	3532.58	237.55	22.86	3.85	26.69	0.06
bt.W	ft.A	3532.58	463.91	22.86	22.54	45.13	0.58
cg.A	bt.W	467.36	3532.58	36.04	22.86	58.98	-0.14
cg.A	ft.A	467.36	463.91	36.04	22.54	58.7	-0.2
ep.A	cg.A	239.95	467.36	61.59	36.04	107.74	-10.34
ep.A	lu.W	239.95	2013.23	61.59	58.87	120.58	-0.09
ep.A	mg.B	239.95	1778.41	61.59	102.02	167.79	-2.55
ep.A	ua.W	239.95	544.02	61.59	27.03	88.57	0.05
ep.S	cg.A	237.55	467.36	3.85	36.04	42.46	-6.43
ep.S	ep.S	237.55	237.55	3.85	3.85	7.7	-0.15
ep.W	cg.A	238.93	467.36	7.66	36.04	50.26	-14.98
ep.W	ep.A	238.93	239.95	7.66	61.59	69.32	-0.09
ep.W	ep.S	238.93	237.55	7.66	3.85	11.5	0.1
ep.W	is.A	238.93	4803.74	7.66	6.16	13.85	-0.16
ep.W	sp.W	238.93	931.35	7.66	54.18	63.42	-2.55
ft.A	bt.W	463.91	3532.58	22.54	22.86	44.44	2.1
ft.A	cg.A	463.91	467.36	22.54	36.04	57.82	1.3
ft.A	ep.A	463.91	239.95	22.54	61.59	83.01	1.33
ft.A	ep.S	463.91	237.55	22.54	3.85	25.29	4.14
ft.A	ep.W	463.91	238.93	22.54	7.66	29.06	3.77
ft.A	ft.A	463.91	463.91	22.54	22.54	46.27	-2.66
ft.A	is.A	463.91	4803.74	22.54	6.16	27.45	4.35
ft.A	is.B	463.91	5316.15	22.54	25.21	46.51	2.57
ft.A	mg.B	463.91	1778.41	22.54	102.02	121.49	2.46
is.A	cg.A	4803.74	467.36	6.16	36.04	36.83	12.73
is.A	ft.A	4803.74	463.91	6.16	22.54	27.33	4.77
is.A	mg.A	4803.74	1413.96	6.16	29.16	33.87	4.12
is.A	sp.W	4803.74	931.35	6.16	54.18	63.01	-4.43
is.A	ua.W	4803.74	544.02	6.16	27.03	33.16	0.08
is.B	cg.A	5316.15	467.36	25.21	36.04	56.36	7.98
is.B	ep.A	5316.15	239.95	25.21	61.59	86.87	-0.08
is.B	ft.A	5316.15	463.91	25.21	22.54	46.54	2.52
is.B	mg.A	5316.15	1413.96	25.21	29.16	55.71	-2.48
is.B	sp.W	5316.15	931.35	25.21	54.18	83.51	-5.2
lu.W	ft.A	2013.23	463.91	58.87	22.54	80.89	0.64
mg.A	cg.A	1413.96	467.36	29.16	36.04	64.28	1.42
mg.A	ep.W	1413.96	238.93	29.16	7.66	35.63	3.24
mg.A	is.A	1413.96	4803.74	29.16	6.16	34.5	2.33
mg.A	mg.A	1413.96	1413.96	29.16	29.16	57.98	0.59
mg.A	ua.S	1413.96	523.12	29.16	4.39	33.34	0.64
mg.A	ua.W	1413.96	544.02	29.16	27.03	55.22	1.72
mg.B	mg.A	1778.41	1413.96	102.02	29.16	138.8	-5.81
sp.W	cg.A	931.35	467.36	54.18	36.04	85.29	5.46
sp.W	ep.S	931.35	237.55	54.18	3.85	59.29	-2.18
sp.W	ep.W	931.35	238.93	54.18	7.66	63.27	-2.31
sp.W	lu.W	931.35	2013.23	54.18	58.87	116.25	-2.83
sp.W	sp.W	931.35	931.35	54.18	54.18	111.47	-2.87
sp.W	ua.S	931.35	523.12	54.18	4.39	61.22	-4.52
ua.S	ep.A	523.12	239.95	4.39	61.59	66.03	-0.08
ua.S	ft.A	523.12	463.91	4.39	22.54	25.99	3.48
ua.S	is.A	523.12	4803.74	4.39	6.16	10.43	1.09
ua.S	mg.B	523.12	1778.41	4.39	102.02	111.14	-4.45
ua.W	ft.A	544.02	463.91	27.03	22.54	48.05	3.06
ua.W	is.A	544.02	4803.74	27.03	6.16	32.68	1.52
ua.W	is.B	544.02	5316.15	27.03	25.21	51.74	0.93
ua.W	sp.W	544.02	931.35	27.03	54.18	83.3	-2.58

### 9.2.1 NAS Parallel Benchmarks

Altogether, NPB results on AArch64 are much better than on X86-64. The reason is that A53 is an in-order pipeline design. There is less hardware support for ILP than X86-64 such as that CSSMT can help converting TLP to unused ILP. The second observation is that variation is also larger than on X86-64, which confirms that in-order pipeline are more sensitive to scheduling and the workload performance patterns.

Specifically, `ft.A`, `mg.A`, `is.B` are CSSMT friendly. It mostly matches the result of X86-64. `ft.A` is the exception. On x86-64, `ft.B` has performance lost in all cases. This shows that CSSMT, like hardware-based SMT, is highly target-dependent. Both CSSMT and hardware-based SMT are workload-sensitive.

### 9.2.2 SPEC2006

SPEC2006 results on AArch64 are listed in table 9.7. Like NPB results on AArch64, SPEC2006 on AArch64 shows better overall speedup than on X86-64. CSSMT boosts `470.1bm` again on AArch64. All benchmarks paired with `470.1bm` gains performance increase. Like on X86-64, `433.milc` and `462.libquantum` degraded their speed again. `456.hmmmer` becomes a new example where self-merge causes severe slowdown. `429.mcf` self-merge does not work well, but not as worse as X86-64. Those results have shown that the inherent performance patterns of the thread decide the ultimate benefits of CSSMT. Also, the architecture plays a role in the CSSMT effectiveness.

## 9.3 Mix SPEC2006 And NPB On X86-64

We pick three benchmarks (`cg.B`, `mg.B`, `is.C`) from NPB to represent scientific workloads. `456.hmmmer` and `429.mcf` of SPEC2006 are picked for the similar reason. The results of applying CSSMT on chosen benchmarks are in table 9.8. P1 and P2 are benchmarks being merged. **Standalone** column lists their execution time running alone on an SMT-disabled core. To contrast CSSMT with hardware SMT implementation, **Hyper-Threading** column lists their respective execution time when started

Table 9.7: SPEC2006 Results on ARM Cortex-A53  
(L: host thread; R: guest thread; CPI: Cycle Per Instruction)

L Orig	R Orig	L Orig CPI	R Orig CPI	L Orig Time(s)	R Orig Time(s)	CSSMT Time(s)	Speedup (%)
401.bzip2	401.bzip2	123.33	123.33	12.61	12.61	27.18	-7.74
401.bzip2	429.mcf	123.33	55.45	12.61	45.51	62.74	-7.95
401.bzip2	433.milc	123.33	1473.85	12.61	84.13	96.09	0.67
401.bzip2	456.hmmmer	123.33	399.96	12.61	11.56	24.7	-2.19
401.bzip2	462.libquantum	123.33	235.75	12.61	8.93	21.92	-1.77
401.bzip2	470.lbm	123.33	1119.82	12.61	19.33	30.39	4.86
429.mcf	401.bzip2	55.45	123.33	45.51	12.61	64.92	-11.71
429.mcf	429.mcf	55.45	55.45	45.51	45.51	108.1	-18.77
429.mcf	433.milc	55.45	1473.85	45.51	84.13	129.5	0.1
429.mcf	456.hmmmer	55.45	399.96	45.51	11.56	55.56	2.63
429.mcf	462.libquantum	55.45	235.75	45.51	8.93	52.33	3.85
429.mcf	470.lbm	55.45	1119.82	45.51	19.33	62.6	3.45
433.milc	401.bzip2	1473.85	123.33	84.13	12.61	91.69	5.22
433.milc	429.mcf	1473.85	55.45	84.13	45.51	124.93	3.63
433.milc	433.milc	1473.85	1473.85	84.13	84.13	184.34	-9.56
433.milc	456.hmmmer	1473.85	399.96	84.13	11.56	101.26	-5.83
433.milc	462.libquantum	1473.85	235.75	84.13	8.93	91.58	1.58
433.milc	470.lbm	1473.85	1119.82	84.13	19.33	96.44	6.78
456.hmmmer	401.bzip2	399.96	123.33	11.56	12.61	24.89	-2.99
456.hmmmer	429.mcf	399.96	55.45	11.56	45.51	55.49	2.76
456.hmmmer	433.milc	399.96	1473.85	11.56	84.13	105.83	-10.6
456.hmmmer	456.hmmmer	399.96	399.96	11.56	11.56	27.63	-19.58
456.hmmmer	462.libquantum	399.96	235.75	11.56	8.93	22.5	-9.87
456.hmmmer	470.lbm	399.96	1119.82	11.56	19.33	28.9	6.44
458.sjeng	401.bzip2	52.41	123.33	17.52	12.61	32.6	-8.2
458.sjeng	429.mcf	52.41	55.45	17.52	45.51	68.47	-8.64
458.sjeng	433.milc	52.41	1473.85	17.52	84.13	111.86	-10.05
458.sjeng	456.hmmmer	52.41	399.96	17.52	11.56	35.85	-23.3
458.sjeng	462.libquantum	52.41	235.75	17.52	8.93	29.41	-11.2
458.sjeng	470.lbm	52.41	1119.82	17.52	19.33	36.1	2.02
462.libquantum	401.bzip2	235.75	123.33	8.93	12.61	22.13	-2.76
462.libquantum	429.mcf	235.75	55.45	8.93	45.51	53.06	2.52
462.libquantum	433.milc	235.75	1473.85	8.93	84.13	96.65	-3.86
462.libquantum	456.hmmmer	235.75	399.96	8.93	11.56	22.54	-10.07
462.libquantum	462.libquantum	235.75	235.75	8.93	8.93	22.16	-24.13
462.libquantum	470.lbm	235.75	1119.82	8.93	19.33	26.52	6.14
470.lbm	401.bzip2	1119.82	123.33	19.33	12.61	30.61	4.18
470.lbm	429.mcf	1119.82	55.45	19.33	45.51	62.93	2.94
470.lbm	433.milc	1119.82	1473.85	19.33	84.13	101.34	2.04
470.lbm	456.hmmmer	1119.82	399.96	19.33	11.56	28.99	6.14
470.lbm	462.libquantum	1119.82	235.75	19.33	8.93	26.99	4.47
470.lbm	470.lbm	1119.82	1119.82	19.33	19.33	35.98	6.93

Table 9.8: Experiment Results on SPEC2006 and NAS Benchmark

P1	P2	Standalone		Hyper-Threading			CSSMT		
		P1	P2	P1	P2	Speedup	P1	P2	Speedup
is.C	456.hmmmer	21.49	36.41	25.39	42.25	61.64%	22.5	51	30.67%
is.C	429.mcf	21.49	10.27	21.7	13.07	76.97%	34.71	26.9	-10.97%
cg.B	456.hmmmer	90.01	36.41	101.86	58.85	41.73%	125	89.64	1.58%
cg.B	429.mcf	90.01	10.27	98.12	15.97	13.53%	102.1	18.57	-9.80%
mg.B	456.hmmmer	38.67	36.41	54	50.06	42.11%	85.49	108.79	-39.43%
mg.B	429.mcf	38.67	10.27	49.48	22.26	-2.43%	54.17	24.35	-21.48%

at the same time on the shared SMT-enabled core. **CSSMT** column lists results for this work. The **Speedup** column is calculated with the formula:

$$\text{speedup} = (O1 - (T1 - T2) + O2) / T2 - 1$$

where  $O1, T1$  are total time of program  $P1$  when executed alone and with CSSMT respectively;  $O2, T2$  are total time of program  $P2$  when executed alone and with CSSMT respectively; assume  $T1 \geq T2$ , if not, swap both  $O1, O2$  and  $T1, T2$ , then apply formula again. Considering there is execution time imbalance between two programs, the formula calculates the speedup for the time window that two programs actually have fused instruction stream together. The portion of time where only one program is running does not affect the speedup calculation.

We saw speedup for pairs [is.C,456.hmmmer] and [cg.B,456.hmmmer].

[is.C,456.hmmmer] has the largest speedup 30% for a reason. They both are not MEM\_Bound but L1\_Bound. The ALU is seldom used. After CSSMT, when one thread is waiting for data from L1, the other could do ALU with underutilized ports. The “nice” bottleneck property of is.C make it work well with mg.B and cg.B too. The L1 bound property of is.C does not require a perfect fine-grained merge strategy of CSSMT, as long as the hot spots from threads going together. The unused ports get more chances to be utilized.

Like is.C, mg.B have even more room for improvement. Retiring of mg.B is only 8.69%, comparing with 24.47% of is.C. Then why mg.B does not go well with hmmmer?

The reason is that `mg.B` is bounded by memory bandwidth. Any change of less utilized bandwidth due to merging would be punished by slowdown since fine-grained merging could cause easily generated a less memory intensive kernel. In this case, “disruptive” instructions of `hmm` reduces sustained memory traffic of `mg.B`, causing combined significant slowdown.

The rest of the cases can be analyzed similarly. The takeaway lesson is that CSSMT does not do a good job of scheduling fused instructions well enough that keep the good performance pattern of respective programs while gaining the benefits of merging. A compiler backend that is fully CSSMT aware could really these slowdown cases.

The results may look disappointing, but it shows that coarse-grained merge works really well, causing radical change to the performance. If the coarse-grained merge is not effective, then the combined performance should not make much difference. However, we still need better fine-grained merge strategy.

## 9.4 Conclusion

On X86-64 and AArch64, we saw both performance boost and degradation across all possible benchmark mixes. AArch64 has benefit more from CSSMT due to its in-order pipeline setup, which exposes more issue slots for CSSMT manipulate. The same benefit would cost a lot more design and validation effort to be deployed on AArch64 in hardware. It is also observed that switching host and guest thread in CSSMT sometimes generate different code (For instance `is.A` and `cg.A` on X86-64 figure 9.3). This is due to the CSSMT transformation on CFG introducing a varying amount of overhead for host and guest thread. We should address this problem in future work.

## Chapter 10

### DISCUSSION

This chapter discusses rationals and design choices behind CSSMT.

#### 10.1 Employ Source-to-Source Instead Of IR Transformation?

Compiler infrastructures for Source-to-source transformation such as [55] have language dependent in their representation. For example in Rose compiler, AST has special nodes to support Fortran code. This is not desirable for CSSMT since CSSMT being a TLP-to-ILP tool should be language agnostic to apply to as many applications as possible. Hardware-based SMT works at the instruction level. So it could boost performance for applications written in any language. Moreover, the CSSMT implement its fine-grained merge step in code generation part of LLVM. The source-to-source transformation has no control over the generated native instructions and allocated registers which is indispensable for CSSMT.

#### 10.2 Micro-architecture profiling information Helps Performance?

CSSMT is a profiled guided transformation (PGO). It is currently guided by only sampling the cycles event. The intuition here is that the hot spot code should also be the most pipeline resource constrained. Even if the other parts of the program have shown resource constrained symptom, since it does not consume a considerable amount of cycles, it is not worthwhile to merge them in CSSMT. So in CSSMT, the micro-architecture analysis such as Top Down [78] is not used for transformation. Top-Down analysis is only used to predict if a program would benefit from CSSMT. It gives hints about how the program is using pipeline resource overall. Furthermore, the bottleneck identification does not imply anything about the benefits of its mitigation

[78]. In conclusion, in CSSMT, bottleneck such cache miss, memory hierarchy latency, etc. are not considered during the merging process since it would not bring further performance boost in currently available performance modeling framework such as Top Down [78].

### 10.3 Hot Spot In the Precompiled Library Such As Glibc?

CSSMT merge programs in intermediate representation (IR) level. The pre-compiled library is in binary format which is not recognized by CSSMT. For example, embarrassingly parallel (EP) kernel in NAS benchmark spends 50% execution time in the math library. One possible solution is to have compiler emitting IR in ELF sections (GCC LTO does this). In linking stage, with profiling information, LTO could choose to internalize or inline hot functions in the shared library such as Libc. However, this should be performed only when CSSMT could prove it is beneficial to merge the code in the shared library. For the case of merging math library, if profiling information reveals that math is the bottleneck operation, then it is not a good idea to apply CSSMT since the bottleneck could be relieved by mixing another SMT thread. It would only make it slower.

### 10.4 How To Merge Recursive Function?

The CSSMT merging algorithm works equally correctly for both recursive and non-recursive functions. In figure 5.2 and figure 5.3, `mpm_over` is used to control the ending of either SMT thread. Supposing either merged function is recursive, `mpm_over` would still correctly stop the execution of the other thread. The correctness of merged function is preserved.

### 10.5 Execution Time Imbalance Between Two Threads

The imbalance issue could affect the performance of the mixed program. It comes from the fact that the merge overhead of CSSMT varies according to the call graph and the control flow graph (CFG) of the hot spot functions. These are static program information. However, execution time is runtime information that determined

by factors such as input size. The relatively shorter execution time of either thread would cause the reduced chance of performance boost whereas the CSSMT overhead remains the same. Overall, the potential benefits of CSSMT are diminished.

## Chapter 11

### RELATED WORK

SMT is a hot topic in computer architecture community ([54, 45, 70, 26]). However, most works focus on improving SMT performance. Researches regarding software based SMT are more related to this work. We summarize these as follows.

[46] implements the software simultaneous multithreading by using a thin binary translation layer above architecture. It is a dynamic method to implement SMT contrary to the static method of CSSMT. It adds running time overhead to the executed program due to the software layer in exchange for the flexibility of no modification to the program binary. So recompilation is not necessary, but the burden of code generation which is most likely the job of a compiler is shifted to this thin layer. By being dynamic, [46] also lacks the ability to reorganize the data layout of two separate memory sections when the instruction streams are merged ([79] for example). CSSMT is perfect to accommodate these kinds of optimizations using compilation in the future.

[19] uses a static method to fuse merge independent instruction stream, but it is done inside a *single* program. Thus the benefits would be program speedup instead of throughput in the SMT context. However both CSSMT and [19] are static methods. Removing data dependence is the prerequisite of [19] before the actual control and data flow merge. However, there is a limit [17] of how much dependence can be removed and it is still an ongoing research topic. On the contrary, CSSMT, by manipulating the fact that distinct programs have no control and data dependence, achieve a general-purpose merging schema that not rely on results of other advanced compiler analysis. The applicable scope of [19] is global (function level); CSSMT looks at whole program.

[28] statically merges two independent threads in assembly level. The purpose is to save context switching overhead in the embedded system when satisfying the

real-time requirement. The target program of [28] usually have well-defined hot spots, and the real-time constraint is inherently more coarse-grained than the SMT, in which cycle level granularity is absolutely required to obtain benefits. The transformation is rendered in a customized program dependence tree representation.

[48] does whole function vectorization for data-parallel languages such as GLSL [49], HLSL [62] and Metal [6]. The target is the SIMD instruction sets such as Intel’s SSE and AVX rather than the GPU these languages intended for. In another word, [48] is targeted at DLP instead of TLP. It uses LLVM IR to perform target independent transformation. So it applies to all SIMD-enabled architectures. It uses mask and select generation to convert control flow dependency to data dependency, thus to generate a long sequence of vector instructions.

[65] uses scheduling techniques to simulate SMT effect to increase the power efficiency of embedded workloads. The scheduling unit is different mixes of threads where each thread represents a small task in the embedded machine. Before the workloads are running, [65] compiles all potentially beneficial mixes beforehand. At the running time, [65] uses appropriate heuristics to select mixes. However, the paper lacks detail about the offline compilation part.

To exploit the advanced SMT feature on IBM POWER8, [45] use an offline training & online prediction hybrid approach to auto-tune big data analytics workloads on POWER8. POWER8’s [69] supports up to eight-way SMT (SMT8) for each core. SMT4 and SMT2 are also supported. The core dynamically switches among single-thread, SMT2, SMT4 and SMT8 to achieve increased throughput and energy efficiency. Unlike previous generations in POWER series, POWER8 SMT supports scheduling a thread on any SMT thread position on the core. In the offline training phase, it feeds micro-architecture level data to a machine learning model. The inferred model is utilized for online SMT configuration prediction.

[31, 44] propose to define metrics for whole system workloads. In a system equipped with SMT and multiple cores, the workloads running on the system affect each other’s performance. [31] defines two metrics: latency and throughput to estimate

the system performance as a whole. [44] compares the performance of co-executed applications across different machines. Specifically, it defines exactly how a pair of workloads interact with each other. It is helpful to evaluate techniques such hardware-based SMT and CSSMT where the optimizations are targeted at more than one program.

## Chapter 12

### CONCLUSION AND FUTURE WORK

To bridge the gap between hardware and software in the modern age, we not only need a heterogeneous computing platform to accommodate new workload. The workloads themselves should also have some flexibility to adapt to the underlying hardware. SMT has been proven to be an effective measure to allow TLP and ILP balance. In this thesis, we propose a general method to implement SMT in software using compilation. We evaluate the results on X86-64 and AArch64 machine to show that CSSMT works well in coarse-grained merge stage but not so well on fine-grained merge stage. A maximum 30% speedup is observed.

There is a lot of room to improve. An immediate future work is to make the compiler backend, in particular, register allocation and instruction scheduling aware of the CSSMT. Both coarse-grained merge and fine-grained merge are for control flow. We also need data flow merge. Global data reconstruct could be really helpful since data access pattern in the hot loop could correspond to their layout in memory. So that normal compiler optimization may also improve performance such as these optimizations for cache locality. We also need to extend CSSMT to handle many-to-many hot spots merging so that irregular programs can be applicable.

For the CSSMT profiling step, it would be interesting to adopt [45]’s machine learning approach towards micro-architectural data to direct CSSMT transformation.

We still need a precise mapping from runtime view of the program to the source code in order to figure out the timing order and iteration relationships among hot spots. With this, [28] could readily be implemented using our framework. Besides, we could solve the many-to-many hot spots merging cases, extending CSSMT application to irregular programs.

CSSMT could potentially be very useful for GPU. When GPGPU earns increasingly more attention, the control flow divergence problem becomes the more severe performance bottleneck. CSSMT could be really effective at merging GPU kernels considering the control flow is usually less complex than their CPU counterpart. In that regard, CSSMT causes minimal overhead and have more room to improve performance by executing useful instructions on the otherwise wasted divergent code path.

## BIBLIOGRAPHY

- [1] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero. Thread to core assignment in smt on-chip multiprocessors. In *2009 21st International Symposium on Computer Architecture and High Performance Computing*, pages 67–74, October 2009.
- [2] Adrian Prantl. Bug 31268 - Umbrella: debug info for optimized code, 2017.
- [3] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiawicz. April: a processor architecture for multiprocessing. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.
- [4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In *Proceedings of the 4th International Conference on Supercomputing, ICS '90*, pages 1–6, New York, NY, USA, 1990. ACM.
- [5] Glenn Ammons, Thomas Ball, and James R Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI '97*, pages 85–96, New York, NY, USA, 1997. ACM.
- [6] Apple. Metal Shading Language. <https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf>, 2017. Last accessed 2017-06-30.
- [7] ARM. Arm compiler software development guide version 6.9, 2017.
- [8] D. H. Bailey, R. S. Schreiber, H. D. Simon, V. Venkatakrisnan, S. K. Weeratunga, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, and T. A. Lasinski. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing - Supercomputing '91*. ACM Press, 1991.
- [9] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, MICRO 29*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.
- [10] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: the showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '98*, pages 134–148, New York, NY, USA, 1998. ACM.

- [11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.
- [12] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C. Y. Cher, and M. Valero. Software-controlled priority characterization of power5 processor. In *2008 International Symposium on Computer Architecture*, pages 415–426, June 2008.
- [13] Boost. Boost C++ Libraries. <http://www.boost.org>, 2017. Last accessed 2017-06-30.
- [14] PETER BRIGHT. Skylake, Kaby Lake chips have a crash bug with hyperthreading enabled. <https://arstechnica.com/information-technology/2017/06/skylake-kaby-lake-chips-have-a-crash-bug-with-hyperthreading-enabled/>. [Online; Accessed 19-Nov-2017].
- [15] Doug Burger, James R Goodman, and Alain K Agi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 78–89, New York, NY, USA, 1996. ACM.
- [16] D. Callahan, B. L. Chamberlain, and H. P. Zima. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*, pages 52–60, April 2004.
- [17] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1988.
- [18] Steve Carr and Ken Kennedy. Improving the Ratio of Memory Operations to Floating-point Operations in Loops. *ACM Trans.Program.Lang.Syst.*, 16(6):1768–1810, November 1994.
- [19] P Carribault, A Cohen, and W Jalby. Deep jam: conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 291–300, September 2005.
- [20] F J Cazorla, A Ramirez, M Valero, and E Fernandez. Dynamically Controlled Resource Allocation in SMT Processors. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 171–182, December 2004.

- [21] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization - CGO 2016*. ACM Press, 2016.
- [22] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for FDO compilation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 42–52, New York, NY, USA, 2010. ACM.
- [23] Seungryul Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 239–251, 2006.
- [24] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, July 1963.
- [25] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January 1998.
- [26] M. Danelutto, T. De Matteis, D. De Sensi, and M. Torquati. Evaluating concurrency throttling and thread packing on smt multicores. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 219–223, March 2017.
- [27] Arnaldo Carvalho de Melo. The new linuxperftools. In Slides from Linux Kongress, 2010.
- [28] A G Dean and J P Shen. Techniques for software thread integration in real-time embedded systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 322–333, 1998.
- [29] Chen Ding and K Kennedy. The memory of bandwidth bottleneck and its amelioration by a compiler. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 181–189, 2000.
- [30] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.
- [31] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, May 2008.

- [32] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for smt processor scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 91–102, New York, NY, USA, 2010. ACM.
- [33] Ehsan Fatehi and Paul Gratz. ILP and TLP in Shared Memory Applications: A Limit Study. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 113–126, New York, NY, USA, 2014. ACM.
- [34] J Feliu, S Eyerman, J Sahuquillo, and S Petit. Symbiotic job scheduling on the IBM POWER8. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 669–680, 2016.
- [35] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.
- [36] Johan De Gelas. Assessing IBM’s POWER8. <https://www.anandtech.com/show/10435>, July 2016.
- [37] GNU. ucontext\_t. [https://www.gnu.org/software/libc/manual/html\\_node/System-V-contexts.html](https://www.gnu.org/software/libc/manual/html_node/System-V-contexts.html). [Online; Accessed 19-Nov-2017].
- [38] Google. Android NDK. <https://developer.android.com/ndk/index.html>, 2017. Last accessed 2017-06-30.
- [39] Google. NDK Simpleperf. <https://developer.android.com/ndk/guides/simpleperf.html>, 2017. Last accessed 2017-06-30.
- [40] R. H. Halstead and T. Fujita. Masa: a multithreaded processor architecture for parallel symbolic computing. In *[1988] The 15th Annual International Symposium on Computer Architecture. Conference Proceedings*, pages 443–451, May 1988.
- [41] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.
- [42] Intel. Hyperthreading. Intel Technology Journal, 2002.
- [43] Intel. VTune Amplifier. <https://software.intel.com/en-us/intel-vtune-amplifier-xe>, 2017. Last accessed 2017-06-30.
- [44] A N Jacobvitz, A D Hilton, and D J Sorin. Multi-program benchmark definition. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 72–82, 2015.

- [45] Zhen Jia, Chao Xue, Guancheng Chen, Jianfeng Zhan, Lixin Zhang, Yonghua Lin, and Peter Hofstee. Auto-tuning spark big data workloads on power8: Prediction-based dynamic smt threading. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 387–400, New York, NY, USA, 2016. ACM.
- [46] Emmett Witchel Kaashoek and M. Frans. Using Software-Extended Architectures for Software Simultaneous Multithreading. Technical Report MIT-LCS-TR-878, MIT, 1997.
- [47] R Kalla, Balaram Sinharoy, and J M Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.
- [48] R Karrenberg and S Hack. Whole-function vectorization. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 141–150, April 2011.
- [49] Khronos. OpenGL Shading Language. <https://www.khronos.org/registry/OpenGL>, 2017. Last accessed 2017-06-30.
- [50] D Koufaty and D T Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.
- [51] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004.
- [52] Chris Lattner. The Architecture of Open Source Applications – LLVM. <http://www.aosabook.org/en/llvm.html>. [Online; Accessed 19-Nov-2017].
- [53] H Q Le, W J Starke, J S Fields, F P O’Connell, D Q Nguyen, B J Ronchetti, W M Sauer, E M Schwarz, and M T Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, November 2007.
- [54] E. A. LeÅşn, I. Karlin, and A. T. Moody. System noise revisited: Enabling application scalability and reproducibility with smt. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 596–607, May 2016.
- [55] LLNL. ROSE Compiler Infrastructure. <http://rosecompiler.org>. [Online; Accessed 19-Nov-2017].
- [56] LLVM-Project. LLVM Link Time Optimization: Design and Implementation. <https://llvm.org/docs/LinkTimeOptimization.html>, 2018. [Online; Accessed 19-Nov-2017].

- [57] Jack L Lo, Joel S Emer, Henry M Levy, Rebecca L Stamm, Dean M Tullsen, and S J Eggers. Converting Thread-level Parallelism to Instruction-level Parallelism via Simultaneous Multithreading. *ACM Trans. Comput. Syst.*, 15(3):322–354, August 1997.
- [58] S Maleki, Yaoqing Gao, M J Garzaran, T Wong, and D A Padua. An Evaluation of Vectorizing Compilers. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 372–382, October 2011.
- [59] Jason Mars, Lingjia Tang, and Mary Lou Soffa. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 167–176, New York, NY, USA, 2011. ACM.
- [60] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 257–265, New York, NY, USA, 2010. ACM.
- [61] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel. Characterization of simultaneous multithreading (smt) efficiency in power5. *IBM Journal of Research and Development*, 49(4.5):555–564, July 2005.
- [62] Microsoft. High Level Shading Language. [https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx), 2017. Last accessed 2017-06-30.
- [63] Alessandro Morari, Carlos Boneti, Francisco J Cazorla, Roberto Gioiosa, Chen-Yong Cher, Alper Buyuktosunoglu, Pradip Bose, and Mateo Valero. SMT Malleability in IBM POWER5 and POWER6 Processors. *IEEE Transactions on Computers*, 62(4):813–826, 2013.
- [64] S K Sadasivam, B W Thompto, R Kalla, and W J Starke. IBM Power9 Processor Architecture. *IEEE Micro*, 37(2):40–51, March 2017.
- [65] Daniele Paolo Scarpazza, Praveen Raghavan, David Novo, Francky Catthoor, and Diederik Verkest. *Software Simultaneous Multi-Threading, a Technique to Exploit Task-Level Parallelism to Improve Instruction- and Data-Level Parallelism*, pages 12–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
- [66] S. Seo, G. Jo, and J. Lee. Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148, November 2011.
- [67] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner,

- C. F. Marino, E. Retter, and P. Williams. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1–1:29, May 2011.
- [68] B Sinharoy, R N Kalla, J M Tendler, R J Eickemeyer, and J B Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4.5):505–521, July 2005.
- [69] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. Ibm power8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2:1–2:21, January 2015.
- [70] F. M. Sleiman and T. F. Wenisch. Efficiently scaling out-of-order cores for simultaneous multithreading. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 431–443, June 2016.
- [71] Allan Snaveley and Dean M Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IX*, pages 234–244, New York, NY, USA, 2000. ACM.
- [72] Guangyu Sun, Christopher J Hughes, Changkyu Kim, Jishen Zhao, Cong Xu, Yuan Xie, and Yen-Kuang Chen. Moguls: a model to explore the memory hierarchy for bandwidth improvements. In *Proceedings of the 38th annual international symposium on Computer architecture, ISCA '11*, pages 377–388, New York, NY, USA, 2011. ACM.
- [73] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12*, pages 1–12, New York, NY, USA, 2012. ACM.
- [74] D M Tullsen, S J Eggers, and H M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 392–403. IEEE, 1995.
- [75] David W Wall. Limits of Instruction-level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IV*, pages 176–188, New York, NY, USA, 1991. ACM.
- [76] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput.Archit.News*, 23(1):20–24, March 1995.

- [77] Di Xu, Chenggang Wu, and Pen-Chung Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 237–248, New York, NY, USA, 2010. ACM.
- [78] A Yasin. A Top-Down method for performance analysis and counters architecture. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 35–44, 2014.
- [79] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array Regrouping and Structure Splitting Using Whole-program Reference Affinity. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 255–266, New York, NY, USA, 2004. ACM.
- [80] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM.