SCALABLE AND FLEXIBLE BULK ARCHITECTURE

BY

XUEHAI QIAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Professor Marc Snir
Professor Wen-mei Hwu
Professor David Padua
Professor Satish Narayanasamy, University of Michigan
Professor Per Stenström, Chalmers University of Technology
Abstract

Multicore machines have become pervasive and, as a result, parallel programming has received renewed interest. Unfortunately, writing correct parallel programs is notoriously hard. Looking ahead, multicore designs should take into account support for programmability and productivity, and make it one of the top-class design considerations.

This thesis focuses on efficient and scalable architecture supports to improve the programmability of shared-memory architectures. Specifically, we focus on supporting Sequential Consistency (SC), a strong and intuitive memory consistency model. The first part of the thesis focuses on enforcing SC by chunk-based execution. I propose techniques to remove the scalability bottlenecks of chunk-based architectures. Also, I propose the design of an SMT processor to support chunk operations among the contexts in the same processor. The second part of the thesis focuses on enforcing high performance whole-system SC, from language to architecture, by speculative chunk ordering. The third part of the thesis focuses on dynamically detecting SC violations in a directory-based cache coherence protocol precisely.

For chunk-based execution to be competitive, a machine must support chunk operations very efficiently. In my research, I focus on an environment with lazy conflict detection. In this environment, a major bottleneck in a large manycore with directory-based coherence is the chunk commit operation. The reason is that a chunk must appear to commit all of its writes atomically — even though the addresses written by the chunk belong to different, distributed directory modules. In addition, the commit may have to compete against other committing chunks that have accessed some of the same addresses — hence prohibiting concurrent commit. To resolve this commit bottleneck, I propose two scalable chunk-commit protocols.
The first protocol, called *ScalableBulk*, innovates with a set of primitives that enable a scalable coherence protocol designed for chunks. Specifically, ScalableBulk is the first work that integrates signatures into the directory design. Signatures enable the concurrent commit of any number of chunks that use the same directory module — as long as their addresses do not overlap. In addition, ScalableBulk introduces a commit protocol that groups all the directories relevant to the chunk in a way that ensures: (i) multiple groups of directories with non-overlapping addresses can form successfully concurrently and (ii) if the directory groups have overlapping addresses, at least one of the groups forms.

The second protocol, called *IntelliSense*, targets two inefficiencies in ScalableBulk. First, a ScalableBulk commit grabs the relevant directory modules in a sequential manner to ensure deadlock freedom, which may incur long latency for large directory groups. Second, two chunks with cross-processor write-after-write (WAW) dependences between them cannot commit concurrently; one squashes the other, even though these are name dependences.

To solve the first problem, I propose the *IntelliCommit* protocol, where a commit grabs all the relevant directory modules in parallel. The idea is for the committing processor to send commit-request protocol messages to all of the relevant directory modules in parallel, get their responses directly, and finally send them a commit-confirm message.

To solve the second problem, I propose the *IntelliSquash* mechanism. It uses an idea similar to the store buffers in current processors to serialize, without any squash, the commits of two chunks that only have WAWs. The result is that the write sets of the two chunks are applied in a serial manner without squashes.

To support chunk-based execution in Simultaneous Multithreading (SMT) processors, I propose *BulkSMT* [59]. It exploits the close proximity of the contexts in an SMT processor to concurrently run dependent chunks with simple hardware. I perform a broad design space analysis. The designs analyzed include three different schemes for conflict resolution inside the SMT processor. As a result of the analysis, I show for the first time that SMT processors are very cost-effective in supporting the concurrent execution of dependent chunks.
The chunk-based execution is effective at enforcing SC in hardware. However, since a memory model deals with the whole computing stack, its semantics are well-defined only when the model is specified and enforced consistently in every layer, from the language to the hardware. Therefore, to harness the benefits of SC, hardware-only SC enforcement is not sufficient — the software can easily violate SC even if the hardware implementation is correct. For correctness, we need to guarantee SC in every system layer, which is called whole-system SC.

To enable high performance whole-system SC, I propose UniBlock, the first scheme built from a conventional distributed cache coherence protocol that prevents SC violations due to hardware and software with the same set of techniques. The basic concept in UniBlock is the ordered chunk, which is used by the hardware as the mechanism to enforce hardware SC, and by the compiler as the specification to guide to scope of compiler optimizations that could violate SC. Starting from a conventional relaxed-consistency coherence protocol, UniBlock forms intermittent dynamic chunks when the speculative retirement of an instruction may violate SC. The compiler also marks the optimized code regions as static chunks to ensure correct execution. UniBlock treats static and dynamic chunks in a unified manner, and cleanly supports whole-system SC.

The above techniques are used to enforce SC, and involve some changes in the cache coherence protocol. The last work of this thesis is to detect SC violations based on a conventional cache coherence protocol. To address this problem, I propose Volition \[\text{[60]}\], the first scalable and precise hardware SC violation (SCV) detector that detects SCVs involving an arbitrary number of processors. Volition detects SCVs dynamically as a program runs. While it can be applied to both directory and bus-based coherence protocols, it does not rely on any property that is only available in a bus, such as broadcast. Volition’s idea is to dynamically detect data-dependence cycles across processors by piggybacking information on the coherence transactions. When an SCV is detected, an exception is raised to the software. For a given dynamic execution, Volition suffers no false positives or negatives.
To my loving parents, Depei Qian and Yunsheng Zhang,

and my dear wife, Yangyan Zhao,

and my adorable daughter, Chloe Kouyi Qian.
Acknowledgments

Many people have contributed to this work. Here, I can only mention a few.

Josep Torrellas, my advisor, has provided invaluable technical and professional advice, which has guided me throughout my Ph.D years. By his guidance, I became mature in the research, especially, I learned how to debug the idea, how to design the experiments and how to write a paper so that it meets the top conference standard. The most important lesson I learned from him is that, we should not give up no matter what happens. His rigorous and persistence in the research will continue to influence the rest of my career and life.

I would like to thank my committee members, – Marc Snir, Wenmei Hwu, David Padua, Per Stenström and Satish Narayanasamy. They invested the time for my preliminary exam and final defense out of their busy schedules. They also gave me the invaluable suggestions and feedbacks that improved my thesis significantly. In addition, I would like to thank Thomas Wenisch to agree to write the recommendation letter for me together with the committee members. Without their recommendation, my job search would not be successful.

I also have to mention about my friends and group mates in the department. Inside the IACOMA group, I have the great opportunity to interact with some of the most brilliant minds of computer architecture. I still remember the good times that I was talking to Wonsun, Abdullah, Shanxiang, Ulya, Nima, Yuelu, Amin about the topics from good food to academic gossips. Outside the group, I am specially thankful to Shuo, Haohui for their invaluable help and feedback during my practice talks. I also deeply appreciate Sherry for her help in all the arrangements of seminar, exams and conference travels.

Of course, nothing would have been possible without the support from my family. Yangyan
Zhao, my loving wife, stood by me all the time during the long journey of the Ph.D. With her constant love and support, together with the delicious meals she prepared, my life has been much easier and I can purely focus on the research. Probably I would have gone insane a long time ago without my wife. Her constant presence through all the ups and downs of the Ph.D journey has made the ride so much smoother. I would like to thank my parents in-law, Ziguang Zhao and Aiping Chen, for coming here to take care of my expected first daughter. Finally, the support and encouragement I have received from my parents, Depei Qian and Yunsheng Zhang, are immeasurable. Their pride in my research made every little progress so much more pleasurable.
# Table of Contents

**Chapter 1  Introduction**  
1.1 Problem and Limitations  
1.2 Proposed Approaches  

**Chapter 2  ScalableBulk: A Scalable Chunk Commit Protocol**  
2.1 Introduction  
2.2 Contribution  
2.3 Background & Related Work  
  2.3.1 Main Proposals for Concurrent Commits  
  2.3.2 Is Commit Really Critical?  
  2.3.3 Our Approach  
2.4 The ScalableBulk Protocol  
  2.4.1 Preventing Access to a Set of Directory Entries  
  2.4.2 Grouping Directory Modules  
  2.4.3 Optimistic Commit Initiation  
  2.4.4 Putting it All Together: Scalable Commit  
2.5 ScalableBulk Implementation  
  2.5.1 Message Types  
  2.5.2 Directory Module Design  
2.6 Results  
  2.6.1 Performance and Scalability  
  2.6.2 Directories Accessed per Chunk Commit  
  2.6.3 Chunk Commit Latency  
  2.6.4 Chunk Commit Serialization  
  2.6.5 Traffic Characterization  
2.7 Conclusion  

**Chapter 3  IntelliSense: A Scalable and Fast Chunk Commit Protocol with Reduced Squash**  
3.1 Introduction  
3.2 Contribution  
3.3 Background and Related Work  
3.4 A Model of the Lifetime of a Chunk  
  3.4.1 The Model
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2</td>
<td>Stalling in the Grouping Stage</td>
<td>46</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Application of the Model to Existing Protocols</td>
<td>46</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Sources of Inefficiency</td>
<td>47</td>
</tr>
<tr>
<td>3.5</td>
<td>IntelliSquash: No Squash on WAW</td>
<td>48</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Basic Idea</td>
<td>48</td>
</tr>
<tr>
<td>3.5.2</td>
<td>IntelliSquash Design</td>
<td>49</td>
</tr>
<tr>
<td>3.6</td>
<td>IntelliCommit: Parallel Grouping</td>
<td>54</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Basic Idea</td>
<td>54</td>
</tr>
<tr>
<td>3.6.2</td>
<td>IntelliCommit Commit Protocol</td>
<td>55</td>
</tr>
<tr>
<td>3.6.3</td>
<td>States of a Committing Chunk in a Directory</td>
<td>56</td>
</tr>
<tr>
<td>3.6.4</td>
<td>BlockSort: Distributed Algorithm to Order Chunks</td>
<td>58</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Example</td>
<td>61</td>
</tr>
<tr>
<td>3.6.6</td>
<td>IntelliCommit Correctness Properties</td>
<td>62</td>
</tr>
<tr>
<td>3.7</td>
<td>Analysis of Design Complexity</td>
<td>65</td>
</tr>
<tr>
<td>3.8</td>
<td>Evaluation</td>
<td>66</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Evaluation Setup</td>
<td>66</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Performance for a Single Active Block</td>
<td>68</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Performance for Two Chunks</td>
<td>70</td>
</tr>
<tr>
<td>3.8.4</td>
<td>Scalability</td>
<td>72</td>
</tr>
<tr>
<td>3.8.5</td>
<td>Directory Group</td>
<td>73</td>
</tr>
<tr>
<td>3.9</td>
<td>Conclusion</td>
<td>74</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>75</td>
</tr>
<tr>
<td>4.2</td>
<td>Contribution</td>
<td>76</td>
</tr>
<tr>
<td>4.3</td>
<td>Background &amp; Related Work</td>
<td>76</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Continuous Execution of Chunks</td>
<td>76</td>
</tr>
<tr>
<td>4.3.2</td>
<td>An Opportunity for Chunked Execution</td>
<td>77</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Other Related Work</td>
<td>78</td>
</tr>
<tr>
<td>4.4</td>
<td>Chunked-Execution SMT Processors</td>
<td>79</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Design Space</td>
<td>79</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Basic Hardware Mechanisms</td>
<td>84</td>
</tr>
<tr>
<td>4.5</td>
<td>Chunked-Execution Multicores of SMTs</td>
<td>91</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Global Protocols Examined</td>
<td>92</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Integrating the Local and Global Protocols</td>
<td>92</td>
</tr>
<tr>
<td>4.6</td>
<td>Implementation Issues</td>
<td>93</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Cycle Detection Algorithm Implementation</td>
<td>93</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Cache Conflicts</td>
<td>95</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Handling High-Contention Synchronizations</td>
<td>96</td>
</tr>
<tr>
<td>4.7</td>
<td>Results</td>
<td>98</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Performance Comparison</td>
<td>100</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Dependence Analysis in STALL and ORDER</td>
<td>104</td>
</tr>
<tr>
<td>4.7.3</td>
<td>Dependence Cycles in ORDER</td>
<td>105</td>
</tr>
<tr>
<td>4.7.4</td>
<td>Squash Set Size</td>
<td>105</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Problem and Limitations

Multicore chips as commodity architecture for platforms ranging from handhelds to supercomputers herald an era when parallel programming and computing will be the norm. While the community has periodically focused on advancing the technology for parallel processing [34], it is critically important at the current time, since there is no obvious route to higher performance other than through parallelism. However, for parallel computing to become mainstream, the programmability of the system should be improved. Breakthroughs are needed in all layers of the computing stack, including languages, programming models, compilation and runtime software, programming and debugging tools, and hardware architectures.

At the hardware-architecture layer, designs were primarily for performance or energy efficiency in the past. Looking ahead, designs should take the support for programmability and productivity into account and make it one of the top-class design considerations. However, programmability is usually not well-defined and hard to measure. In our view, good programmability at the hardware-architecture level means two things: (1) the architecture is able to ensure high performance and efficiency while relieving programmers or compiler writers from having to manage too many low-level tasks; (2) the architecture should help to minimize the chance of parallel programming errors. In the following, we show three specific aspects that are important to improve programmability.

First, the memory consistency model (memory model) defines the order in which memory accesses performed by one processor become visible to the others. Sequential Consistency (SC) is the most intuitive memory model assumed by most programmers. SC ensures the property that
all the memory accesses by all the processors appear to have executed in a global sequential order consistent with the per-processor program order \([35]\). Strong memory model support, like SC, is critical toward a user-friendly programming environment.

Second, concurrency bugs \([39]\) are another aspect relevant to programmability. These bugs are software bugs related to synchronization and communication operations of parallel programs. Tasks in parallel programs communicate with each other through shared memory. An important type of concurrency bug is atomicity violation. An atomicity violation \([50, 41]\) can occur when the programmer fails to enclose in the same critical section all of the memory accesses that should occur atomically. During execution, such accesses get interleaved with accesses from another thread that alter the program state, making it inconsistent. Architecture support for concurrency bug detection can directly help to eliminate many program errors.

Third, record and replay \([29]\) of the nondeterministic events in the shared-memory multithreaded execution can help ease the pain of developing parallel software. A system with a deterministic replay capability can record sufficient information during an execution to enable a replayer to (later) create an equivalent execution despite the inherent sources of nondeterminism that exist. This technique greatly helps programmers in developing, debugging and monitoring parallel programs.

Recent proposals show that all of the above three aspects for good programability can be enhanced efficiently with a unique concept, Atomic Block (or Chunk) \([72]\). A chunk is a set of consecutive dynamic instructions executed by a processor. Each chunk executes on the processor atomically and in isolation. Atomic execution means that none of the chunk’s actions are made visible to the rest of the system (processors or main memory) until the chunk completes and commits. Execution in isolation means that if the chunk reads a location and (before it commits) a second chunk in another processor that has written to the location commits, then the local chunk is squashed and must re-execute.

Chunk is a notion for both architecture and software. In hardware, it can efficiently implement high-performance sequential consistency \([15]\) and enable a record and replay system with high re-
play speed and very small log size [48]. In software, it enables aggressive (and potentially unsafe) compiler optimizations for high-performance and whole-system SC by containing the optimization effects in the chunks [4]. Moreover, programmers or automatic tools can mark or learn the atomic regions to detect or avoid concurrency bugs [50, 41]. Due to the important uses of chunk, we believe the architecture should support them widely and efficiently.

This thesis focuses on efficient and scalable architecture supports to improve programmability. We identify the following limitations of the existing designs (before the work in this thesis). First, BulkSC [15] enforces high-performance SC by chunks. However, the centralized arbiter limits the scalability. Second, all of the many previous architecture proposals for chunk (or transaction) execution have assumed single-context cores as their building blocks, ignoring the widely-available Simultaneous Multithreading (SMT) cores. The solution was not complete without an understanding of how to make it work for an SMT design. Third, while the chunk-based approach has many merits, traditional designs require drastic changes to the existing cache coherence protocols. Designs based on the conventional coherence protocol may have better applicability in the short-term.

1.2 Proposed Approaches

To remedy the limitations, we propose the Scalable and Flexible Bulk Architecture. In this architecture, we focus on supporting Sequential Consistency (SC), a strong and intuitive memory consistency model, or dynamically detect any non-SC behavior. In all the proposals, we emphasize the scalability to ensure they can be applied to the many-core systems becoming available soon.

ScalableBulk [58] is proposed to improve the scalability of the chunk-based architecture. All of the previous schemes, such as Scalable TCC [16] and SRC [57] are limited in that two independent chunks (i.e., chunks that accessed disjoint addresses) that happen to use the same directory module cannot commit simultaneously. This is because the directory has to conservatively assume that the two chunks have conflicting accesses and, therefore, has to order them. ScalableBulk
uses signatures to relax this limitation and allows chunks with overlapping directories to commit concurrently, as long as their signatures do not conflict. Due to this fact, ScalableBulk can better tolerate chunks that access many directory modules.

*IntelliSense* is proposed to eliminate two inefficiencies in ScalableBulk. First, a ScalableBulk-commit grabs the relevant directory modules in a sequential manner to ensure deadlock freedom, which may incur long latency for large directory groups. Second, two chunks with cross-processor write-after-write (WAW) dependences between them cannot commit concurrently; one squashes the other, even though these are name dependences. The *IntelliCommit* protocol is used to solve the first problem, where a commit grabs all the relevant directory modules *in parallel*. The *IntelliSquash* mechanism is used to solve the second problem. It uses an idea similar to the store buffers in current processors to serialize, without any squash, the commits of two chunks that only have WAWs. The result is that the write sets of the two chunks are applied in a serial manner without squashes.

*BulkSMT* [59] is proposed to enable chunk-based (transactional) execution on the widely-available Simultaneous Multithreading (SMT) cores. It is a stake in the ground that serves to inspire and influence the likely wide deployment of Hardware Transactional Memory (HTM) and chunk support on commercial SMT processors in the future. We observe and show how to exploit the close proximity of the contexts in an SMT processor to concurrently run dependent transactions effectively and with hardware simplicity. We performs a broad design space analysis. The designs analyzed include three different schemes for conflict resolution inside the SMT processor (*SQUASH*, *STALL*, and *ORDER*), and two across the SMT processors in a multicore (EAGER and LAZY). The analysis offers an understanding of what design is best. As a result of the analysis, this paper shows for the first time that SMT processors are very cost-effective in supporting the concurrent execution of dependent transactions. It also shows that *ORDER* is the most competitive scheme, and that it combines well with LAZY or EAGER at the multicore level.

*UniBlock* is proposed to enable high performance whole-system SC. It is the first scheme built from a conventional distributed cache coherence protocol that prevents SC violations due to hard-
ware and software with the same set of techniques. The basic concept in UniBlock is the ordered chunk, which is used by the hardware as the mechanism to enforce hardware SC, and by compiler as the specification to guide to scope of compiler optimization that could violate SC. Starting from a conventional relaxed-consistency coherence protocol, UniBlock forms intermittent *dynamic chunks* when the speculative retirement of an instruction may violate SC. The compiler also marks the optimized code regions as *static chunks* to ensure correct execution. UniBlock treats the static and dynamic chunks in a unified manner, and cleanly supports whole-system SC. UniBlock attains high performance through the concurrent execution of dependent chunks in a distributed directory coherence protocol.

*Volition [60]* is proposed as the first *scalable* hardware scheme that detects *precise* SCV involving an *arbitrary* number of processors from a dynamic execution on a relaxed-consistency machine. Such support can help the programmers to find subtle and notorious concurrency bugs due to SC violation. The scheme can be used either in directory or bus-based cache coherence protocols, and does not rely on any property that is only available in buses. The idea is to dynamically detect a special pattern of memory access cycles across processors by piggybacking the coherence messages with the local serial number of the memory access, and checking it against a remote processor’s local execution information. Volition suffers no false positives or negatives. It requires no significant coherence protocol changes and does not affect the scalability of the coherence protocol.

All of our proposals improve the system programmability without sacrificing system scalability. We cautiously hope that, in due course, BulkSMT, Volition and UniBlock will enable a wide deployment of architecture supports for SC and general debug/test.
Chapter 2

ScalableBulk: A Scalable Chunk Commit Protocol

2.1 Introduction

There are several recent proposals for shared-memory architectures that efficiently support con-
tinuous atomic-block operation [6, 15, 16, 20, 28, 57, 74, 76]. In these architectures, a proces-
sor repeatedly executes blocks of consecutive instructions from a thread (also called chunks) in an atomic manner. These systems include TCC [16, 28], BulkSC [15], Implicit Transactions (IT) [74], ASO [76], InvisiFence [6], DMP [20], and SRC [57] among others. This mode of execution has performance and programmability advantages. For example, it can support transactional memory [16, 28, 57]; high-performance execution, even for strict memory consistency models [6, 15, 76]; a variety of techniques for parallel program development and debugging such as determinism [20], program replay [48], and atomicity violation debugging [41]; and even provide a substrate for new high-performance compiler transformations [4, 53].

For these machines to deliver scalable high performance, the cache coherence protocol must support chunk operations very efficiently. It must understand and operate with chunk transactions like conventional machines operate with individual memory accesses.

There are several ways to design a chunk cache coherence protocol. In particular, an important decision is whether to use lazy or eager detection of chunk conflicts. In the former, chunks execute obliviously of each other and only check for conflicts at the time of chunk commit; in the latter, conflict checks are performed as the chunk executes. There are well-known pros and cons of each approach, which have been discussed in the related area of transactional memory [8, 70, 54, 26] — although there is no consensus on which approach is the most promising one. In this work, we
focus on an environment with lazy conflict detection \[15, 16, 20, 28, 57, 74, 55\].

In such an environment, a major bottleneck is the chunk commit operation, where the system checks for collisions with all the other executing chunks. Early proposals used non-scalable designs. For example, TCC \[28\] relies on a bus, while BulkSC \[15\] uses a centralized arbiter. Later designs such as Scalable TCC \[16\] and SRC \[57\] work with a directory protocol and, therefore, are more scalable. However, Scalable TCC requires broadcasting and centralized operations, and SRC has message serialization. More importantly, these schemes add unnecessary commit serialization by disallowing the concurrent commit of chunks that, while collision-free, happen to use the same directory module(s). This problem is worse for applications with lower locality and for higher processor counts.

2.2 Contribution

To address this problem, we present a novel directory-based protocol that enables highly-overlapped, scalable commit operations for chunks. In our design, the commit operation uses no centralized structure and communicates only with the relevant directory modules. Importantly, it enables the concurrent commit of any number of chunks that use the same directory module — as long as their addresses do not overlap. Our goal is to emulate for chunks what conventional directories do for individual write transactions.

The protocol, called *ScalableBulk*, builds on the previously-proposed BulkSC protocol — effectively extending it to work with directories. ScalableBulk introduces three new generic hardware primitives for scalable chunk commit: (1) preventing access to a set of directory entries, (2) grouping directory modules, and (3) initiating the commit optimistically.
2.3 Background & Related Work

In a chunk-based cache coherence protocol that performs lazy conflict detection, the chunk commit operation is critical. Indeed, during the execution of a chunk, cache misses bring individual lines into the cache, but no write is made visible outside the cache. At commit, the changes in coherence states induced by all the writes must be made visible to the rest of coherent caches atomically — squashing any other chunk that has a data collision. Note that a commit does not involve writing data back to memory. However, it requires updating the states of the distributed caches in a way that appears that chunks executed in a total order.

Architectures that support continuous chunk operation in this environment [15, 16, 20, 28, 57, 74] must critically provide efficient chunk commit. In general, while they must commit dependent chunks serially, they attempt to overlap the commit of independent chunks. In the following, we describe the main existing proposals for concurrent commits, reconsider whether commit is critical, and describe our approach.

2.3.1 Main Proposals for Concurrent Commits

**BulkSC** [15] uses a centralized arbiter that receives every commit request. The arbiter allows the concurrent commit of multiple chunks, as long as the addresses that an individual chunk wrote do not overlap with the addresses accessed by any other chunk. To detect overlap at a fine grain, BulkSC uses hardware address signatures [14].

**Scalable TCC** [16] supports chunk commits in a directory-based coherence protocol. The protocol overlaps the commit of independent chunks but has several scalability bottlenecks. First, the committing processor contacts a centralized agent to obtain a transaction ID (TID), which will enforce the order of chunk commit. Second, the processor contacts *all* the directory modules in the machine — each one possibly multiple times. Specifically, it sends a *probe* message to the directories in the chunk’s write- and read-set, and a *skip* message to the rest. Third, for every cache line in the chunk’s write-set, the processor sends a *mark* message to the corresponding directory.
More importantly, however, is that this protocol limits the concurrent commit of independent chunks. Indeed, two chunks can only overlap their commits if they use different directories. In other words, if two chunks access different addresses but those addresses are in the same directory module, their commit gets serialized.

**SRC [57]** focuses on removing the TID centralization and the message multicasts from Scalable TCC. The idea is for each committing processor to send messages to the directories in the chunk’s read- and write-sets to sequentially occupy them. For example, if the chunk’s sets include directories 1, 4 and 6, the processor starts by occupying 1, then 4, then 6. The transaction gets blocked if one directory is taken. This protocol, called SEQ-PRO, reduces centralization. However, it introduces sequentiality. Importantly, it has the same shortcoming as Scalable TCC: two chunks that accessed different addresses from the same directory are serialized.

The authors present an optimization called SEQ-TS where the committing processor sends a request in parallel to all the directories in its read- and write-sets, and can steal a directory from the chunk that currently occupies it. However, this approach seems prone to protocol races, and there are little details on how it works.

### 2.3.2 Is Commit Really Critical?

There are two papers that show experimental data on the scalability of these protocols. One is the Scalable TCC paper [16], which shows simulations of SPEC, SPLASH-2, and other codes for 64 processors. The other is the SRC paper [57], which shows simulations of synthetic traces for 64-256 processors. The data in both papers appears to suggest that chunk commit is overlapped with computation and does not affect the execution time of applications.

However, the environments described in these papers are different from the one we are interested in. There are two key differences: the size of the chunks and the number of directories accessed per chunk commit.

Consider the chunk sizes first. The chunks (i.e., transactions) in Scalable TCC are large, often in the range of 10K-40K instructions. Such large chunks are attained by manually (or automati-
cally) instrumenting the application, positioning the transactions in the best places in the code. In SRC, the authors consider synthetic models with chunks larger than 4K instructions.

We are interested in an environment like BulkSC, where the code is executed as is, without software instrumentation or analysis. Following BulkSC, we use chunk sizes of 2K instructions. Additionally, cache overflows and system calls can further reduce the average size. With chunk sizes that are one order of magnitude smaller than Scalable TCC, chunk commit is more frequent, and its overhead is harder to hide.

Consider now the number of directories accessed per chunk commit operation. In Scalable TCC, for all but two codes, the 90th percentile is 1–2 directories. This means that, in 90% of the commits, only one (or at most two) directories are accessed. In SRC, the synthetic model is based on the Scalable TCC data.

This is in contrast to the larger numbers that we observe in our evaluation. In this work, the average number of directories accessed per chunk commit is typically 2–6. We believe the difference is because, in our environment, we cannot tune what code goes into a chunk. With these many directories, we often concurrently commit chunks that accessed disjoint addresses but use the same directory. Scalable TCC and SRC would serialize them.

Overall, from this discussion, we conclude that the commit operation is indeed time-critical.

### 2.3.3 Our Approach

We devise a directory-based coherence protocol that works with chunks efficiently. A truly scalable chunk commit operation should (i) need no centralized structure, (ii) communicate only with the relevant directories, and (iii) allow the concurrent commit of chunks that use the same directory, as long as their addresses do not overlap. Moreover, we believe that hardware address signatures [14, 46, 77] provide a good means to implement a chunk protocol efficiently. They perform operations on footprints inexpensively.
2.4 The ScalableBulk Protocol

We describe ScalableBulk by focusing on its three new generic protocol primitives: (1) preventing access to a set of directory entries, (2) grouping directory modules, and (3) initiating the commit optimistically. In our discussion, we assume a multicore architecture with distributed directory modules as in Figure 2.1.

![Generic multicore architecture considered.](image)

2.4.1 Preventing Access to a Set of Directory Entries

In a chunk protocol like the one considered, there are no individual write transactions. Instead, all the writes in a chunk are processed with a single commit transaction at the directory. To understand such a transaction, consider the operation of a write in a conventional directory protocol. When a write arrives at a directory, the controller starts a transaction that involves (in a four-hop protocol):
setting the directory state for the line to transient, identifying the sharers, sending invalidations to
the sharers, receiving acknowledgments (acks), updating the directory state for the line to dirty,
and notifying the writer. While the state is transient, the directory controller blocks all requests to
the same line — either by buffering the requests or by bouncing them (i.e., nacking). However, at
all times, the controller can process transactions for other lines.

In a chunk protocol, there is a single transaction for each chunk commit. Let us assume for now
that the machine has a single directory module. When the directory receives the commit request,
the controller must identify the addresses of the lines written by the chunk. In ScalableBulk, this
is done by expanding the write (W) signature [15] (while in other schemes, the controller may
receive the list of written addresses). Then, the controller compiles the list of sharers of such lines,
sends W to them for cached line invalidation and chunk disambiguation, and finally collects all
acks. In the meantime, the directory controller updates the state of the directory entries for these
lines.

During this process, and until all acks are received and the directory state for all of these lines
is updated, the directory controller must disable access to these lines. It does so by nacking both (i)
reads to the lines and (2) commits of chunks that have read or written these lines. However, com-
mits (and reads) that do not have any address overlap with these lines should proceed in parallel.
Moreover, the decision of whether or not to nack should be quick.

Figure 2.2 shows how these operations are performed in ScalableBulk. In the figure, two
chunks given by W signatures $W_0$ and $W_1$ are committing concurrently. Signature expansion is
performed in a module like the DirBDM in BulkSC [15]. Any incoming load to the directory
module is checked for membership in $W_0$ and $W_1$. If there is no match, the access is allowed
to proceed. Any incoming read/write signature pair $(R_2, W_2)$ for a chunk is intersected with $W_0$
and $W_1$. If all the intersections are null, $W_2$ is allowed to join $W_0$ and $W_1$ in committing. Note
that all these operations are fast. Moreover, false positives due to signature aliasing cannot affect
correctness. At worst, they nack an operation unnecessarily.
Figure 2.2: A ScalableBulk directory module allows the commit of multiple, non-overlapping chunks, and nacks overlapping accesses and overlapping chunk commits.

2.4.2 Grouping Directory Modules

In a conventional directory protocol, a write access engages a single directory module; in a chunk protocol, a chunk commit can require the participation of multiple directories — the home directories of the lines read or written in the chunk. Coordinating the multiple directories in a commit is the biggest challenge in any chunk protocol.

For commit scalability, ScalableBulk only communicates with the home directories of the lines read or written in the chunk — rather than with all the directories as in Scalable TCC. Coordinating
multiple directories is done in two steps, namely identifying the directories and then synchronizing their operation in what we call Directory Grouping. To identify the relevant directories, the hardware could encode the signatures in a way that made it easy to extract the home directory numbers of the constituting addresses. However, this approach is likely to require a non-optimal signature encoding. Consequently, ScalableBulk works differently: as a chunk executes, the hardware automatically collects in a list the home directory numbers of the reads and writes issued. At chunk commit time, the compressed R and W signatures and this list are sent to the directory modules in the list.

For any group of directories that receive a (R, W) signature pair, ScalableBulk designates a Leader. The leader is set by a simple, default hardware policy. The baseline policy is for the leader to be the lowest-numbered module in the group. The leader initiates a synchronization step using the Group Formation protocol (Group for short). The protocol, which is described next, attempts to group all participating directories. If it succeeds, the leader sends a commit success message to the committing processor, which then clears its signatures; if it fails, the leader sends a commit failure message to the committing processor, which prompts it to wait for a while and then retry the commit request (unless the committing processor is asked to squash the chunk before).

![Diagram of Group formation protocol](image)

Figure 2.3: Group formation protocol.

As the leader sends the commit success message, it sends the W signature to all the sharer processors, to trigger cached line invalidation and chunk disambiguation. Later, when the leader
receives all acks, it multicasts a commit done message to the directory group. The directories in the group silently break down the group and discard W.

From the time a directory receives the (R, W) signature pair and tries to form a group until it receives the commit done message (or the group formation fails), it nacks incoming overlapping requests and overlapping commits.

**Group Formation Protocol**

At any time, there may be multiple sets of directory modules trying to form groups. Some of these groups may be incompatible with each other. Two groups that are trying to use a given directory module are Incompatible if their W signatures overlap or if the R signature of one and the W signature of the other overlap. Otherwise, they are compatible and can commit concurrently.

The Group protocol ensures that: (i) all the compatible groups form successfully concurrently, and (ii) given a set of incompatible groups, at least one of them forms. To guarantee deadlock- and livelock-freedom, the Group protocol follows well-known deadlock-free resource-allocation algorithms by requiring a fixed directory-module traversal order. Specifically, the algorithm forms a group by always starting from the leader and traversing directory modules in hardware from lower to higher numbers.

The algorithm is shown in Figure 2.3 which depicts six directory modules. All arrows are hardware messages. In Chart (a), a committing processor sends the (R, W) signature pair to participating directories 1, 2, and 5. The message also contains the list of participating directories. Each of these directory modules expands the W signature and, after checking its local directory state, determines the list of processors that need to be invalidated to commit the chunk (namely, the sharer processors). In addition, the leader starts by putting its list in a g (or grab) message and sending it to the next directory module in the sequence (Chart (b)). Each directory module, when it receives g, if it has already received the signatures and expanded W to find the sharer processors, augments the processor list in g, and passes the updated g to the next directory in the sequence. Note that computing the sharer processors is done by all directory controllers in parallel, typically
before they receive the $g$ message. Therefore, it is not in the critical path.

Eventually, the $g$ message returns to the leader. The leader then multicasts a short $g\_success$ message to all participating directory modules (Chart (c)). The group is now formed, and the directories start updating their state based on the W signature. At the same time, the leader sends a commit success message to the committing processor and W to the sharer processors (Chart (d)). On reception of all the acks from the sharers, the leader multicasts a commit done message to the directory group (Chart (e)). All the directories in the group then break down the group and deallocate the W signature.

It is possible that the $g$ message does not return to the leader. This occurs when the group being formed collides with a second group and the latter wins. The collision occurs in the lowest-numbered directory module that is common to both groups. We call it the Collision module. It declares, as the winner group, the first group for which it sees both (i) the (R, W) signature pair coming from the committing processor and (ii) the $g$ message coming from the previous directory module in the group.

As soon as the Collision node sees both messages from one group, it pushes the $g$ message to the next node in that group, irrevocably choosing that group as the winner. Later, when it receives both messages from the losing group, it simply multicasts a $g\_failure$ message to all the directories in the losing group. The directories then deallocate the losing W signature and the leader of the losing group sends a commit failure message to the committing processor. Chart (f) shows this case assuming that module 2 detected the collision.

**Forward Progress, Starvation, and Fairness**

The Group algorithm guarantees forward progress because, when several groups collide, at least one is able to commit successfully. This is guaranteed because $g$ messages are strictly passed from lower- to higher-numbered modules, and they are only sent when the sender has received both (R, W) and $g$.

As an advanced example, Figure 2.3(g) shows a system with nine directory modules and three
colliding groups. The latter are G0 (which tries to combine directories 0, 2, 3, and 4), G1 (trying directories 1, 2, 3, 7, and 8), and G2 (trying directories 6 and 7). The Collision module for Groups G0 and G1 is Module 2 — the lowest-numbered, common module. Suppose that Module 2 receives the combination of (R, W) and g for Group G1 first. At that point, Module 2 passes g for G1 to Module 3, effectively choosing G1 over any future colliding group. Later, when Module 2 receives the combination of (R, W) and g for G0, it multicasts g_failure for G0 to Modules 0, 3, and 4. The next decision occurs in Module 7. The module chooses between G1 and G2 based on which of the groups first provides (R, W) and g. Overall, at least one group will form successfully.

The algorithm described tends to favor small groups over large ones. This is because large groups are likely to encounter more Collision modules as they form and, therefore, have higher chances of failing to form. To prevent the commit starvation of such chunks, the Group algorithm works as follows. After a given directory module has seen the squash of a given chunk commit for a total of MAX times, then, it reserves itself for the commit of that chunk. It responds to all other requests for commit as if there was a collision and the requester lost. It does this until it receives the request from the starving chunk and commits it. Since all the directories in the group see every single squash of the group, they all reserve themselves for the starving chunk at the same time.

The algorithm also tends to favor the commit of chunks from processors close to low-numbered directories. This is because these processors can push signatures to low-numbered directories faster, hence pre-empting other processors. To solve this problem and ensure long-term fairness, the Group algorithm can change the relative priority of the directory-module IDs at regular intervals. Specifically, it can use a modulo rotation scheme where, during one interval, the highest-to-lowest priority is given by IDs 0, 1, ... n; during the next interval, it is given by 1, 2, ... n, 0; and so on. At any time, the Group algorithm chooses the leader of a group to be the one with the highest-priority ID in the group, and the g messages are sent from higher to lower priority modules.
2.4.3 Optimistic Commit Initiation

In existing chunk protocols such as BulkSC, the commit operation proceeds conservatively. Specifically, the processor sends a permission to commit request to an arbiter and, while the processor is waiting for an OK to commit or Not OK to commit message from the arbiter, it nacks all incoming messages — such as signatures for cache line invalidation and chunk disambiguation. This action limits concurrent commit.

In a machine with many cores, communication latencies may be high, and determining whether a chunk can commit takes some time. In ScalableBulk, it takes the time to form (or fail to form) a group.

To address this issue, ScalableBulk introduces Optimistic Commit Initiation (OCI), where a committing processor assumes that its commit transaction will succeed. After the processor sends its commit request with signatures to the target directories, it continues to consume incoming messages — including signatures from concurrently committing transactions that attempt to perform bulk invalidation (i.e., invalidate cached lines and disambiguate against the local chunk). Note that the local chunk’s R and W signature registers are not deallocated until the processor receives a commit success message and, therefore, are available for disambiguation.

OCI increases performance by increasing the overlap of multiple commits. Moreover, by doing so, it also reduces the time that signatures are buffered in directory modules (Section 2.4.1). This in turn reduces the time during which requests and signatures are nacked from directories, and decreases the possibility of collisions.

However, OCI complicates the protocol when the committing processor consumes a bulk invalidation message and finds that it needs to squash the chunk that it recently sent out for commit. In this case, as the processor squashes and restarts the chunk, it sends a Commit Recall message to ask for the cancellation of the commit. This recall message is piggy-backed on the ack to the bulk invalidation message that caused the squash. As we show in Section 2.4.4, ScalableBulk ensures that this message is propagated to the correct directories. Later, when the processor receives a
commit failure message from the leader of its failed directory group, it discards it.

The OCI protocol is illustrated in Figure 2.4. Consider two processors that initiate commits with overlapping addresses. Processor P0 sends its signatures to directory modules 0, 2, and 3, while P1 sends its own to modules 1, 2, and 3 (Chart (a)). The first set of directories succeed in forming Group G0. Its leader (Module 0) sends commit success to P0, and W₀ for bulk invalidation to P1 (Chart (b)). At this point, a conservative protocol proceeds as in Chart (c), while one with OCI proceeds as in Chart (d).

![Diagram of Figure 2.4: Operation of the Optimistic Commit Initiation (OCI).](image)

Specifically, in Chart (c), P1 nacks the W₀ message repeatedly until it receives a commit failure.
message from the leader of the group that failed (Group G1). At that point, it consumes the \( W_0 \) message and squashes the local chunk — therefore enabling the completion of the Group G0 chunk commit.

On the other hand, in Chart (d), P1 immediately consumes the \( W_0 \) message, piggy-backs a commit recall on the ack to the \( W_0 \) (bulk invalidation) message, and squashes and restarts its chunk. We show in Section 2.4.4 that this recall message is routed to the directories of Group G1, to tell them that the committing processor has squashed its chunk. Later, when P1 receives the commit failure message for the chunk from Module 1, it discards it.

Overall, OCI reduces the critical path to complete the successful commit of the chunk in Group G0. Specifically, it removes from the critical path the following potential latencies of failed Group G1 operation: the initial request from P1 to the directory modules participating in Group G1, the (failed) formation of Group G1, and the transfer of the commit failure message to P1.

### 2.4.4 Putting it All Together: Scalable Commit

![Diagram of scalable chunk commit](image)

(a) Groups G0 and G1 do not collide  
(b) Groups G0 and G1 collide  
This state follows from Figure 4(d)

Figure 2.5: Scalable chunk commit, where both commits succeed (a) and where only one does (b).

The ScalableBulk features described fulfill the requirements for a truly scalable commit operation listed in Section 2.3.3. First, there is no centralization point. Second, a committing processor communicates only with the directory modules in its signatures, with point-to-point messages; there is no message broadcasting. Third, any number of chunks that share directory modules
but have non-overlapping updated addresses ($R_i \cap W_j \cup W_i \cap W_j$ is null for every $i, j$ pair) can commit concurrently — just as conventional protocols support any number of concurrent write transactions to different addresses. Finally, OCI maximizes the overlap of commits by removing operations from the critical path of commits.

To show how the complete chunk commit operation works, we revisit Figure 2.4(a), where two chunks that have accessed data from common directory modules (Modules 2 and 3) try to commit. Let us consider two cases: in one, they do not have overlapping updated addresses; in the other, they do. When they do not, they commit concurrently (Figure 2.5(a)). Each processor sends the signatures to the relevant directory modules. The leader module in each group forms the group. Then, each leader sends a commit success to its own originating processor, and bulk invalidations to all sharer processors. When the sharer processors ack, the leader multicasts a commit done to all directory modules in the group, which deallocate the signature. These operations proceed in parallel for the two groups.

Consider now that the two chunks have overlapping updated addresses. Each processor sends the signatures to the relevant directory modules. Assume that, as shown in Figure 2.4(b), Group G0 succeeds and G1 fails. P1 receives the $W_0$ signature (i.e., the bulk invalidation message), squashes the chunk it is committing (Figure 2.4(d)), and piggy-backs a commit recall in the ack to G0’s leader (Module 0).

Figure 2.5(b) shows the state after Figure 2.4(d). As Module 0 multicasts the commit done to the G0 members, it includes the commit recall from P1 in the message. All modules deallocate signature $W_0$ and consider the commit complete. The commit recall triggers no action in any module except in the lowest-numbered one of the set of modules common to both G0 and G1 (the Collision one). In our example, this is Module 2. The recall tells Module 2 that P1 started a commit and its chunk has already been squashed. If Module 2 has already seen both ($R_1$, $W_1$) and $g$ for G1, then it has already sent $g_{\text{failure}}$ to all the members of Group G1. Consequently, it simply discards the commit recall. Otherwise, the commit recall tells Module 2 to be on the lookout for the reception of ($R_1$, $W_1$) and $g$ for G1; when it receives them both, it sends the $g_{\text{failure}}$ message.
to all the G1 group members. This is why the commit recall is needed: Module 2 deallocates signature $W_0$ and, therefore, would not be able to detect the collision if it has not yet observed ($R_1$, $W_1$).

### 2.5 ScalableBulk Implementation

To get a flavor of ScalableBulk’s implementation, this section describes the message types and the design of a directory module. Appendix A describes the ordering of messages in a directory module.

#### 2.5.1 Message Types

Table [2.1](#) shows all of the message types needed in ScalableBulk. There are ten types. In the table, $C.Tag$ is the unique tag assigned to a chunk. It is formed by concatenating the originating processor ID and a processor-local sequence number. $W.Sig$ and $R.Sig$ are the write and read signatures of a chunk. $g.vec$ is the set of directory modules in a chunk’s read- and write-sets. It is formed by the processor as it executes a chunk. invalid.vec is a bit vector with the set of sharer processors that need to be invalidated once a group has been formed. It is built incrementally at each participating module, and passed with the $g$ message. The commit recall message is piggy-backed on the bulk_inv_ack message and then on the commit_done message, so that it reaches the Collision directory module (indicated by $Dir.ID$). Finally, $Proc$ and $Dir$ mean processor and directory module, respectively.

As an example, the first row describes the request-to-commit message (commit_request), sent by a committing processor to all the directories in the read- and write-sets of the chunk. The message includes the chunk tag, the signatures, and the set of directories in the chunk’s read- and write-sets. The other rows can be easily followed.
<table>
<thead>
<tr>
<th>Message Type</th>
<th>Description</th>
<th>Format</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit_request</td>
<td>Processor requests to commit a chunk. Message is sent to all the directory</td>
<td>$C_{Tag}, W_{Sig}, R_{Sig}, g_{vec}$</td>
<td>Proc → Dir(s)</td>
</tr>
<tr>
<td></td>
<td>modules in the read- and write-sets of the chunk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g (or grab)</td>
<td>Source directory module is part of a group, and tries to grab the destination</td>
<td>$C_{Tag}, inval_{vec}$</td>
<td>Dir → Dir</td>
</tr>
<tr>
<td></td>
<td>module to put it into the same group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g_failure</td>
<td>A module detects that group formation has failed and notifies of the failure to</td>
<td>$C_{Tag}$</td>
<td>Dir → Dir(s)</td>
</tr>
<tr>
<td></td>
<td>all the modules in the group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g_success</td>
<td>The leader informs all the modules in the group that the group has been</td>
<td>$C_{Tag}$</td>
<td>Dir → Dir(s)</td>
</tr>
<tr>
<td></td>
<td>successfully formed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit_failure</td>
<td>The leader notifies the commit-requesting processor that the commit failed</td>
<td>$C_{Tag}$</td>
<td>Dir → Proc</td>
</tr>
<tr>
<td>commit_success</td>
<td>The leader notifies the commit-requesting processor that the commit is</td>
<td>$C_{Tag}$</td>
<td>Dir → Proc</td>
</tr>
<tr>
<td></td>
<td>successful</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bulk_inv</td>
<td>The leader sends out a bulk invalidation to all the sharer processors</td>
<td>$C_{Tag}, W_{Sig}$</td>
<td>Dir → Proc(s)</td>
</tr>
<tr>
<td>bulk_inv_ack</td>
<td>The leader receives a bulk invalidation acknowledgment from a sharer</td>
<td>$C_{Tag}$</td>
<td>Proc → Dir</td>
</tr>
<tr>
<td></td>
<td>processor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit_done</td>
<td>The leader releases all the modules in the group and requests the deallocation</td>
<td>$C_{Tag}$</td>
<td>Dir → Dir(s)</td>
</tr>
<tr>
<td></td>
<td>of the signatures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit_recall</td>
<td>A processor with a squashed chunk notifies the leader module of the squash.</td>
<td>$C_{Tag}, Dir_{ID}$</td>
<td>Proc → Dir,</td>
</tr>
<tr>
<td></td>
<td>The message is piggy-backed on $bulk_{inv}<em>{ack}$ and $commit</em>{done}$</td>
<td></td>
<td>Dir → Dir</td>
</tr>
<tr>
<td></td>
<td>messages</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Message types in ScalableBulk.

### 2.5.2 Directory Module Design

Figure 2.6 shows the design of a ScalableBulk directory module. It has three components, namely buffers for the incoming and outgoing messages, the Chunk State Table (CST) that contains one entry per committing or pending chunk, and the ScalableBulk protocol engine. The protocol engine implements the protocol state machine. It accepts incoming messages, triggers the state transitions for the corresponding chunks in the CST, and potentially generates new outgoing messages based on the state of the chunks. To design the state machine, we follow the methodology summarized in [68], and derive the set of states, events, messages, transitions, and actions with each transition.
Figure 2.6: Directory module in ScalableBulk.

Each CST entry corresponds to a chunk being processed by this directory module. There is an analogy between a CST entry in a ScalableBulk directory and a regular entry in a conventional directory. A CST entry is allocated when the directory module receives either a signature pair \((R_{Sig}, W_{Sig})\) or a \(g\) message for a chunk. It is deallocated at one of two points: (1) the chunk commits successfully and the directory has received all \(bulk_{inv\_ack}\) messages (if it is the leader) or a \(commit\_done\) message (if it is not), or (2) the chunk commit fails and the directory receives a \(commit\_recall\) message (if it is the Collision module) or a \(g\_failure\) message.

A CST entry contains several fields. \(C\_Tag\) is the chunk’s unique tag. \(Sigs\) is the R and W signatures. \(Chunk\ State\) is the state of the chunk. As indicated before, \(g\_vec\) is a bit vector with the set of directory modules in the chunk’s read- and write-sets, while \(inval\_vec\) is a bit vector with the set of sharer processors that need to be invalidated (based on the state in this directory). The final \(inval\_vec\) is built by accumulating the \(inval\_vec\) fields of all participating directories through the \(g\) message. Finally, there are three status bits for the chunk. \(l\ (leader)\) indicates whether this directory is the leader of the group. \(h\ (hold)\) indicates that no conflicts were found in this directory and that this directory was admitted into the group. It is set right before sending a \(g\) message. Finally, \(c\ (confirmed)\) indicates that the group has been successfully formed. For the leader, it is set after a \(g\) message is received from the last module in the group; for a non-leader, it is set after
a `g_success` message is received from the leader.

Overall, the system operates similarly to a conventional directory-based protocol, but maintains “coherence” at the granularity of chunks.

## 2.6 Results

We evaluate ScalableBulk using a cycle-accurate execution-driven simulator based on SESC \[63\]. We model a multicore system like the one in Figure 2.1 in which we can configure the number of cores to be 32 or 64. The cores issue and commit one instruction per cycle. Memory accesses can be overlapped with instruction execution through the use of a reorder buffer. Each core has private L1 and L2 caches that are kept coherent using a directory-based scheme that implements the ScalableBulk protocol. The cores are connected using an on-chip 2D torus modeled with the simulator of Das \textit{et al} \[18\]. A simple first-touch policy is used to map virtual pages to physical pages in the directory modules. Table 2.2 shows more details.

<table>
<thead>
<tr>
<th>Processor &amp; Interconnect</th>
<th>Memory Subsystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores: 32 or 64 in a multicore</td>
<td>Private write-through D-L1:</td>
</tr>
<tr>
<td>Signature:</td>
<td>size/assoc/line:</td>
</tr>
<tr>
<td>Size: 2 Kbits</td>
<td>32KB/4-way/32B</td>
</tr>
<tr>
<td>Organization: Like in [15]</td>
<td>Round trip: 2 cycles</td>
</tr>
<tr>
<td>Max active chunks per core: 2</td>
<td>MSHRs: 8 entries</td>
</tr>
<tr>
<td>Chunk size: 2000 instructions</td>
<td>Private write-back L2:</td>
</tr>
<tr>
<td>Interconnect: 2D torus</td>
<td>size/assoc/line:</td>
</tr>
<tr>
<td>Interconnect link latency: 7 cycles</td>
<td>512KB/8-way/32B</td>
</tr>
<tr>
<td>Coherence protocol: ScalableBulk</td>
<td>Round trip: 8 cycles</td>
</tr>
<tr>
<td></td>
<td>MSHRs: 64 entries</td>
</tr>
<tr>
<td></td>
<td>Memory roundtrip: 300 cycles</td>
</tr>
</tbody>
</table>

Table 2.2: Simulated system configurations.

For the evaluation, we run 11 SPLASH-2 applications and 7 PARSEC applications. The applications run with 32 and 64 threads. We run the applications with reference data sets for all runs. For LU and Ocean from SPLASH-2, we use the more locality-optimized contiguous versions. For the PARSEC applications, we use the small input sets, except for Dedup and Swaptions, which run
with the medium and large input sets, respectively, due to scalability reasons.

We also implement and evaluate several protocols proposed in previous work. They are shown in Table 2.3.

<table>
<thead>
<tr>
<th>Name</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScalableBulk</td>
<td>Protocol proposed</td>
</tr>
<tr>
<td>TCC</td>
<td>Scalable TCC [16]</td>
</tr>
<tr>
<td>SEQ</td>
<td>SEQ-PRO from [57]</td>
</tr>
<tr>
<td>BulkSC</td>
<td>Protocol from [15] with arbiter in the center</td>
</tr>
</tbody>
</table>

Table 2.3: Simulated cache coherence protocols.

2.6.1 Performance and Scalability

Figures 2.7 and 2.8 show the execution time of the applications on ScalableBulk, TCC, SEQ, and BulkSC for 32 and 64 processors — normalized to the execution time of single-processor runs on the same architecture with ScalableBulk. Each bar is labeled with the name of the application and the number of processors. The last two bars show the average. The bars are broken down into the following categories, from bottom to top: cycles executing one instruction (Useful), cycles stalling for cache misses (Cache Miss), cycles stalling waiting for a chunk to commit (Commit) and cycles wasted due to chunk squashes (Squash). The number on top of each bar is the speedup.

From the average values, we see that the BulkSC protocol does not scale well going from 32 to 64 processors due to its centralized nature. Distributed protocols such as ScalableBulk, TCC, and SEQ scale better going from 32 to 64 processors, but TCC and SEQ show significant commit overhead for applications such as Radix, Barnes, Canneal, and Blackscholes. ScalableBulk suffers almost no commit overhead even for these applications due to its overlapped nature. Squash overhead is generally minimal, since data conflicts between two chunks are relatively rare and not very costly even when they happen — given the small chunk size (2000 instructions). In ScalableBulk for 64 processors, only 1.5% of all chunks were squashed due to data conflicts and 2.3% were squashed due to address aliasing in the signatures.
(a) ScalableBulk

(b) TCC
Figure 2.7: Execution times of the SPLASH-2 programs normalized to single-processor runs with ScalableBulk.
(a) ScalableBulk

(b) TCC
Figure 2.8: Execution times of the PARSEC programs normalized to single-processor runs with ScalableBulk.
Radix shows a large commit overhead for TCC and SEQ. This is because, as we will see in Section 2.6.2 chunks in Radix use a large number of directory modules compared to other applications. Moreover, most of these modules record writes. Radix implements a parallel radix sort algorithm that ranks integers and writes them into separate buckets for each digit. The writes to these buckets are random depending on the integer, and have no spatial locality. This results in a large number of directory modules that record writes per chunk, and in serialization for non-overlapped protocols.

Previous work on Scalable TCC [16] shows smaller overheads for Radix and Barnes, but this was in the context of software-defined transactions. Such transactions are much larger than the automatically-generated chunks of this work. As a result, the transactions do not commit as frequently, leading to better scalability. However, they need software to define them.

Ocean, Cholesky, and Raytrace attain superlinear speedups. The reason is that the single-processor runs can only use a single L2 cache, while the parallel runs use 32 or 64 L2 caches.

2.6.2 Directories Accessed per Chunk Commit

Figures 2.9 and 2.10 show the average number of directory modules accessed per chunk commit in the ScalableBulk protocol. The data is shown for the SPLASH-2 and PARSEC applications. The figures show data for each application for 32 and 64 processors, and the average across applications. Each bar shows the number of directories that record at least one write (Write Group) and of those that record only reads (Read Group).
The scalability of distributed commit protocols such as ScalableBulk, TCC, and SEQ depends on chunks accessing a small number of directory modules. We see from the figures that most applications access an average of 2–6 directories per chunk commit. These numbers are significantly larger than the ones reported in the Scalable TCC paper [16]. Moreover, in some applications such as Barnes, Canneal, and Blaekscholes, chunks access a much larger set of directories. Radix is especially challenging in that practically all of the directories in the group record writes.
Large groups are especially harmful for TCC and SEQ, due to their inability to concurrently commit two overlapping chunks. However, ScalableBulk suffers no noticeable commit overhead thanks to its ability to overlap groups through the use of signatures.

Figures 2.11 and 2.12 show the distribution of the number of directory modules accessed per chunk commit in the ScalableBulk protocol for the SPLASH-2 and PARSEC applications. We can see that, in some applications, there is a significant tail of chunks with high numbers of directories accessed.
Figure 2.11: Distribution of number of directories accessed per chunk commit in SPLASH-2 for 64 processors.

Figure 2.12: Distribution of number of directories accessed per chunk commit in PARSEC for 64 processors.
2.6.3 Chunk Commit Latency

Figure 2.13 shows the distribution of the latency of a chunk commit operation. The data corresponds to the average of all the applications running on 64 processors. The mean latencies for ScalableBulk, TCC, SEQ, and BulkSC are 91, 411, 153, and 2954 cycles respectively. For 32 processors, the mean latencies are 74, 402, 107, and 98 cycles, respectively. ScalableBulk not only has lower latencies than all of the existing protocols, but also scales well. BulkSC has the worst scaling behavior.

2.6.4 Chunk Commit Serialization

In order to analyze the divergent commit latencies shown in Section 2.6.3, we measure two additional metrics: the bottleneck ratio and the chunk queue length.

Bottleneck Ratio

We define the Bottleneck Ratio as “the number of chunks in the process of forming groups” over “the number of chunks that have successfully formed groups and are in the process of completing the commit”. We exclude from the numerator those chunks that are forming groups that will later be squashed. This ratio is sampled every time that a new group is formed. A high ratio signifies that groups are taking a long time to form, most likely due to a bottleneck — e.g., in the case of the less-overlapped protocols, group formation is stalled waiting for another group to commit. A low ratio signifies that groups are getting formed and processed quickly through the system.

A high bottleneck ratio does not necessarily imply a high commit overhead because the commit latency can be hidden by the execution of the next chunk in the processor. However, given an application, a high bottleneck ratio in a protocol compared to another protocol is a good indicator that group formation is taking a longer amount of time.

Figures 2.14 and 2.15 show the bottleneck ratios in SPLASH-2 and PARSEC for ScalableBulk, TCC and SEQ. We do not show data for BulkSC because it does not form groups.
Figure 2.13: Distribution of the latency of a chunk commit operation.
From the figures, we see that the bottleneck ratio in ScalableBulk is uniformly low and is about 1 on average. This roughly means that chunks spend about the same amount of time forming a group as committing the group. In contrast, SEQ and especially TCC, have higher bottleneck ratios. Some of the applications that have a high bottleneck ratio are those with the most commit overhead in Figures 2.7 and 2.8 such as Radix, Barnes, FMM, Blackscholes, and Canneal.
**Chunk Queue Length**

The Chunk Queue Length is the number of chunks in the whole machine that are queued waiting to commit. A completed chunk gets queued in the TCC and SEQ protocols when the directory modules that it accessed overlap with those accessed by earlier, yet-uncommitted chunks. Chunks do not get queued in ScalableBulk because ScalableBulk enables full overlap of chunk commits using signatures. We sample the chunk queue length every time that a new group is formed. A long chunk queue means that a completed chunk has to wait for a long time to commit. Therefore, it signifies commit serialization.

Figures 2.16 and 2.17 show the average chunk queue lengths in TCC and SEQ for all the applications with 64 processors. Long chunk queues do not necessarily imply high commit overhead because the commit latency can be hidden by the execution of other chunks. However, we can see that applications such as Radix, Blackscholes, and Canneal, which have high commit overheads in Figures 2.7 and 2.8, do in fact have long chunk queues.

![Figure 2.16: Chunk queue lengths in SPLASH-2 codes.](image)
Figure 2.17: Chunk queue lengths in PARSEC codes.
2.6.5 Traffic Characterization

Figure 2.18: Message characterization in SPLASH-2. $S$, $T$, $Q$, and $B$ stand for ScalableBulk, TCC, SEQ, and BulkSC, respectively.

`LargeCMessage` and `SmallCMessage` are large and small messages, respectively, in the commit protocol. For example, in ScalableBulk, the `LargeCMessage` are those that carry signatures, namely `commit_request` and `bulk_inv` in Table 2.1, while `SmallCMessage` are the rest of the messages in Table 2.1.
Figures 2.18 and 2.19 show the number and distribution of the messages in the network for the different coherence protocols. The data is shown for each application running on 64 processors. For a given application, the bars are labeled as S (for ScalableBulk), T (for TCC), Q (for SEQ), and B (for BulkSC), and are normalized to the number of messages in TCC. The messages are classified into five classes: reads of a cache line from memory (MemRd), reads of a cache line from another cache, either in state shared (RemoteShRd) or in state dirty (RemoteDirtyRd), and
two classes of commit-related messages (*LargeCMessage* and *SmallCMessage*).

From the figures, we see that TCC generates significantly more messages than the other protocols, and that these messages are mostly commit-related small messages. These messages are mostly the skip and probe messages. This results in a more congested network, potentially increasing the commit overhead and delaying the read messages.

### 2.7 Conclusion

To boost programmability and performance, researchers have proposed architectures that continuously operate on chunks of instructions. These systems must support chunk operations efficiently. In particular, in lazy conflict-detection environments, they must provide scalable chunk commits. Unfortunately, current proposals very often limit the overlap of conflict-free chunk commits.

This section presented ScalableBulk, a novel directory-based protocol that enables highly-overlapped, scalable chunk commits. ScalableBulk uses hardware address signatures to optimize chunk operations. It introduces three general hardware primitives for scalable commit: preventing access to a set of directory entries, grouping directory modules, and initiating commit optimistically. Our results with SPLASH-2 and PARSEC codes with up to 64 processors show that ScalableBulk enables highly-overlapped chunk commits and scalable performance. Unlike previously-proposed schemes, it removes practically all commit stalls.
Chapter 3

IntelliSense: A Scalable and Fast Chunk Commit Protocol with Reduced Squash

3.1 Introduction

A class of recently-proposed shared-memory architectures attempts to improve both performance and programmability by continuously executing chunks of consecutive instructions from the program in an atomic manner. In an environment with lazy conflict detection, there are several proposals that specifically target the commit bottleneck, namely Scalable TCC [16], SRC [57], and ScalableBulk [58]. While these designs have improved the commit operation, it still remains a major source of overhead, especially when dealing with high processor counts, committing chunks that have accessed data mapped to many directories, and small chunk sizes.

This section focuses on the challenge of providing scalable and fast chunk commit with reduced squash for a large manycore in a lazy environment.

3.2 Contribution

This section makes three contributions:

• To understand the problem, it presents a novel model of chunk commit in a distributed directory-based protocol, and shows how past schemes map to it.

• It introduces two new, general techniques to attain scalable and fast commit. The first one, called IntelliSquash, is the serialization of the write sets of WAW-dependent chunks. This technique eliminates chunk squashes due to WAWs.

• The second technique, called IntelliCommit, is the full parallelization of how committing chunks
attain ownership of directory modules. This technique speeds-up the critical path of the commit operations.

3.3 Background and Related Work

Most of the background and related works on the distributed commit protocol have been discussed in the corresponding section in Chapter 2. Beyond the distributed commit protocol in directory-based lazy environment listed in Section 3.3, the most relevant work to IntelliSquash includes techniques that try to increase the concurrency of conflicting transactions in hardware transactional memory systems. There are three proposals, DATM [61], SONTM [5] and BulkSMT [59]. These systems use the conflict serialization to avoid the squashes by transaction (chunk) ordering. However, such ordering incurs extra constrains in the execution and may incur more squashes eventually. For example, if two chunks have cyclic WAW dependences, both chunks need to squash. IntelliSquash avoids squashes without incurring any extra ordering requirement. Moreover, these systems incur considerable complexity. One such example is DATM [61], which needs 11 more stable states beyong the basic MSI protocol and more extra transient states may be needed in the implementation. In IntelliSense, the three state (H,R,G) is for each commit transaction, instead of each cache line. Several hybrid TM systems [65, 71] moves some operations (e.g. conflict detection) in the grouping stage in our model to the execution stage and they don’t perfectly fit into our model. Those systems don’t support the squash avoidance on WAW-only conflicts.

3.4 A Model of the Lifetime of a Chunk

3.4.1 The Model

The lifetime of a chunk in a directory-based lazy environment can be thought of as three sequential stages: Execution, Grouping, and Propagation (Figure 3.1). The last two, combined, form the Commit stage.
Execution is the time during which the processor executes the chunk’s instructions. Reads and writes bring lines into the cache, but no written line is made visible to other processors, which continue to read the non-speculative value of the line from memory or other caches. Execution terminates when the last instruction of the chunk completes.

Grouping is the first stage of the commit. It sets the relative order of any two (or more) chunk commits that cannot proceed concurrently. The Grouping stage of a chunk commit attempts to establish a coordination between all the directory modules that map the lines accessed by the chunk. We call this process grabbing the directories. Two chunks that are not allowed to commit concurrently because they have accessed common memory lines will conflict when they try to grab the same directory. Only one of the chunks will succeed and proceed to the Propagation stage; the other will stall until the first one has completed its Propagation. As soon as the Grouping stage completes successfully, the commit is guaranteed to succeed (i.e., the chunk will not be squashed). Hence, at this point, the processor can start to execute the next chunk of the program.

Propagation involves making the stores in the chunk visible to the rest of the system. While Propagation appears atomic, it takes a certain time, as it requires updating directory entries and sending invalidations to other caches in the machine. Propagation may involve no data transfer. While Propagation takes place, the directory modules prevent accesses by other processors to the directory entries of the lines accessed by the committing chunk; this ensures the atomicity of the Propagation. After Propagation completes, no processor can see the old values of the updated lines. Propagation can overlap with the execution of the next chunk in the initiating processor.

A chunk in the Propagation stage squashes other processors’ chunks in the Execution or Group-
ing stages if the latter are data dependent on the former. In current systems, this is implemented by comparing the Propagating chunk’s write set (i.e., the addresses written) to the read and to the write sets of the other chunk. If there is an overlap, that other chunk is squashed — ostensibly because it has accessed stale data.

3.4.2 Stalling in the Grouping Stage

Two chunks are said to be conflicting if they have overlapping footprints. This means that some addresses in the write set of one chunk are also present in the read or in the write sets of the other. When two conflicting chunks perform Grouping concurrently, only one succeeds, and the other stalls in the middle of its Grouping until the first one completes the Propagation.

If the stalled chunk (call it $C_1$) only has WAR dependences with the successful one (call it $C_0$), it will not be squashed. Hence, $C_1$ could be allowed to proceed with its Grouping as soon as $C_0$ finishes Grouping, without waiting for the end of $C_0$’s Propagate. While this is possible, we discourage it because overlapping Propagates may result in processors receiving messages out-of-order, which complicates the protocol.

3.4.3 Application of the Model to Existing Protocols

We apply our model to different protocols in the literature.

**Scalable TCC** [16]. In the Grouping stage, a chunk first obtains a global-order transaction ID (TID) from a central agent. This operation logically serializes competing commits. In addition, the committing processor sends messages to all of the directory modules in the machine, to identify when the Grouping stage is over and it can start Propagation. These messages are a Skip to the directories not in the chunk’s write set, and (potentially several) probes to the directories in the chunks write and read set. When all of the directories have completed the conflicting previous commits, then the probes succeed. Then, the Propagation stage starts. It involves sending the chunk’s write addresses to the directories in the write set. In this protocol, two chunks can commit
concurrently only if they use different directory modules. If any directory in the write set of one chunk appears in the read or write set of the other, the commits are serialized — even if the addresses accessed by the two chunks are different.

**SRC** [57]. SRC optimizes the Grouping stage by eliminating the centralized agent and broadcasting step. A processor with a committing chunk attempts to grab the directory modules in the chunk’s access set in a *sequential* manner, from the lowest-numbered one to highest-numbered one. Two committing chunks that attempt to grab the same directory get ordered. The one that fails waits until the successful one fully completes its commit. Like in Scalable TCC, two chunks that use the same directory module but access non overlapping addresses cannot commit concurrently. Once a committing chunk grabs all of its directories, it proceeds to the Propagation stage. The Propagation stage executes as in Scalable TCC\(^1\).

**ScalableBulk** [58]. ScalableBulk further optimizes the Grouping in two ways. First, although the directories are still grabbed in a *sequential* manner, the transaction does not involve repeated round trips from the initiating processor to the directories — the directories organize themselves as a group. Secondly, two chunks can concurrently grab the same directory module as long as the addresses they accessed do not overlap. This is accomplished by using address signatures to represent access sets, and intersecting them to detect overlap. In the Propagation stage, ScalableBulk does not propagate addresses; only signatures are passed between nodes, which is cheaper.

### 3.4.4 Sources of Inefficiency

As we have seen, successive proposals have progressively optimized the commit — especially the Grouping stage, which is the one in the critical path. However, there are still major bottlenecks. An obvious one is that the Grouping stage still requires grabbing directory modules in a *sequential* manner.

A second, subtler one, is that a chunk in the Propagation stage squashes a chunk in the Execu-

\(^1\)The authors outline an optimization called SEQ-TS that improves on SRC. However, there are few details on how it works.
tion or Grouping stages even if there are only write-after-write (WAW) dependences between the chunks. These are name dependences rather than true dependences, and should not cause a squash.

In this paper, we eliminate these bottlenecks with two new designs, called IntelliCommit and IntelliSquash, respectively. With them, we make chunk commit highly efficient and, therefore, retain high performance in executions with high processor counts and with small-sized chunks. The latter is important because, in some workloads, large-sized chunks are not an option: they suffer frequent dependences that lead to squashes. In the following, we present the two new techniques.

3.5 IntelliSquash: No Squash on WAW

3.5.1 Basic Idea

In existing chunk-based protocols, if a chunk currently in the Propagation stage wrote to an address that a later chunk \( C \) has read, \( C \) will get squashed. This is needed because \( C \) has read stale data. However, most protocols also squash \( C \) even if it has only written to the address that the Propagating chunk wrote. This is a WAW and should not induce a squash.

The only exceptions to this behavior are protocols such as DATM [61], SONTM [5], and BulkSMT [59], which dynamically forward data between concurrently-executing, dependent chunks, and then force an order to their commits. Such protocols, however, add an extra layer of complexity that we want to avoid. Our goal is to augment a baseline chunk protocol with a general primitive that prevents squashing the chunk if there are only WAWs — without having to record the dependences and order the chunks. We call our technique IntelliSquash.

To understand how IntelliSquash works, consider a conventional multiprocessor where the write buffers of two processors (\( P_0 \) and \( P_1 \)) each have a store (\( w_0 \) and \( w_1 \)) to the same line. Suppose that the line is in state Shared in the caches of both processors. The stores drain and get ordered without requiring squash and re-execution. Specifically, both stores issue requests for line ownership. The one that reaches the line’s home directory first (say \( w_0 \)) gets serialized before the
other, and invalidates the line from the other cache (P₁’s). The line becomes owned by P₀ and w₀ is applied. Later, w₁ misses in its cache, reaches P₀’s cache, obtains the line (and invalidates it), and brings it to P₁’s cache, where w₁ is applied. The effects of the two stores are serialized without store re-execution.

Current chunk commits do not work like this. In a typical implementation, the L1 cache serves the same purpose for a chunk as the write buffer for an individual write. The difference is that, in a conflict, the data in the write buffer is not affected by an external invalidation, while the cache lines that the chunk updated get invalidated. Hence, the chunk’s updates are lost. This data can only be recovered by squashing and re-executing the chunk.

IntelliSquash uses the idea of the write buffer to serialize, without any squash, the commits of two chunks that only have WAWs. Specifically, when a chunk C₁ only has WAWs with a chunk C₀ currently in the Propagation stage, some lines in C₁’s cache will be invalidated, but the current write set of C₁ (set of speculative data produced by C₁) is not lost. Later, when C₁ commits, its final write set will be merged with the memory system state. Overall, the write sets of the two chunks are applied in a serial manner, without squashes.

3.5.2 IntelliSquash Design

To support IntelliSquash, we extend the cache with some bits that trigger certain transactions. In the following, we explain the design. Without losing generality, we explain it as extensions to a protocol with signatures.

Additional Cache Bits.

The cache must have the ability to retain speculatively-written data even while it invalidates the rest of the line. For this reason, we extend each cache line with one Absent (A) bit, and with fine-grain dirty bits (one bit per word or per byte, depending on the granularity that we want to support). Figure 3.2 shows the design. In Figure 3.2(a), we show a conventional 4-word cache line with a Speculative (Sp), Valid (V) and Dirty (D) bit. In Figure 3.2(b), we show the line
under IntelliSquash: it adds the Absent (A) bit and per-word Dirty (d) bits (since we assume word granularity).

![Cache line augmented for IntelliSquash.](image)

To see how these bits work, assume that chunk $C_1$ writes `foo` to the second word of a line and misses in its cache. As in conventional schemes, the line is brought from the memory, the word is updated, and $Sp$, $V$ and $D$ are set. In addition, the second word’s $d$ bit is set (Figure 3.2(c)). Suppose that, at this point, the local processor receives the write set (e.g., in a signature) from a remote chunk $C_0$ that is in the Propagation stage. Further, the write set overlaps with $C_1$’s write set (at a line granularity) but not with $C_1$’s read set.

In this case, IntelliSquash sets the line’s A bit, transitioning it to the Dirty Absent state — effectively invalidating the words with $d=0$. As seen from the directory, this state is the same as Invalid (the processor is not a sharer of the line anymore in the directory). However, as $C_1$ continues to execute, accesses to words with $d=1$ are satisfied as cache hits. Accesses to words with $d=0$ induce a cache miss that brings-in the line (but only overwrites the part of the line with $d=0$) and marks the processor as sharer in the directory. This transaction is called a Merge transaction. Finally, when $C_1$ commits, it issues Merge transactions for all its Dirty Absent lines.
**Merge Transaction on a Miss.**

When a chunk accesses a word with $d=0$ in a line in Dirty Absent state, the cache initiates a Merge transaction. The transaction proceeds like a miss, obtaining the current non-speculative version of the line, and recording the requesting processor as a sharer in the directory. Once the line arrives, it only updates the words that had $d=0$ and clears the $A$ bit, transitioning to the Dirty state. The $Sp$, $V$ and $D$ bits remain set. If the access was a store, the $d$ bit of the word is set.

The lazy protocol that we assume is like BulkSC, in that a write miss to a line by a chunk appears as a read miss outside the cache: the state of the line in the directory is set to Shared, although the line is (speculatively) Dirty in the cache.

**Merge Transactions on a Commit.**

When a chunk commits, the commit automatically generates Merge transactions for all its lines in Dirty Absent state. These transactions obtain individual lines from main memory (if the directory has them as Shared) or from another cache (if the directory has them as Dirty in a cache). As usual, all of the lines in the write set of the committing chunk (including those in Dirty Absent state) are invalidated from the other caches, and are marked by the directory as Dirty in the committing processor.

To see how this is done with signatures, consider a protocol like BulkSC. In the Propagation stage, the committing chunk’s write signature ($W$) is expanded in the directory into its constituting addresses. For each such address, the line is marked as Dirty in the directory (and owned by the committing processor). In addition, $W$ is sent to the sharer processors to invalidate the cache lines whose addresses are in $W$. While the invalidation is in progress, the directory uses $W$ to make the entries for the addresses in $W$ inaccessible to other processors. In IntelliSquash, it is during this time that the Dirty Absent lines are sent to the committing processor. The directory recognizes such lines because, although they belong to $W$, they are marked in the directory as not present in the committing processor. Recall also that a Dirty Absent line may be (non-speculative) Dirty.
in another cache, in which case the directory first asks the owner to write it back (and invalidate itself) before forwarding the line to the committing processor. At the end of the commit, all the lines in the write set of the committing chunk are marked in the directory as Dirty in the committing processor.

**Effects of False Positives.**

When W expands in the directory, it may generate line addresses that were not in the committing chunk’s write set. They are false positives. Such false positives can at most create unnecessary traffic, but never affect correctness. To see why, consider a line address emerging from the expansion. Its directory entry can be in one of the four possible states in Table 3.1 — depending on whether the Dirty bit (D) and/or the presence bit of the committing processor (K) in the Bit Vector are set. The table shows the actions that IntelliSquash takes. Note that, in each case, it may be a false positive.

<table>
<thead>
<tr>
<th>Current Entry State</th>
<th>Action in IntelliSquash</th>
<th>Action in BulkSC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty Bit</td>
<td>Committing Proc Bit in Bit Vector</td>
<td></td>
</tr>
<tr>
<td>D=0 K=1</td>
<td>Invalidate sharer caches Reset rest of bit vector Set Dirty bit</td>
<td>Same as IntelliSquash</td>
</tr>
<tr>
<td>D=0 K=0</td>
<td>Merging transaction: Provide line to committing proc Invalidate sharer caches Clear bit vector and set the bit for the committing processor Set Dirty bit</td>
<td>Do nothing: false positive</td>
</tr>
<tr>
<td>D=1 K=0</td>
<td>Merging transaction with owner: Owner writes back &amp; invalidates Provide line to committing proc Clear bit vector and set the bit for the committing processor</td>
<td>Do nothing: false positive</td>
</tr>
<tr>
<td>D=1 K=1</td>
<td>Do nothing: false positive</td>
<td>Same as IntelliSquash</td>
</tr>
</tbody>
</table>

Table 3.1: Directory entry states after signature expansion.

If the directory has D=0 and K=1, we assume it is a normal write. IntelliSquash invalidates the
other sharer caches, clears the rest of the Bit Vector (only its bit remains), and sets D. These are the same actions as in BulkSC. If it is a false positive, we are changing the directory to an inaccurate state, but one from which the protocol can exit gracefully \[15\].

If the directory has D=0 and K=0, IntelliSquash initiates a Merging transaction. It involves the same operations as in the previous case plus providing the line to the committing processor and setting its K bit in the directory to 1.

If the directory has D=1 and K=0, IntelliSquash initiates a Merging transaction with owner. The directory requests the line from the owner processor and sends it to the committing one. The old owner invalidates its cache entry. The K bit of the old owner is cleared, while the one for the committing processor is set. The Dirty bit stays set.

The two cases just described are false positives in BulkSC and no action is taken. In IntelliSquash, if they are false positives, we have simply forwarded an unrequested line to the committing processor and marked it as the owner. The cache in the committing processor may choose to take-in the line or silently reject it. In the latter case, the directory is left pointing to the processor, which is inaccurate but easy to recover from: it is like a processor reading a line in Exclusive state and then evicting it from its cache silently. On a future request, the protocol realizes that the cache does not have the line and provides it from the memory.

The final case, where the directory has D=1 and K=1, is a false positive. Hence, IntelliSquash (like BulkSC) does nothing. Overall, we see that IntelliSquash always works correctly.
3.6 IntelliCommit: Parallel Grouping

Figure 3.3: IntelliCommit commit protocol. The figure shows five nodes with processor (P) and directory (D).

3.6.1 Basic Idea

While the critical path of the commit operation in a lazy environment is the Grouping stage (Section 3.4.1), the recently-proposed designs still execute it inefficiently. Specifically, the Grouping stage for a chunk in SRC [57] and ScalableBulk [58] grabs the directory modules in the access set of the chunk in a sequential manner. Hence, if a chunk commit needs to grab many directory modules, Grouping is slow.

To speed-up Grouping, this section proposes IntelliCommit, the first design where the Grouping transaction grabs the directory modules in parallel. The idea is for the committing processor to send commit_request protocol messages to all of the relevant directory modules, get their responses directly, and finally send them a commit_confirm message. The main challenge is to correctly arbitrate multiple concurrent chunk commits. In the following, we present the commit protocol and its arbitration mechanism, and discuss its correctness.
3.6.2 IntelliCommit Commit Protocol

Our IntelliCommit protocol uses address signatures like ScalableBulk [58] to encode the read and write sets of a chunk. Figure 3.3 shows the commit protocol, using a scalable architecture where each node has a processor and a directory module. During the execution of a chunk, the processor records the set of directory modules that are the homes of the lines accessed in the chunk. Then, when the chunk finishes execution, IntelliCommit sends a commit_request with the chunk’s signatures \( R_{\text{com}} \) and \( W_{\text{com}} \) to such directories. Figure 3.3(a) shows the case with three such directories. Then, these directories respond with commit_ack messages (Figure 3.3(b)), which include Invalidation Sets (i.e., the set of nodes that should receive invalidations, according to the local directory’s sharing information). In addition, the directories use the \( R_{\text{com}} \) and \( W_{\text{com}} \) signatures to disallow subsequent accesses to the lines in \( R_{\text{com}} \) and \( W_{\text{com}} \), preventing a conflicting chunk from Grouping concurrently.

When the processor receives all of the expected commit_acks, a group has been formed, and the Grouping stage ends. Then, the Propagation stage starts, while the processor proceeds to execute the next chunk. Propagation starts with a commit_confirm message from the processor to the same directories (Figure 3.3(c)). As in ScalableBulk, there is a default policy that designates a Leader in each group — e.g., the lowest-numbered node. As part of the commit_confirm message to the Leader, the committing processor includes the union of all of the Invalidation Sets.

The Leader then sends \( W_{\text{com}} \) to all of the nodes in the combined Invalidation Set, for disambiguation (and possible chunk squash) and cache invalidation. Figure 3.3(d) shows the case with two such nodes. When all the nodes in the set indicate with an acknowledgment that such operation is completed, the Leader sends a done message to the other directory modules in the group, so that they make the corresponding directory entries accessible again (Figure 3.3(e)). The commit is done.

Between the time that a processor sends the commit_requests for a chunk (Figure 3.3(a)) and the time it receives all the commit_acks (Figure 3.3(b)), the processor may receive a W signature.
from the commit of another chunk that squashes the current chunk. In this case, the processor immediately sends a commit_cancel message to all the directories it had sent the commit_requests. In addition, it will discard any incoming commit_acks. The directories, on reception of the commit_cancel, discard the signatures of the chunk, make its directory entries accessible, and consider the commit aborted.

A processor cannot receive a \( W \) signature that squashes its chunk \( C_1 \) after it has already received all the commit_acks for \( C_1 \) (Figure 3.3(b)). The reason is that the directories cannot form the group for \( C_1 \) (by all sending the commit_ack to the processor), if another chunk \( C_0 \), which has signatures overlapping with those of \( C_1 \), is currently in the Propagation stage and, therefore, can squash \( C_1 \). If such a chunk \( C_0 \) existed, it would have prevented the directories where \( C_0 \) and \( C_1 \) overlap from sending a commit_ack to the processor of \( C_1 \) in the first place. Hence, \( C_1 \) cannot have formed its group. To see this, we now describe how chunks compete in the Grouping stage.

### 3.6.3 States of a Committing Chunk in a Directory

From a directory module’s viewpoint, a chunk commit starts when the directory receives a commit_request, and it ends when the directory receives acks from all of the invalidated caches (if it is the leader) or the done message (otherwise). It may also end early if the directory receives a commit_cancel message. During the commit of the chunk, the chunk goes through the states of Figure 3.4 in a directory module.
Assume that a directory module receives a *commit_request* for a chunk \( C_0 \) (which includes its signatures). If there is no other chunk in that directory, the directory responds with a *commit_ack* and sets the chunk state to Ready (R). Later, when the directory receives the *commit_confirm*, the commit is irreversible, and the chunk transitions to Granted (G). It will exit G when the commit completes.

The same process also takes place if the arriving chunk’s signatures do not overlap with any of the signatures of the chunks \( C_1 \) that are currently in R or G states — specifically, \( R_0 \cap W_1 \), \( R_1 \cap W_0 \), and \( W_0 \cap W_0 \) are null.

However, if the signatures overlap, the arriving chunk \( C_0 \) is set to the Hold (H) state, and the directory does not send *commit_ack*. Each chunk in H has a *Wait Set* list, with the chunks currently in G or R that it overlaps with. Moreover, \( C_0 \) compares its priority to that of its overlapping chunks in R. If \( C_0 \)’s priority is higher, it attempts to preempt them, by sending preempt requests. We will see in Section 3.6.4 how we assign priorities and how preemption works.
$C_0$ moves from $H$ to $R$, and the directory sends a *commit_ack* as soon as (i) all of its higher-priority conflicting chunks have left the G+R states and (ii) all of its lower-priority conflicting chunks in $R$ have been preempted.

At all times, the signatures of the chunks in $G$, $R$, or $H$ in the directory module are buffered. When the directory module receives a *commit_cancel* for a chunk in $H$ or $R$, the commit of that chunk terminates.

### 3.6.4 BlockSort: Distributed Algorithm to Order Chunks

We now describe how we order conflicting chunks.

**Significance.**

In a directory protocol for lazy-conflict chunks, it is challenging to devise a low-overhead, scalable Grouping stage. The difficulty comes from having to handle concurrent requests from potentially conflicting chunks in a fully distributed environment. As an example, consider two committing chunks that need to grab two directory modules each. If the two chunks either (i) use different directory modules or (ii) use the same directory modules but their signatures do not overlap, then the Grouping stages of the two chunks can proceed in parallel. Otherwise, one of chunks will succeed in building a group, while the other has to stall.

The decision of which chunk succeeds must be the same in all of the relevant directories. Unfortunately, messages may arrive with different timings at different directories and, initially, different directories may make opposite decisions. To reach a single decision, the protocol needs two supports. First, there has to be a known policy that, on a conflict, mandates which chunk has priority. Secondly, the protocol has to be able to change the decision taken by a directory if, due to reasons of message timing, that directory initially took an inconsistent decision. This is the *preemption* operation mentioned above. Consequently, preemption is not a performance issue, but a correctness one.

In this section, we describe the priority policy and the preemption algorithm, called *BlockSort*. 
**Priority Policy.**

IntelliCommit needs a default priority policy that directories can use *locally* when two chunks with overlapping signatures want to grab the same directory module. The main requirement of the policy is fairness. In our design, each processor generates a random number and attach the number with the *commit_request*. When two requests conflict, the directory gives priority to the one with the smaller random number. In a tie, the lower ID wins.

![Diagram](image)

**Figure 3.5:** Example of a chunk preempting another.

**Preemption Algorithm: BlockSort.**

BlockSort’s approach is to make decisions based on information available locally in the directory module, and only introduce a small number of inter-node messages to accomplish the preemption. The BlockSort algorithm is run by a directory controller when there is a chunk in state H that could be in state R except for a conflicting, lower priority chunk already in R. In this case, correctness requires that preemption be attempted, since it is possible that, in another directory that both chunks also need to grab to commit, the higher and lower priority chunks are in the opposite states due to message timing differences. Hence, not performing preemption could result in deadlock.

Let us call \( C_{low} \) the low-priority chunk in state R, \( P_{low} \) the processor where \( C_{low} \) runs, \( C_{high} \) the high-priority chunk in state H, and \( P_{high} \) the processor where \( C_{high} \) runs. Preemption starts with the directory sending a *preempt_request* message to \( P_{low} \) on behalf of \( C_{high} \). If \( P_{low} \) has already started its Propagation stage (because it has already received all its *commit_acks*), then \( P_{low} \) simply responds with a *preempt_nack*. It is too late to perform a preemption, and \( C_{high} \) has to wait until
the commit of $C_{low}$ completes, unless it is squashed before.

Otherwise, the preemption occurs. $P_{low}$ responds with a $preempt_ack$ and records which directory requested the preemption and for which block. $P_{low}$ will not proceed to Propagation; it may keep receiving $commit_acks$ for $C_{low}$, but will take no action. When the directory receives the $preempt_ack$, it swaps the states of $C_{high}$ and $C_{low}$. It also sends a $commit_ack$ to $P_{high}$ for $C_{high}$. This same process may occur in several directories.

It is guaranteed that $C_{high}$ will eventually enter the R state in all of its directories, as it will preempt $C_{low}$ everywhere. Thus, the Grouping stage of $C_{high}$ will be able to complete. Once $C_{high}$ completes the full commit, the directories that preempted $C_{low}$ send a $preempt_finished$ message to $P_{low}$.

$P_{low}$ has to receive $preempt_finished$ messages from all of the directories that initially sent $preempt_requests$ and were granted. Once $P_{low}$ has received all of the $preempt_finished$ and also all its $commit_acks$, then $P_{low}$ proceeds to Propagate, by sending $commit_confirm$.

**State Required for BlockSort: Preemption Vector (PV).**

To support preemptions, a processor needs to record which other chunk(s) preempted its chunk, and know when they complete their commit. We support this with a *Preemption Vector (PV)* in the processor. In a machine with $N$ processors, the PV has $N-1$ counters (one for each of the other processors). Each counter can count up to $N-1$.

Suppose that chunk $C$ running in processor $P$ is committing and it gets preempted by chunk $C_j$ running in processor $P_j$. In this case, the $PV[j]$ of processor $P$ will count the number of $preempt_request$ messages that $P$ has received and granted for $C_j$. Later, as processor $P$ receives $preempt_finished$ messages for the chunk committed by $P_j$, it decrements $PV[j]$. Note that a chunk may be preempted by multiple chunks and, therefore, multiple entries in PV may be non-zero. Hence, only when a processor’s PV reaches zero for *all of its entries* can the processor restart the commit of its preempted chunk.

In addition, a processor also needs to record the number of $commit_acks$ that it has received.
for the chunk that it tries to commit. As it restarts the commit of the preempted chunk, it can only
send the \textit{commit\_confirm} when it has received all of the \textit{commit\_acks}.

Finally, a preempted chunk may be squashed due to a dependence. In this case, the processor
sends the usual \textit{commit\_cancel} and clears its PV and the count of the number of \textit{commit\_acks}
received.

\textbf{Preempting Multiple Chunks.}

It is possible that a chunk in state H in a directory needs to preempt multiple chunks that are in state
R in the directory. In this case, BlockSort works seamlessly. The directory sends \textit{preempt\_requests}
to multiple processors. Similarly, a chunk in state R in two directories may be preempted by a
different chunk in each directory. Here, BlockSort also works seamlessly. The processor executing
the chunk receives \textit{preempt\_requests} from the two directories and updates its PV entry.

\textbf{3.6.5 Example}

To illustrate IntelliCommit and its relation to IntelliSquash (Section 3.5), we show in Figure 3.5 an
example of a chunk preempting another. As shown in Figure 3.5(a), processors P_0 and P_3 want to
commit chunks C_0 and C_3, respectively, and both need to grab directories D_1 and D_2. We assume
that their signatures overlap and that C_0 has higher priority.

Both chunks start the Grouping stage at the same time. Let us assume that a \textit{commit\_request}
from P_0 arrives at D_1 first, and one from P_3 arrives at D_2 first. The directories place the chunks
in R state and respond with \textit{commit\_acks} (shown in a simplified format in Figure 3.5(b)). As the
second pair of \textit{commit\_requests} arrive at the directories, since the incoming signatures overlap with
those in state R, the chunks are placed in H state (C_0 is H in D_2 and C_3 is H in D_1). Since C_0 has a
higher priority than C_3, C_0 attempts to preempt C_3 in D_2 by sending a \textit{preempt\_request} from D_2 to
P_3 (Figure 3.5(c)).

Since P_3 has not received all of its \textit{commit\_acks} yet, it allows the preemption, replying with a
\textit{preempt\_ack} to D_2. On reception of the message, D_2 places C_0 in R state and, on behalf of C_0,
sends a commit_ack to P₀ (Figure 3.5(d)). At this point, the commit of C₀ enters the Propagation stage and P₀ sends commit_confirms to the two directories (Figure 3.5(e)). In the meantime, C₃ is waiting in both directories, unable to complete its Grouping stage.

During C₀’s Propagation stage, C₃’s commit can be affected in three different, exclusive ways — depending on the type of overlap that exists between C₀ and C₃’s signatures.

- **If W_{C₀} ∩ R_{C₃} is not null**, it means that C₃ has true dependences with C₀ and has to get squashed. When P₃ receives the W_{B₀} signature for disambiguation, it squashes C₃, terminating its commit. P₃ sends commit_cancel to directories D₁ and D₂. When it later receives a commit_ack from D₁ or a preempt_finished from D₂, it discards them.

- **Otherwise, if W_{C₀} ∩ W_{C₃} is not null**, it means that C₃ has output dependences and no true dependences with C₀. In this case, the IntelliSquash technique of Section 3.5 prevents the squash of C₃. The lines written by both processors become Dirty Absent in P₃’s cache, and C₃’s updates are not lost. C₃ continues to be stalled in the Grouping stage, with P₃ waiting for a commit_ack from D₁ and a preempt_finished from D₂.

- **Otherwise, R_{C₀} ∩ W_{C₃} is not null.** In this case, C₃ only has WAR dependences with C₀. As per Section 3.4.2, IntelliCommit stalls the Grouping of C₃ to make the protocol simpler, and C₃ does not get squashed. P₃ simply waits for the same messages as in the previous case.

Once C₀ completes the Propagation stage, its commit is completed. At this point, the directories send messages on behalf of C₃ to P₃: a commit_ack from D₁ and a preempt_finished from D₂ (Figure 3.5(f)). If C₃ has not been squashed in the meantime (last two cases above), P₃ transitions to Propagate and sends commit_confirms to the two directories (Figure 3.5(g)).

### 3.6.6 IntelliCommit Correctness Properties

In this section, we discuss the correctness of IntelliCommit.
**Atomicity of Commit.** If two chunks that attempt to commit concurrently have dependences, their signatures will overlap. Hence, in the common directories, one of the chunks will be forced to stall the Grouping until the other one completes the Propagation. Therefore, the commits are serialized. Signature intersections can have false positives, in which case IntelliCommit stalls (and perhaps squashes) a chunk unnecessarily. However, correctness is not affected. Signature intersections cannot have false negatives.

**Consensus of Commit Order for Conflicting Chunks.** In the Grouping stage, two conflicting chunks are ordered in the same way in all of the common directories. This is guaranteed for two reasons. First, each directory uses the same information, available locally, to order two chunks that are in R or H states. Second, the preemption algorithm guarantees that if a chunk C₀ preempts a second one C₁ in any directory module, then such a preemption will also succeed in all other overlapped directory modules. This is because as soon as the processor executing C₁ issues a single preempt_ack, it will not send the commit_confirm until it is informed that C₀’s commit has completed (with a preempt_finished).

**Liveness.** In IntelliCommit, a commit eventually succeeds or fails. To see why, consider the property of consensus of commit order. It ensures that one of the conflicting chunks completes. At this point, if the second chunk has not been squashed, it will resume its Grouping stage. This is because the directories where the first chunk stalled the second one will send preempt_finish messages to the second chunk’s processor. The processor will respond with a commit_confirm.

**Deadlock Freedom.** The property of consensus of commit order plus the property of liveness ensure that deadlock cannot happen.

**Starvation Freedom.** If we use a fair policy to decide which of two chunks has priority when they conflict in a directory, there is no starvation. The policy of Section 3.6.4 does not cause starvation.
<table>
<thead>
<tr>
<th></th>
<th>Hardware Structures</th>
<th>Control Logic</th>
</tr>
</thead>
</table>
| **Base** (Inherited from BulkSC and Scalable-Bulk) | – R/W signatures  
  – Functional units to operate on signatures in cache controller and directory controller | – Cache controller: checkpoint and rollback, address disambiguation using signatures  
  – Directory controller: update directory state using signatures |
| **IntelliSquash** | – Per L1 cache line:  
  • A bit  
  • d bit per byte (or word) | – Cache controller:  
  • Merge transaction on a miss: incoming line does not overwrite words with d=1 (Sec. 3.5.2)  
  • Incoming invalidation: set A bit  
  – Directory controller:  
  • Merge transactions on chunk commit (Sec. 3.5.2): Send some lines from memory to the cache to merge |
| **IntelliCommit** | – Cache controller:  
  • # of commit_confirm & preempt_finished received  
  • Counter of # of chunks committed so far.  
  • Preemption Vector: N-1 counters | – Directory controller:  
  • Block commit protocol (Fig. 3.3) (also in cache controller)  
  • State of a committing chunk in a directory module (Fig. 3.4)  
  • Preemption state machine (Fig. 3.5) (also in cache controller) |
| Removed complexity from ScalableBulk | – Cache controller: Two active chunks per processor  
  • Additional set of W/R signatures | – Cache controller: Two active chunks per processor  
  • Rollback of partial state  
  • Stall on overlap of two active chunks (Set Restriction)  
  – Directory Controller:  
  • ScalableBulk commit protocol (Sec. 3.4.3): More transient states |

Table 3.2: Estimated design complexity of IntelliSense.
3.7 Analysis of Design Complexity

Table 3.2 estimates the design complexity of IntelliSense and also compares it to that of Scalable-Bulk. We divide complexity into hardware structures and control logic. IntelliSense is composed of a base design from BulkSC [15] and ScalableBulk [58] (Row 1), IntelliSquash (Row 2), and IntelliCommit (Row 3), minus some components from ScalableBulk (Row 4).

The top three rows are largely self-explanatory. The last row shows two components of ScalableBulk not present in IntelliSense: two active chunks per processor and the ScalableBulk chunk commit protocol. The former requires an additional set of R/W signatures. It also needs control logic to roll back only the state of one of the chunks, and to stall one of the active chunks if it wants to write to a cache set that contains a speculative line from the other active chunk. This is the Set Restriction [15], which requires that a cache set hold the speculative dirty data from only one chunk.

The second component is the ScalableBulk chunk commit protocol, which is more complex than the IntelliSense protocol: it has many more transient states. The reason is that it has a higher number of different types of messages received by a directory module before the commit succeeds or fails. In addition, the messages can be received in different orders. To get a qualitative insight, we compare the IntelliSense protocol in Figure 3.3 to the ScalableBulk protocol in Figure 3 of [58]. In the latter, the directory modules coordinate among themselves to form a group and to resolve collisions. As glimpsed from the figure and shown in Table 5 of [58], the messages exchanged are of different types and can come in a variety of orders, creating transient states. The IntelliSense protocol is simpler.


3.8 Evaluation

3.8.1 Evaluation Setup

We evaluate IntelliSense using simulations of a 16- or 64-core multicore using a network simulator with processor and cache model [18]. The cores issue and commit two instructions per cycle. Memory accesses can be overlapped with instruction execution through the use of a reorder buffer. Each core has a private L1 cache and one bank of the shared L2 cache. Caches are kept coherent using a directory-based scheme that implements the IntelliSense protocol. The cores are connected using an on-chip 2D mesh. A simple first-touch policy is used to map virtual pages to physical pages in the directory modules. Table 3.3 shows more details. We also implement several protocols proposed in previous work (Table 3.4).

<table>
<thead>
<tr>
<th>Processor &amp; Interconnect</th>
<th>Memory Subsystem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores: 16 or 64 in a multicore</td>
<td>Private write-back D-L1:</td>
</tr>
<tr>
<td>Signature:</td>
<td>Size/assoc/line:</td>
</tr>
<tr>
<td>Size: 2 Kbits</td>
<td>32KB/4-way/32B</td>
</tr>
<tr>
<td>Max active chunks per core:</td>
<td>MSHRs: 8 entries</td>
</tr>
<tr>
<td>1 or 2</td>
<td></td>
</tr>
<tr>
<td>Chunk size: 2000 instruc.</td>
<td>Shared write-back L2:</td>
</tr>
<tr>
<td>Interconnect: 2D mesh</td>
<td>Size/assoc/line of local bank:</td>
</tr>
<tr>
<td>Interconnect link latency: 7 cycles</td>
<td>256KB/8-way/32B</td>
</tr>
<tr>
<td>Coherence protocol: IntelliSense</td>
<td>Round trip local: 8 cycles</td>
</tr>
<tr>
<td>Memory roundtrip: 300 cycles</td>
<td>MSHRs: 64 entries</td>
</tr>
</tbody>
</table>

Table 3.3: Simulated system configuration.

We execute 11 SPLASH-2 and 7 PARSEC applications, running with 16 or 64 threads. Their input sets are shown in Table 3.5. The applications are run to completion.
<table>
<thead>
<tr>
<th>Name</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>IntelliSense</td>
</tr>
<tr>
<td>INT-IS</td>
<td>IntelliCommit only</td>
</tr>
<tr>
<td>ST</td>
<td>Scalable TCC [16]</td>
</tr>
<tr>
<td>SB</td>
<td>ScalableBulk [58]</td>
</tr>
</tbody>
</table>

Table 3.4: Simulated cache coherence protocols.

<table>
<thead>
<tr>
<th>App.</th>
<th>Input Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>-s -m16</td>
</tr>
<tr>
<td>radix</td>
<td>-n262144 -r1024 -m524288</td>
</tr>
<tr>
<td>cholesky</td>
<td>-s tk23.O</td>
</tr>
<tr>
<td>lu</td>
<td>-n512 -b16</td>
</tr>
<tr>
<td>barnes</td>
<td>16384 123 0.025 0.05 1.0 2.0 5.0 0.075 0.25</td>
</tr>
<tr>
<td>fmm</td>
<td>two cluster plummer 16384 1e-6 64 5 0.025 0.0 cost zones</td>
</tr>
<tr>
<td>ocean</td>
<td>-n256 -e1e-07 -r20000 -t28800</td>
</tr>
<tr>
<td>radiosity</td>
<td>-batch -room</td>
</tr>
<tr>
<td>raytrace</td>
<td>balls4</td>
</tr>
<tr>
<td>water-ns</td>
<td>1.5e-16 512 3 6 -1 3000 3 0 64 6.212752</td>
</tr>
<tr>
<td>water-sp</td>
<td>1.5e-16 512 3 6 -1 3000 3 0 64 6.212752</td>
</tr>
<tr>
<td>PARSEC</td>
<td>simmedium</td>
</tr>
</tbody>
</table>

Table 3.5: Applications and input sets.

Figure 3.6: SPLASH-2 execution time with 64 threads.
3.8.2 Performance for a Single Active Block

Figure 3.6 and Figure 3.7 show the execution time of the applications on INT, INT-IS, SB and ST for 64 processors, assuming one active chunk in each processor. We show results for two chunks in Section 3.8.3. The execution time is normalized to INT. Each bar is labeled with the name of the application and the protocol. The last four bars show the average. The bars are broken down into the following categories: cycles when the processor commits at least one normal instruction (Useful), cycles stalling for cache misses (CacheMiss), cycles wasted due to chunk squashes (Squash) and cycles stalling waiting for a chunk to commit (Commit).

For SPLASH-2 applications (Figure 3.6), on average, IntelliSense reduces the execution time by 27% and 19% compared to Scalable TCC and ScalableBulk, respectively. For almost all the applications, we observe very small commit overhead in IntelliSense. One exception is Radiosity, where a committing chunk often accesses many directories. A large directory group size may increase the chance of preemption, which may result in slower commit. For a large directory
group size, ScalableBulk shows substantial commit overhead, greater than Scalable TCC. It is because the Grouping stage of ScalableBulk is sequential and especially sensitive to a large group size. In general, we observe that ScalableBulk performs better than Scalable TCC, but IntelliSense eliminates most of the commit overhead.

For PARSEC applications (Figure 3.7), IntelliSense reduces the execution time by 56% and 16% compared to Scalable TCC and ScalableBulk, respectively. The commit overheads in PARSEC applications are larger than in SPLASH-2. For Scalable TCC, they are mainly due to high contention in the overlapped directory modules of two chunks, especially the overlapped write groups. In these cases, the commits need to be completely serialized. If the write groups overlap directories but the address sets do not overlap, ScalableBulk allows the two chunks to commit concurrently. Such effect is seen particularly in Blackscholes, Canneal and Swaptions. For all of the applications, IntelliSense reduces the commit overhead by more than half compared to ScalableBulk.

Finally, we consider IntelliSquash (difference between INT-IS and INT). While the majority of the applications have a modest amount of squash, IntelliSquash does eliminate substantial squashes for several applications, including Barnes, Ocean, Radix and Streamcluster. The large squash present in these applications with INT-IS is mainly due to false sharing (since we use line-based addresses). Barnes uses an array of locks to protect dynamic structures. Those locks are contiguously stored in memory. Several threads share the same cache line and when the threads release the critical section they all write to the lock variable in the same cache line. Ocean has a similar problem but only with two locks (psiailock and psibilock). Those locks share the same cache line and two threads may release the critical section at the same time. Streamcluster has a conditional variable waiting inside a critical section. Several threads may be waiting at the same time and when they are released, they all write at the same time to the lock variable that shares the cache line with the conditional variable. Radix implements a parallel radix sort algorithm that ranks integers and writes them into separate buckets for each digit. The writes to these buckets create false sharing.
Overall, IntelliSense reduces the execution times by 40% and 18% on average for the applications compared to Scalable TCC and ScalableBulk, respectively. In addition, IntelliSquash is effective at reducing the squashes due to false sharing for a number of applications.

### 3.8.3 Performance for Two Chunks

This section evaluates the performance when each processor has two chunks. Having two chunks hides commit latency, since while the first chunk is committing, the second chunk can continue executing. But, due to the set restriction [15], when the second chunk writes to a line in a cache set that contains a dirty line from the first chunk, the second chunk stalls.

In the one chunk scenario, the commit latency is fully exposed to the critical path of the execution. With two active chunks, the second chunk may stall due to two reasons. First, it may stall because, when it finishes, the first chunk is not committed yet (commitStall). Second, it may stall when it is about to write a line due to the Set Restriction (writeStall).

Figure 3.8 characterizes the stall reason for the second chunk. Each figure shows a distribution of stall cycles for different reasons, the curves are generated by the data points from all the applications. We consider the INT, SB and ST. We see that there is nearly no commitStall for each protocol, it means that in each protocol, before the active chunk is committed, the second chunk already stalls due to the write to a dirty set. Therefore, while active chunks can hide some commit latency, it is in fact not very effective. The latency shown in the curves will appear in the critical path of the execution. We do see that the curve for INT stay on the left, indicating that when the second chunk is stalled due to writeStall, it needs to wait for less time before resuming execution. We see that the curves of SB and ST are moved right, due to the longer commit latency.

On average, for two chunks, the writeStall for INT, SB and ST are 97, 176 and 245 cycles. For one chunk, the commitStall (no writeStall) for INT, SB and ST are 130, 406 and 2225 cycles. Figure 3.9 shows the performance comparison of one and two chunks over all applications in Splash-2 and PARSC. We see that two active chunks can reduce the commit overhead but there is still considerable commit latency exposed to the critical path.
Figure 3.8: Average Stall time for Commit and Write with 64 threads

Figure 3.9: Performance with one and two chunks
3.8.4 Scalability

Figure 3.10: Performance Scalability of IntelliSense

Figure 3.11: Network Traffic
To understand the scalability of IntelliSense, we consider three aspects: (a) performance; (b) commit latency; (c) network traffic.

Figure 3.10 shows the performance scalability of IntelliSense for SPLASH-2 and PARSEC applications. Each bar is composed of the same categories as in execution time. We see good scalability for all the applications. On average, from 16 to 64 processors, SPLASH-2 and PARSEC applications achieve 3.51 and 3.77 speedups.

Figure 3.11 shows the network traffic of different protocols. Each bar is divided into Normal, the messages due to normal execution, CommitLarge, the address sets sent in commit and CommitSmall, the small auxiliary messages in the commit (e.g. Grab, Skip, Preempt, etc.). We show the average for all the applications in SPLASH-2 and PARSEC for 16 and 64 processors. We see that Scalable TCC incurs more traffic due to commit and it increases drastically with the number of processors. It is mainly due to the small broadcast messages (e.g. Skip). IntelliSense and ScalableBulk generate very small amount of traffic in the commit.

### 3.8.5 Directory Group

![Figure 3.12: Group Size](image-url)
Figure 3.12 shows the directory group size of different applications. Each bar is divided into read and write groups. Most applications have group size less than 5. A few applications have large group size, in particular, Radiosity and Canneal. We show that IntelliSense performs well even for these applications.

3.9 Conclusion

Architectures that continuously execute chunks can improve performance and programmability. However, in a large manycore with directory-based coherence and lazy conflict detection, chunk commit is a major bottleneck.

This chapter has focused on providing scalable and fast chunk commit for these systems. It made three contributions. First, it presented a novel model of chunk commit in these machines. Second, it introduced two new, general techniques. One is the serialization of the write sets of WAW-dependent chunks, which eliminates chunk squashes due to WAWs. The other is the parallelization of how the committing chunk grabs directory ownership, which speeds-up the commit’s critical path. Our results with PARSEC and SPLASH-2 codes for 64 processors show that we eliminate most of the squash and commit stall times. Codes run an average of 40% and 18% faster than on previously-proposed schemes.
Chapter 4

BulkSMT: Designing SMT Processors for Chunk Execution

4.1 Introduction

There has been much interest recently in a class of shared-memory architectures where processors continuously execute chunks of instructions — often called chunks or transactions. Broadly speaking, these architectures include research proposals such as TCC [28], Bulk [72], Implicit Transactions [74], ASO [76], InvisiFence [6], DMP [20] and SRC [57] among others. Their chunked execution mode can improve performance and software productivity. For example, it supports transactional memory [28, 57], high performance under strict memory-consistency models [6, 72, 76], deterministic execution [20], parallel program replay [48], and atomicity-violation debugging [41].

All of the proposals for such architectures have implicitly used as their building blocks single-context cores — rather than Simultaneous Multithreading (SMT) cores [73]. This is unfortunate, given that SMT cores are widely deployed [30, 32] and would likely be used in a commercial implementation of these architectures. It is therefore important to understand how SMT cores would support chunked operation, both individually and integrated into a multicore of SMTs.

SMT processors are attractive for chunked execution for some of the same reasons as they are interesting for conventional execution. Specifically, they enable a better utilization of the hardware resources in a core. Moreover, they support fast communication between contexts, which improves performance and minimizes energy consumption. However, they are also attractive for chunked execution in their own right. Indeed, by minimizing the cost of interaction between the multiple contexts of the same core, they can enable more aggressive, higher-concurrency forms of chunked
On the other hand, the fact that SMT threads share caches and other hardware structures makes it intrinsically more difficult to support the execution of atomic, isolated chunks.

4.2 Contribution

Given this state of the art, this work contributes with the first SMT design that supports chunked (or transactional) execution of its contexts. We call it BulkSMT, and can be used either in a single-core processor or in a multicore of SMTs. In this work, we first perform an analysis of the design space, and propose three BulkSMT configurations with different cost and performance: squash on conflict, stall on conflict, and order on conflict. Then, we describe a set of novel architectural primitives that enable chunked execution in an SMT core. Finally, we show how to augment the resulting SMT core to work in a multicore of SMTs that executes chunks.

4.3 Background & Related Work

4.3.1 Continuous Execution of Chunks

In blocked (or chunked) execution, a core continuously executes chunks of instructions, also called transactions or chunks. There are several recent proposals of architectures that operate or can operate in this mode (e.g. [6, 20, 28, 57, 72, 74, 76]). These architectures have interesting capabilities related to performance and parallel software productivity.

In these systems, before a chunk starts, the processor hardware takes a register checkpoint. Then, as the chunk executes, the architecture records the addresses read and written by the chunk, and prevents the written data from being irreversibly merged with the memory system before the chunk is proven safe to commit. In most designs, no other processor is allowed to observe the intermediate state of the chunk as it executes. Consequently, the architecture watches for data conflicts (i.e., RAW, WAW, and WAR dependences) between concurrently-executing chunks. If a conflict
is found, typically one of the chunks is squashed and restarted. Squashing involves discarding the
data updated by the chunk in the cache or buffer, and restoring the register checkpoint.

Another reason for squashes is the overflow of the cache or buffer that keeps the updates of the
chunk (or a log of the values prior to such updates). When a chunk is squashed by an event that
re-appears on re-execution (e.g., cache overflow), execution transfers to a special version of the
code that guarantees forward progress.

A system with chunked execution needs to perform version management, conflict detection,
and conflict resolution. Each of these operations can be performed eagerly or lazily. Version
management deals with the storage of speculative and non-speculative data. The lazy policy buffers
the speculative data in special storage, separate from the shared memory, until the chunk commits;
the eager one stores the speculative data in place in the shared memory, and saves the prior values
in a special buffer or log. Conflict detection refers to the detection of inter-thread conflicts. The
eager policy detects them as soon as a chunk tries to perform the conflicting memory accesses; the
lazy policy checks for conflicts when a chunk is ready to commit. Finally, conflict resolution refers
to the action taken to deal with the conflict. The eager policy takes the action as soon as the eager
conflict detector has detected the conflict; the lazy one takes the action when a chunk is ready to
commit.

4.3.2 An Opportunity for Chunked Execution

Chunked-execution multiprocessors can attain higher performance if they can withstand data con-
flicts between concurrently-executing chunks without squashing. Recently, there have been sev-
eral proposals for such systems. For example, some proposals use the Conflict Serialization model
from database systems [5] [61]. The idea is to record and manage the conflicts between chunks as
they execute, and then ensure that the chunks commit in the correct order. In practice, supporting
this additional level of concurrency has resulted in a complicated cache coherence protocol as in
DATM [61] or in non-trivial bookkeeping requirements to ensure correct ordering as in SONTM [5].

Other proposals involve a “state correcting” step. Specifically, in RetCon [7], execution pro-
ceeds after a data conflict occurs. However, when the chunk is ready to commit, RetCon attempts to fix its state by obtaining the current value of the variables that were involved in the conflict. Alternately, in transaction value prediction [56], a chunk uses a value that it predicts it will need in a future conflict.

In general, many of these schemes involve significant conceptual and hardware complexity — much of it resulting from the distributed nature of the multiprocessor hardware involved. In contrast, an SMT processor contains multiple hardware contexts that are in close proximity and share hardware structures such as caches and buffers. If such contexts run chunks that conflict with each other, the hardware can efficiently and quickly detect the conflicts, record them, and manage them, in order to attain some concurrency between the chunks.

Remarkably, none of the proposals for chunked-execution architectures has used SMT cores as its building block. There is, therefore, an opportunity to leverage SMT to support higher levels of chunk concurrency with simpler hardware than in the past. Uncovering such opportunity is the goal of this work.

4.3.3 Other Related Work

Beyond the chunked-execution architectures discussed, the most relevant work includes techniques that try to increase the concurrency of conflicting transactions in hardware transactional memory systems. There are two proposals, DATM [61] and SONTM [5], which apply to multicore systems with single-context processors.

DATM manages the dependences between uncommitted transactions, sometimes forwarding data between them to be able to safely commit conflicting transactions. It uses a bus-based shared-memory machine and proposes the FRMSI snoopy-based cache coherence protocol. This is a new protocol with 11 stable states. Such protocol supports the forwarding of lines between caches like an update-based protocol. It also needs to select the correct version of a datum among the several that exist in the different caches of the machine. It has per-word access bits to support the ability to merge cache lines that have been partially updated by different processors. Finally, to keep the
order of transactions, it has an order vector of transactions stored in each cache.

The SONTM system maintains an upper bound and a lower bound Serializability Order Number (SON) for each transaction. They are updated when a transaction performs a memory operation and when a transaction commits. During a transaction’s execution, when the upper bound is smaller than the lower bound, the transaction is aborted because it cannot be serialized with other dependent transactions. While SONTM does not modify the cache coherence protocol, it adds substantial overhead. Specifically, each memory location accessed has a read-number and a write-number stored in memory. While some optimizations are possible, each load and store instruction needs to get the read-number or write-number to potentially update the upper and lower bounds. Moreover, a validation step at a transaction commit involves broadcasting write-numbers of all the updated data and receiving read-number responses from other processors.

Overall, compared to these schemes, we focus on optimizing dependent chunks executing on the same core, rather than across cores. Hence, our hardware is substantially simpler and has much less overhead.

4.4 Chunked-Execution SMT Processors

Given an off-the-shelf SMT processor with L1 and L2 caches, we want to extend it to support chunked execution. In this section, we examine the design space and the basic hardware mechanisms required. The processor can be part of a multicore, although we ignore multicore effects until later sections.

4.4.1 Design Space

The extensions needed in the SMT processor to support chunked execution depend on our choices for the operations of Section 4.3.1. The preferred design points are shown in Table 4.1. For version management, it is simpler for the hardware to adopt an eager policy. This means allowing a speculative write from one context to update the cache and be immediately visible to the other
contexts of the SMT processor. There is no need to save the old value of the written variable in a log, as long as speculative data is prevented from spilling from L2; we can always get the old value of the data from main memory. If L2 is about to overflow, the chunk is squashed. The alternative, lazy version management, would require separately buffering the state that each context is generating. For conflict detection, it is natural to do it eagerly between contexts, as soon as a conflicting access executes. Similarly, for conflict resolution, it is simpler for the hardware to do it eagerly, rather than keeping state and resolving the conflict at commit time.

<table>
<thead>
<tr>
<th>Version Management</th>
<th>Eager, but without updating main memory with speculative data. No log is needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conflict Detection</td>
<td>Eager</td>
</tr>
<tr>
<td>Conflict Resolution</td>
<td>Eager squash, eager stall, or eager order</td>
</tr>
</tbody>
</table>

Table 4.1: Preferred design points.

We propose three distinct eager conflict resolution schemes, as shown in Figure 4.1. They generate three very different BulkSMT design points. Consider a data dependence between two concurrent chunks as in Figure 4.1(a). In **SQUASH**, the hardware squashes and restarts one of the conflicting chunks (Figure 4.1(b)). In **STALL**, the hardware stalls the consumer chunk until the producer commits (Figure 4.1(c)). However, if the stall induces a cycle between two or more stalled chunks, the consumer is squashed. In **ORDER**, the hardware records the order of the two chunks, lets them continue and enforces the order at commit (Figure 4.1(d)). However, if the conflict forms a cycle between two or more ordered chunks, then one or more chunks are squashed. The three schemes are summarized in Table 4.2.
(1) Record the order of two chunks;
(2) Stall the consumer chunk;
Eager Stall
Eager Squash
Eager Order
Squash one of the conflicting chunks

(a)
(b) SQUASH
(c) STALL
(d) ORDER

Figure 4.1: Conflict resolution schemes.

<table>
<thead>
<tr>
<th>Design Point</th>
<th>Conflict Policy</th>
<th>Resol.</th>
<th>Action on a Conflict Between Chunks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SQUASH</strong></td>
<td>Eager Squash</td>
<td></td>
<td>Squash one of the conflicting chunks</td>
</tr>
<tr>
<td><strong>STALL</strong></td>
<td>Eager Stall</td>
<td></td>
<td>(1) Stall the consumer chunk;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(2) if there is a cycle between two</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or more stalled chunks, squash the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>consumer chunk</td>
</tr>
<tr>
<td><strong>ORDER</strong></td>
<td>Eager Order</td>
<td></td>
<td>(1) Record the order of two chunks;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(2) if there is a cycle between two</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or more ordered chunks, squash one</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or more chunks;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(3) enforce the order at chunk commit</td>
</tr>
</tbody>
</table>

Table 4.2: Conflict resolution design points.

The main tradeoff is one of performance versus hardware cost. As we go from **SQUASH** to **STALL** and **ORDER**, we enable more concurrency and, therefore, higher performance — as seen in Figure 4.1. However, the hardware is more costly and we need to keep more state. Specifically, **STALL** needs to record if a thread is stalled and, if so, which other thread stalled it, and detect cycles of stalled threads. **ORDER** needs to record if threads are currently ordered and, if so, by what type of dependence. In addition, it needs to enforce the commit ordering, and also detect cycles of dependent threads. Fortunately, thanks to the tight coupling of the contexts in an SMT,
the hardware needed is simple and highly localized.

Next, we describe the three schemes. When we refer to a data dependence, we include RAW, WAR, and WAW, and they are at memory line granularity. The thread at the receiving end of the dependence is called consumer.

**SQUASH Design**

On a conflict, the chunk from one of the conflicting threads is squashed. Then, all of the processor resources used by the chunk (e.g., ROB entries, registers, and load/store queue entries) are released, and the cache lines written by the chunk are discarded (Section 4.4.2). Finally, the chunk restarts. We use the policy of oldest transaction wins, which helps make forward progress.

**STALL Design**

When a conflict is detected, the consumer chunk stalls before the actual consumer memory access is performed. The hardware records which chunk is stalled and which chunk is the producer one. When the producer chunk commits, the hardware resumes the consumer chunk, starting from the consumer memory access.

A chunk may stall on an already-stalled chunk. This is acceptable as long as the stalled chunks do not form a cycle. Consider Figure 4.2(a). In the figure, thread $T_0$ is about to write $x$ and stalls on $T_1$. Then, $T_2$ is about to write $y$ and stalls on $T_0$. This is fine because there is no cycle. Eventually, $T_1$ will commit and then $T_0$ will resume. When $T_0$ commits, $T_2$ resumes.

Figure 4.2: Examples of stalls.
When the hardware detects a cycle, it squashes the chunk that closes the cycle and resumes the chunks that were stalled on it. For example, in Figure 4.2(b), T0 is stalled on T1. Then, T1 attempts to write y, which would stall it on T0, creating a cycle. Consequently, T1 gets squashed and T0 resumes. A cycle can involve more than two chunks.

When BulkSMT stalls a chunk, the processor pipeline completes all the instructions in the chunk that are before the stalling one in program order. All the instructions that are after it must be flushed from the pipeline. Keeping them in the pipeline would lock up entries in resources that are shared by all of the contexts, such as the instruction queue. The result could be deadlock, as other contexts could fail to make progress. When the stalled chunk resumes, all of these instructions are reloaded again into the pipeline.

**ORDER Design**

When a conflict is detected, the hardware records the type and direction of the dependence. The chunks involved are allowed to proceed, but the hardware will enforce that they commit in the same order. The hardware also watches for a dependence that creates a cycle of ordered chunks (with two or more threads). If this happens, the hardware breaks the cycle by squashing and restarting one or more chunks — which may not include the one with the reference that closed the cycle.

To understand which chunks should get squashed in a cycle, consider the type of dependence. Figure 4.3 shows a RAW, WAW and WAR dependence and the squash rules that are easiest to support in hardware. Recall that a chunk squash also invalidates the cache lines updated by the chunk. In a RAW, one chunk wrote to the cache and a second one read. If we choose to squash the producer chunk, then we also have to squash the consumer. However, we can squash the consumer and not the producer. In a WAW, since dependences are at line granularity, if we choose to squash one of the chunks, we also have to squash the other. In a WAR, the squash of either one of the chunks does not cause the squash of the other. Section 4.4.2 uses these rules to decide which chunks to squash when a cycle is detected.
Figure 4.3: How squashes are affected by the type of dependence.

### 4.4.2 Basic Hardware Mechanisms

The basic mechanisms for BulkSMT operation are shown in Table 4.3. For each mechanism, the table shows its function, its implementation, and the designs it applies to. We consider each in turn.

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Function</th>
<th>Implementation</th>
<th>Designs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Recording and Conflict Detection</td>
<td>Record the addresses accessed by each chunk and detect when two chunks have a data conflict</td>
<td>Access Bits in cache and related logic</td>
<td>All (More in ORDER)</td>
</tr>
<tr>
<td>Cycle Detection</td>
<td>Record data conflicts and their ordering, and detect conflict cycles</td>
<td>Dependence Table and Cycle Table</td>
<td>STALL and ORDER</td>
</tr>
<tr>
<td>Advanced Conflict Recording</td>
<td>Represent the type of conflict between different chunks compactly</td>
<td>Enhanced Dependence Table</td>
<td>ORDER</td>
</tr>
<tr>
<td>Squash Set Generation</td>
<td>On a cycle of chunks with conflicts, decide the set of chunks to squash</td>
<td>Logic that operates on the Dependence Table</td>
<td>ORDER</td>
</tr>
</tbody>
</table>

Table 4.3: Basic mechanisms to support chunked execution in an SMT processor.
Access Recording and Conflict Detection

Past work has used Bloom-filter based hardware signatures to detect conflicts between chunks or transactions executing on different cores (e.g., [14]). In BulkSMT, since the chunks are executing all in the same core and share the (multi-level) cache, it is easier to record the accesses with cache bits and use simple logic to detect conflicts. Hence, we augment each cache line with a Last Writer (LW) context-ID, a read bit-mask with as many bits as contexts ($R[i]$), and a speculative bit ($Sp$). In an SMT with 4 contexts, this represents 7 bits per cache line (Figure 4.4(a)). When thread $k$ reads from the cache, it sets $R[k]$; when it writes to the cache, it sets $Sp$ and writes its context ID to $LW$. These bits are in the L1 and L2 caches and write buffers.

![Diagram](a) Access Bits
![Diagram](b) Enhanced Dependence Table

Figure 4.4: Two mechanisms for BulkSMT operation.

With this support, conflicts are detected as follows. When a load accesses a cache line, if $Sp$ is set and $LW$ is not the requester’s ID, a RAW conflict is declared. When a store accesses a cache line, if the $R[i]$ for any other context is set, a WAR is declared. Moreover, if $Sp$ is set and $LW$ is not the requester’s ID, a WAW is (also) declared. In addition, when a chunk commits, the hardware performs two multi-cycle operations: (1) a flash clear of the $R[i]$ bit corresponding to the chunk’s context for all the cache lines, and (2) a flash conditional clear of the $Sp$ bit of any cache line whose $LW$ is equal to the committing context ID. Finally, when a chunk is squashed, the hardware performs two operations: (1) a flash clear of the $R[i]$ bit corresponding to the chunk’s context for
all the cache lines, and (2) a flash conditional clear of the valid and $Sp$ bits of any cache line whose $Sp$ bit is set and the $LW$ is equal to the squashed context ID. An example of SRAM cells augmented with similar support is shown in [6].

In a WAR dependence in ORDER, we may want to squash the writer chunk after it has written and not squash the reader chunk (Figure [4.3]). In this case, we need to make sure that, after the squash of the writer (and invalidation of the lines it updated), we do not lose the record of any prior reader to those lines. To support this performance optimization, we add one additional bit per line called $Read\_But\_Missing$ (RM). When a chunk is squashed, as the hardware invalidates a line, it checks the line’s $R[i]$. If $R[i]$ has a set bit for a thread $Ti$ that is not being squashed, such bit is left unmodified and RM gets set. RM indicates that $R[i]$ is up-to-date but the data is invalid. A future access to the line brings the line from memory, while clearing RM but not $R[i]$ and, if applicable, recording a data conflict with $Ti$.

**Cycle Detection**

In both Stall and Order, we need a structure to record inter-thread data conflicts and their order — so that we stall the consumer thread in Stall and order the commit of producer and consumer in Order. Such structure also needs to detect conflict cycles. BulkSMT uses two low-cost hardware structures: the Dependence Table (DT) to record conflicts and the Cycle Table (CT) to detect conflict cycles. They are two-dimensional arrays, with as many rows and as many columns as hardware contexts in the processor. In the baseline design, each entry has one bit.

Figure 4.5(a) explains how they work. If there is a conflict where the chunk in thread $T_i$ is the producer and the one in $T_j$ is the consumer (represented as $T_i \rightarrow T_j$), the BulkSMT hardware sets the bits DT[$i$][$j$] and CT[$i$][$j$]. Every time that a new conflict is detected, the DT sets the corresponding bit. As the processor continues, in the background, the CT attempts to find if the new dependence has created a cycle. The CT does it by setting: (i) the bit corresponding to the latest conflict and (2) the bits corresponding to dependences transitively implied by all the recorded conflicts. A cycle is detected if a bit is set in the diagonal of the CT — i.e., a dependence has the
same producer and consumer thread.

Figure 4.5: Operation of the Dependence Table and Cycle Table.

As an example, consider Figure 4.5(b). As conflict $d1$ occurs, DT[0][1] and CT[0][1] get set. Later, as conflict $d2$ occurs, DT[1][2] and CT[1][2] get set, and CT tries to find transitive dependences. This is done by taking the newest dependence ($d2$) and examining, in turn, its source and its destination, checking for other arrows connected there. Starting at the source ($T_1$), we consider all the arrows that point to it. In our example, the only one is $d1$. For this arrow, the transitive dependence is shown as $dA$. Specifically, any arrow whose destination is $T_1$ (i.e., $dI$) creates a new one (i.e., $dA$), whose source is unchanged and whose destination is the destination of the newest dependence (i.e., $d2$). In hardware terms, CT takes the column corresponding to $T_1$’s ID (i.e., second column) and bit-ORs it into the column corresponding to the destination of the newest dependence $d2$ (i.e., third column). This is shown in Figure 4.5(b).

The next step is to consider the arrows that start at the destination of the newest dependence and create transitive dependences. Our example does not have any. If it had (call it dependence
we would create an arrow from the source of \( d_2 \) to the destination of \( d_3 \). Specifically, any arrow whose source is \( T_2 \) creates a new arrow whose destination is unchanged and whose source is the source of the newest dependence. In hardware, CT would take the row corresponding to \( T_2 \)’s ID (i.e., third row) and bit-OR it into the row corresponding to the source of dependence \( d_2 \) (i.e., second row).

The process described proceeds recursively: every time that a new transitive dependence is found in the CT, the algorithm proceeds to analyze its source and destination as described above to find new dependences. The process terminates when the CT no longer changes. Since an SMT processor has few contexts, CT is small (e.g., 4x4), and very few steps are typically needed.

Figure 4.5(c) shows an example of a cycle with two threads. On the left, we show the dependences \( d_1 \) and \( d_2 \) and, on the right, the evolution of the CT. When \( d_1 \) is flagged, bit CT[0][1] is set. When \( d_2 \) is flagged, bit CT[1][0] is set and the algorithm proceeds by ORing the second column into the first one. The set bit CT[0][0] flags the cycle, which corresponds to arrow \( d_A \). While the algorithm stops as soon as it finds a cycle, for completion, we note that there is another cycle. It is an arrow not shown in the figure that goes from \( T_1 \) to \( T_1 \). It is obtained by processing the destination of \( d_2 \). It appears as we bit-OR the first row into the second row and bit CT[1][1] gets set.

Figure 4.5(d) shows a three-thread cycle. Dependence \( d_1 \) sets CT[0][1]; dependence \( d_2 \) sets CT[1][2] and uncovers \( d_A \), setting CT[0][2]; finally, dependence \( d_3 \) sets CT[2][0] and uncovers \( d_B \) and \( d_C \), setting CT[1][0] and CT[0][0] — hence flagging a cycle.

**Additional Issues Related to the DT and CT**

The CT is not a time-critical structure. While the DT must be updated as soon as the dependence occurs, the CT can buffer its inputs and only later get updated and run the cycle detection algorithm. It is always correct to find a few cycles later that a cycle occurred. At that point, the Squash Set Generation algorithm (Section 4.4.2) will be run based on the up-to-date state of the DT.

The DT and CT are also updated when a chunk commits or gets squashed. Consider first that
the chunk in thread $T_i$ is ready to commit. In **ORDER**, BulkSMT first checks if it can commit. If any bit in column $i$ of the DT is set, the thread has to stall — and post that it is stalled. In all other cases of **ORDER** and **STALL**, the chunk commits and then BulkSMT runs the algorithm of Figure 4.6. Similarly, after the chunk in thread $T_i$ is squashed, BulkSMT runs the algorithm of Figure 4.6.

<table>
<thead>
<tr>
<th>1) Wake up any chunks that are stalled and can now proceed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1) Consider row $i$ in the DT. Find all columns where their only set bit is in row $i$</td>
</tr>
<tr>
<td>1.2) For each column $j$ in this set</td>
</tr>
<tr>
<td>1.2.1) If in <strong>STALL</strong>: wake up thread $T_j$</td>
</tr>
<tr>
<td>1.2.2) If in <strong>ORDER</strong>: if thread $T_j$ is stalled, wake up thread $T_j$</td>
</tr>
<tr>
<td>2) Clear row DT[$i$][.] and column DT[.][$i$]. Clear the CT</td>
</tr>
<tr>
<td>3) Copy the DT to the CT. Regenerate all the transitive dependences in CT</td>
</tr>
</tbody>
</table>

Figure 4.6: Actions at the commit/squash of thread $T_i$’s chunk.

**Advanced Conflict Recording**

In **STALL**, each DT entry only needs to record if there is a dependence or not. Therefore, one bit per entry suffices. In **ORDER**, each DT entry also needs to record what type(s) of dependence there are between the two chunks — RAW, WAW, or WAR. This information is needed in case of a cycle, to decide what chunks to squash (Section 4.4.2). Recall that the dependence type impacts which chunk to squash (Figure 4.3).

Two chunks may have multiple dependence types (on the same or different variables). Hence, in **ORDER**, the DT has three bits per entry, one per each type of dependence (Figure 4.4(b)). Note that the CT is unaffected, and it still has one bit per entry. The bit in the CT entry is set if any of the three bits in the DT entry is set.
**Squash Set Generation**

In *ORDER*, when the CT detects a cycle, the hardware stalls the processor and uses the DT (which is consistent with the current speculative memory state) to decide which chunks to squash to break the cycle. The algorithm used to select such chunks is called *Squash Set Generator (SSG)*. It reads the bits currently in the DT and applies the rules of Figure 4.3 for RAW, WAW, and WAR dependences.

The Baseline SSG algorithm starts by putting in the set of chunks to squash (the squash set) the chunk that closed the cycle. Then, it follows forward dependences from that chunk, using the rules in Figure 4.3 to put additional chunks in the squash set. The squash propagation stops when forward dependences either bring us to chunks already in the set or they do not propagate squashes because they are WAR dependences. Then, SSG goes back to the original chunk and follows backward dependences from there, again using the rules. The backward propagation stops when we reach chunks already in the set or the dependences do not propagate the squash because they are RAW or WAR dependences. The algorithm is recursive.

Figure 4.7 shows an example of a cycle with three chunks. The read in thread *T0*’s chunk closes a cycle. Consequently, *T0* is put in the squash set. From *T0*, SSG then follows the RAW to thread *T1*, which is also put in the set. The next forward dependence is a WAR to *T2*, which stops the propagation. Then, SSG goes to *T0* and propagates backward. Since we find a RAW to *T2*, back-propagation stops. Consequently, only *T0* and *T1* get squashed.

![Figure 4.7: Example of breaking the cycle of ordered chunks.](image-url)
In reality, a given cycle can be broken in multiple ways, possibly resulting in different numbers of squashed chunks. Consequently, the Advanced SSG algorithm does not simply squash the chunks found in the squash set as described after the first try. Instead, it then picks each chunk in the set in turn and re-runs the algorithm starting from that chunk. These new runs may result in fewer chunks to squash. For example, in Figure 4.7 if SSG starts from \( T1 \), it finds that it only needs to squash \( T1 \) to break the cycle. Consequently, the Advanced SSG algorithm breaks the cycle in the way that minimizes the number of squashed chunks.

Since over 95% of the cycles that we found only involve two chunks, the Advanced SSG adds very little overhead. Moreover, Section 4.6 shows that the Advanced SSG helps ORDER handle high-contention locks.

After BulkSMT has squashed the chunks to break the cycle, it uses the resulting DT to regenerate the CT. If the CT finds that there is still a cycle, the whole process is repeated. The CT may still find a cycle if the last dependence recorded ended up creating two cycles, and with the use of the Advanced SSG algorithm, we broke only one. Overall, we use the Advanced SSG algorithm. Its hardware cost is modest, since it only accesses the DT and the number of contexts in an SMT is fairly small. Moreover, it only runs in the relatively rare case of a cycle.

### 4.5 Chunked-Execution Multicores of SMTs

SMT cores with chunked-execution support should be amenable to integration into multicores and multi-socket systems. We now examine the additional microarchitecture needed to use BulkSMT as a building block for a chunked-execution multicore. In our discussion, we refer to the hardware actions across SMT cores as *global*, while those across the contexts of an SMT core as *local*.

There has been much research on designing multiprocessor hardware that supports chunks or transactions using single-context (non-SMT) cores (e.g., [15, 16, 28, 49, 57, 58]). To cover a broad design space, we consider two global designs. The first one (EE) uses eager version management and eager conflict resolution, and is like LogTM [49]. The other (LL) uses lazy
version management and lazy conflict resolution, and is like BulkSC [15]. Next, we outline the relevant parts of the two global protocols and then describe the integration with the local protocol.

### 4.5.1 Global Protocols Examined

The EE scheme uses the Access Bits in the caches to flag conflicts between threads running on different cores. As in LogTM, we need to augment each core with hardware-based undo logs that save the old values when a speculative thread writes a variable. Since we use SMT cores, a core has as many undo logs as hardware contexts.

The LL scheme could also use the Access Bits in the caches to flag inter-core conflicts like TCC [28]. However, since we model BulkSC [15], we use hardware-based address signatures to detect conflicts. Hence, in an SMT core, each context has a R an a W signature. When chunks commit, they send out their signatures, which are intersected with those in the receiving cores. To detect conflicts between the contexts of a core, we still use the Access Bits in the caches.

### 4.5.2 Integrating the Local and Global Protocols

Table 4.4 lists our two rules for a design that integrates local and global chunk-based protocols. The first rule applies when a chunk wants to commit: it should first initiate a commit globally (across cores) and, when it succeeds, commit locally among the threads in the SMT. In the EE scheme, this implies waiting for the completion of all the buffered previous memory accesses by the chunk, and then performing the local commit in the SMT core. In the LL scheme, it implies sending the chunk’s signature out to the global network, waiting for the global commit confirmation, and then performing the local commit in the SMT core.

In the LL scheme, since commit is costly, we augment ORDER with Commit Combining. This event occurs when a consumer chunk completes execution before its producer chunk in the same SMT core does. The consumer has to wait to commit until after the producer commits. With commit combining, when the producer completes, both chunks perform the commit together —
Table 4.4: Rules for integrating local and global chunk-based protocols.

<table>
<thead>
<tr>
<th>Event</th>
<th>Actions</th>
</tr>
</thead>
</table>
| Chunk wants to commit | Initiate a global commit; when it succeeds, commit locally  
| | EE: Wait for the completion of all the buffered previous memory accesses, then perform local commit  
| | LL: Send signature out, wait for the global confirmation, then perform local commit. For performance, perform commit combining under ORDER |
| Reception of a coherence event that may cause a squash | Squash any local chunk that needs to be squashed (even the stalled ones)  
| | EE: Use the address of the coherence message to index the cache and read the Access Bits; if a conflict is detected, squash the corresponding local chunk(s)  
| | LL: Intersect the incoming signature with local signatures; if intersection is not null, squash the corresponding local chunk(s)  
| | Propagate the squash inside the SMT processor |

The second rule applies when a core receives a coherence event that may cause a squash. The core must check against all of the local chunks — even the stalled ones. In the EE scheme, this means using the address of the coherence message (e.g., invalidation) to index the cache and check the Access Bits; if a conflict is found, we squash the corresponding chunk(s). In the LL scheme, the hardware intersects the incoming signature with all the local signatures; if an intersection is not null, we squash the corresponding chunk(s). Moreover, in both EE and LL, the squash needs to be propagated. Specifically, in ORDER, we run the SSG algorithm (Section 4.4.2) to detect other chunks to squash; in STALL, we wake up all the chunks that are stalled waiting only on the squashed chunks.

4.6 Implementation Issues

4.6.1 Cycle Detection Algorithm Implementation

To complement the description of the cycle detection algorithm, we outline its hardware implementation in a Cycle Table Module (Figure 4.8). The module contains the Cycle Table (CT), combinational logic to perform the steps of the cycle detection algorithm (Dependence Generator)
and detect a cycle (Cycle Checker), and the Shadow Cycle Table (SCT). The latter is a table like the CT that contains temporary state as the algorithm runs.

In idle state, CT holds a certain bit pattern and SCT is clear. When the program generates a new dependence, it is encoded as a \((\text{src}, \text{dst})\) code and goes through the Dependence Generator. The latter finds if the dependence itself sets a new bit in CT. If so, the new bit is set in both CT and SCT, and the multi-step process of finding transitive dependences starts. Such process involves the SCT feeding the new bit to the Dependence Generator in two steps: (1) first to combine with existing dependences at the source of the new dependence (which triggers the Dependence Generator to bit-OR two CT columns as in Figure 4.5(b)) and (2) then to combine with dependences at the destination of the new dependence (which triggers the Dependence Generator to bit-OR two CT rows). After these two steps, the bit is cleared from the SCT.

If, in any of these two steps, a bit that was not set in CT gets set, we have found a new dependence by transitivity. Such bit is set in both CT and SCT, and the SCT will process the new bit once it is done with the first one. The process continues until no new dependence is found and SCT becomes clear. At all times, the Cycle Checker uses simple logic to check if a bit gets set in CT’s diagonal. If one gets set, a cycle is flagged and the whole process stops.

Figure 4.8: Cycle Table Module.
If, during this process, the program generates a new dependence, the dependence is buffered. Completing the current process of uncovering all transitive dependences has higher priority. The new dependence will be processed immediately after.

Based on this description, Figure 4.9 lists the hardware cost of the Cycle Table (CT) Module, and of all the other hardware structures required by BulkSMT. We assume a BulkSMT processor with \( n \) contexts. The CT Module only needs two arrays of \( n^2 \) bits and the combinational logic for the Dependence Generator (DG) and Cycle Checker (CC). The Enhanced Dependence Table (DT) needs an array of \( 3 \times n^2 \) bits. Finally, the Access Bits need \( n + \log_2 n + 1 \) bits per cache line. Overall, the hardware requirements of BulkSMT are very modest.

<table>
<thead>
<tr>
<th>Structure Name</th>
<th>Cost</th>
<th>Hardware Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT Module</td>
<td>( n^2 )</td>
<td>Array of bits</td>
</tr>
<tr>
<td>SCT</td>
<td>( n^2 )</td>
<td>Array of bits</td>
</tr>
<tr>
<td>CC + DG</td>
<td>Combinat. logic</td>
<td>—</td>
</tr>
<tr>
<td>Enhanced DT</td>
<td>( 3 \times n^2 )</td>
<td>Array of bits</td>
</tr>
<tr>
<td>Access Bits per cache line</td>
<td>( n + \log_2 n + 1 )</td>
<td>Extension to tags</td>
</tr>
</tbody>
</table>

Figure 4.9: Hardware requirements of the BulkSMT mechanisms.

### 4.6.2 Cache Conflicts

While our discussion has focused on chunk squashes due to dependences, chunks may also get squashed on cache overflow. Specifically, when a cache conflict displaces a line with non-null Access Bits from the lowest cache level, the chunks that accessed the line get squashed. In addition, in *ORDER*, the hardware follows the rules of Figure 4.3 to find dependent chunks to squash. Finally, after the squashes, in both *STALL* and *ORDER*, chunks that were stalled on the squashed ones are released. Given the potential cost of these actions, it may be beneficial to tune the cache replacement algorithms to avoid these cases. In this work, we have not done so.
4.6.3 Handling High-Contention Synchronizations

High-contention synchronizations are a concern for chunked-execution architectures because they introduce frequent dependences between chunks. Such dependences cause squashes or stalls. One way to minimize their impact is to explicitly terminate the chunk with a software command after or before a high-contention synchronization. The shortcoming of this approach is that it needs either a profiling pass to identify high-contention synchronizations or support to learn the frequent dependences dynamically.

In this work, we do not use a profiling pass or hardware to learn the frequent dependences dynamically. Hence, we do not terminate chunks in software at high-contention synchronizations. The one exception is at barriers: since it is clear that chunks conflict at barriers, we place chunk termination commands inside the barrier library call or macro. By terminating the chunk, we ensure the work before the barrier is not squashed due to a conflict in the barrier.

For an interesting illustration of how chunked-execution works, consider a high-contention lock under ORDER. BulkSMT uses Test&Test&Set for the lock. In Figure 4.10(a), the chunk in thread T0 grabs the lock. Then, T1 spins on it, creating a RAW dependence. When T0 releases it, it creates a cycle. The Advanced SSG algorithm (Section 4.4.2) squashes the minimum number of chunks to break the cycle, namely just the one in T1. As the chunk restarts (Figure 4.10(b)), an ordered dependence is created, which still allows both chunks to eventually commit successfully.
Table 4.4 lists our two rules for a design that integrates local and global chunk-based protocols. The first rule applies when a chunk wants to commit: it should first initiate a commit globally (across cores) and, when it succeeds, commit locally among the threads in the SMT. In the EE scheme, this implies waiting for the completion of all the buffered previous memory accesses by the chunk, and then performing the local commit in the SMT core. In the LL scheme, it implies sending the chunk’s signature out to the global network, waiting for the global commit confirmation, and then performing the local commit in the SMT core.

In the LL scheme, since commit is costly, we augment ORDER with Commit Combining. This event occurs when a consumer chunk completes execution before its producer chunk in the same SMT core does. The consumer has to wait to commit until after the producer commits. With commit combining, when the producer completes, both chunks perform the commit together — i.e., they send a combined signature out and then commit locally together.

The second rule applies when a core receives a coherence event that may cause a squash. The core must check against all of the local chunks — even the stalled ones. In the EE scheme, this means using the address of the coherence message (e.g., invalidation) to index the cache and check...
the Access Bits; if a conflict is found, we squash the corresponding chunk(s). In the LL scheme, the hardware intersects the incoming signature with all the local signatures; if an intersection is not null, we squash the corresponding chunk(s). Moreover, in both EE and LL, the squash needs to be propagated. Specifically, in ORDER, we run the SSG algorithm (Section 4.4.2) to detect other chunks to squash; in STALL, we wake up all the chunks that are stalled waiting only on the squashed chunks.

4.7 Results

<table>
<thead>
<tr>
<th>Configurations Used: XX-YY, where:</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX: Type of core:</td>
</tr>
<tr>
<td>SQ: 4-context SQUASH BulkSMT design</td>
</tr>
<tr>
<td>ST: 4-context STALL BulkSMT design</td>
</tr>
<tr>
<td>OR: 4-context ORDER BulkSMT design</td>
</tr>
<tr>
<td>BK: Single-context core with chunked-execution support</td>
</tr>
<tr>
<td>YY: Type of global (inter-core) protocol:</td>
</tr>
<tr>
<td>EE: Eager version management and eager conflict resolution</td>
</tr>
<tr>
<td>LL: Lazy version management and lazy conflict resolution</td>
</tr>
</tbody>
</table>

Table 4.5: Names of configurations used.

Figure 4.11: Configurations used.

98
In our evaluation, we model a 4-context BulkSMT core alone or in a 4-core multicore chip. We model all combinations of SQUASH, STALL, and ORDER with the EE and LL global protocols. In addition, we compare the performance to machines built out of single-context cores with chunked-execution support called BK. The configurations used are shown in Figure 4.5 and Figure 4.11.

We evaluate these designs using a cycle-accurate execution-driven simulator based on SESC [63] with detailed models for the processor, the memory subsystem and the interconnect. The architectural parameters are shown in Table 4.6. The BulkSMT and BK cores have the same issue width and hardware structure sizes. However, in BulkSMT, the ROB and load-store queue are partitioned equally among the 4 contexts. Each core has private L1 and L2 caches. The global protocol is similar to BulkSC’s [15] in LL and LogTM’s [49] in EE.

We use the applications from SPLASH-2 and PARSEC that have a noticeable degree of interactions between threads. From SPLASH-2, the applications and inputs we use are: Barnes (16k particles), Cholesky (tk29.0), Ocean (258x258 ocean), Radiosity (room), Radix (256K keys) and Raytrace (car). From PARSEC, they are: fluidanimate (simmedium) and streamcluster (simmedium). For the other applications, the total squash time is very small and the different core designs discussed make no difference. The applications run with 1, 4 or 16 threads. The applications are dynamically broken down into chunks of 10K dynamic instructions automatically in hardware. However, the software places chunk termination commands inside the library calls for barriers, to minimize any work squashed at barriers. We use these relatively large chunks because, as

<table>
<thead>
<tr>
<th>Core</th>
<th>Memory Subsystem</th>
<th>Chunk Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency: 5.0 GHz</td>
<td>Private write-back D-L1:</td>
<td># of outstanding</td>
</tr>
<tr>
<td># of contexts:</td>
<td>Size/assoc/line: 32KB/4-way/32B</td>
<td>chunks/thread: 1</td>
</tr>
<tr>
<td>4 for BulkSMT, 1 for BK</td>
<td>Hit round trip: 2 cycles</td>
<td>Target chunk size:</td>
</tr>
<tr>
<td>Fetch/issue/comm width: 4/4/5</td>
<td>Private write-back L2:</td>
<td>10k instructions</td>
</tr>
<tr>
<td>I-window: 80</td>
<td>Size/assoc/line: 256KB/8-way/32B</td>
<td>Commit latency:</td>
</tr>
<tr>
<td>ROB: 176 (1/4 per thread)</td>
<td>Hit round trip: 9 cycles</td>
<td>50 cycles (1 core)</td>
</tr>
<tr>
<td>Ld/St/Int/FP units: 2/3/3</td>
<td>L2 miss delay:</td>
<td>200 cycles (4 cores)</td>
</tr>
<tr>
<td>Ld/St queue: 56 (1/4 per thread)</td>
<td>Hit other L2s (avg): 16 in 4 cores/chip</td>
<td>250 cycles (16 cores)</td>
</tr>
<tr>
<td>Int/FP registers: 96/80</td>
<td>20 in 16 cores/chip</td>
<td>Signature size: 2K bits R and W</td>
</tr>
<tr>
<td>Branch penalty: 17 cyc (min)</td>
<td>To memory: 500 cycles round trip</td>
<td>Signature config: S14 from [14]</td>
</tr>
</tbody>
</table>

Table 4.6: Simulated system configurations.
discussed in [3], they more accurately represent future uses of chunked architectures, where the compiler optimizes the code, and the commit cost is more effectively amortized.

Figure 4.12: Execution time of 4-core architectures. The BulkSMT designs (SQ-LL, ST-LL, OR-LL, SQ-EE, ST-EE, OR-EE) run with 16 threads, while the BK designs (BK-LL, BK-EE) run with 4 threads. In each application, the bars are normalized to SQ-LL. The SQ-EE bar for Radiosity reaches 4.23.

Figure 4.13: Execution time of single-core architectures. The BulkSMT designs (SQ, ST, OR) run with 4 threads, while the BK design runs with 1 thread. In each application, the bars are normalized to SQ. The BK bar for Radix reaches 3.39.

4.7.1 Performance Comparison

We want to find out which of the BulkSMT designs performs best, and how does the performance of BulkSMT and BK compare (i) for a fixed number of cores (which is a proxy for the amount...
of hardware) and (ii) for a fixed number of total threads. Due to space limitations, we do not compare a chunked-execution platform to a non-chunked one. Our focus is chunked-execution environments and our goal is to find out the impact of SMT on them.

In our plots, we break an application’s execution time into the following types of processor cycles: cycles retiring instructions (Useful), stalled due to pipeline hazards (ProcPipe), stalled due to memory accesses (ProcMem), stalled because the chunk is stopped in STALL or ORDER (ChunkStall), and performing work that will be squashed (Squashed). The total time that a processor is stalled while committing a chunk is negligible. The plots also show the geometric mean of the applications, which cannot be broken down.

Comparing BulkSMT Designs

Figure 4.12 compares the execution time of all of the 4-core architectures: BulkSMT designs running with 16 threads as in Figure 4.11(c) and BK designs running with 4 threads as in Figure 4.11(d). Each application has 8 bars, organized as LL first and EE later, all normalized to SQ-LL.

Comparing the different BulkSMT designs, we see that SQUASH suffers from Squashed time, since it is not tolerant of dependences. As we move to STALL, Squashed decreases, but some ChunkStall time appears — often resulting in faster execution. Finally, as we move to ORDER, both Squashed and ChunkStall largely disappear, resulting in the fastest design. These trends are clearest in Barnes and Raytrace.

In Radiosity, the large changes across bars are due to the frequent enqueue and dequeue operations in a task queue. Each operation involves the update of shared variables in critical sections, which translates into squash and stall in SQUASH and STALL.

The particular characteristics of each application determine whether the LL or EE designs are better.

Overall, ORDER is the recommended design. On average, its LL design reduces the execution time by 38% relative to SQUASH or STALL. In the EE environment, the reductions attained by
ORDER are 48% relative to SQUASH and 35% relative to STALL.

Figure 4.14: Execution time of different 16-thread architectures. The BulkSMT designs (SQ-LL, ST-LL, OR-LL, SQ-EE, ST-EE, OR-EE) use 4 cores, while the BK designs (BK-LL, BK-EE) use 16 cores — hence about 4 times more hardware. In each application, the bars are normalized to SQ-LL. The SQ-EE bar for Radiosity reaches 4.23.

BulkSMT vs BK for a Fixed Number of Cores

We return to Figure 4.12 to compare the BulkSMT and BK designs. They use the same core count, which is a proxy for hardware amount, although BulkSMT runs with 16 threads and BK with 4.

Since the applications run with more threads in BulkSMT, they can attain higher performance. Moreover, the tightly-coupled SMT hardware enables fast inter-thread communication. However, these applications do not exhibit linear speedup curves up to 16 threads. Instead, their speedups saturate. Moreover, more inter-thread dependences appear, which the BulkSMT designs have to handle. Finally, with BulkSMT, multiple threads compete for the fixed resources of a core. The relative impact of these factors determines the execution time.

For example, in Radix, the BulkSMT designs perform better. With a single context executing in each core, processor resources are underutilized because the ILP is low; when 4 contexts are executing per core, processor resources are utilized better. On the other hand, in Raytrace, BulkSMT designs perform worse because of the increased contention for locks among the more threads.

Looking at the geometric mean, we see that ORDER is faster than BK, although SQUASH and STALL are not. Specifically, in the EE environment, ORDER reduces the execution time of
the applications by an average of 26% compared to \textit{BK}. The corresponding number in the LL environment is 10%.

Figure 4.13 repeats the experiments for 1 core, such that the BulkSMT designs run with 4 threads as in Figure 4.11(a), and \textit{BK} with 1 as in Figure 4.11(b). Each application only has 4 bars because there are no EE or LL effects. The bars are normalized to SQ.

We largely observe the same trends as in Figure 4.12, except that the BulkSMT designs perform relatively better than \textit{BK}. The reason is that the applications scale much better from 1 to 4 threads than from 4 to 16. From the mean, \textit{ORDER} is the best design, followed by \textit{SQUASH}, \textit{STALL}, and \textit{BK}. On average, \textit{SQUASH}, \textit{STALL}, and \textit{ORDER} reduce the execution time of the applications by 23%, 17%, and 32%, respectively, relative to \textit{BK}. Hence, supporting BulkSMT is cost effective.

**BulkSMT vs BK for a Fixed Number of Threads**

Figure 4.14 shows the execution time of 16-threaded architectures, where BulkSMT designs use 4 cores as in Figure 4.11(c) and \textit{BK} designs use 16 cores as in Figure 4.11(e). As a result, the \textit{BK} designs use about 4 times more hardware. The figure is organized as usual.

The figure shows that the OR designs, with much less hardware than the \textit{BK} systems attain, on average, about the same performance as the \textit{BK} systems. Specifically, the average execution time of OR-LL is 15% higher than that of BK-LL, while OR-EE’s execution time is 20% lower than BK-EE’s. The other BulkSMT designs are slower.

Comparing the \textit{BK} designs of Figures 4.12 and 4.14 we see that, as applications move from 4 to 16 threads, they reduce their \textit{Useful} and other stall times. However, they often increase their \textit{Squashed} time. On the other hand, \textit{ORDER} often avoids squashes thanks to its technique of ordering chunks. This is the case for Radiosity, Radix, Raytrace and Streamcluster.

Overall, combining all the findings in this performance section, we conclude that the \textit{ORDER} BulkSMT design is attractive. For 16-threaded applications, it performs significantly better than single-context core platforms with the same core count, and performs about the same as single-context core platforms with four times more hardware.
4.7.2 Dependence Analysis in \textit{STALL} and \textit{ORDER}

Figure 4.15 shows the number of dependences (\textit{WAW}, \textit{RAW} and \textit{WAR}) observed between the 4 threads running on a BulkSMT core. The data corresponds to the 16-thread ST-LL and OR-LL environments. We do not show data for \textit{SQUASH} because, on a dependence, one of the threads gets squashed. We also do not show data for the EE environment because the trends are qualitatively similar. For each application and architecture, the bars are normalized to 1 and broken down into the type of dependence. The number on top of each bar is the average number of dependences per 100K instructions.

![Figure 4.15: Types of dependences.](image)

We see that the number of dependences in \textit{ORDER} is larger than in \textit{STALL}. This is because, in \textit{STALL}, one of the chunks is stopped after the dependence. In \textit{ORDER}, both chunks can continue...
execution. Therefore, more dependences can be established between the two chunks. We also see that applications with a large difference between the number of dependences in \textit{STALL} and \textit{ORDER} have a much faster \textit{ORDER} architecture than \textit{STALL} in Figure 4.12. This is because many same-direction dependences between two concurrently-executing chunks are formed in \textit{ORDER}, while \textit{STALL} has to stall. Finally, we see that the dominant dependence type in \textit{ORDER} is RAW, while the three types of dependences are more equally distributed in \textit{STALL}.

\subsection*{4.7.3 Dependence Cycles in \textit{ORDER}}

Figure 4.16 characterizes the dependence cycles observed between the 4 threads running on a BulkSMT core. The data corresponds to the 16-thread OR-LL environment. For each application, the bars are normalized to 1 and broken down into the different types of cycles. The large majority of the cycles are formed between two chunks, and are classified according to the type of dependence. For example, RAW $\to$ WAR means that the cycle is formed by a RAW dependence followed by a WAR one in the opposite direction. Accordingly, there are 9 types of cycles between two chunks. The topmost class is cycles with more than two chunks. On top of each bar, we show the number of cycles per 100K instructions.

Some of the major types of cycles are WAR $\to$ RAW (where two consecutive reads are interleaved by a remote write) and RAW $\to$ RAW (where information is transferred from one processor to another and then back to the first one). Comparing Figure 4.16 and Figure 4.15, we see that the average number of cycles is typically much lower than the average number of dependences.

\subsection*{4.7.4 Squash Set Size}

Figure 4.18 shows the number of chunks that need to be squashed to break a cycle (i.e., the Squash Set size) in \textit{ORDER}. The figure corresponds to 16-thread OR-LL. For each application, the bar is normalized to 1 and broken down into squash set sizes: one (\textit{SqSet:1}), two (\textit{SqSet:2}), three (\textit{SqSet:3}), or four (\textit{SqSet:4}). We see that, in practically all cases, only one chunk is squashed to
break the cycle; the other chunk(s) can continue.

### 4.7.5 Comparing Local vs Global Squashes

Figure 4.17 compares the number of chunks squashed by intra-SMT conflicts (*Locally Squashed*) to those squashed by inter-core conflicts (*Globally Squashed*). The figure shows data for each of the BulkSMT designs with 16 threads: SQ-LL, ST-LL, OR-LL, SQ-EE, ST-EE, and OR-EE. There is a bar for each design, normalized to 1, and broken down into locally- and globally-squashed chunks.

The figure shows that, while most of the squashes in *SQUASH* and *STALL* are local, the opposite is true for *ORDER*. *ORDER*’s ability to allow the two chunks involved in a dependence to continue executing enables it to eliminate practically all of the local squashes. Interestingly, *STALL* still suffers local squashes. The reason is that stalled chunks, as they wait, are effectively vulnerable to squashes due to new dependences that appear.
Figure 4.17: Comparing the number of locally-squashed and globally-squashed chunks.

Figure 4.18: Squash sets sizes.

4.8 Conclusions

None of the previously-proposed architectures that continuously execute chunks of instructions or transactions use SMT cores — although SMT cores are widely deployed and would likely be used in a commercial implementation of these architectures.

To address this problem, this chapter has presented the first SMT design that supports continuous chunked execution. The design, called BulkSMT, can be used either in a single-core processor or in a multicore of SMTs. We have proposed three BulkSMT configurations with different cost and performance: SQUASH, STALL, and ORDER. We have described a set of novel architectural
primitives that enable chunked execution in an SMT core. Finally, we have shown how to augment
the resulting SMT core to work in a multicore of SMTs that supports chunked execution. Our
results, based on simulations of SPLASH-2 and PARSEC codes, showed that BulkSMT supported
this mode of execution cost-effectively. For example, in a 4-core multicore with eager chunked ex-
ecution, BulkSMT reduces the execution time of the applications by an average of 26% compared
to running on single-context cores. The corresponding number for lazy chunked execution is 10%.
In a single-core machine, the average execution time reduction is 32%.
Chapter 5

UniBlock: Unified and High Performance Whole-System Sequential Consistency by Speculative Chunk Ordering

Due to compiler optimizations, codes can violate sequential consistency (SC) even on a machine supporting SC. Therefore, it is important to ensure whole-system SC from the language level to architecture. This chapter proposes UniBlock, the first unified approach to support whole-system SC using a single set of mechanisms. The central concept in UniBlock, the Ordered Chunk, is used as both the mechanism to implement hardware SC and as a specification for the compiler to guide the hardware to ensure correct execution. Our solution is based on a conventional directory-based cache coherence protocol. In this chapter, while the architecture executes chunks, we do not assume as continuous execution of chunks.

5.1 Introduction

The memory consistency model (or memory model) specifies how the memory accesses performed by one thread become visible to other threads. Sequential Consistency (SC) is a strong and intuitive memory model that requires that all the memory accesses of a program appear to have executed in a global sequential order consistent with the per-thread program order. SC is critical for program productivity because it reflects programmers’ natural expectations of the program behavior when memory accesses from different threads are interleaved.

The memory model deals with the whole computing stack. Its semantic are only well-defined when the model is specified and enforced consistently in every layer, from language to the hardware. Therefore, to harness the benefits of SC, hardware-only SC enforcement is not sufficient, — the software can easily violate SC even if the hardware implementation is correct. We call the
SC guarantee in every system layer as whole-system SC. It is challenging to achieve the goal of whole-system SC with high performance, since it disallows even simple compiler transformations involving shared variables [44, 75]. Accordingly, modern languages (e.g. C++ and Java) only provide SC for data-race-free programs [9, 42]. For programs with races, these languages provide no or weak semantics that are difficult to reason about. To support whole-system SC, we need to consider two aspects: SC hardware and the mechanisms to prevent software from breaking SC semantics. In general, all the many previous proposals that enforce SC can serve as the SC hardware component. However, the hardware schemes that work smoothly with the software SC violation prevention mechanisms are more usable. Next, we first consider the software mechanisms to prevent SC violation and then consider the incorporation of the two components.

There are two approaches to prevent SC violation in software. We call them the safe and the speculative approach. The safe approach only allows the compilers to apply safe optimizations that do not violate SC [44]. Specifically, it requires the classification of the shared and private accesses and only allows optimizations on the thread private or read-only variables. This approach is simple but may limit the potential of compiler optimizations, because the access classification may be conservative. The speculative approach uses chunks to contain the effects of compiler optimizations [4]. Such chunks are always executed atomically and in isolation. Requiring speculation support, this approach enables better performance.

In hardware, different approaches, from naive SC implementation to in-window speculation [25, 62], to post-retirement speculation [6, 15, 27, 76], progressively enable better performance by more complex and aggressive hardware. The safe approach can work with any SC hardware. Recently, two proposals [38, 67] try to reduce the stall in in-window speculation by identifying the safe accesses early and allowing safe reorderings of those accesses. As shown in Section 5.3.1, they either suffer from the unnecessary serialization due to conservative access classification [67] or the latency to fetch of pending list from distributed directory modules and extra store serialization [38]. The limited compiler optimization capability and the performance bottleneck in current non-speculative SC hardware motivate supporting whole-system SC by speculation.
The speculative approach only works with the SC hardware with post-retirement speculation support. While considered more complex, this approach is better. First, with chunks, the compiler does not have to identify any shared access and can do aggressive optimizations that are not available in the safe approach. Second, post-speculation normally delivers better performance. However, the current post-retirement speculation designs (e.g. [6, 15]) do not work well with the compiler-marked chunks. InvisiFence [6] squashes a chunk on conflict, but compiler-marked chunks are usually much larger than the dynamic chunks generated by hardware. It may incur large amount of squashes.

5.2 Contribution

To enable high performance whole-system SC, this chapter presents UniBlock, the first speculative scheme based on conventional distributed cache coherence protocol that prevents SC violations due to hardware and software by a same set of techniques. The key and unique concept in UniBlock, ordered chunk, is used by the hardware as the mechanism to enforce hardware SC and by the compiler as the specification to guide hardware to execute the optimized code regions in a way that does not violate SC. Based on any conventional relaxed-consistency machine, UniBlock forms intermittent dynamic chunks when the speculative retirement of an instruction may violate SC. The compiler marks the optimized code regions as the static chunks to ensure the correct execution. The key technique to ensure high performance is the concurrent execution of dependent chunks in a distributed directory coherence protocol. UniBlock treats the static and dynamic chunks in a unified manner and cleanly supports whole-system SC.

5.3 Background and Related Work

5.3.1 Hardware SC Enforcement

The existing proposal for hardware SC can be classified into five categories.
Naive. The naive SC implementation ensures SC by forcing every memory operation to *issue* in order. Each load and store can only be issued when all the previous memory operations complete. The performance of this implementation is low since it destroys all the parallelism in memory accesses.

**In-window speculation** [25]. To improve the performance, this optimization allows the hardware to prefetch and execute memory operations out of order but complete in order. After a store is retired from the processor the following loads and stores still have to wait for it to complete. A high latency store can cause significant execution stall. If the processor receives an external coherence (or replacement) request that conflicts with a memory operation that has been executed out of order, the processor needs to replay from the oldest conflict operation.

**Post-retirement speculation** [6, 27, 62, 76]. To further reduce the ordering constraints, this optimization allows the processor to complete the load or store out of order. If an instruction is retired from processor when the store buffer is not empty, this instruction is considered to be speculative, and the processor creates a checkpoint of the state before the speculative retirement. During speculation, a remote access that conflicts with the speculatively retired instructions causes a processor roll back to the execution state recorded in the checkpoint.

**In-window speculation + safe reordering** [38, 67]. This approach does not allow post-retirement speculation, but the processor tries to proactively get some extra information which can be used to infer whether a memory operation can be safely reordered. If so, the instruction can be safely retired from the processor with pending operations.

Lin *et al.* [38] proposes to check the cache hits with the pending lists from directory modules before they can retire from the processor. If it does not conflict with the pending list, it can retire with the presence of pending operations. While this approach reduces the stall time that would have been incurred in plain in-window speculation scheme, it has two drawbacks.

First, this scheme does not work well with the distributed directory protocol since the fetch of a pending list from one directory will invalidate the one from another, potentially leading to the fetch of pending lists alternatively. Such requirement is because the processor must *read* the
pending lists sequentially.

Second, it requires that the store miss addresses are registered in directory modules in order. Such requirement ensures that the global pending list is always updated in a sequential manner. Before a store miss can start performing, it needs to wait for all the previous stores to contact the directory modules in sequence. This serialization may prevent the following instructions from retiring and reduce performance.

Singh et al. [67] proposes another approach. The idea is to use two separate store buffers for safe and unsafe accesses. The access classification is done in TLB translation at page level. The unsafe stores enter the in-order store buffer, and only the unsafe loads cannot bypass the in-order store buffer. The safe stores can complete out-of-order in the unordered store buffer, both safe and unsafe loads can bypass the unordered store buffer. This approach is simple but the coarse-grain sharing information may incur unnecessary stalls. More importantly, once a page is marked as shared read/write, it can never transition back to private. As the system runs for a long period of time, most pages in the system will be marked as shared due to dynamic memory allocation, it can make most accesses be marked unsafe and the scheme is close to the plain in-window speculation. Finally, to provide whole-system SC, it still relies on the conservative SC compiler [44].

Back-to-back chunk-based architecture (e.g. BulkSC [15] and InvisiFence-Continuous [6]). This approach is a variant of post-retirement speculation. It enforces SC at the granularity of consecutive groups of dynamic instructions (chunk), which is executed atomically and in isolation.

Fence-based Approach [69]. Hardware SC can be enforced by inserting enough fences to prevent the reordering. The naive way is to insert fences between any two memory operations that may be reordered, more advanced compiler analysis (e.g. escape analysis) can reduce the number of fences needed. It is worth noting that this approach is oblivious to the hardware memory model, as long as the fences are selected, inserted and implemented correctly in the hardware. To further improve performance, additional hardware supports can reduce the cost of the fences. This chapter does not consider this direction because the compiler does not enable any extra compiler optimization than the safe approach. Second, the hardware components (relaxed-consistency hardware plus
potential supports to reduce fence cost) are not SC hardware, which means that it may violate SC when executing even the plain code with no compiler transformation and no fences.

### 5.3.2 Compiler Optimizations and SC

Compiler optimizations, typically involving multiple memory access reordering, can generate codes that may produce non-SC behavior even in SC hardware. It is because the SC-compliant interleaving of memory accesses can violate SC when the memory access orders are recovered to the original order in the program. In the following, we show two examples.

In Figure 5.1 (a), the non-SC behavior is due to the reordering of the stores by compiler in the codes executed by $P_0$. The original order of stores is $A0 \rightarrow A1$. After the reordering, load $B0$ can observe the value produced by $A1$ and load $B1$ still sees the value of address "x" before $A0$. Such results satisfy SC in the reordered codes, but no valid SC interleaving in the original code can produce such results. Therefore, SC in the original codes is violated.

More interestingly, in Figure 5.1 (b), we show that removing the redundant store can also violate SC. In $P_0$, there are two stores ($A0$ and $A2$) to address "x". The compiler can easily decide to remove the first redundant store $A0$. After that, the $B0$ and $B1$ can be interleaved between $A1$ and $A2$.

![Figure 5.1: Compiler-introduced SC Violation.](image-url)
and A2. Such interleaving is valid in SC hardware, in which B1 observes the old value of “x” before the new value is produced by A2. However, the behavior can potentially violate SC in the original code when the value observed by B1 is before A0. In the optimized code, such dependence is lost due to redundant store elimination and SC hardware can produce non-SC behavior.

Marino et al. [44] tries to prohibit the compiler optimizations that may violate SC (the safe approach). Specifically, the compiler preserves SC by only allowing the optimization on the thread private or read-only variables. We believe that this approach may limit the potential of compiler optimizations because the compiler has to be conservative in classifying share and private variables. Ahn et al. [4] proposes to use chunk to contain the effects of aggressive optimizations (the speculative approach). Such solution is supposed to work with a non-conventional back-to-back chunk architecture [15].

5.4 UniBlock: Unified Whole-System SC by Ordered Chunk

This section presents UniBlock, the first framework based on conventional directory cache coherence protocol that supports high performance whole-system SC by a unique technique, ordered chunk. Such chunk serves both as the mechanism in hardware to enforce SC and as the specification for compilers to guide the hardware to execute the aggressively optimized codes in a correct way that never violates SC.

5.4.1 Static Chunk: Specification from Compiler

To fully realize the potential of compiler optimizations, UniBlock allows the compiler to use the chunk as the specification to mark the code regions where optimizations may violate SC. The compiler-marked chunk is called static chunk. Unlike the previous approach [44], UniBlock does not require the compiler to distinguish the shared and private variables and does not place any restriction on the compiler optimization (e.g. it can involve the reordering of multiple potentially shared variables). All a compiler needs to do is to wrap the code region with a static chunk. With
chunk, the hardware gives the compiler the promise that the effects of the reordering inside the chunk is never exposed to the other processors in program execution. Therefore, any compiler optimization never violates SC in UniBlock hardware.

Our goal is to seek the efficient hardware implementation based on conventional coherence protocol. Indeed, the static chunk can be supported in the current schemes (i.e. InvisiFence [6]), where any conflict remote access to the data touched by a static chunk will squash the chunk. However, even if the static and dynamic chunk have the same semantic, they have very different characteristics. To harness more performance enhancements by optimizations, the compiler has to perform the code transformation in a large region, which can make the static chunks much larger than the chunks generated by hardware. The larger static chunks can increase the cost of chunk squash. This may easily offset the expected performance gain from the compiler optimizations. Therefore, a better post-retirement speculation scheme is needed.

Any architecture supporting the chunk has a structure to buffer the local speculative dirty data. The overflow of the structure may cause the static chunk to repeatedly squash and never be able to commit. To make progress in these cases, we would have to commit a downsized chunk, — i.e. the code up until the cache overflow. However, this would break the atomicity of the chunk and potentially expose inconsistent or non-SC state. To address these cases, similar to BulkCompiler [4], a safe version of the code is generated for each static chunk. The compiler can only perform safe optimizations for the safe code. If the static chunk needs to be truncated for any of the repeatable reasons, the static chunk is squashed and execution is transferred to the PC of the safe version entry point. The details in supporting the safe version can be found in [4].

5.4.2 Ordered Chunk

Based on post-retirement speculation, UniBlock achieves higher performance by enabling the concurrent execution of dependent chunks. On conflict between two chunks, instead of squashing one of them, the hardware lets both chunks continue the execution and enforces the order of chunk commits. Only when a cyclic dependence is formed, one or more chunks are squashed. The model
can be applied in the same way to static or dynamic chunk. To realize the idea, we need to consider the issues of chunk initiation (both local and remote), speculative data management and cycle (involving arbitrary number of processors) detection. We discuss them in detail in Section 5.5. We argue that UniBlock is particularly important for the larger static chunk, since it can avoid the squash on the first conflict and only incurs the squash on much more rarer dependence cycles, which implies the execution is not serializable.

UniBlock is the first scheme supporting concurrent execution of dependent chunks with post-retirement speculation and can also be applied to the transactional memory system. DATM [61] tries to concurrently execute the dependent transactions, but it is based on bus and relies on the snoopy protocol to update the structures that enforce transaction commit order. The version management of UniBlock has some similar requirements with Cherry-MP [31] in that some of the data held in caches may be volatile, — that is, subject to rollback. Again, Cherry-MP is also a design based on bus. Tracking the data dependences in a distributed directory protocol is a much harder problem.

### 5.4.3 Whole-System SC Schemes

![Whole-system SC Design Space](image)

Figure 5.2: Whole-system SC Design Space.
With the compiler-marked static chunk and the UniBlock hardware, our system cleanly supports whole-system SC. The overall view of the paradigm is shown in Figure 5.2, the gray boxes are the UniBlock components. For comparison, we also show the alternative approaches toward whole-system SC.

First, the static chunk can be used with the existing post-retirement speculation schemes (i.e. InvisiFence, ASO, etc.). This solution may incur repeated static chunk squashes, eliminating the benefits of the optimizations. The Commit-On-Violation (COV) [6] delays the squashes for a number of cycles to allow the local chunk to commit. However, this technique may not be applied well for the larger static chunks. Moreover, COV is essentially a hybrid conflict resolution policy in between squash and stall, it may incur subtle corner cases that make it not as simple as it is described (e.g. when a stalled chunk needs to stall another chunk).

The second option is to use the static chunk with the back-to-back chunk architecture to support whole-system SC. Some hardware implementations (e.g. BulkSC [15]) requires non-conventional cache coherence protocol. Moreover, neither InvisiFence-Continuous nor BulkSC allows the concurrent execution of dependent chunks in different processors.

As the third option, SC preserving compiler [44] can be used with any SC hardware. An interesting design point is to combine it with the simple hardware [67] that enables in-window speculation and safe reordering. This approach can limit the potential of compiler optimization and may not offer high performance due to the coarse-grained access classification.

For completeness, we also list the scheme that enforces SC by inserting sufficient fences. Only this approach can be used with relaxed-consistency hardware. We do not consider this approach since the compiler is only used to avoid SC violation on relaxed-consistency machine but not to enable extra compiler optimization.

We believe UniBlock is best among all the choices in the design space, because it offers better performance compared with the existing post-retirement speculation schemes due to the concurrent execution of dependent chunks.
5.5 UniBlock Hardware Design

This section discusses the hardware and protocol design of UniBlock. The main challenges in directory protocol are speculative version management and cycle detection.

5.5.1 Chunk Initiation

In UniBlock, a chunk can be initiated locally or by a remote chunk.

Local Chunk. An chunk is initiated whenever the processor retires an instruction that is unsafe to retire in SC. More specifically, whenever a load or store is about to retire, if the store buffer is not empty, a chunk is created. The checkpoint for the chunk records the processor state before the instruction is retired. Figure 5.3 (a) shows an example of locally created chunk.

Remote Chunk. When there is a conflict (dependence) from a speculative to a non-speculative access, if the processor executing the non-speculative access is not currently in an chunk, it needs to create one. The chunk is created due to the remote access of speculative data. If the remote access is a load, the load is the first instruction of the chunk (Figure 5.3 (b)). If the remote access is a store, the remote chunk starts with the first memory instruction after the store (Figure 5.3 (c)). Note that the chunk can be always created: If the remote access is a load, it is still in ROB, a checkpoint can be taken before it retires from the processor. If it is a store, before the processor (P_1) knows that it (the first instruction in P_1) conflicts with a speculative access, when a memory instruction is about to retire (e.g. the second instruction in P_1), the remote processor (P_1) will create a chunk according to the local chunk creation condition. It is because the first store is guaranteed to be still in store buffer. The remotely initiated chunk is forced to commit after the local chunk. If there is already a chunk in the remote processor, no chunk is created and the system just enforces the commit order from the local chunk to the existing remote chunk.

In both cases, the remote access observes the speculative value. It is the key distinction between UniBlock and the existing post-retirement speculation schemes. In UniBlock, when the local speculative data are accesses by the remote processors, the system creates a remote chunk (if there
isn’t one) and establishes the data dependence and chunk commit order. The local chunk is not squashed. In the current post-retirement speculation schemes (e.g. InvisiFence), any access to the local speculative data causes the local chunk to squash.

The chunk initiation policy guarantees the following three invariants: (1) If the source of a conflict is in a chunk, the destination must be also in a chunk. (2) The commit order of the two chunks is always the same as the direction of the first conflict. (3) Each processor only has one chunk at any time.

![Chunk Initiation Diagram](image)

**Figure 5.3: Chunk Initiation.**

### 5.5.2 Conflict Detection and Chunk Ordering

After a processor enters a chunk, it keeps the read and write sets. The conflicts are detected when the addresses of remote accesses hit the local address sets. In practice, the sets can be implemented as bloom filters or extra bits in the cache line. Since we only have one chunk in any processor at any time, we use the extra bits to represent the address set. The speculative access bits are marked when the memory operations are executed. This policy ensures that any consistency violation will be detected without requiring an in-window mechanism.

To record the chunk ordering information, each processor maintains two sets $Pred\_Set$ and $Succ\_Set$. They represent the predecessor and successor set for the current chunk in the processor.

The directory protocol ensures that whenever a conflict occurs, both the source and destination processor know the processor ID of the other one. Each of the processor records the other processor
in its *Pred Set* or *Succ Set*. We call the source and destination of a dependence as \( P_s \) and \( P_d \).

For RAW, \( P_s \) knows the \( P_d \) when it is about to provide the speculative data, at which point *Succ Set*[\( d \)] in \( P_s \) is set. The request to create a remote chunk is piggybacked in the response message. When \( P_d \) receives the data response, it creates a chunk and set *Pred Set*[\( s \)].

For WAW, when a speculative written line in \( P_s \)’s cache is invalidated by a remote write from \( P_d \). *Succ Set*[\( d \)] in \( P_s \) is set. The \( P_d \) sets *Pred Set*[\( s \)] when it receives the data response, a chunk is created if \( P_d \) does not have a chunk. The chunk dependence information can be similarly piggybacked and recorded for WAR.

### 5.5.3 Chunk Commit

A chunk can be committed when all of the following conditions are true: (1) all its predecessor chunks are committed (all bits in *Pred Set* are cleared); (2) the store buffer drain; (3) the chunk size reaches the maximum threshold.

When a chunk is committed, it sends a message, *commit*, to all the processors in *Succ Set* and clears the current read and write set. When a processor receives *commit* from \( P_i \), it removes \( P_i \) from *Pred Set* and checks whether the current chunk can be committed by the same conditions. Similarly, when a chunk is squashed, it also sends *squash* to all its successor processors so that the dependent chunks do not need to wait. The maximum chunk size is to prevent high cost squashes, we use 1000 as the threshold.

The commit and squash of dependent chunks incur some message overhead. However, this overhead is expected to be small and well-deserved. First, conflicts are low in common cases, therefore, most chunks are committed without dependence. Second, for the chunks with dependence, most of such extra overhead is paid to avoid the squashes, which hurts performance much more negatively. The cyclic dependences are expected to be very rare for dynamic chunks.
5.5.4 Speculative Version Management

In UniBlock, the speculative data can be forwarded around the system. This section first presents the correctness requirements of the speculative data forward then discusses the designs based on a directory protocol. Finally, we validate the design by showing that all the correctness requirements are satisfied.

Protocol Requirements

At any point in the execution, the system should maintain the following requirements.

1. When a chunk is committed, the data produced by the chunk is never lost and will be eventually merged to the non-speculative memory state.

2. When a chunk is committed, all the data it reads should be non-speculative.

3. When a chunk is squashed, all the speculative data needs to be cleared in the system.

4. Every access should get the most recent version of data, either speculative or non-speculative.

5. Every dependence established through speculative data forwarding should be recorded.

Main Idea: Logging the History of Speculative Updates

The conventional directory protocol naturally forwards the most recent version of data to the reader, however, it is challenging to support the rollback. On a chunk squash, the speculative data produced may have been transferred to several other processors, the protocol needs to find the data, invalidate them and possibly squash some remote chunks. On the other side, the protocol needs to make sure that the only copy of non-speculative data is not lost by local invalidations on chunk squash. The directory state (e.g. the ownership pointer) also needs to be recovered.

These difficulties did not exist in the existing post-retirement speculation schemes (e.g. InvisiFence), where the protocol never allows the speculative data to be provided. For instance, if a
remote processor reads the local speculative dirty cache line, InvisiFence squashes the local chunk and lets the shared cache to provide the old version of the data. This is why before a speculative dirty store writes to the local cache, a non-speculative version is written back.

The version management of UniBlock is carefully designed to handle these issues. The main idea is to carry a log of speculative updates around with the ownership of a cache line, so that the owner can ultimately produce the non-speculative version of the line when speculative chunks from the predecessor processors commit. To avoid the potential chain recovery of data version, UniBlock transfers the cache line around the system exactly the same as the unmodified directory protocol but never merges the speculative data with the line before the chunk is committed. The directory always has the illusion that different processors write the line as usual and will forward all requests of a cache line to the current owner. In serving the forwarded requests, the private cache provides both the non-speculative cache line plus the log of updates. The receiver uses the most recent version, which may be speculative. The logs of the updates to different lines are cleared on chunk commit or squash.

Sending the update logs incur the bandwidth overhead. This overhead is low, because such information is only needed on a dependence, — not frequent in the well-behaved applications. The protocol allows the logs containing several updates, this implies that a write chain is formed, — even rarer in practice.

**Hardware Structures**

This section discusses the hardware structures needed in UniBlock, we assume a basic MESI directory protocol. Each node has a processor with a private L1 cache and a bank of shared L2 cache.

Different from the previous post-retirement speculation schemes, the processor can no longer use the private cache as the buffer to store the speculative dirty data. In UniBlock, each processor needs to have a private ”store buffer” holding the speculative dirty words produced by the local chunk. It is because the original non-speculative data need to be preserved until the current chunk
is ready to commit. In addition, the processor can receive the update logs, which may contain multiple versions of the same word. The hardware structures needed in UniBlock is shown in Figure 5.4.

**Figure 5.4: UniBlock Hardware Structures.**

**L0 Cache.** We propose to have a private small and fast *L0 cache* for each processor to serve as the private store buffer. It is between the store buffer and the L1 cache. It avoids the non-scalable associatively-searched store buffer. It also ensures the fast accesses to the locally produced speculative dirty data. For simplicity, the line sizes of L0 and L1 are the same. Each word in a L0 cache line has a valid bit since it holds the speculative dirty values in word granularity. On an external invalidation, the presented cache line in both L0 and L1 are invalidated, and the speculative dirty words in L0 are sent to the next writer as the update log. On chunk commit, the dirty words are merged with the line in L1 and the whole L0 is invalidated. Please refer to Section 5.5.4 for detailed discussion of protocol operations.

**Speculative Version Buffer (SVB).** To handle the multi-version words in the logs, each processor also has a *Speculative Version Buffer (SVB)*. This buffer only holds the different versions of speculative words produced by the local processor’s predecessors. Each SVB entry keeps the word address, producer processor ID, speculative dirty data and a Previous Writer (PW) bit. The word produced by the most recent previous writer is marked with PW bit. With PW bit, when the line is read by future accesses, the processor can provide the correct data version.

SVB does not need to be large because the conflict is not common. For each local access, L0,
L1 and SVB are checked in parallel. If it hits in L0, the word is returned, otherwise, if it hits in SVB, the most recent version (with PW set) is returned. If neither L0 nor SVB has the data, the non-speculative version from L1 is returned. To avoid unnecessary associative search of SVB, we use a counting bloom filter (with the deletion capability) to record the word addresses presented in the SVB.

**Speculative Dependence Table (SDT).** It exists in directory and is needed due to the read of speculative data, where the cache line will be shared in both source and destination processor. The source is not the owner of the line anymore and will not observe the future requests to the line.

**Dependence Forwarding Table (DFT).** It is the unified low cost hardware component recording the *Pred Set*, *Succ Set*, the direct and transitive dependences. Squash dependence specifies when a chunk is squashed, what other chunks need to squash due to data dependences. It will be discussed in Section 5.5.4, here we just show how that dependences are recorded.

DFT is organized as a bit array, the DFT in $P_i$ is denoted as $DFT_i$. The first bit (P-bit) in each row indicates the *Pred Set*. $P[j]$ ($DFT_i[j][0]$) indicates that $P_j$’s chunk has to commit before $P_i$’s chunk. The second bit (S-bit) indicates the squash dependence, $S[j]$ ($DFT_i[j][1]$) indicates that the squash of $P_j$’s chunk will cause the squash of $P_i$’s chunk.

The next $n$ bits (in dark gray) in each row summarize the transitive dependence (called TD bits). $TD[j][0]$ to $TD[j][n]$ are the bits from $DFT_i[j][2]$ to $DFT_i[j][n+1]$. $TD[j][k]$ is set when some speculative words in SVB produced by the predecessor $P_j$ are accessed and transferred to remote processors ($P_k$). The $i$-th row (light gray) of TD in $P_i$ indicates *Succ Set*.

The last $n$ in each row are called *Dependent Directory (DD)* bits. $DD[j][0]$ to $DD[j][n]$ are the bits from $DFT_i[j][n+2]$ to $DFT_i[j][2n-1]$. $DD[j][k]$ is set when some SDT entries of speculative words produced by $P_j$ is transferred to directory module $k$’s SDT. The n-bit DD for each processor naturally work for the distributed directory. If the system only has a single directory, each predecessor only needs one-bit DD for each processor.

For a system with $n$ processors and $n$ directory modules the cost of the DFT in one processor is $(2n + 2) \times n$ bits. For a 64 processor system, the overhead for each processor is about 1 KB, a
very modest overhead.

The overflow of the hardware structures is handled as follows. If it is due to dynamic chunk, the processor stalls the execution until the local chunk is committed. If a static chunk causes the overflow, the chunk is squashed and the processor then executes the safe version (see Section 5.4) without creating a static chunk.

**Protocol Operations**

This section discusses UniBlock protocol operations.

**Speculative Write.** The stores in a chunk write the speculative words in L0 cache and bring the non-speculative version to the L1 cache by normal coherence operation.

When a remote processor writes on a local speculative dirty line, a Write-After-Write (WAW) conflict happens, we call the source and destination writer as P_s and P_d. Since the directory owner pointer is updated as normal coherence protocol, the write miss in P_d will be forward to P_s. On receiving the invalidation, P_s invalidates the cache line in both L0 and L1. In the response, both the non-speculative cache line from L1 and speculative words in L0 in the line are sent to P_d. When P_d receives the response, it places the speculative dirty words from P_s in its SVB and its own speculative dirty word is written to its L0 cache. After the transactions, P_d has the only copy of the non-speculative cache line and both versions of the speculative dirty words (P_s’s in SVB and its own in L0 cache).

When the WAW happens, if P_s has some words in the cache line in SVB, these entries are also sent to P_d. The PW bits are properly set for the words with the most recent versions. More specifically, if a word is produced by P_s, then the words in L0 are marked with PW bit. Otherwise, the version in SVB with PW bit set is the most recent version. In addition, the transitive dependence (TD) bits are set according to the owners of the SVB entries. For example, suppose one of the owners is P_i, then TD[i][d] is set after P_d gets the update log. This is used in forwarding the commit and squash messages and will be discussed later.

**Speculative Read.** When the speculative data produced by P_s is read by the remote processor (P_d),
a Read-After-Write (RAW) conflict happens. Different from WAW, \( P_s \) generates the most recent version of the cache line by merging the non-speculative line in L1 and most recent speculative words in SVB and L0, and sends to \( P_d \). In addition, it sends all the speculative versions of the words in the line to SDT in the directory and invalidates those entries in SVB without PW bit set. The most previous version (with PW set) and the data produced by itself in L0 need to stay in the processor because the processor should be able to read the words. The transitive dependence bits are set in the same way as WAW case.

Note that the line provided to \( P_d \) is a single most recent version potentially mixed with non-speculative words and most recent speculative words. It is safe because \( P_d \) will never become the source that can provide such line. If \( P_d \) later writes the line, it first needs to contact directory to get the ownership, as a part of that, the directory will provide the SDT entries to \( P_d \) in the same way as \( P_d \) would get from \( P_s \). After the transaction, the directory invalidates the entries in SDT.

**Chunk Commit.** When a processor (\( P_i \)) is ready to commit a local chunk, it first merges the speculative dirty words in L0 to L1. After that, the whole L0 cache is invalidated and the local commit completes. Then, it sends commit messages to all the processors in \( \text{Succ Set} \) and directory modules according to \( \text{DD}[i][0] \sim \text{DD}[i][n] \).

When \( P_j \) receives the commit from the predecessor (\( P_i \)), it does the following operations. (1) Merge the words in its SVB produced by \( P_i \) with PW bit set to the non-speculative cache line in L1, all the lines are guaranteed to be in L1, since the SVB entries always are transferred with the cache line as the update log. (2) Forward the commit to other processors according to \( \text{DT}[i][0] \sim \text{DT}[i][n] \) in DFT\(_j\). (3) Forward the commit to all the directory modules according to \( \text{DD}[i][0] \sim \text{DD}[i][n] \) in DFT\(_j\). The processors receiving the forwarded commit messages perform the same set of operations. The committing processor ID is indicated in commit and in the forwarded commit. The second and third operation ensure that all the speculative words produced by \( P_i \) are merged with the non-speculative cache line, no matter where the line is.

When a directory module receives the commit message of \( P_i \), it checks the SDT, finds the words produced by \( P_i \) and merges them to the line in shared L2 cache. Since the SDT entries are still in
directory, the relevant L2 line is guaranteed to be still in Shared state.

**Chunk Squash.** When a processor squashes a chunk, the L0 cache is invalidated. Then the *squash* messages are sent and forwarded to other processors and directory modules in the exact same way as in commit operation. The only difference is that when they receive the *squash*, the SVB (in processors) and SDT (in directory modules) entries are invalidated without merging. For both commit and squash, after the forwarded messages are sent, the DFT bits for these messages are cleared.

**Squash Dependence**

It specifies the relation that the squash of local chunk may cause the squashes of one or more remote chunks. Squash dependence of a protocol is determined by the dependence types between two chunks and version management policy. In the following, we discuss it for UniBlock. For each dependence, the source and destination chunks are denoted as $B_s$ and $B_d$ and they execute on $P_s$ and $P_d$.

If the two chunks have a RAW, the squash of $B_s$ will cause the squash of $B_d$ because it is a true dependence. However, the squash of $B_d$ will not cause the squash of $B_s$. For WAR, the squash of either chunk will not cause the squash of the other.

If the two chunks have a WAW, the squash of $B_s$ will not cause $B_d$ to squash because the speculative dirty data produced by $B_d$ are not lost, — still in its L0 cache or other processors’ SVB. Similarly, the squash of $B_d$ does not cause $B_s$ to squash because it never overwrites the data produced by $B_s$.

If $B_s$ is squashed, it will send *squash* message to $P_d$, which will invalidate the dirty words produced by $B_s$ and possibly forward $B_s$’s squash to the other processors or directory modules.

Therefore, in UniBlock, the only squash dependence is due to the true dependence. When a chunk in $P_j$ reads the speculative data produced by $P_i$, $S[i]$ in $P_j$ (DFT$_j[i][1]$) is set. If later $P_i$’s chunk is squashed, $P_j$’s chunk also needs to squash. It is different from the previous proposals [59, 61] and minimizes the negative performance penalty due to the squash dependence.
A Running Example

This section considers a running example of UniBlock protocol, we show all the operations and state changes of the relevant hardware structures.

The example is shown on the left of Figure 5.5: four processors are all in a chunk. They try to access the same cache line, which contains two words [A:B]. The event order is marked with circled numbers. On the right, we show the L0, SVB and SDT state changes after each step. The state after the SVB entry is the state of the line in L1 cache. On the bottom, we show the final states in the DFT in each processor. The write value is indicated below each operation.

After P₀ writes the word A (1), the speculative dirty word is put in P₀’s L0 cache. When P₁ tries to write the same word (2), both L0 and L1 cache line are invalidated in P₀ and the speculative dirty word A produced by P₀ is sent as the update log with the non-speculative cache line. When P₁ receives it, the log is put to SVB. The PW bit is not set since P₁ writes the same word.

Then P₂ writes a different word B (3), the speculative dirty value is put in P₂’s L0 cache. The
update log with the non-speculative cache line transfer includes the two version of word A produced by \( P_0 \) and \( P_1 \), only the PW bit of \( P_1 \)'s version is set because it is the most recent speculative version.

After the first three events, the cache line moves around the system without being written, the state is Dirty in the most recent writer’s L1 cache. Finally, \( P_3 \) tries to read the line (\( \Theta \)). The read miss request is forwarded by directory to \( P_2 \). In the response, it generates the most recent version of the whole line, which includes word A produced by \( P_1 \) and word B produced by \( P_2 \). Then, the SVB entries are sent to the directory. The most recent version of A and B still stays in \( P_2 \). After the RAW, the SDT has three entries for the words in the cache line, there are two version of word A produced by \( P_0 \) and \( P_1 \) and one version of word B produced by \( P_2 \). The line is Shared in both directory and \( P_2 \) and \( P_3 \)'s L1 cache.

Next, we look at the final states of the DFTs. After \( \Phi \), DFT\( _0 \) is unchanged since there is no data forwarding. After \( \Theta \), \( P_1 \)'s chunk becomes the successor of \( P_0 \)'s chunk, therefore, the bit for \( P_1 \) in \( P_0 \)'s Succ_Set is set. The P[0] bit is set in DFT\( _1 \) to indicate that \( P_0 \) is its predecessor. After \( \Theta \), since \( P_1 \) forwards the SVB entry of \( P_0 \)'s update and its own speculative dirty word in L0 to \( P_2 \), both TD[0][2] and TD[1][2] are set. The first bit indicates that if \( P_1 \) receives the commit or squash from \( P_0 \), the message needs to be forwarded to \( P_2 \). The second bit indicates that when \( P_1 \)'s own chunk is ready to commit or has to squash, it needs to send commit or squash message to \( P_2 \). After \( \Theta \), since \( P_2 \) sends a generated cache line including the words produced by \( P_1 \) and \( P_2 \), both TD[1][3] and TD[2][3] are set. It requires that \( P_3 \)'s chunk must commit after these two chunks. In DFT\( _3 \), P[1], P[2] and S[1], S[2] are set, indicating that the chunks in \( P_1 \) and \( P_2 \) are in its predecessor set and also, if any of the two is squashed, the chunk in \( P_3 \) also needs to squash. Note that the TD[0][3] in DFT\( _2 \) is not set and \( P_0 \) is not a predecessor of \( P_3 \), it is because \( P_3 \) does not read the data produced by \( P_0 \). Suppose the directory module has index 1, the DD[0][1], DD[1][1] and DD[2][1] are all set, it is because the SVB entries produced by all \( P_0 \), \( P_1 \) and \( P_2 \) are sent to the directory module.

Finally, we briefly review the situation for chunk commit and squash. When \( P_0 \)'s chunk is squashed, since \( P_1 \) is in its Succ_Set, the squash message is sent to \( P_1 \). When \( P_1 \) receives the
message, there is no entry in its SVB produced by P₀, so it does not do any merge. Because the TD[0][2] is set, P₁ forwards the *squash* to P₂. In DFT₂, the DD[0][1] bit is set, the *squash* needs to be forwarded to directory module with index 1. On receiving it, SDT entry produced by P₀ is removed.

When P₀ commits, the *commit* message is propagated in the same way. When it reaches the directory, the entry is merged with the shared L2 cache line even if it is not the most recent version. It is to guarantee the correct system state if the newer writer (i.e. P₁) is squashed. Eventually, the commit of P₁ will merge the word A again to the shared line in L2, overwriting the previous version produced by P₀.

**Property Validation**

This section informally validates UniBlock design by checking all the protocol requirements in Section 5.5.4 are satisfied.

**Req. 1**: It is always guaranteed because the speculative words produced by a chunk are always passed around the system together with the non-speculative cache line as the update logs. The current holders of the speculative words can never discard the data unless it receives the *squash* from the processor that produced the data. If the chunk is committed, the current holders can only receive a *commit* for the chunk. DFT ensures that all relevant processors and directory modules will receive the *commit* message. Therefore, the data produced by a committed chunk are never lost and will be eventually merged to the non-speculative memory state.

**Req. 2**: It is guaranteed because if a chunk reads the speculative data, it can only commit after the producer of the data is committed, which implies that the data are non-speculative. If the producer chunk is squashed, the consumer chunk also squashes according to squash dependence.

**Req. 3**: It is guaranteed for the similar reason as **Req. 1**. Eventually the words produced by the squashed chunk are cleaned in the system on receiving *squash* messages in the relevant processors or directories.
**Req. 4:** It is guaranteed because all the update logs are transferred together with the non-speculative cache line. The current owner is always able to provide the most recent version of data.

**Req. 5:** It is satisfied since all the dependence information is recorded in DFT.

### 5.5.5 Distributed Cycle Detection

When there is a cycle between two or more chunks, they are no longer serializable and one or more chunks need to squash. This section discusses the distributed cycle detection algorithm. Our algorithm is better than the previous proposal in DATM [61] in two ways. First, it detects the cycles precisely, instead, DATM uses the order vector to conservatively detect cycles. Second, DATM’s mechanism can only work for snoopy protocol on bus, since state change of the cycle detection related structures relies on the broadcast. BulkSMT [59] also has a precise cycle detection, however, that algorithm can only work among the threads in the same SMT processor. UniBlock’s cycle detection algorithm is the first to detect precise cycles in a distributed directory protocol. We considers the cycles between two processors and more processors separately.

The cycles between two processors are easy to detect. As shown in Figure 5.6 (a), when the cycle is formed, each processor finds another processor in both its Pred Set and Succ Set.
The cycles between more processors are more difficult to handle. Figure 5.6(b) shows a cycle among three processors as an example. When the cycle is formed, none of the processors have the same processor in their \textit{Pred\_Set} and \textit{Succ\_Set}. To detect the transitive cycle, we propose to have the following two extra operations.

**Update.** When a chunk’s \textit{Pred\_Set} is changed, it needs to notify all the chunks in its \textit{Succ\_Set} about the updated \textit{Pred\_Set}.

**Forward.** When a chunk’s \textit{Succ\_Set} is changed, the source chunk’s \textit{Pred\_Set} needs to be sent to the new destination chunk.

The above two operations are performed as the program executes. The **cycle condition** is when a processor finds itself in its \textit{Pred\_Set}.

With the update and forward operations, each processor in Figure 5.6(b) will eventually have all the processors involved in the cycle in its \textit{Pred\_Set}.

We address two issues. First, our algorithm can only guarantee that the cycles between multiple processors can be \textit{eventually} detected, but does not have a guarantee on the detection latency. This property does not affect the correctness because, when a cycle is formed, not detecting it will only cause the deadlock, but the commit order requirements can never be violated. The key insight is the commit order requirement is not transitive and is recorded in both the source and destination.

Second, when a cycle is detected, it is favorable to have a deterministic policy to decide the chunk needs to squash, otherwise, more chunks than necessary may be squashed. Based on our algorithm, we use the following simple policy to decide the chunk to squash. For two-processor cycles, the local processor compares its own processor ID with the overlapped processor ID in \textit{Pred\_Set} and \textit{Succ\_Set}, the chunk in the processor with smaller ID is squashed. For the example in Figure 5.6(a), P₀’s chunk is squashed. Note that both processors involved in the cycle can make the same decision if the policy is predetermined.

For more-processor cycles, our algorithm has the property that each processor involved in the cycle eventually has the whole set of processors in the cycle, we can similarly require that the chunk in the processor with the smallest ID in the set is squashed. For the example in Figure 5.6(b),
although all the processors will have three processors in their $Pred_{Set}$, only $P_0$ will be squashed.

5.5.6 Interactions Between Static and Dynamic Chunks

When a processor encounters a static chunk, it is possible that the processor is already in a dynamic chunk and the chunk hasn’t committed. In this scenario, the processor should stall before the static chunk and let the previous dynamic chunk to commit first. This policy will avoid the unnecessary squashes of the static chunk due to the conflicts with the tail of unfinished dynamic chunk. Similarly, at the end of a static chunk, the processor should stall and commit the static chunk before executing the following instructions. In summary, the processor should place a conceptual fence before and after a static chunk to reduce the unnecessary squashes.

5.6 Evaluation

5.6.1 Evaluation Setup

In the evaluation, we want to understand three problems. First, the performance comparisons of all hardware SC schemes. Second, how does UniBlock work with static chunks? Finally, how does UniBlock work with the continuous chunk execution?

<table>
<thead>
<tr>
<th>Name</th>
<th>Protocol Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive-SC</td>
<td>Naive SC without speculation</td>
</tr>
<tr>
<td>SC</td>
<td>SC with in-window speculation</td>
</tr>
<tr>
<td>TSO</td>
<td>Total-Store-Order</td>
</tr>
<tr>
<td>RC</td>
<td>Release Consistency</td>
</tr>
<tr>
<td>CO</td>
<td>Conflict Ordering [38]</td>
</tr>
<tr>
<td>ENDSC</td>
<td>&quot;End-To-End SC&quot; [67]</td>
</tr>
<tr>
<td>IF</td>
<td>InvisiFence [6] applied to RC</td>
</tr>
<tr>
<td>IF_COV</td>
<td>IF with Commit-On-Violation</td>
</tr>
<tr>
<td>UB</td>
<td>UniBlock applied to RC</td>
</tr>
</tbody>
</table>

Table 5.1: Simulated SC Hardware Designs.

To evaluate hardware SC, we implemented seven SC designs, RC and TSO in SESC [63] summarized in Table 5.1. The designs are evaluated using 12 applications from Splash-2 and 4
applications from Parsec. Since our simulator does not run an operating system, we use the static address space partition to model the effect of paging in END_SC [67], we use 16KB page size. To evaluate the static chunk execution, we use STAMP [47]. We do not have a compiler that can do optimization and generate the chunk yet, instead, we consider the transactions in STAMP applications as the static chunk and execute the codes outside the transactions by dynamic chunk. The transactions are larger than the dynamic chunks, this could in some extent model the effects of static chunk. To execute STAMP, we change the SESC to enable the transaction squash and re-execution in functional model. On transaction overflow, the transaction is re-executed by acquiring the lock when no other processors are in the transaction, so that the processor can cut the chunk when the overflow happens again. The code version with lock is considered to be the safe version. Finally, we compare the UniBlock with InvisiFence in a continuous chunk mode (InvisiFence-Continuous mode [6]), we use the chunk size of 1000 dynamic instructions. Table 5.2 shows the modeled machine configurations, all the runs are in 64 processors.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Multicore chip 64 cores.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core width</td>
<td>4-issue. pipeline</td>
</tr>
<tr>
<td>ROB</td>
<td>176 entries.</td>
</tr>
<tr>
<td>L0 Cache</td>
<td>8 KB</td>
</tr>
<tr>
<td>Store Buffer</td>
<td>32 entries.</td>
</tr>
<tr>
<td>SVB</td>
<td>128 entries</td>
</tr>
<tr>
<td>SDT</td>
<td>32 entries</td>
</tr>
<tr>
<td>Priv. L1 cache</td>
<td>64KB, 4-way asso., 2-cycle round trip.</td>
</tr>
<tr>
<td>Shar. L2 cache</td>
<td>256KB per-bank, 64 banks, 8-way asso., 11-cycle round trip.</td>
</tr>
<tr>
<td>Cache line size</td>
<td>32 Bytes.</td>
</tr>
<tr>
<td>Coherence</td>
<td>Directory-based MESI protocol.</td>
</tr>
<tr>
<td>Main memory</td>
<td>200-cycle round trip.</td>
</tr>
</tbody>
</table>

Table 5.2: Architecture Parameters.

5.6.2 Hardware SC Performance
Figure 5.7: Performance Comparison of SC Hardware.
Figure 5.7 shows the performance of seven SC scheme and RC/TSO. For each application, we report the results for all the nine simulated models. The bars are normalized to the performance of SC (SC with in-window speculation). For each scheme, the execution time is broken into the following categories: cycles retiring instructions (Useful), stalled due to pipeline hazards (Pipeline), stalled due to memory accesses (Memory), stalled due to store buffer full (SBFull), squashed cycles (Squashed), stalled due to non-empty store buffer (SBNonEmpty).

For all applications, we see that the in-window speculation can improve the performance significantly compared with Naive SC. The TSO and RC memory model progressively improves the performance. In nearly all TSO hardware, we see visible portion of store buffer full. The more the execution is stalled due to store buffer full in TSO, the more RC can improve the performance.

Next, we consider two optimized schemes with only in-window speculation. For CO, eight applications (Blackscholes, Cholesky,FFT,Fluidanimate,Radix, Swaptions,Water-ns,Water-sp) show better performance than the plain SC with in-window speculation (SC). The performance gain is because the instructions do not have to wait for all to pending stores to complete. However, we see considerable squashes in several applications, they are Barnes, FFT, Ocean, Radiosity, Raytrace and Volrend. In FFT, despite the squash, the performance is still slightly better than TSO but worse than RC. In the other applications, the squash even make the performance a little worse than SC. We found that it is due to the fast-fetched pending list forces some load hits to replay after invalidating the cache line. Such replay is not always necessary, since the remote stores can be ordered after the local load hits without violating SC. This lead us to believe that the eagerly fetched pending list has both positive and negative impact on performance. While it can reduce the stall due to non-empty store buffer, it may also incur unnecessary squashes, especially for the high contention region. For END_SC, it can achieve better performance than SC on 4 applications (Cholesky,Fluidanimate,Ocean,Water-ns). However, for the other ones, it is almost the same as SC. The visible portion of SBNonEmpty is due to the coarse-grained access classification on page level.

Next, we consider the speculative schemes. We see that for most applications (except Barnes,
Ocean, Raytrace), the performance of IF and UB are very close and similar to RC. The behavior is reasonable, since in these applications, there is nearly no squashes due to conflict, therefore, UB cannot perform much better than IF. For these three applications, IF incurs visible amount of squashes, COV technique can reduce the squashes by certain amount but cannot remove all. UB can practically remove all the squashes in these three applications.

Overall, by comparing all the hardware SC schemes, we see that there is still a gap between the performance of non-speculative and speculative schemes. For the applications with little conflicts, both IF and UB can achieve performance similar to RC. There are several cases where IF incurs relatively large amount of squashes, UB is able to eliminate nearly all squashes in these cases.
### 5.6.3 Static Chunk Performance

Figure 5.8 compares the performance of IF, IF_COV and UB on STAMP benchmark. The transactions are considered to be static chunk. We see that all the three schemes incur little squashes and perform similarly in three applications (Genome, Ssca2 and Vacation). They either have quite small transactions or the contention is very low. For applications (Intruder, Kmean and Yada), we see a mediate amount of squashes in the IF and IF_COV, but UB can remove most of the squashes. Intruder implements a network intrusion detection algorithm. The three transactions that capture, reassemble and move the network packets have a relatively regular producer-consumer pattern, which can benefit from UB’s concurrent transaction execution supports. Kmeans algorithm groups
objects in an N-dimensional space into K clusters. The transaction is used to protect the update of the cluster center that occurs during iterations. The amount of squash is determined by the chance that two threads concurrently operate on the same cluster center. Such operation is occasional, therefore, the squash is not very high in IF. UB can remove nearly all the squashes because the transactions are small and the conflicts are often in one direction. For two applications (Bayes and Labyrinth), all schemes observe large portion of squashes, the ability of UB in squash avoidance is limited. It is due to the high contention and irregular communication pattern between the transactions. Bayes implements an algorithm for learning the structure of Bayesian networks from observed data. The transaction is used to protect the calculation and addition of new dependencies. Because calculations of new dependencies take most of the execution time, this application spends long time in transactional execution. The conflict is high because the subgraphs change frequently and the pattern is irregular.

Overall, for transactions (which mimic the static chunks), the UB can reduce considerable amount of squashes for all several applications in STAMP. On average, UB performs better than IF and IF_COV by 11% and 8% respectively. It is not very effective in handling high contention and irregular communication pattern. In reality, we expect that the static chunks generated by compilers are smaller than the transactions in some STAMP benchmarks.
5.6.4 Continuous Chunk Performance

Figure 5.9: Performance Comparison of InvisiFence and UniBlock with Continuous Chunks.

Figure 5.9 shows the performance of IF, IF_COV and UB using continuous-chunk execution mode. Each processor commits a chunk every 1000 dynamic instructions. The larger fixed-size continuous chunk will increase the conflicts compared with the selective dynamic chunk mode. We see that UB can reduce the squashes effectively for a number of applications (Barnes, FFT, FMM, Ocean, Radiosity, Raytrace, Volrend, Water-ns and Water-sp). It is because a majority of the conflicts are in single direction. UB still has the squashes for some applications, where the cyclic dependence is formed. On squashing the chunks to remove the cycle, UB causes the squash dependence only for RAW, we found that this property reduces the squash cost significantly. On average, UB performs better than IF and IF_COV by 17% and 11%, respectively.
5.6.5 Speculative Data Forwarding

Figure 5.10: Average Speculative Dependences.

Figure 5.10 shows the average of total number of dependences established from a finally committed ordered chunk. The results are from the continuous chunk mode. We can see that for most of the applications, the average number of dependences is less than 20. It implies that the size requirement for SVB and SDT are very small. Using a 128-entry SWB and 32-entry SDT, we did not see any overflow case.
5.6.6 Bandwidth Analysis

Figure 5.11 shows the bandwidth break down and message overhead of UniBlock. \textit{Fwd} shows the coherence overhead due to data forwarding in UniBlock. Each bar is normalized to the sum of three other portions of bandwidth consumption. \textit{MemAcc} is the memory accesses, \textit{Read} and \textit{Write} are the accesses through caches and the normal coherence overhead. We see that in all applications, the extra bandwidth consumption by UniBlock is very small. In comparison, the applications with more conflicts have more overhead. On average, the overhead due to UniBlock is 7%.

5.7 Conclusion

This chapter proposes UniBlock, the first \textit{unified} approach to support whole-system SC using a single set of mechanisms. The central concept in UniBlock, the Ordered Chunk, is used as both
the mechanism to implement hardware SC and the specification for the compiler to guide the hardware for correct execution. Our solution is based on a conventional directory-based cache coherence protocol. Compared with previous hardware SC proposals, UniBlock delivers better performance by allowing the concurrent execution of dependent chunks. Since the system treats hardware-generated dynamic chunks and compiler-marked static chunks in a unified manner, UniBlock seamlessly and cleanly supports whole-system SC. UniBlock performs better than the best InvisiFence by 8% and 11% using STAMP benchmark and continuous chunk, respectively.
Chapter 6

Volition: Precise and Scalable Sequential Consistency Violation Detection

The previous chapters discuss techniques to enforce SC, which require certain changes to the coherence protocol. This chapter proposes a scheme that detects SC violations based on a conventional cache coherence protocol.

6.1 Introduction

When programmers write and debug applications with shared-memory threads, they intuitively assume the Sequential Consistency (SC) model. SC requires that the memory operations of a program appear to execute in some global sequence, as if the threads where multiplexed on a uniprocessor \cite{35}. In practice, however, processors and memory systems overlap, pipeline, and reorder the memory accesses of threads. As a result, the execution of a parallel program can violate SC.

As an example, consider Figure 6.1(a). Processor P\textsubscript{0} initializes variable \( p \) and then sets flag \( OK \); later, P\textsubscript{1} tests \( OK \) and, if it is set, uses \( p \). While the interleaving in Figure 6.1(a) produces the expected results, the interleaving in Figure 6.1(b) does not. Here, while the two writes in A\textsubscript{0} and A\textsubscript{1} are retired in order, they complete out of order: the first one after B\textsubscript{0} and B\textsubscript{1}, and the second one before B\textsubscript{0} and B\textsubscript{1}. In this interleaving, P\textsubscript{1} ends up using an uninitialized \( p \). This order is an SC Violation (SCV).
While this example is trivial, SCVs often appear under subtle thread interleavings and timing conditions, even in popular codes. For example, Muzahid et al. [51] discovered SCVs in the Pthread and Crypt libraries of glibc. SCVs are often found in double-checked locking constructs [64], some synchronization libraries, and code for lock-free data structures. In Section 6.3, we show a typical example of SCV.

From the hardware perspective, an SCV occurs only when multiple conditions are met. First, there needs to be two or more data races — e.g., the races on variables $p$ and $OK$ in Figure 6.1. Second, these races must overlap in time. Finally, the order of the references in these races has to form a cycle at runtime [66].

Specifically, for two threads, an SCV requires a pattern like that in Figure 6.2(a) where, if we follow program order, the two threads reference the same two variables in opposite orders, and each variable is written at least once. Moreover, the references in these two racing pairs have to form a cycle as shown in Figure 6.2(b) — where we have arbitrarily picked reads and writes. Specifically, A1 must occur before B0, and B1 must occur before A0. This what happens in Figure 6.1(b), where $y$ is $OK$ and $x$ is $p$. 

Figure 6.1: Example of an SC violation.
However, if the timing at runtime is such that at least one of the two dependence arrows occurs in the opposite direction, there is no SCV. For example, Figure 6.2(c) shows the case when A1 executes before B0, but A0 executes before B1. Since there is no cycle, SC is not violated. This case corresponds to the timing in Figure 6.1(a).

It is important to detect SCVs because, in virtually all cases, SCVs are programming mistakes — as shown in Figure 6.1(b), they are the result of memory-access orders that contradict a programmer’s intuition. In addition, given their subtlety, they can potentially cause great harm to the program without being obvious to the programmer. Finally, the programmer cannot reproduce them using a single-stepping debugger, and has to largely rely on mental analyses of interleavings to uncover them.

Most prior work has attempted to find SCVs by focusing on detecting data races (e.g., [6, 15, 24, 27, 40, 43, 76]). However, using data races as proxies for SCVs is very imprecise. As discussed above, the specific race pattern and interleaving required for an SCV is not necessarily common. In large codes, race-detection tools typically flag a very large number of data races, often causing the programmer to spend time examining races that are much less likely to cause code malfunctioning than SCVs [22, 52].
A second reason for not using data races as proxies is that we may want to uncover SCVs in codes that have intentional data races — perhaps in lock-free data structures. We may want to debug such codes for SCVs, while being less concerned about non-SC-violating races. Here, a race-detection tool would not be a good instrument to use. If we want to detect SCVs, we need to precisely zero-in on the data races and interleavings that cause them.

Given the importance of these bugs and the difficulty in isolating them, there have been two recent proposals for hardware-supported detection of data-race cycles \cite{Lin2018,Muzahid2019}. The first one, by Lin et al. \cite{Lin2018}, focuses on detecting overlapping data races, even if they involve disjoint sets of processors. Hence, the approach is fairly conservative, resulting in false positives. However, false positives are not a problem because the goal of that approach is to avoid SCVs (by flushing the processor pipeline) rather than to detect them and report them to the programmer.

The second approach, by Muzahid et al. \cite{Muzahid2019} detects cycles that cause SCVs, precisely. However, it is only designed to work for two-processor cycles and relies on a broadcast-based cache coherence protocol in the machine. We compare our work to these two approaches in Section 6.3.

6.2 Contribution

In this section, we advance the state of the art by proposing the first hardware scheme that detects SCVs in a relaxed-consistency machine precisely, in a scalable manner, and for an arbitrary number of processors in the cycle. We call our scheme Volition. Volition leverages cache coherence protocol transactions to dynamically detect cycles in memory-access orders across threads. When a cycle is about to occur, an exception is triggered, providing information to debug the SCV. Volition can be used in both directory- and snoopy-based coherence protocols; it does not rely on any property of snoopy protocols such as the broadcast ability.

The current Volition design does not consider speculative loads from mispredicted branch paths. In addition, it is unconcerned with SCVs due to compiler transformations; it only reports SCVs due to hardware-initiated access reordering. Within these constraints, and with large-enough
hardware structures, Volition suffers neither false positives nor false negatives for a given execution.

6.3 Background & Related Work

An SCV occurs when the memory accesses of a program have executed in an order that does not conform to any SC interleaving. It is virtually always a programming mistake, since it involves an unintuitive interleaving. Given its subtlety, an SCV can potentially cause great damage to the program and not be obvious to the programmer. Finally, an SCV cannot be reproduced using a single-stepping debugger, and has to be identified with mental analyses of possible interleavings.

Shasha and Snir [66] showed what causes an SCV: overlapping data races where the dependences end up ordered in a cycle. Recall that a data race occurs when two threads access the same memory location without an intervening synchronization and at least one is writing. Figure 6.2 showed the required program pattern and order of dependences at runtime for two threads. We arbitrarily assigned reads and writes to the references.

An SCV is avoided by placing one fence instruction between the two references that participate in the cycle in each thread. For example, in Figure 6.2, we need a fence between A0 and A1, and another between B0 and B1. The algorithm that finds where to put the fences is called the Delay Set [66].

SCVs are very subtle. A major source of SCVs is the commonly-used Double-Checked Locking (DCL) [64]. This is a programming technique to reduce the overhead of acquiring a lock by first testing the locking criterion without actually acquiring the lock. Only if the test indicates that locking is required does the actual locking logic proceed. Figure 6.3(a) shows a DCL example. The code checks variable \( x \) in access B0 and, if it is not null, it reads its field \( x \rightarrow m \) in B1. If, instead, \( x \) is null, we grab a lock, check again and, if \( x \) is null, allocate a new object and assign it to \( x \) in A1. As part of the constructor, in A0, the field of the object is initialized.
B0: if(x == NULL){
    lock()
    if(x == NULL){
A0,A1:   x = new Object()
                /*initializes Object->m*/
                }
    unlock()
}
B1: .. = x->m

pthread_cancel_init(){
B0:      if(libgcc_s_getcfa != NULL)
return
A0:      libgcc_s_resume = fence
A1:      libgcc_s_getcfa =
B0: if(libgcc_s_getcfa != NULL)
B1:     libgcc_s_resume()

_Unwind.Resume(){
    pthread_cancel_init()
B1:       libgcc_s_resume()

(a)

(b)

(c)

Figure 6.3: SCV uncovered by Muzahid et al. [51] in the Pthread library.

The DCL code has a structure like in Figure 6.1(a), with equivalent A0, A1, B0, and B1 references. In Figure 6.3(a), A0 sets the field Object->m, and then A1 assigns Object to x. Unfortunately, the updates of the accesses A0 and A1 can get reordered. This causes the same problem as in Figure 6.1(b). According to our discussion, to guarantee SC execution, the software needs to place a fence between A0 and A1, and another between B0 and B1. Unfortunately, because the code is typically complicated, such fences end up occasionally missing.

As an example, Muzahid et al. [51] found one of such fences missing in the Pthread and Crypt libraries of glibc. The Pthread code is shown in Figure 6.3(b). It shows two subroutines that
construct a DCL pattern. We mark the references A0, A1, B0, and B1. We see that the code has a fence between A0 and A1, but not between B0 and B1. Since the second fence is missing, an erroneous reorder with an SCV can happen. This is shown in Figure 6.3(c), where we picked the four relevant references. The SCV occurs when the condition in B0 is predicted true by the branch predictor (although it is currently false) and B1 is executed before A0. After A0 and A1 execute, the B0 branch resolves, confirming that B1 is in the correct path. However, B1 used the old value and the code crashes. To fix this, we put a fence between B0 and B1.

Given the importance of SCVs, there has been significant work in this area. We discuss related work in architecture, compilation, testing, and hardware verification.

In architecture, the most related work is Vulcan by Muzahid et al. [51] and Conflict Ordering (CO) by Lin et al. [38]. Both works are based on identifying SCVs in hardware using Shasha and Snir [66] delay sets.

Vulcan [51] is the most similar work to Volition. It is a hardware scheme to detect SCVs at runtime, in programs running on a relaxed-consistency machine. It also leverages the cache coherence transactions to detect dependence cycles between processors and, from there, SCVs. It also supports multiple-word cache lines. While it has similar metadata structures as Volition, it works differently. It relies on a snoopy-based coherence protocol. Moreover, the design presented only operates with 2-processor SCVs. With Volition, we have taken a different approach, focusing on scalability and on handling SCV cycles with an arbitrary number of processors. The resulting Volition design is scalable, as it works with a scalable directory-based cache-coherence protocol and its hardware does not need all-to-all structures. In addition, it works seamlessly for any number of processors in the SCV cycle.

CO [38] is a technique that detects upcoming SCVs and enforces SC in a relaxed-consistency machine. As a potential SCV is about to occur, CO avoids it by squashing and replaying certain instructions. Although it is an SC enforcement scheme, it can be used as an SCV detection scheme if it reports the SCV when the replay is needed to retain SC semantics. However, CO has substantial false positives, which are fine in an SC enforcement approach but not in an SCV detection
scenario. To see why, consider Figure 6.4. Initially, there is a race between $P_0$ and $P_1$ on variable $x$. When CO sees this, it gets from the directory the set of pending writes. In this example, it gets the writes to variables $y$ and $z$. If $P_1$ then tries to access $y$ or $z$, CO conservatively assumes an SCV is about to occur, and causes a replay. Clearly, these dependences do not cause a cycle; we need a new dependence between $P_1$ and $P_0$ for a cycle.

![Figure 6.4: Operation of CO.](image)

CO also requires the serialization of some of the accesses from the same processor to operate correctly. Finally, it is unclear how CO works for a distributed directory design: since a processor gets the pending sets asynchronously from the directory modules, the information seen by different processors can easily become inconsistent.

Other work has focused on identifying data races as proxies for SCVs. However, data races and SCVs are very different, and programs have more data races than SCVs. Specifically, one line of work detects incoming coherence messages on data that has local outstanding loads or stores. This work includes that of Gharachorloo and Gibbons [24] and many aggressive speculative designs (e.g., [6, 15, 27, 76]). Another line of work detects a conflict between two concurrent synchronization-free regions. This includes DRFx [43] and Conflict Exceptions [40]. In general, all of these works look for a data race with two accesses that occur within a short time — but still, only a single race. Overall, while focusing on these races may be a good way to discard many irrelevant ones, it is still a very different problem than focusing on uncovering SCVs.

There are compiler techniques to identify race pairs that could cause SCVs, typically using the
Delay Set algorithm, and then insert fences to prevent cycles (e.g., [23 33 36 69]). They are conservative because they only use static information, and typically cause large slowdowns. Lin et al. [37] can hide some of the resulting fence delay with architectural support. Duan et al. [21] use a race detector to construct a graph of races dynamically. Then, off-line, they traverse the graph to find potential SCVs. Our work differs in that: (1) it is an on-the-fly scheme, while Duan’s SCV detection is off-line; (2) it needs no software support; and (3) it has no false positives, while Duan’s scheme may point to SCVs that never occur.

The software testing community has proposed static and off-line techniques to check for SCVs (e.g., [11 12 13]). While promising, these techniques are not designed for on-the-fly SCV detection in large codes with negligible overhead. The hardware verification community has designed techniques to verify if a memory system hardware is correctly implemented (e.g., [17 19 45]). While related, these works have a different goal: we focus on debugging software as it runs on a relaxed-consistent machine; they focus on verifying that the hardware correctly implements a memory model.

6.4 Volition: Scalable and Precise SCV Detection

In this section, we present the basic design of Volition. For now, we assume a cache line size equal to the granularity of processor accesses — e.g., one word. In Section 6.5, we extend the design to support multi-word cache lines.

6.4.1 Design Goals and Basic Assumptions

We are interested in an always-on hardware monitoring scheme usable for production runs. The scheme should detect all the SCVs that occur in the current dynamic execution, rather than attempting to find all the potential SCVs in the code. As a result, for a given binary, the scheme may detect different SCVs in different runs on the same machine and on different machines. Based on this usage model, an ideal SCV detection scheme has four traits.
**1. Precise.** The scheme should report SCVs, not conservative estimates of SCVs such as data races. Moreover, it should have no false positives or false negatives for a particular run.

**2. Scalable.** The scheme should not rely on any specific property of snoopy coherence protocols, which have limited scalability. It should be applicable to both snoopy- and directory-based protocols. In addition, the size of the hardware structures should increase only slowly with the processor count.

**3. Low overhead.** The scheme should have low overhead in terms of execution time and network bandwidth consumption.

**4. Decoupled from the coherence protocol.** Since the coherence protocol is difficult to design and verify, the SCV detection scheme should be decoupled from it.

In our design, we assume a multicore with directory-based coherence (although our scheme can also work with a snoopy-based protocol) and a relaxed memory consistency model, such as RC or TSO. We assume an MSI coherence protocol, which has a clean state (S, the line is coherent with memory and can be in multiple caches) and a dirty state (D, the line is not coherent and can be in only one cache). Each core is an out-of-order superscalar. For stores, the value is only written to the cache after retirement. Retired stores are held in the store buffer while they are being globally performed. They enter the store buffer in program order but may update the cache out-of-order. Reads can get their value before retiring. When an SCV is detected, an exception is raised.

The compiler can itself induce SCVs with certain optimizations [75]. However, Volition is a pure hardware scheme and, as a result, is not able to detect those. Hence, in this paper, we assume that the compiler does not perform SCV-inducing transformations, and we are only concerned with hardware-induced reorderings that cause SCVs. We leave the problem of preventing SCVs cooperatively by the compiler and hardware as future work.

**6.4.2 Basic Insight to Identify an SCV**

Volition detects an SCV by continuously trying to identify the pattern of Figure 6.2(b) across two or more processors. Such a pattern involves an interaction between processors on different
memory addresses. In general, it is challenging to identify such a pattern, since the information is distributed. However, the insight of Volition is to start by first detecting a special access pattern that is a necessary condition for an SCV to subsequently occur. Such a pattern is called Suspicious Pattern (SP). It can be detected locally within a processor and inexpensively. When a processor detects an SP, it piggybacks a small amount of local state information with the response coherence message. If a true SCV eventually occurs, the processors involved in the cycle will detect it. Before describing our solution, we define some terms.

**Completion of a memory operation.** A load completes when it gets its value from the memory system and no store from any processor can alter it; a store completes when its value updates the memory system and no load from any processor can return the value before the store. In relaxed memory consistency, memory operations from a thread can complete out of order.

**Active access.** A memory access \( A \) is Active when either itself or an older local access (according to program order) have not completed, or they are the destination of a data dependence from a remote access that is still active. For example, in Figure 6.5(a), B1 is active while B1 is not completed, or B0 is not completed, or (following the A1→B0 dependence) A1 is active.

![Figure 6.5: Basic ideas in Volition.](image-url)
Pending set (P-Set). The P-Set of an access is the set of older local accesses that are still active.

Active Data Race (AR). An AR is a data race where the source access is active. We refer to the source and destination processors of an AR as AR\textsubscript{s} and AR\textsubscript{d}, respectively.

Suspicious Pattern (SP). A processor detects an SP when the processor is the source of an AR. The processor identifies the SP locally, when it responds to a coherence event (i.e., it provides and/or invalidates a cache line). If the local access involved in the transaction is active, then an SP is identified. For example, in Figure 6.5(a), assume that A1 is completed and A0 is not. Hence, A1 is active. When the dependence A1→B0 occurs, it is an AR and, therefore, P\textsubscript{0} detects an SP. As a result, in the response coherence message, AR\textsubscript{s} piggy-backs some information that AR\textsubscript{d} will need to detect an SCV, if it ever occurs. When A1 ceases to be active, then the SP is considered expired. At that point AR\textsubscript{s} informs AR\textsubscript{d}.

SCV pattern. An SCV occurs when multiple ARs form a cycle across two or more processors. Figure 6.5(a) shows a two-AR cycle.

Vulnerability Window (VW). Given an AR, the VW is the global physical time period during which an SCV is possible. In the AR\textsubscript{s} processor, the VW is from the time when it identifies the SP until when the source access of the AR becomes inactive. In AR\textsubscript{d}, the VW is from the time when it is notified about the SP (by AR\textsubscript{s}) to when it receives the SP expiration notification by AR\textsubscript{s}.

Volition works as follows. When an AR occurs, AR\textsubscript{s} includes in its coherence response to AR\textsubscript{d} some information about the P-Set of the AR’s source access. Then, AR\textsubscript{d} checks its local state against the information received, to flag if this AR closes a cycle. When the SP induced by this AR expires, AR\textsubscript{s} notifies AR\textsubscript{d} via a small SP\_expire message so that AR\textsubscript{d} no longer tries to check for a cycle. Figure 6.5(b) shows a timeline of the VWs and SP\_expire message.

6.4.3 Volition Hardware Structures

To support the algorithm described, we need structures to (i) represent the local execution state, and to (ii) detect and record ARs.
**Representing the Local Execution State**

In Volition, each processor assigns a monotonically increasing *Sequence Number (SN)* to every memory access as it is issued. The SN reflects the order of local accesses in program order. If the access produces a network transaction, its SN is piggy-backed in the request message. With the proper way to handle the occasional wrap-around of SN (Section 6.6), the SN does not need to be very long.

The main hardware structure in Volition is the per-processor *Active Table (ACT)*. The ACT is in the core, and maintains state for all of the local accesses that are currently active. An ACT entry is allocated for each access in program order as it is issued. When an access ceases to be active and is the oldest access in the ACT, it is deallocated. At any time, the ACT may contain some entries that are completed and some that are not.

The goal of the ACT is to help record ARs. As shown in Figure 6.6(a), the ACT entry for an access contains its SN, the address of the location it loads or stores (*Addr*), a bit to specify whether the access is completed (*C*) and a writeback bit (*wb*). The functionality of *wb* is described later. The granularity of *Addr* (byte, half-word, word, etc.) depends on the granularity of the access. In our evaluation, we will assume word granularity only.

![Figure 6.6: Hardware structures in Volition.](image)

157
Detecting and Recording ARs

A processor detects an AR as follows. When it receives a request from the network, it checks the ACT for a race. We will later see how to avoid most of the unnecessary checks. The ACT is scanned from younger to older access, trying to find the first completed local access to the address of the request. If such an entry $E$ is found, we have detected an AR. The $P$-Set is the set of accesses in the ACT that precede $E$. However, the $P$-Set is effectively encoded with the SN of $E$, which we represent as $SN_s$.

If an AR is found, the Volition hardware performs two actions. First, it appends the information in Figure 6.6(b) to the response coherence message. Such information is a bit (SP) to indicate that this is an SP, the SN of the source of the AR ($SN_s$), the processor ID ($AR_s$), and the memory address of the dependence ($Addr$).

In addition, Volition records the AR information in a local table called the $AR$ Source Table ($ARST$). As shown in Figure 6.6(c), an ARST entry contains: the AR source’s SN ($SN_s$) and processor ID ($AR_s$), the AR destination’s SN ($SN_d$) and processor ID ($AR_d$), and the memory address ($Addr$). $AR_s$ obtained the values of $SN_d$ and $AR_d$ from the incoming message. We will explain later why we need to store $AR_s$: in a cycle with more than two processors, $AR_s$ may not be the ID of the local processor.

When the destination processor of the dependence receives a coherence response with SP=1, Volition records the AR information in a local table called the $AR$ Destination Table ($ARDT$). As shown in Figure 6.6(d), an ARDT entry contains: the AR source’s SN ($SN_s$) and processor ID ($AR_s$), the AR destination’s SN ($SN_d$) and the memory address ($Addr$). We do not need to store the processor ID of the AR destination because it is the local processor.

Ensuring Correct Monitoring for ARs

For Volition to work correctly, a processor $P$ with an ACT entry for address $Addr$ has to be able to see subsequent coherence transactions to $Addr$ that can cause ARs. Unfortunately, this is not
guaranteed without additional support. Specifically, consider a line in state Dirty (D) in P’s cache that is written back to the shared cache — either (i) because it is evicted from P’s cache or (ii) because another processor reads it. In the first case, P will not be sharer in the directory anymore and, therefore, will be unable to see future reads or writes to the line; in the second case, P will still be a sharer in the directory, but will be unable to see future reads.

To solve this problem, when P writes back a D line for which it has ACT entries, Volition allocates an entry in the directory’s AR Table (ART). As shown in Figure 6.6(e), the entry contains the processor ID (ARs) and the line address (Addr). In addition, P sets the wb bit in its youngest ACT entry for Addr.

From then on, when reads to the Addr by other processors reach the directory, the directory will read the ART entry and inform P. P will check its ACT and possibly send a message like the one in Figure 6.6(b) to the reader, informing it of an AR. Similarly, when the first write to Addr by another processor reaches the directory, the directory will read the ART entry and add P to the list of sharers that need to be notified. The ART entry will then be removed. The sharers (including P) may send messages like the one in Figure 6.6(b) to the writer if they find ARs.

When the entry in P’s ACT that had the wb bit set becomes inactive, P sends an SP_expire message to the processors it has informed of ARs, and to the directory. The latter deallocates the ART entry if it still exists.

Note that if P evicts a clean shared (S) line from its cache, it requires no action. The reason is that the directory is not updated, and still records P as a sharer.

**Table Operations**

Table 6.1 shows how the tables described are used. Specifically, for each of the ACT, ARST, ARDT, and ART, the table shows: (i) the condition for inserting an entry, (ii) the actions when an entry is inserted, (iii) the condition for deleting an entry, and (iv) the actions when an entry is deleted.

In the ACT, an entry is inserted when a memory instruction is issued. An entry is removed only
### Table 6.1: Table operations in Volition.

when it satisfies the following three conditions: it is at the head of the ACT, its access is completed, and its access is not the destination of any AR. The latter means that its SN is not the SN of any local ARDT entry.

When an entry is deleted from the ACT, three actions need to be taken (Table 6.1). First, if the entry’s `wb` field is set, it means that the corresponding line in L1 had been written back and, therefore, an `SP_expire` message is now to be sent to the ART to deallocate the entry. Second, the ARST is checked for entries that need to be removed; these are the entries representing ARs whose source is the removed ACT entry. Their AR and SN are equal to the local processor ID and to the SN of the removed ACT entry, respectively. Finally, Volition tries to delete the new entry at the head of the ACT, repeating the process above.
In the ARST, an entry is inserted when a new AR is detected whose source is a local access. In addition, when we discuss cycles with more than two processors (Section 6.4.5), we will see that we also allocate an ARST entry when an AR is propagated from another processor to the local one. In either case, when an ARST entry is allocated, Volition sends a message with SP=1 (format in Figure 6.6(b)) with information on the new AR to the AR_d processor.

An ARST entry representing an AR is deleted in two cases. One is when the source reference of the AR has been removed from the ACT, as discussed above. The other is when, in environments with cycles with more than two processors (Section 6.4.5), an SP_expire message for the AR is propagated from another processor to the local one. Finally, when an entry is deleted from the ARST, an SP_expire message for the AR is sent to the AR_d processor.

In the ARDT, an entry is inserted when the processor receives a message with SP=1 from the source of the AR. An entry is deleted when an SP_expire message for the AR is received. Finally, when an ARDT entry is deleted, Volition checks if the entry at the head of the ACT can now be removed; this will be possible if the reference at the ACT head is the destination of the removed AR and of no other existing AR.

We insert an entry in the ART in the directory module when a dirty cache line is written back from a cache whose processor has the line’s address in its ACT. Moreover, an entry is deleted from the ART when the directory receives (i) either an SP_expire message for the entry (ii) or a write request for the line in the entry.

Figure 6.7 repeats the information in the table for the ACT, ARST, and ARDT in the form of a state diagram. We use the diagram to describe the examples below.
Figure 6.7: State diagrams for insertion and deletion of table entries. The transitions with dashed lines will be discussed later.
Examples

To better understand the operations, we consider several example access streams in Figure 6.8. In the streams, \( w \) and \( r \) are writes and reads, and arrows are data races. Moreover, white, gray, and black circles indicate incompleted accesses, completed but active accesses, and inactive accesses, respectively.

In Figure 6.8(a), assume that \( P_1 \) issues \( r_1 \). This causes the insertion of an entry in \( P_1 \)’s ACT (edge (E1) in Figure 6.7(a)). Since \( r_1 \) reads the value produced by \( w_1 \) in \( P_0 \), and \( w_1 \) is active because \( w_0 \) is incompleted, \( P_0 \) detects an AR. Hence, \( P_0 \) inserts an entry in its ARST ((E2) in Figure 6.7(a)) and responds to \( P_1 \) with a message with SP=1 ((E3) in Figure 6.7(a)). When \( P_1 \) receives the message with SP=1, it inserts an entry in its own ARDT ((E4) in Figure 6.7(a)).

Later, as shown in Figure 6.8(b), assume that \( w_0 \) in \( P_0 \) has completed and been deleted from \( P_0 \)’s ACT. At this point, \( w_1 \) is completed and at the head of \( P_0 \)’s ACT. Since \( w_1 \) is not the destination of any AR (it has no associated entry in \( P_0 \)’s ARDT), it is deleted from \( P_0 \)’s ACT (edge (E5) in Figure 6.7(b)). Such removal causes the deletion of any entry in \( P_0 \)’s ARST whose source is the deleted ACT entry ((E6) in Figure 6.7(b)). Hence, in our example, we delete the ARST entry corresponding to the AR that goes from \( w_1 \) to \( r_1 \). After removing the ARST entry, \( P_0 \) sends an \( SP_{\text{expire}} \) message to \( P_1 \) ((E8) in Figure 6.7(b)). At the same time, \( P_0 \) also tries to remove its next
entry in the ACT, which is w2 ((E7) in Figure 6.7(b)). When \( P_1 \) receives the \( SP_{\text{expire}} \), it deletes the entry for this AR in its ARDT ((E9) in Figure 6.7(b)). Immediately after this, \( P_1 \) has to check its ACT — to see if the deletion of the ARDT entry makes the entry at the head of its ACT eligible for deletion ((E10) in Figure 6.7(b)). In the example, \( P_1 \) can remove r1’s entry in the ACT.

Figure 6.8(c) augments 6.8(b) with another processor (\( P_2 \)) that also had an AR whose destination is r1 in \( P_1 \). In this case, after the deletion of \( P_1 \’s \) ARDT entry for the \( w1 \rightarrow r1 \) AR, \( P_1 \) cannot yet remove r1’s entry from the ACT. It can only be deleted when both races have ceased to be active.

### 6.4.4 Detecting SCVs between Two Processors

An SCV between two processors occurs when there are two ARs in the opposite directions forming a cycle as in Figure 6.9. The cycle requires that, in each of the processors, the source access of the outgoing AR is equal to or younger than the destination access of the incoming AR. Hence, the condition for an SCV is determined in a processor locally by comparing the local ARST and ARDT. Specifically, we are looking for an entry \( arst \) in ARST and an entry \( ardt \) in ARDT that satisfy all of the following four conditions:

- The source processor in \( arst \) is the local processor. This is always the case based on our discus-
sion so far. However, it may not be the case in cycles with more than two processors (Section 6.4.5).

\[ \text{arst}[\text{AR}_{s}] = \text{Local}_{\text{PID}} \]

- The destination processor in \text{arst} is the same as the source processor in \text{ardt}.

\[ \text{arst}[\text{AR}_{d}] = \text{ardt}[\text{AR}_{s}] \]

- The SN of the source access in \text{arst} is equal to or larger than the SN of the destination access in \text{ardt}.

\[ \text{arst}[\text{SN}_{s}] \geq \text{ardt}[\text{SN}_{d}] \]

- The SN of the source access in \text{ardt} is equal to or larger than the SN of the destination access in \text{arst}.

\[ \text{ardt}[\text{SN}_{s}] \geq \text{arst}[\text{SN}_{d}] \]

Figure 6.9 shows the entries in \( P_0 \)'s ARST and ARDT for the example shown. We see that the entries satisfy the four conditions listed above. Specifically, from top to bottom, the conditions find: \( P_0, P_1, 115 \geq 102, \) and \( 120 \geq 101 \). For simplicity, Figure 6.9 does not show \( P_1 \)'s ARST and ARDT. Using such tables, the conditions are also show to be satisfied in \( P_1 \).

In each processor, the condition for SCV is locally checked every time that a new entry is added to its ARST or to its ARDT. Specifically, when a new entry is added to the ARST, it is checked against those currently in the ARDT, and vice-versa.

With this approach, when an SCV occurs, both processors detect it. Like in Vulcan [51], the timing of the detection depends on the relative timing of the ARs. If the two writes in Figure 6.9 complete at approximately the same time, both processors detect the SCV when their write transaction receives the response and causes the allocation of an ARDT entry. However, if one write (say \( W_y \) in Figure 6.9) is already completed by the time its processor receives the invalidation from the other write (at the point of \( R_x \) in the figure), then this processor (\( P_1 \)) detects the SCV as receives the invalidation. The other processor (\( P_0 \)) detects the SCV as it gets the invalidation acknowledgement.

In either case, when each processor detects the SCV, it raises an exception. As in Vulcan [51], the exception may not provide the exact architectural state at the point of the SCV-causing accesses.
Specifically, the information that is available to the debugger in the interrupted processor at the destination of an AR is the address being accessed, the instruction’s PC and the ID of the other processor. If the destination reference of the AR is a read, the exception gets the precise processor state. If it is write, it is not generally possible to get the precise state at the reference because the write is in the store buffer and later operations may have already retired and completed. The information available to the debugger in the interrupted processor at the source of the AR is the address accessed, the ID of the requesting processor, and if we augment the ACT with PCs, the instruction’s PC. The exception in the source processor is not precise because newer instructions may have finished.

Execution can potentially continue after reporting the SCV in the exception handlers. It requires that Volition explicitly remove one the ARs participating in the cycle, by sending an \textit{SP\_expire} for the AR to the AR’s destination processor. Otherwise, the four accesses involved in the SCV would remain active and no entry would ever be removed from the tables.

\section{Detecting SCVs Among Any Number of Processors}

In this section, we discuss the mechanism to detect SCVs involving an arbitrary number of processors. We start with some examples, describe the concept of AR propagation, and then define the conditions for an SCV.

\subsection{Motivating Examples}

Figure 6.10(a) shows an SCV involving three processors. The cycle is composed of active races AR0, AR1, and AR2. Although Volition can find the three ARs, our previous conditions for SCV cannot find the SCV.
Figure 6.10: Finding SCVs across more than two processors.

To be able to detect the SCV, we use the insight that two ARs transitivity imply another AR that combines them. For example, in Figure 6.10(a), ARs Wy→Ry and Rz→Wz transitivity imply Wy→Wz. If P₀ and P₂ have such information, they can easily detect the SCV.

As we attempt to transitivity combine ARs, we need consider the order of the local accesses of the ARs. Specifically, in a processor, the destination of one AR has to precede the source of the other AR. This is seen in Figure 6.10(b) for P₁. If the opposite is the case, as in P₁ in Figure 6.10(c), the two ARs cannot be combined.

Propagation of Active Races

To understand how Volition transitivity combines two ARs, Figure 6.11(a) shows two ARs with the SNs of their accesses. We name these ARs from the point of view of the processor that sees them both, namely Pⱼ: the Predecessor AR (ARₚᵢₖₑᵈ) is the one whose destination is in Pⱼ, and the Successor AR (ARₑᵤₓₑ) is the one whose source is in Pⱼ. Such terminology does not imply the relative time of when the ARs were identified. The goal of Volition is to generate ARₜʳₜᵢₙₜᵢₜ, the AR in dashes in Figure 6.11(b), which connects the source of the predecessor AR (SNᵢ) to the destination of the successor AR (SNₖ).
Figure 6.11: Propagating active races.

Figure 6.11(a) also shows that AR$_{pred}$ has an entry in P$_i$’s ARST and one in P$_j$’s ARDT. Similarly, AR$_{succ}$ has one in P$_j$’s ARST and one in P$_k$’s ARDT. Figure 6.11(b) shows how Volition will represent the new AR$_{trans}$: with a new entry in P$_j$’s ARST and one in P$_k$’s ARDT. They are shown as shaded. The new entries will combine information of P$_i$ and P$_k$, and contain no information on...
P_j, even though one of the entries is in P_j.

Specifically, the new entries for AR_{trans} are shown in Figure 6.11(c), together with those existing for AR_{succ} for comparison. Consider the ARST for AR_{trans}. It contains source information from AR_{pred} (SN_i and P_i) and destination information for AR_{succ} (SN_k and P_k); the address field is unused. Similarly, the ARDT for AR_{trans} contains source information from AR_{pred} (SN_i and P_i) and destination information for AR_{succ} (SN_k). P_k will be unable to distinguish transitive ARs from direct ones; P_j will distinguish transitive ARs because the source information is from another processor. It appears as if Volition had “propagated” the information from P_i to P_j. Hence, we refer to the operation of transitively combining two ARs as AR Propagation.

Table Operations

To propagate ARs, the state diagrams of Figure 6.7 are augmented with the three transitions with dashed lines. Consider insertion first. Assume that, in Figure 6.11(a), P_j has already recorded AR_{pred} and now it detects AR_{succ}. After inserting the usual ARST entry, P_j observes that P_j is the destination of an AR that can be transitively combined. Hence, it creates a new entry in its ARST and sends a second message with SP=1 to the destination of AR_{succ}. This message contains the information in Figure 6.11(c): SN_i, P_i, and SN_k). This operation is shown in edge (E12) in Figure 6.7(a).

Consider, instead, that in Figure 6.11(a), P_j has already recorded AR_{succ} and now it detects AR_{pred}. After inserting the usual ARDT entry, P_j observes that P_j is also the source of an AR that can be transitively combined. Therefore, it performs the same operations as described above. This operation is shown in edge (E11) in Figure 6.7(a).

Finally, consider removal. Among AR_{pred} and AR_{succ}, the one that must become inactive first is AR_{pred}. Hence, P_i sends an SP_expire message to P_j. After P_j deletes its ARDT entry corresponding to (P_i, SN_i), it now has to check for an entry in its ARST with the same source (P_i, SN_i). If it finds one, it is a propagated AR. Therefore, it deletes it and sends an SP_expire message for it to its destination, which is P_k. This operation is shown in edge (E13) in Figure 6.7(b).
reception of the $SP_{\text{expire}}$, $P_k$ removes its entry from ARDT.
Figure 6.12: Detecting an SCV with three processors.
**SCV Condition**

With the AR propagation operations, the SCV is detected when one of the created transitive ARs ends up having the same processor as source and destination. This event occurs when Volition creates a new ARDT entry in a processor, such that the AR source is also the current processor and the AR source SN is larger than the AR destination SN. Specifically, the new ARDT entry ardt is such that:

- The source processor in ardt is the local processor. Recall that there is no destination processor in ardt because it is always the local one.

  \[
  \text{ardt\{AR}_s\} = \text{Local\_PID}
  \]

- The SN of the source access in ardt is equal to or larger than the SN of the destination access in ardt.

  \[
  \text{ardt\{SN}_s\} \geq \text{ardt\{SN}_d\}
  \]

Figure 6.12 shows an example with three processors. Charts (a)-(f) show snapshots of the transitive ARs as they are generated; Chart (g) shows a timeline of events. Consider Chart (a), which corresponds to time \(t_0\). At this time, AR (1) and AR (2) have been used to generate transitive AR (i). The timeline in Chart (g) shows that, at time \(t_0\), processor \(P_1\) took predecessor AR (1) and successor AR (2) and generated AR (i), creating an ARST entry in \(P_1\) and an ARDT entry in \(P_2\).

Chart (b) shows that, at time \(t_1\), AR (3) is detected. There are now enough ARs to create a cycle. It will be uncovered by creating transitive ARs.

Specifically, Chart (c) corresponds to time \(t_2\) in \(P_0\), when \(P_0\) generates AR (ii). The timeline in Chart (g) shows that, at time \(t_2\), processor \(P_0\) took AR (3) and AR (1) and generated AR (ii), creating an ARST entry in \(P_0\) and an ARDT entry in \(P_1\). Chart (d) is also at the same logical time \(t_2\) in \(P_2\), when \(P_2\) generates AR (iv) and AR (v). As shown in Chart (g), at time \(t_2\), processor \(P_2\) took AR (2) and AR (3) and generated AR (iv), augmenting the ARST in \(P_2\) and the ARDT in \(P_0\). \(P_2\) also took AR (i) and AR (3) and generated AR (v), recording it in the ARST in \(P_2\) and the ARDT in \(P_0\). This ARDT entry causes the detection of the SCV in \(P_0\).
Potentially while this is taking place, Chart (e) shows time $t_3$ in $P_1$, where AR (iii) is created. Specifically, as shown in Chart (g), at time $t_3$, $P_1$ takes AR (ii) and AR (2) and generates AR (iii), recording it in the ARST in $P_1$ and the ARDT in $P_2$. This ARDT entry causes the detection of the SCV in $P_2$. Moreover, Chart (f) shows time $t_4$ in $P_0$, where (vi) is created. Specifically, as shown in Chart (g), at time $t_4$, processor $P_0$ takes AR (iv) and AR (1) and generates AR (vi), recording it in the ARST in $P_0$ and the ARDT in $P_1$. This ARDT entry causes the detection of the SCV in $P_1$.

When a processor detects the condition for the SCV, we envision Volition to trigger an exception. The processor then obtains the $SN_s$ and $SN_d$ from the offending ARDT entry. These are the local SNs of the two local accesses involved in the SCV. The processor can then read its ACT and obtain the addresses for these two accesses. Moreover, if the ACT is augmented with the program counters (PC) of the instructions, then it can also obtain the local PCs of the accesses.

From the example, we see that all of the processors involved in the cycle eventually detect the SCV. However, the actual timing and order is not deterministic. Hence, we can think of different usage modes for Volition. In one mode, as soon as the first processor detects the SCV, the processor dumps the addresses and PCs of the local accesses, and then stops all other processors. In a second mode, we let each processor involved in the SCV find the SCV, dump the information, and continue. With this approach, we will get a better picture of the SCV, but the Volition tables of the processors involved in the SCV will eventually fill up and the processors will stop. Finally, in a third mode, as soon as the first processor detects the SCV, it reports it in a log file and sends an $SP_{expire}$ for one of the ARs participating in the SCV. This will break the cycle and allow the processors to continue execution. This mode is attractive when the program runs in a non-interactive mode. The log can later be examined. However, it is not guaranteed that all the processors in the SCV will suffer an exception.

The approach described is applicable irrespectively of the number of processors participating in the SCV. In particular, it works in cycles with only two ARs, where one AR is arbitrarily chosen as predecessor and one successor. Hence, this approach supersedes the one in Section 6.4.4 which was presented to ease the explanation.
6.5 Supporting Multi-Word Cache Lines

6.5.1 The Problem

With single-word cache lines, every inter-processor dependence (not affected by cache displacements) induces a coherence transaction — which Volition uses for AR recording. In multi-word cache lines, the fact that all of the words in a line have to have the same state, may cause a simple design to miss some ARs (false negatives) or to falsely report some ARs (false positives). For example, in Figure 6.13 where a and b are in the same line, there is only one coherence action, at Wa. The Rb access is silently satisfied from the local cache. However, in reality, there are two races in the example. The opposite case, where there are no races but the protocol induces transactions, can occur due to false sharing. Hence, we must extend the Volition scheme of Section 6.4. In the rest of the discussion, we assume that, although the coherence protocol is line-based, coherence transactions include the address of the word accessed within the line.

![Figure 6.13: Missing an AR with multi-word cache lines.](image)

6.5.2 Approach: Metadata Transactions

To solve this problem, we use the general approach proposed in Vulcan [51]. It involves augmenting a cache line with some information on recent accesses performed by processors to each of the words in the line. Such information should be enough to tell a processor that is referencing the line...
whether or not it needs to check for ARs in other processors. If it needs to, but the protocol will not generate a coherence transaction, Volition triggers a *Metadata Transaction*. Such transaction checks and updates the Volition metadata in other processors, possibly recording ARs. However, it involves no data transfer or cache coherence transition. Hence, the cache coherence protocol is unmodified.

In Volition, the information that needs to be associated with a line in a cache is the set of accesses from any processor to any of the words in the line that are currently active — i.e., that are in the Active Table (ACT) of any processor. With this information, when a processor accesses a line, it can check, for the relevant word, whether any AR can be created. In particular, we need the following information for each word:

- The most recent active write (if any).
- The set of active reads (if any) that follow the most recent active write (if there is one) or that currently exist (if there is no active write).

This information is needed when a processor issues a read or a write to the word, to identify a potential AR. Note that all of the accesses before the most recent active write are irrelevant because they cannot source any future race.

Consequently, our basic design augments some of the cache lines with the information shown in Figure 6.14, which we call *Summary of Active Information* (SAI). For each word in the line, it contains: (i) the ID of the processor that has issued the most recent active write (if any), and (ii) the IDs of the processors that have issued the active reads (if any) that follow the most recent active write (if there is one) or that just currently exist (if there is no active write). Note that there is room for only a few readers. Hence, like in limited directory schemes [2], we keep space for only very few readers; if more are needed, we set a Broadcast bit. Since most of the words in a line are likely to need little or no information, the SAI should be encoded to use little space. The SAI has to travel with the line in the cache hierarchy and must be kept up to date.
Figure 6.14: Structure of an SAI associated with a cache line.

When a processor accesses a line in its cache that has an SAI, Volition checks the SAI for the word accessed. If it finds that there are active accesses to this word in other processors that can create an AR with the current access, it needs to communicate with such processors. The communication automatically happens if the current access causes a coherence transaction with such processors; it simply requires including the updated SAI. Otherwise, Volition explicitly triggers a metadata transaction with the updated SAI directed to the processors that can potentially generate ARs. The arrival of the coherence or metadata transaction at these processors triggers Volition operations there.

6.5.3 Basic Operation

To describe the operation of Volition, we initially assume that all the cached lines have SAI entries, and that there are no cache line evictions. These assumptions will be removed later.

With these assumptions, the main challenge of the design is how to keep a line’s SAI information correct. A SAI is associated with a copy of the line, and when such a copy is accessed, its SAI is updated with appropriate read or write information. If the multiprocessor cache hierarchy contained at most a single copy of a line, then keeping the SAI up to date would be easy. In reality, however, in the MSI protocol that we use in this paper (Section 6.4.1), a line repeatedly moves between a situation where there is a single (D or S) copy of the line in the system, and one where the caches have multiple S copies of the line that are identical.
Figure 6.15 shows the transition diagram for the system-wide state of a cache line in our protocol. When there is a single D or S copy of the line (leftmost circle in Figure 6.15), there also a single copy of the SAI. Such line may be accessed by the local processor or may receive an external write, in which case it moves to another cache, followed by its up-to-date SAI.

![Transition diagram for the system-wide state of a cache line.](image)

Figure 6.15: Transition diagram for the system-wide state of a cache line.

When an external read occurs, an extra copy of both the line and its SAI is made (rightmost circle in Figure 6.15). At this point, these two (or more) processors may repeatedly read words in the line. As they do so, each updates its local SAI copy, which starts to diverge from the other SAI copies. However, their divergence only consists of reader IDs, and can only induce RAW metadata transactions with the single recent-most writer of the word.

When one processor (among the sharers or otherwise) writes any of the words of the line, the state transitions back to the leftmost circle in Figure 6.15. As invalidation coherence messages are sent to all the sharers (which potentially record ARs), the current SAI copies are all returned to the writer. The writer accumulates the information from all of the SAI copies, and now keeps the single SAI entry for the line. The SAI information for the particular word written is reset to have a single writer ID, namely the writer processor, and no reader IDs.

We now consider several issues, namely which cache lines have SAI entries, how is the in-
formation removed from SAI entries as accesses become inactive, and the interaction with line eviction from caches.

6.5.4 Allocation of SAI Entries

SAI entries are only needed for lines that are being actively shared between processors. If a line is accessed by a single processor, it does not need an SAI. Even for a shared line, as accesses to it become inactive, Volition progressively removes information from its SAI; when the SAI contains no information, the SAI can be deallocated. Overall, at any given time, only very few lines in a multiprocessor cache hierarchy have SAI entries.

Specifically, as a processor misses on a line, if there are no sharers or the sharers do not have an SAI, then the line is read into the local cache without an SAI. Note that we also require that there is no information in the directory from a past eviction, as we will see in Section 6.5.6. If, instead, an SAI needs to be allocated, it is allocated in a small table in the cache controller called the Summary Active Table (SAT) (Figure 6.16).
As a processor accesses the line, Volition attempts to update the line’s SAI. However, it only does it if it finds one. As local accesses to the line become inactive, Volition attempts to remove information from the line’s SAI, if it exists. Section 6.5.5 discusses this operation in detail. As soon as the line’s SAI becomes empty, it is deallocated from the SAT.

Finally, there are situations when a processor with a line without a SAI needs to build a SAI for it. This occurs when the processor has active accesses to the line in its ACT, and either it receives a coherence transaction for the line, or its cache wants to evict the line. In these cases, Volition allocates the SAI, updates it with information on the current local active accesses, and includes it with the coherence response or with the evicted line.
6.5.5 Removal of Information as Accesses Become Inactive

As an active access (by a processor $P$) to a line becomes inactive, $P$’s information should be removed from the line’s SAI entries. This is because we do not want to keep querying $P$ for ARs anymore in any future access by other processors.

To see how information is removed from an SAI entry, consider four possible cases. The first, trivial case is when the line is in $P$’s cache but has no SAI; in this case, no action is taken. The second case is when the line state tells us that the line is present only in $P$’s cache; in this case, Volition simply removes $P$’s ID from the SAI’s write or the read area (for a write or read, respectively).

A third case occurs when the line is present in $P$’s cache and potentially in other caches as well. If $P$’s access is a read, its information is simply removed from the local SAI. It may be that other copies of the line also have the read’s information, which now becomes redundant. A subsequent write will trigger an unnecessary metadata transaction to $P$’s cache. However, to minimize overall traffic, Volition takes no further action now.

However, if the now-inactive access by $P$ is a write, both the local and all of the remote SAIs are updated. Volition removes the write PID from all of the SAIs to prevent future reads by other processors from initiating unnecessary metadata transactions to $P$’s cache. Hence, after Volition removes $P$’s ID from the local SAI, it initiates a metadata transaction to the directory, which is forwarded to all of the sharers of the line. All the SAI versions are updated.

Finally, the fourth case is when the line is not present in $P$’s cache — because it has been invalidated or it has been evicted. In this case, for both reads and writes, Volition initiates a metadata transaction to the directory, which is forwarded to all of the sharers of the line and removes $P$’s ID from all SAIs. This is done to avoid future unnecessary metadata transactions.
6.5.6 Cache Eviction of Lines with SAI Entries

When a line with SAI information is evicted from a cache, its SAI needs to be saved. The reason is that the SAI may contain unique information: the complete SAI information if this was the only cached copy of the line in the system, or unique reader PIDs otherwise. Hence, Volition stores the evicted SAI in a table in the directory controller called Directory SAT (DirSAT) (Figure 6.16). The DirSAT subsumes the ART structure in Section 6.4.3 which worked for single-word lines only.

Subsequent metadata transactions, as they reach the directory on their way to check for ARs in the relevant processors, they must read the DirSAT. If they find an SAI entry for the line accessed, they read its information (combining it with the information in the transaction’s own SAI). Similarly, messages from processors that indicate that a certain access has become inactive, as they reach the directory, they remove the relevant bits from the SAI entry in the DirSAT.

Multiple evictions of a line’s SAI from multiple caches simply accumulate their state in a single DirSAT entry. Moreover, when a line (with SAI) that is dirty in a processor is read by a second one, as the line and SAI are provided to the reader, the DirSAT also collects a copy of the SAI. The reason is that future read misses on the line will read directly from memory (since the line is not D in any cache), and they also need to obtain a copy of the SAI.

Eventually, as bits for inactive accesses are removed form an SAI entry in the DirSAT, the entry may lose all of its information and be deallocated. In addition, an SAI entry in the DirSAT is also deallocated on any write to the line. Specifically, as a write coherence transaction invalidates all copies of the line in the system and collects (and accumulates) all the SAIs for the line, it also collects and removes the SAI entry in the dirSAT.

Overall, we see that Volition manages multi-word cache lines without modifying the cache coherence protocol.
6.6 Implementation Issues

6.6.1 Wrap-Around of SNs

The wrap-around of SNs could confuse SCV detection because a number that is supposed to be comparable to another is now much smaller. While this problem occurs infrequently with our 4-Byte SN, we still need to handle it. Specifically, we use the most significant bit of the 4-Byte SN as a detector of wrap-around. When the Volition logic compares two SNs and finds that the most-significant bit of one has changed, it knows that it is a larger number that has wrapped. The range of SNs is large enough that a processor is very highly unlikely to wrap around a second time before all the other processors have wrapped around once. If this event appears to be possible, as the SN reaches a certain watermark, all processors are interrupted and the starting point of SNs is reset.

6.6.2 Reducing the Cost of ACT Scan

To avoid searching a processor’s ACT for every incoming coherence message, we follow Vulcan’s design [51] and use a counting Bloom filter (CBF) [10] to represent the current set of addresses in the ACT. If the incoming address does not hit in the CBF, we know the address is not in the ACT. We need to use a CBF because addresses need to be removed from the filter when they are deallocated from the ACT.

6.6.3 Region Expiration to Reduce Bandwidth

As discussed in Section 6.4.3, Volition causes a processor to send an $SP_{\text{expire}}$ message when a local AR expires. While ARs are not generally very common, we can optimize this operation and save some network bandwidth when there are many clustered ARs. Specifically, rather than sending a message to the destination processor ($AR_d$) with the AR’s SN$_a$ immediately when the AR expires, Volition can wait for a small time. At regular intervals, when a few ARs have expired,
it can multicast the $SP_{\text{expire}}$ message with the maximum of the expired ARs’ SNs to all of the ARs. If ARs cluster in time, and there are only very few different ARs, we can save bandwidth. In practice, for our applications, we do not find this optimization beneficial.

### 6.7 Discussion

The current Volition design largely attains the design goals of Section 6.4.1. First, Volition detects SCVs involving an arbitrary number of processors, in a precise manner, and with no false positives or false negatives. Of course, the hardware design has to be adapted to support the finest granularity of program accesses (e.g., bytes). The current design has assumed word-level accesses.

One limitation of the current design is that it does not consider speculative loads from mispredicted branch paths. To be able to support them, we need to extend Volition, possibly delaying the recording of an AR until the source load becomes non-speculative, and discarding an AR whose destination load is proven to be in the wrong path. We consider this extension to be our future work.

Note also that Volition is not concerned with the impact of compiler optimizations on SCVs. It simply takes the executable that the compiler provides to the hardware and reports SCVs due to hardware-initiated reference reordering. Similarly, since Volition is a dynamic scheme, it only provides information for the actual performed runs.

A second goal attained is to have a scalable design. Volition works with a scalable directory-based cache-coherence protocol. In addition, the size of its hardware structures increases only moderately with the processor count, and Volition does not need all-to-all structures.

A third goal is to have low overhead. As we show in Section 6.8.4, Volition incurs little execution time and bandwidth overhead.

The final goal is to be decoupled from the coherence protocol. The Volition hardware is substantial and often fairly involved. However, all of the additional messages used to manage the metadata for SCV detection are largely decoupled from the existing cache coherence protocol.
Hence, the coherence protocol should not have to be revalidated.

## 6.8 Results

### 6.8.1 Evaluation Setup

We focus the evaluation on three aspects, namely, (1) the ability of Volition to detect SCVs, (2) the characteristics of SCVs, and (3) the overhead and scalability of Volition.

We implement Volition in the SESC cycle-level architectural simulator. We model a multicore with 64 cores and an MSI directory-based cache coherence protocol. The hardware uses either the Release Consistency (RC) or the Total-Store-Order (TSO) memory consistency model. We use different store buffer sizes to see their impact on access reordering. For comparison, we also implement the Conflict Ordering scheme of Lin et al. [38]. We name it CO. Such scheme, as we describe in Section 6.3, enforces SC. When a potential SCV is about to occur, CO avoids it by squashing and replaying certain instructions. The configuration of the simulated machine is shown in Table 6.2.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Multicore chip with 64 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core width; ROB size</td>
<td>4-issue; 128 entries</td>
</tr>
<tr>
<td>Consistency</td>
<td>RC or TSO</td>
</tr>
<tr>
<td>Store buffer</td>
<td>32 entries</td>
</tr>
<tr>
<td>Private L1 cache</td>
<td>32KB WB, 4-way, 2-cycle round trip</td>
</tr>
<tr>
<td>Shared L2 cache</td>
<td>1MB module/proc. Module: WB, 8-way</td>
</tr>
<tr>
<td>Latency to L2</td>
<td>Local module: 11-cycle round trip</td>
</tr>
<tr>
<td>Cache line size</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Cache coherence</td>
<td>Directory-based MSI protocol</td>
</tr>
<tr>
<td>Network</td>
<td>2-D mesh with 7-cycle hop latency</td>
</tr>
<tr>
<td>Main memory</td>
<td>200-cycle round trip</td>
</tr>
<tr>
<td>Volition parameters</td>
<td>SN size: 4 bytes; ACT size: 256 entries</td>
</tr>
<tr>
<td></td>
<td>per-node ARST, ARDT: 40 entries each; per-node SAT: 100 entries</td>
</tr>
</tbody>
</table>

Table 6.2: Architecture parameters.

We run the applications shown in Table 6.3. They include several small codes with concurrent algorithms, a kernel with a double-checked lock (DCL), and SPLASH-2 and Parsec applications.
The codes with concurrent algorithms were mostly obtained from [1], which in turn comes from CheckFence [11]. They are small C-code programs where threads share data structures without synchronization, and rely on explicit fences to maintain correct access ordering. Many of these programs insert and remove elements from a linked list. In addition, we added two well-known algorithms for mutual exclusion, namely Dekker (which only runs with two threads) and Peterson.

<table>
<thead>
<tr>
<th>Set</th>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aharr</td>
<td>Variant of Harris</td>
<td></td>
</tr>
<tr>
<td>Dekker</td>
<td>Algorithm for 2 proc. mutual exclusion</td>
<td></td>
</tr>
<tr>
<td>Harris</td>
<td>Non-blocking set</td>
<td></td>
</tr>
<tr>
<td>Lazylist</td>
<td>List-based concurrent set</td>
<td></td>
</tr>
<tr>
<td>Moirbt</td>
<td>Non-blocking sync. primitives</td>
<td></td>
</tr>
<tr>
<td>Moircas</td>
<td>Non-blocking sync. primitives</td>
<td></td>
</tr>
<tr>
<td>Ms2</td>
<td>Two-lock queue</td>
<td></td>
</tr>
<tr>
<td>Msn</td>
<td>Non-blocking queue</td>
<td></td>
</tr>
<tr>
<td>Mst</td>
<td>Non-blocking queue</td>
<td></td>
</tr>
<tr>
<td>Peterson</td>
<td>Algorithm for N proc. mutual exclusion</td>
<td></td>
</tr>
<tr>
<td>Snark</td>
<td>Non-blocking double-ended queue</td>
<td></td>
</tr>
<tr>
<td>Bug</td>
<td>DCL</td>
<td>Double-checked lock without fence</td>
</tr>
<tr>
<td>Full</td>
<td>SPLASH-2</td>
<td>12 programs</td>
</tr>
<tr>
<td>Apps</td>
<td>Parsec</td>
<td>4 programs</td>
</tr>
</tbody>
</table>

Table 6.3: Applications executed.

We use the codes with the concurrent algorithms and DCL to check the ability of Volition to detect SCVs. Specifically, we explicitly remove all of their fences and run each code 100 times. We count the total number of SCVs detected. Note that many of the executions are now incorrect, but they help us understand Volition’s effectiveness. Finally, we use the SPLASH-2 and Parsec codes to evaluate the overheads and scalability of Volition.

### 6.8.2 Ability to Detect SCVs
<table>
<thead>
<tr>
<th>Appl.</th>
<th># of Runs</th>
<th>RC</th>
<th>TSO</th>
<th>CO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># SCVs</td>
<td># Line-SCVs</td>
<td># ARs</td>
</tr>
<tr>
<td>Aharr</td>
<td>100</td>
<td>166</td>
<td>20</td>
<td>4097</td>
</tr>
<tr>
<td>DCL</td>
<td>100</td>
<td>122</td>
<td>48</td>
<td>8529</td>
</tr>
<tr>
<td>Dekker</td>
<td>100</td>
<td>316</td>
<td>140</td>
<td>2653</td>
</tr>
<tr>
<td>Harris</td>
<td>100</td>
<td>123</td>
<td>56</td>
<td>11752</td>
</tr>
<tr>
<td>Lazylist</td>
<td>100</td>
<td>49</td>
<td>26</td>
<td>11688</td>
</tr>
<tr>
<td>Moirbt</td>
<td>100</td>
<td>127</td>
<td>37</td>
<td>9408</td>
</tr>
<tr>
<td>Moircas</td>
<td>100</td>
<td>90</td>
<td>41</td>
<td>9522</td>
</tr>
<tr>
<td>Ms2</td>
<td>100</td>
<td>98</td>
<td>93</td>
<td>26252</td>
</tr>
<tr>
<td>Msn</td>
<td>100</td>
<td>355</td>
<td>251</td>
<td>15277</td>
</tr>
<tr>
<td>Mst</td>
<td>100</td>
<td>2288</td>
<td>1048</td>
<td>110293</td>
</tr>
<tr>
<td>Peterson</td>
<td>100</td>
<td>31</td>
<td>23</td>
<td>2028</td>
</tr>
<tr>
<td>Snark</td>
<td>100</td>
<td>174</td>
<td>103</td>
<td>213082</td>
</tr>
<tr>
<td>Average</td>
<td>100</td>
<td>328</td>
<td>157</td>
<td>35381</td>
</tr>
</tbody>
</table>

Table 6.4: Volition’s ability to detect SCVs.
To assess Volition’s ability to detect SCVs, we take each of the small codes stripped of fences and run them 100 times. We use the simulator to count the number of SCVs observed with Volition — even if the SCVs repeat across runs. When a processor detects an SCV, it sends an \textit{SP\_expire} to remove one of the ARs and ensure that the program continues execution. We model either RC or TSO.

Table 6.4 shows, for each consistency model, the number of SCVs detected with full Volition support (\# of SCVs). It also shows the SCVs detected when Volition does not keep per-word access information (\# of Line-SCVs). This environment is missing the SAT support from Section 6.5 that records which words of the line have been accessed by the processor. In this case, there are no metadata transactions — only the transactions generated by the coherence protocol occur. Hence, Volition misses SCVs. Finally, the table shows the number of active races detected (\# of ARs). Intuitively, these are the races between largely concurrent accesses. They roughly capture the types of races recorded by DRFx [43] and Conflict Exceptions [40].

Looking at the RC columns, we see that Volition detects many SCVs in these codes. If metadata transactions are disabled (Line-SCVs), about half of the SCVs are missed. Hence, we need full Volition support. We also see that the number of ARs is very high — on average, over two orders of magnitude higher than the number of SCVs. This shows that ARs are not good proxies for SCVs.

The number of SCVs detected changes with the memory model. Typically, the more relaxed RC model causes more SCVs and Line-SCVs than the TSO model. However, in some applications, the opposite is the case.

Finally, the table shows the number of replays required to enforce SC in CO from Lin et al. [38], running on RC. We can see that the number of replays is lower than the number of ARs. However, it is, on average, one order of magnitude higher than the number of SCVs in Volition. Therefore, we conclude that, while CO is more precise than just reporting active races, it is much less precise than Volition in its detection of SCVs.
Figure 6.17: Table size requirements.
6.8.3 Characteristics of SCVs

In this section, we characterize the SCVs observed. Figure 6.18 takes the SCVs reported for each code and memory model in Table 6.4 and classifies them based on the number of processors participating in the SCV cycle. Specifically, we have cycles with 2 processors, 3 processors, and 4 or more processors. For each code, the number of SCVs is normalized to the number under RC.

The figure shows that, in these sharing-intensive, fence-free codes, cycles appear with a variety of processor counts. While two-processor SCVs dominate in some applications (e.g., Moircas for RC), four-and-more processor SCVs are dominant in others (e.g., Msn). Hence, Volition’s ability to detect cycles with an arbitrary number of processors is useful for the bug conditions represented.
by these fence-free codes. Note that Dekker can only have 2-processor cycles.

Figure 6.17 shows the use of the Volition hardware tables. Specifically, it shows, for each program and memory model, the maximum number of entries in use in each of the per-node ARST, ARDT, and SAT. We can see that these sizes are modest. In most cases, the maximum number of entries used in the ARST and ARDT is less than 25. For the SAT, it is less than 75. Therefore, our proposed sizes of 40 entries for the ARST and ARDT, and 100 entries for the SAT (Table 6.2) are more than enough.

Figure 6.19 shows the sensitivity of the number of SCVs to the size of the store buffer. For each program under RC, the figure shows the number of SCVs with store buffers of 4, 8, 16, and 32 entries. The latter is the default size. For each program, the bars are normalized to the number of SCVs for 4-entry buffers and are broken down into the number of processors per cycle.

![Sensitivity to the size of the store buffer](image)

**Figure 6.19: Sensitivity to the size of the store buffer.**

Intuitively, larger store buffers should induce more SCVs because they allow more store reordering. While this is the general trend, there are several programs where smaller buffers induce more SCVs. The reason is that smaller buffers also affect the timing of execution significantly, by introducing more access stalls due to full buffers.
6.8.4 Overheads of Volition

We consider two overheads of Volition, namely the increase in network traffic and the increase in program execution time. Figure 6.20 shows the total number of bytes transferred in the network of the machine for different programs and memory consistency models. The figure also includes bars for the average of the SPLASH-2 applications and the average of the Parsec codes. We break down the bytes transferred into those from memory access requests (\textit{MemAcc}), data transferred in a read (\textit{Read}) or write (\textit{Write}), coherence activity in invalidations, acknowledgements, or forwarding to the owner cache (\textit{Coh}), and additional traffic due to Volition (\textit{Overhead}). The latter includes messages such as \textit{SP\_expire}, AR propagation in cycles with more than two processors, or SAI information transfer. The sum of the first four categories is normalized to 100. From the figure, we see that, even with 64 processors, the additional traffic induced by Volition (\textit{Overhead}) is largely negligible. For fewer processor counts, it is even smaller.
Finally, we consider the execution time overhead of Volition in Figure 6.21. The figure shows the execution time of the different applications under the RC memory model on a multiprocessor without Volition (Baseline) and on one with Volition. We show bars for all the small programs, their average, the average of SPLASH-2, and the average of Parsec. For each application, the bars are normalized to the Baseline.
The main source of execution overhead in Volition is the stall due to a full ACT. We use a 256-entry ACT, and we can see that, for most applications, the execution time overhead of Volition for 64 processors is negligible. Some of the applications with visible overhead are those with many SCVs which, according to Table 6.4, include Mst and Msn. Still, for the large applications (SPLASH-2 and Parsec), there is no visible overhead. For the small applications, the average overhead is only about 2%.

Overall, based on the results of the traffic and execution time overheads, we conclude that Volition has low overhead and good scalability.
6.9  Conclusions

This section proposed Volition, the first hardware scheme that detects SCVs in a relaxed-consistency machine precisely, in a scalable manner, and for an arbitrary number of processors in the cycle. Volition uses cache coherence protocol transactions to detect cycles in memory-access orders across threads. When a cycle is about to occur, an exception is triggered. For the conditions considered in this paper, Volition suffers neither false positives nor false negatives. We simulated Volition on a 64-processor multicore with directory-based coherence and 32-byte cache lines, running some small codes and SPLASH-2 and Parsec applications. Our results showed that Volition induces negligible network traffic and execution time overhead and is scalable. In addition, it detects SCV cycles with several processors. Volition is suitable for on-the-fly use.
Chapter 7

Conclusion

As parallel computing becomes mainstream, the programmability of shared-memory multiprocessor systems should be improved. This thesis focuses on efficient and scalable architecture supports to improve the programmability.

We focus on supporting SC, a strong and intuitive memory consistency model. First, the thesis proposes two scalable cache coherence protocols for chunk-based execution, which can be used to enforce SC efficiently. Moreover, it also proposes the design of an SMT processor to support chunk operations among the contexts in the same processor. Second, the thesis proposes a scheme to enforce high-performance whole-system SC, from language to architecture, by speculative chunk ordering. Third, the thesis proposes a scheme to dynamically detect SC violations based on the directory-based cache coherence precisely.
References


