Exploiting Reference Idempotency to Reduce Speculative Storage Overflow*

S. W. Kim[1], C.-L. Ooi[1], R. Eigenmann[1], B. Falsafi[2], T. N. Vijaykumar[1]
[1]School of ECE, Purdue University, West Lafayette, IN
[2]Department of ECE, Carnegie Mellon University, Pittsburgh, PA
mux@ecn.purdue.edu

ABSTRACT

Recent proposals for multithreaded architectures employ speculative execution to allow threads with unknown dependences to execute speculatively in parallel. The architectures use hardware speculative storage to buffer speculative data, track data dependences and correct incorrect executions through roll-backs. Because *all* memory references access the speculative storage, current proposals implement speculative storage using small memory structures to achieve fast access. The limited capacity of the speculative storage causes considerable performance loss due to *speculative storage overflow* whenever a thread's speculative state exceeds the speculative storage capacity. Larger threads exacerbate the overflow problem but are preferable to smaller threads, as larger threads uncover more parallelism.

In this paper, we discover a new program property called *memory reference idempotency*. Idempotent references are guaranteed to be eventually corrected, though the references may be temporarily incorrect in the process of speculation. Therefore, idempotent references, even from non-parallelizable program sections, need not be tracked in the speculative storage, and instead can directly access non-speculative storage (i.e., conventional memory hierarchy). Thus, we reduce the demand for speculative storage space in large threads. We define a formal framework for reference idempotency and present a novel compiler-assisted speculative execution model. We prove the necessary and sufficient conditions for reference idempotency using our model. We present a compiler algorithm to label idempotent memory references for the hardware. Experimental results show that for our benchmarks, over 60% of the references in non-parallelizable program sections are idempotent.

*This work was supported in part by NSF grant #9974976-EIA. This article is an extended version of a paper presented at the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'01). The algorithm for labeling idempotent references has been improved and a discussion of its complexity and correctness has been added.

Seon Wook Kim is now with KAI Software Lab, A Division of Intel Americas, Inc., Champaign, IL, seon.w.kim@intel.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPOPP '01 Snowbird, Utah USA
Copyright 2001 ACM 0-89791-88-6/97/05 ...$5.00.

1. INTRODUCTION

Technological advancements in semiconductor fabrication are giving rise to an abundance of transistors in a single chip. To harness performance from the large number of transistors, computer designers are innovating novel *multithreaded* chip architectures. As in shared-memory multiprocessors, some of the proposed multithreaded chip architectures (e.g., the IBM Power 4) support the conventional parallel execution models in which a programmer or compiler partitions the program into distinct parallel threads. Unfortunately, many programs include code fragments that have dependences unknown at compile time and are therefore not entirely parallelizable [1, 2]. Runtime data dependence tests can parallelize certain unanalyzable code sections [3, 4]. However, these tests cannot be applied to general program patterns.

Alternatively, recent proposals for multithreaded architectures (e.g., the proposed SUN MAJC chip, Wisconsin Multiscalar, CMU Stampede, I-ACOMA, and Stanford Hydra [5, 6, 7, 8]) employ *speculative execution* to allow threads with unknown dependences to execute speculatively in parallel. These architectures use hardware *speculative storage* to produce and consume data speculatively while tracking and enforcing data dependence. On successful speculation, the threads commit the speculative data from speculative storage to *non-speculative storage* (i.e., conventional memory hierarchy). Upon misspeculations, the hardware discards speculative computation and rolls back the machine state.

A key shortcoming of the proposed speculative multithreaded architectures is the limited capacity of speculative storage used to hold speculative state. Because data dependence must be tracked and enforced on *all* memory references (both reads and writes), the speculative storage needs to provide high-speed access. Accordingly, current proposals use small structures to achieve fast access — e.g., custom hardware buffers [7, 9] or level-one data caches [10]. If a thread's speculative state exceeds the speculative storage capacity, the thread stalls for many cycles (typically hundreds of cycles) until its speculation is resolved, incurring considerable performance loss. Larger threads exacerbate the *speculative storage overflow* problem because they ac-

cess more speculative data. This problem is especially critical because larger threads are preferable to smaller threads, as larger threads uncover more parallelism.

Speculatively-threaded applications usually contain sections that are provably parallel, while the rest are not analyzable. Advanced compilers can easily avoid placing the references made by the provably-parallel sections in speculative storage, and direct such references to non-speculative storage, avoiding speculative storage overflow for those sections. In the non-parallelizable sections, however, the hardware blindly tracks data dependence for *all* memory references, increasing the chances of speculative storage overflow.

In this paper, we discover a new program property called *memory reference idempotency*. Idempotent references can be directly placed in non-speculative storage instead of speculative storage, *even if* the references are from non-parallelizable sections. Reference idempotency is based on our fundamental insight that in speculative execution, incorrect values are created due to ependence violations, and propagated through subsequent computation. Idempotent references' key feature is that they do not cause any data-dependence violations on their own, although they may propagate incorrect values. Because the initial incorrect values are eventually corrected and propagated, idempotent references need not be tracked in speculative storage, even if the reference is temporarily incorrect. Needless to say, if a reference is not involved any dependence across processors (e.g., read-only and private references), such a reference is straightforwardly idempotent. By filtering out idempotent references, we reduce the demand for speculative storage space even for large threads, uncovering more parallelism without incurring much overflow.

The key contributions of this paper are:

- We define a formal framework for reference idempotency to alleviate speculative execution overhead.
- We present a novel *compiler-assisted speculative execution* model, in which the compiler communicates idempotent references to the architecture.
- We prove the necessary and sufficient conditions for reference idempotency using our model.
- We present a compiler algorithm to label idempotent memory references, so that the hardware can place the references directly in the non-speculative storage.
- We show results that, for our benchmarks, over 60% of the references in non-parallelizable code sections are idempotent.

This paper is organized as follows. Next, we present an introductory example of hardware-only speculative execution and idempotent references. Section 2 formally defines and verifies the hardware-only model. Section 3 presents the formal definition and proof of correctness of the compiler-assisted speculative execution model. Section 4 introduces reference idempotency, proves the necessary and sufficient conditions for reference idempotency, and describes a compiler algorithm for idempotency analysis. Section 5 shows experimental results on the frequency of idempotent references. Section 6 presents conclusions.

An Introductory Example

Current proposals for speculative multithreaded processors assume a *hardware-only speculative execution* (HOSE) model. In HOSE, the software is unaware of speculative execution. It assumes sequential execution semantics and sees the usual program state (i.e., the values of all program variables) in the memory system. The hardware, which we call the *speculation engine*, selects program segments and executes them speculatively in parallel. Segments can range in size from a single instruction to entire subroutines. Threads are the dynamic instances of segments.

Consider the program in Figure 1. The program is split into two segments that are executed speculatively in parallel by a two-processor system. Segment 2 follows segment 1 in sequential program order and therefore all inter-segment dependences must be satisfied in that order while the segments are executing. The program has several read references to variable `B`, a data dependence across the two segments involving variable `A`, and a write and read reference to variable `C` in segment 2.

A typical speculative execution scenario in HOSE is illustrated in Figure 1 (a). The system executes the two segments in parallel while keeping *all* data values produced or referenced in speculative storage. The data values remain in speculative storage until the speculation is verified, all dependences are satisfied in program order, and the execution is known to be correct. Upon verifying speculation, the data values in the speculative storage are transferred or "committed" to non-speculative storage. To track and enforce dependences in program order, in addition to the data values, the speculative storage also keeps information about every reference type and the order in which references are made.

In the example shown, because the two program segments execute concurrently, upon the write reference to `A` in segment 1, the processor may see that a later program-order read reference to `A` by segment 2 has already happened. This is a dependence violation, to which the system reacts by aborting and re-starting segment 2. Since all accessed data values have been buffered in the speculative storage, the restart simply clears all the buffered references corresponding to segment 2.

Figure 1 (b) illustrates several examples of idempotent references — i.e., references that do not require buffering in speculative storage and that can directly access non-speculative storage. First, the compiler can identify all references to variable `B` to be idempotent because `B` is a read-only variable and as such, does not have any data dependences. Second, the first write reference to `A` in segment 1 is idempotent because there are no previous program-order references to `A` in the segment. To enforce dependences, the write reference, however, does look through speculative storage to check for data dependence violations by segment 2's references to `A` (i.e., the read reference). The actual value of the write reference resides in non-speculative storage, without occupying any space in speculative storage. Third, variable `C` in this example is private to segment 2 — i.e., there are no dependences across segments on this variable — and all references to it are idempotent. Although segment 2 may re-execute due to incorrect speculation, the write reference `C` always occurs first whenever the segment is re-executed. Hence, even if an incorrect value were written initially, the value of `C` will be corrected in the final execution of the

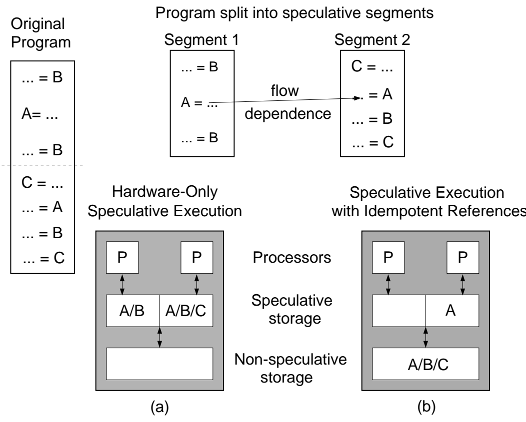

Figure 1: Basic idea of labeling idempotent references. (a) In hardware-only speculation, all data is placed in the speculative storage. (b) After labeling idempotent references, these references can go directly to the non-speculative storage.

segment.

2. HARDWARE-ONLY SPECULATION

2.1 HOSE Model

In the following, we formally define the structure of the software and the execution model of hardware-only speculative execution. We show that the execution produces the same answer as a sequential program.

DEFINITION 1 (PROGRAM STRUCTURE). *A program is structured into one or several* regions*, which are substructured into several* segments*. A region has a single entry and exit. A segment has a single entry, but may have multiple exits. Segments are related by age. An* older *segment would execute before a* younger *segment in a sequential execution of the program. All older segments are referred to as* ancestors*.*

In this definition, segments represent speculative units. These can be individual instructions in low-level speculation or entire subroutines in large-grain speculation models. For HOSE the entire program is a single region. Multiple regions will be important for the compiler-assisted speculative execution model, introduced in Section 3.

DEFINITION 2 (HOSE MECHANISM). *Hardware-Only Speculative Execution is an execution mechanism for programs given in Definition 1 with the following properties:*

0. Overall Execution: *Regions execute sequentially with respect to other regions. Segments can be executed speculatively in parallel with other segments within the same region, that is, they may be started in an order that is different from the sequential order and they may execute concurrently. Internally, segments execute sequentially and perform memory references in program order.*

1. Segment Execution and Roll-Backs: *The speculative parallel execution of segments may violate data and control dependences, resulting in incorrect values generated and incorrect control paths taken. The speculation engine detects these violations (see Property 4) and* rolls back *incorrect segments. Upon a roll-back, all data generated by the segment are discarded (see Property 3). This process may repeat several times.*

2. Final Execution: *A correct,* final execution *follows all incorrect executions of a segment. The final execution satisfies all cross-segment flow and control dependences. If the segment was incorrectly started due to misspeculation, the final execution may execute a different segment or it may be empty.*

3. Data Access: *Each segment has its own* speculative storage*. It is empty at the beginning of each segment's execution and after each roll-back. During the execution of a segment, all data references go to the speculative storage. They do not affect the non-speculative storage until the segment is* committed *(see Property 5). If a read reference accesses a location not present in the speculative storage, then the value is fetched from the youngest ancestor that contains a value for this location, or from non-speculative storage if no ancestor contains that location. A write reference affects only the segment's own speculative storage.*

4. Dependence Tracking: *In addition to the actual data values, the speculative storage contains access information (time and type of reference), which allows the speculation engine to track dependences. If a write reference detects that a read reference to the same storage location by a younger segment has prematurely happened, then a data-dependence (flow dependence) violation has occurred. If, at the completion of a segment, the speculation engine detects that the successor segment is different from the speculatively chosen one, then a control dependence violation has occurred. The speculation engine reacts to both violations by rolling back all younger segments currently in execution. Cross-segment anti and output dependences are satisfied because the segments have separate speculative storage (Property 3), which are committed in sequential order (Property 5).*

5. Segment Commit: *When the oldest segment in execution has completed all instructions, speculation of that segment is said to have succeeded and the segment has performed its final execution. At this point the segment's speculative storage is committed (i.e., conceptually moved) to the non-speculative storage. A segment cannot commit before all older segments have committed. Note, that only the values generated by the segment's final execution are committed.*

2.2 Correctness of HOSE

DEFINITION 3 (CORRECT PROGRAM EXECUTION). *A region $\mathcal{R}$ is executed correctly if, given that all older regions are executed correctly, at their last reference in $\mathcal{R}$ all live program variables in the non-speculative storage have the same value as in a sequential execution of the program.*

Similarly, a segment $\mathcal{R}_x$ in region $\mathcal{R}$ is executed correctly if, given that all older segments in $\mathcal{R}$ and all regions older than $\mathcal{R}$ are executed correctly, at their last reference in $\mathcal{R}_x$

all live program variables in the non-speculative storage have the same value as in a sequential execution of the program.

LEMMA 1 (CORRECTNESS OF HOSE). *A region $\mathcal{R}$ and all segments in $\mathcal{R}$ are executed correctly under the hardware-only speculative execution model.*

PROOF. Let $\mathcal{R}_1 ... \mathcal{R}_n$ be the segments in region $\mathcal{R}$ from oldest to youngest. To satisfy the correctness criterion of Definition 3, we need to show that, for any segment $\mathcal{R}_x, 1 \leq x \leq n$, the values of the program variables generated and committed to non-speculative storage locations at the end of $\mathcal{R}_x$ are correct, that is, they are the same as the values of these variables in a sequential program execution. HOSE discards all values generated by segments that are being rolled back. The only values to be committed are those generated in final executions. We show correctness of these values in two steps. We show that (1) the final executions of all segments produce correct values in the speculative storage and (2) these values are committed correctly.

(1) Internally, segments execute sequentially (HOSE property 0). All data references use the segment's own speculative storage, and this storage cannot be modified by any other segment (HOSE Property 3). Hence, the segments execute and produce the same final values as a sequential program if we can show that data values not initially present in the segment's speculative storage are consumed correctly (i.e., as in a sequential program). This follows from two facts. (a) All cross-segment time orderings are satisfied (HOSE Property 4). (b) By HOSE Property 3, values for locations not present in the speculative storage are consumed either from the youngest ancestor that contains a value for this variable (which is the producer of this value in a sequential execution,) or from non-speculative storage (where they are correct, given the preceding region's correct execution).

(2) By HOSE Property 5, all segments commit in sequential order. Therefore, all segments' values will be seen in the non-speculative storage correctly after all ancestors have placed their values.

Correctness of a region $\mathcal{R}$ follows from the correctness of the segments in $\mathcal{R}$. By HOSE Property 5, the segment last touching any memory location x is the same segment as in a sequential execution. Since x is correct at the end of this segment, it is also correct at the end of $\mathcal{R}$. □

3. COMPILER-ASSISTED SPECULATION

3.1 CASE Execution Model

The Compiler-Assisted Speculative Execution (CASE) model is an extension of the hardware-only model introduced in Section 2. The software structure is the same as in Definition 1. As in HOSE, segments are the primary units of speculative execution. Regions are important for enclosing code sections in which certain data attributes hold (e.g, read-only, or dependence-free). The execution mechanism is defined as follows:

DEFINITION 4 (CASE MECHANISM). *Compiler-Assisted Speculative Execution is a program execution mechanism with the basic properties of HOSE as given in Definition 2. Certain data references are labeled as idempotent, and all other references are speculative with the same properties as in HOSE. Idempotent references have the following properties:*

Idempotent read references *completely bypass the speculative storage and instead directly reference the non-speculative storage. Unlike speculative reads, idempotent reads do not leave any information in the speculative storage.*

Idempotent write references *enforce data dependences by first checking in the speculative storage, much like speculative write references. However, then their value is directly placed in the non-speculative storage and no information about the references is kept in the speculative storage.*

From the definition of idempotent references, we see that the references access non-speculative storage, and do not occupy any space in speculative storage. Thus, idempotent references help reduce speculative storage overflow, as motivated in Section 1. Note, for ease of presentation we use the term *idempotency* for both a program property (the referenced variable is correct despite repeated accesses caused by roll-back and re-execution) and a hardware property (the memory reference accesses non-speculative storage.)

3.2 Correctness of CASE

In CASE, programs contain both speculative and idempotent references. The hardware guarantees correctness for speculative references, like HOSE. But idempotent references are not tracked by speculative storage, and therefore correctness of idempotent references is no longer guaranteed by the hardware. Instead, the compiler must correctly label idempotent references to guarantee correct execution. To that end, the following labeling conditions must be satisfied by references to be identified as idempotent.

LC1: *A write reference*[1] $\hat{x}$ *to a variable* v *in region* $\mathcal{R}$ *is correctly labeled as idempotent only if it is guaranteed that* v *will eventually be correct — i.e., an incorrect* v *must be overwritten with the correct value, before it is consumed by the final execution of any segment.* (Speculative read references may obtain incorrect values in a misspeculated execution and propagate the incorrect values to idempotent write references. Because such incorrect idempotent writes are not discarded but written to non-speculative storage, LC1 ensures that the write reference is eventually corrected.)

LC2: *A reference* $\hat{x}$ *is correctly labeled as idempotent only if, in the final execution, all time orderings as dictated by data dependences involving* $\hat{x}$ *are satisfied.* (An idempotent reference does not keep any information about the reference in speculative storage. Because the hardware can no longer enforce data dependences for the reference, LC2 ensures that the reference is ordered correctly with respect to its dependences.)

LC3: *A write reference is correctly labeled as idempotent only if any subsequent read reference to* v *consumes this value from non-speculative storage. A read reference is correctly labeled as idempotent only if it obtains from non-speculative storage the value generated by any prior write reference.* (If one of the source and sink of a flow dependence is a speculative reference and

[1]We use the notation v for variables and $\hat{x}$ for memory references.

the other is an idempotent reference, the source and sink access different storages. LC3 ensures that the sink reference correctly obtains the value produced by the source reference.)

Recall our fundamental insight that in speculative execution, incorrect values are created due to data-dependence violations, and propagated through subsequent computation. LC1, LC2, and LC3 together guarantee that idempotent references do not cause any data-dependence violations on their own, although the references may propagate incorrect values. Because the initial incorrect values are eventually corrected and propagated, idempotent references need not be tracked in speculative storage, even if the reference is temporarily incorrect. If a reference is not involved in any dependence across processors (e.g., read-only and private references), such a reference is straightforwardly idempotent.

LEMMA 2 (CORRECTNESS OF CASE). *CASE is correct under Definition 3 if and only if all idempotent references satisfy the three labeling conditions LC1 through LC3.*

PROOF. The values in non-speculative storage generated by a segment are those committed from speculative storage and those written by idempotent references.

We proceed in two steps, (1) we show that the values generated by idempotent references are correct and (2) we show that the values generated in speculative storage and then committed are correct.

(1) Because idempotent references directly write into non-speculative storage, we must consider all segment executions. This contrasts with HOSE, which considers only final executions. By LC1, a segment produces correct values for all variables that incur idempotent references. That is, even though a variable v may be written in a misspeculated segment, LC1 guarantees that, in all final executions of segments referencing v, this variable is correct.

(2) The only difference to the values produced in speculative storage in HOSE is that instructions may consume input values through read references involved in idempotent references. These values are correct as follows. By LC2, all time orderings as dictated by data dependences are satisfied. By LC3 values are correctly communicated if either the producer or the consumer is an idempotent reference. Therefore, the values committed from speculative storage are correct for the same reason as they are correct in HOSE.

The proof of the converse is simple, and is only sketched. The descriptions of the three labeling criteria make obvious that if any of them is not satisfied then an incorrect value is produced, consumed, or a data dependence may be violated. Hence, correct program execution would no longer be guaranteed.

The proof of correctness of a region is identical to the one for HOSE. □

4. REFERENCE IDEMPOTENCY

In this section we present the analysis methods and algorithms for identifying variable references in a program that have the idempotency property. Idempotent references do not need to be buffered in speculative storage. To prove correctness we will show that such references satisfy the labeling criteria LC1 through LC3.

Theorems 1 and 2 give the necessary and sufficient conditions for a data reference to be labeled as idempotent. The following lemmas will be helpful in proving the two theorems. In addition, the term *re-occurring first write* will be useful. It is defined as follows.

DEFINITION 5 (RE-OCCURRING FIRST WRITE (RFW)). *A write reference to the variable v in segment $\mathcal{R}_i$ is a RFW if, following any roll-back of $\mathcal{R}_i$, a live v is guaranteed to be written before the end of the enclosing region $\mathcal{R}$ without a preceding read reference.*

Note, that by Definition 2 the segment $\mathcal{R}_i$ may get rolled back to the end of any ancestor segment in $\mathcal{R}$. Hence, a write reference to v in $\mathcal{R}_i$ is a RFW if v is first written on all possible paths p, where p is a path from the end of any ancestor of $\mathcal{R}_i$ to the end of $\mathcal{R}$. If v is not live then its value is irrelevant for correctness by Definition 2.

The RFW attribute will allow us to identify a write reference as idempotent, even though it may be performed in a misspeculated segment with an incorrect value. The RFW attribute ensures that a write reference to the same variable is guaranteed to happen in the final execution of some segment following the RFW reference. Hence, the variable value will be corrected. It further guarantees that no read reference can consume the incorrect value before the correct value is written. Note, that determining the RFW attribute is non-trivial in the presence of pointers and subscripted subscripts. The compiler must guarantee that the references in the misspeculated and in all possible final executions go to the same storage location. We will present a compiler algorithm in Section 4.2.

For the following presentation we consider a region at a time. Data dependences are assumed to have been analyzed for the region on an reference by reference basis. Note, that this means that there are only data dependences between references to the same variable. Only intra-region dependences are considered.

LEMMA 3 (CROSS-SEGMENT DEPENDENCE SINK). *The sink of a cross-segment dependence must be labeled speculative.*

PROOF. Assume the dependence sink can be labeled idempotent. Suppose the dependence source executes after the sink. If the sink is a read reference, no information about its access time is kept in speculative storage. Hence, the hardware will not enforce the dependence per HOSE Property 4. If the sink is a write reference, it directly writes to the non-speculative storage, violating the dependence. In both cases the labeling criterion LC2 is not satisfied, which contradicts the assumption. □

LEMMA 4 (INDEPENDENT READ). *A read reference* $\hat{x}$ *that is not the sink of any dependence can be labeled idempotent.*

PROOF. LC1 does not apply to read references. Considering LC2, suppose the reference $\hat{x}$ is involved in a dependence with sink $\hat{y}$. Intra-segment dependences are always satisfied because of the sequential execution of segments. A cross-segment dependence is also satisfied because $\hat{y}$ is labeled speculative per Lemma 3. This means that the value of $\hat{y}$ is committed at the end of the final execution of the enclosing segment, which happens *after* $\hat{x}$ (HOSE Property 3 and 5). Hence LC2 is satisfied. LC3 is not applicable because there is no write reference preceding $\hat{x}$. □

LEMMA 5 (INDEPENDENT RFW). *A re-occurring first write (RFW) that is not the sink of a cross-segment dependence can be labeled idempotent.*

PROOF. LC1 is satisfied because the write reference is a re-occurring first write. By Definition 5, even after a misspeculated value is written, a new value is guaranteed to be written prior to all reads in any final execution, hence the value is corrected.

For LC2, intra-segment dependences are always satisfied. For cross-segment dependences we consider two cases. Case 1: the reference $\hat{x}$ is the source of a flow dependence with sink $\hat{y}$. This dependence is enforced per Definition 4 as long as the sink is speculative. This is the case by Lemma 3. Case 2: there is an output dependence from $\hat{x}$ to $\hat{y}$. This dependence is also satisfied. Since $\hat{y}$ is speculative, it will be written to the non-speculative memory upon the commit of the segment containing $\hat{y}$, which is *after* the reference $\hat{x}$ (HOSE Property 5). Hence LC2 is satisfied.

LC3 needs to be considered for the case of a flow dependence from $\hat{x}$ to $\hat{y}$. By HOSE Property 3, the speculative read reference consumes the value from the non-speculative storage location if no ancestor segment contains a speculative value for this location. This is indeed the case because $\hat{x}$ is not the sink of any other dependence, which means it is the first reference to this variable in the region. Hence LC3 is satisfied as well. □

LEMMA 6 (COVERED READ). *A read reference* ŷ *that is dependent on an idempotent RFW reference* x̂ *within the same segment can be labeled idempotent.*

PROOF. LC2 and LC3 need to be considered. For LC2, all intra-segment dependences are satisfied because of the sequential execution of segments. Write references are only labeled idempotent with Lemma 5. Such references do not depend on older segments, hence $\hat{y}$ cannot be the sink of a cross-segment dependence. On the other hand, $\hat{y}$ can be the *source* of a cross-segment dependence. Such a dependence is satisfied; the proof is the same as in Lemma 4. Hence, LC2 is satisfied. LC3 is also satisfied, because an idempotent $\hat{y}$ correctly reads the value generated by an idempotent $\hat{x}$ in non-speculative storage. □

For completeness, the following simple lemma deals with fully independent regions.

LEMMA 7 (FULLY INDEPENDENT). *All references of a region whose segments do not carry any data dependences or control dependences can be labeled idempotent.*

PROOF. A region without any data and control dependences across segments is completely non-speculative. That is, all segments are executed only in their correct, final form without any violations of data and control dependences. The execution will not perform roll-backs. Hence all shared references happen exactly once in their final and correct form. Labeling them as idempotent satisfies all three labeling criteria trivially. □

Lemmas 3 through 6 provide the basis for proving necessary and sufficient conditions for idempotent read and write references in segments that include dependences.

THEOREM 1 (IDEMPOTENT WRITE). *A write reference is idempotent if and only if it is a re-occurring first write and it is not the sink of a cross-segment dependence.*

THEOREM 2 (IDEMPOTENT READ). *A read reference is idempotent if and only if it is not the sink of any data dependence or it is dependent on an idempotent write reference within the same segment.*

PROOF (IDEMPOTENT WRITE). By Lemma 5, a RFW that is not the sink of an cross-segment dependence can be labeled idempotent.

We prove the converse by contradiction. We show that a write reference that is the sink of a cross-segment dependence or is not a RFW cannot be labeled idempotent. By Lemma 3 a cross-segment dependence sink cannot be labeled idempotent. If a reference $\hat{x}$ to variable v is not a RFW then, after the enclosing segment rolls back, execution can take a path that (case 1) does not reference v, or that (case 2) first reads from v. Case 1 cannot be labeled idempotent because $\hat{x}$ may have written an incorrect value that is never corrected. Case 2 cannot be labeled idempotent because the read reference would consume the incorrect value written by $\hat{v}$. In both cases LC1 would be violated. □

PROOF (IDEMPOTENT READ). By Lemma 4 a read reference that is not the sink of a data dependence can be labeled idempotent. By Lemma 6 a read can also be labeled idempotent if it is dependent on an idempotent write reference within the same segment.

We prove the converse by contradiction. We show that a read reference cannot be labeled idempotent if it is dependent on a source that is not an idempotent write reference within the same segment. There are two cases. (1) The source is in a different segment and (2) the source within the same segment is labeled speculative. By Lemma 3 a dependence sink cannot be labeled idempotent in case 1. Case 2 directly violates LC3 because an idempotent read will not consume the value written by a preceding speculative write reference. □

4.1 Discussion: Idempotency Categories

We can describe idempotent references in the form of the following categories. The first category deals with the simple case of program regions that can be detected as fully parallel by a compiler.

Fully independent: If there are no cross-segment data and control dependences, then all references in a region $\mathcal{R}$ are idempotent. No individual access labeling would be necessary for this category. No data needs to be placed in speculative storage. Essentially this means that the region can be run as in a conventional multiprocessor.

The next three categories are applicable to regions that have data dependences.

Read-only: All references to read-only variables in a region are idempotent. These references are not sinks of any dependence. Note that, although very intuitive, the idempotency property for read-only variables in partially-dependent code sections is not trivial because of the interaction of idempotent and speculative references. This is shown in the proof of Lemma 4.

Private: All references to segment-private data are idempotent. This category is relevant for compilers that can recognize private variables and express this information such that the architecture or runtime system can provide a private

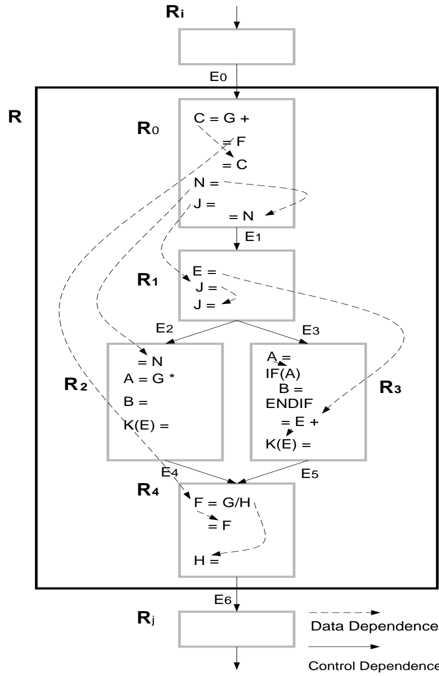

Figure 2: Example code with control and data dependence graphs. The region $\mathcal{R}$ contains five segments, $\mathcal{R}_i$, $\cdots$, $\mathcal{R}_{i+4}$.

address space for each segment. Alternatively, the compiler can apply data renaming, with the result that the references will fall into the next category. Important in our analysis are the facts that private variables do not have any cross-segment dependences and are not live past the end of the segment.

Shared Dependent: The fact that there are data-dependent references that do not need to be placed in speculative storage is most remarkable. Essentially, only sinks of cross-segment data dependences need to be labeled speculative. Within a segment all references following a write that is guaranteed to happen, and happen again after a misspeculation, can be labeled idempotent. It is important to note that these write references may produce temporarily incorrect values in the non-speculative memory. The idempotency property guarantees that correctness is still ensured.

Examples

Figure 2 shows several examples. By Definition 5, $\texttt{RFW}(\mathcal{R}_i) = \{\texttt{C, N, J}\}$, $\texttt{RFW}(\mathcal{R}_{i+1}) = \{\texttt{E, J}\}$, $\texttt{RFW}(\mathcal{R}_{i+2}) = \{\texttt{A}\}$, $\texttt{RFW}(\mathcal{R}_{i+3}) = \{\texttt{A}\}$, and $\texttt{RFW}(\mathcal{R}_{i+4}) = \{\texttt{F}\}$. The reference to `B` in $\mathcal{R}_{i+2}$ is not a RFW, because the reference to `B` is not guaranteed to execute if $\mathcal{R}_{i+2}$ is rolled back. Similarly, the reference to `B` in $\mathcal{R}_{i+3}$ is not a RFW. The write reference to `H` in $\mathcal{R}_{i+4}$ is preceded by a read. The references to `K(E)` in $\mathcal{R}_{i+2}$ and $\mathcal{R}_{i+3}$ are not RFW because `E` is not read-only. All these RFW references are idempotent except for `J` in $\mathcal{R}_{i+1}$ and `F` in $\mathcal{R}_{i+4}$. These references to `J` and `F` are not idempotent by Lemma 5 because they are the sink of output and anti dependences from $\mathcal{R}_i$.

The read references to `N` in $\mathcal{R}_{i+2}$ and `E` in $\mathcal{R}_{i+3}$, and a write reference to `F` in $\mathcal{R}_{i+4}$ are speculative by Lemma 3 because they are sinks of cross-segment dependences. All references to variable `G` in $\mathcal{R}$, `F` in $\mathcal{R}_i$ and the read of `H` in $\mathcal{R}_{i+4}$ are idempotent by Lemma 4 because they are independent reads. The read references to `N` and `C` in $\mathcal{R}_i$, `A` in $\mathcal{R}_{i+3}$ and `F` in $\mathcal{R}_{i+4}$ are idempotent by Lemma 6 because they are covered reads.

4.2 Compiler Algorithms

4.2.1 Prerequisite Analysis

The prerequisites for our algorithm are as follows. The compiler identifies regions and segments. The algorithm for defining regions and segments is not part of the presented paper. In our evaluation, regions are loops and segments are loop iterations. Furthermore, we assume that a state-of-the-art compiler (e.g., [11, 12]) has analyzed read-only and private variables, and also the data dependences of every reference in each region. The following algorithm determines re-occurring first write references.

4.2.2 Analyzing Re-occurring First Write References

Recall that by Definition 5 a write reference to a variable x in segment $\mathcal{R}_i$ is a RFW x is first written on all possible paths p, where p is a path from the end of any ancestor of $\mathcal{R}_i$ to the end of $\mathcal{R}$. The basic idea of the following graph algorithm is to mark all successors of a segment as non-RFW for a given variable x, if any successor has an exposed read reference to x.

ALGORITHM 1. *Identifying re-occurring first write references in a region $\mathcal{R}$:*
Let G be a graph with nodes V representing segments $\mathcal{R}_i$ and edges E representing control paths between segments. An extra node v_{exit} is placed at the exit of $\mathcal{R}$. E^r refers to the reversed edges, and $G^r = (V, E^r)$ is referred to as the reversed segment graph, with v_{exit} becoming the first node. Nodes have the following two attributes for each variable: color `(Black, White)` *and reference type* `(Write, Read, Null)`. *For a given node v and variable x, either all write references to x in v are RFW* `(White)` *or none is RFW* `(Black)`. *The algorithm finds this property.*

1. *Initially, for each node v and for each variable x, set the color to* `White` *and set the reference type as follows:*
   - *If x is defined on all paths through segment v without exposed read*[2]*, then set the reference type to* `Write`.
   - *Else, if there is an exposed read of x, then set* `Read`.
   - *Else, (no reference to x in v) set* `Null`.

   Set v_{exit} for x as `Read` *if x is live-out of $\mathcal{R}$, and* `Null` *otherwise.*

2. *Search G^r (depth or breadth first). At each node with reference type* `Read`, *mark all* `Null` *successor nodes* `Read` *as well.*

3. *Search G (depth or breadth-first). At each node v, if it is* `Black` *or any successor has reference type* `Read`, *color all successors* `Black`.

[2]We refer to standard compiler techniques for analyzing must-definitions and exposed reads.

4. *All write references to x in `White` nodes are re-occurring first writes.*

The complexity of the algorithm is $\mathcal{O}(|V| + |E|)$ for each variable because the steps of Algorithm 1 have the following complexity:

1. $\mathcal{O}(|V|)$, visits each node once
2. $\mathcal{O}(|V| + |E|)$, graph search
3. $\mathcal{O}(|V| + |E|)$, graph search
4. $\mathcal{O}(|V|)$, visits each node once

LEMMA 8 (2 (CORRECTNESS OF THE ALGORITHM 1)). *The write references in `White` nodes are re-occurring first writes of a region $\mathcal{R}$.*

PROOF. We prove that, after a segment containing RFW references identified by Algorithm 1) to a variable x is rolled back, x will get written again before any read access to x. Proof by contradiction.

Suppose the segment W_m containing an RFW write to variable x is rolled back and the first subsequent access to x is a read reference in segment R_n. W_m has `White` color, R_n has reference type `Read`. Let P be the node to the end of which the execution is rolled back. W_1 through W_{m-1} are the nodes on the path from P to W_m. R_1 through R_{n-1} are the nodes on the path from P to R_n. Initially, the nodes R_1 through R_{n-1} had reference type `Null`. Step 2 marked all of them `Read` since R_n is `Read`. Step 3, when visiting P, colored W_1 `Black` because R_1 is `Read`, and subsequently colored all nodes W_2 through W_m `Black` as well. This contradicts the assumption that W_m is `White`. □

Note that the algorithm relies on the compiler's ability to identify references that go to the same address. Two references $\hat{x}$ and $\hat{y}$ cannot be assumed to access the same variable if there is any execution scenario in which the address may be different. Examples of such scenarios are subscripted array subscripts or variables whose address itself may be speculative. Both the used programming language and the architecture may give guarantees that certain addresses are always correct. In our initial implementation we use Fortran programs, whose variable addresses are statically known. In addition, we rely on our architecture's ability to guarantee that loop variables are non-speculative (this is implemented through proper synchronization). Therefore, our compiler can assume that all array references using affine subscript expressions have correct addresses and thus candidates for re-occurring first writes.

Figure 3 shows examples of finding RFW for variables x, y, and z. In Figure 3 (b) and (d), the write references in all successors of the segment 3 cannot be RFW because the segment 3 has two paths to `Read` and `Write` passing a `Null` node. In Figure 3 (c), the write references in all successors of the segment 3 may be RFW because the segment 3 has only paths to `Write` nodes.

4.2.3 Labeling Idempotent References

Given a region $\mathcal{R}$, the algorithm labels all idempotent references.

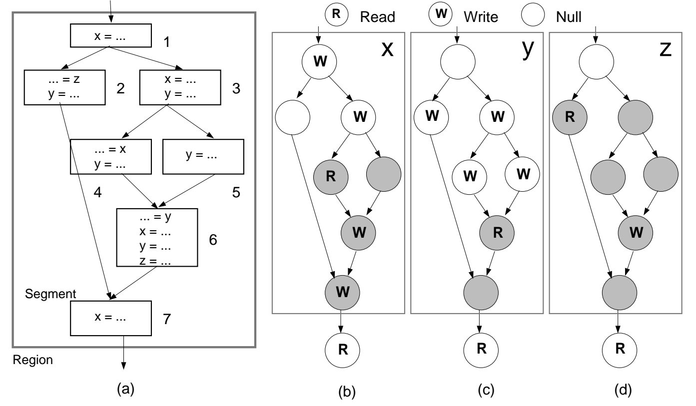

Figure 3: Example of re-occurring first writes. Small boxes represent segments, showing Must Definitions (`MDef` and Exposed Reads `ExR`. (a) Control flow diagrams of segments. (b) Graph marked for variable `x`, (c) variable `y`, and (d) variable `z`.

ALGORITHM 2. *Identifying idempotent references in region $\mathcal{R}$. At the beginning all references are labeled speculative.*

1. *Analyze read-only and private variables, and reference-by-reference dependence of shared variables in $\mathcal{R}$.*
2. *Analyze first re-occurring write references.*
3. *If $\mathcal{R}$ is fully independent with respect to data and control dependences, then*
   - *Label all references in $\mathcal{R}$ as idempotent.*
4. *otherwise (dependent region),*
   - *Label all read-only references as idempotent.*
   - *Label all private references as idempotent.*
   - *For each RFW reference, if the reference is not the sink of a cross-segment dependence, label the reference idempotent.*
   - *For each read reference, label the reference idempotent if*
     - *the reference is not the sink of any dependence, OR*
     - *the reference is the sink of an intra-segment dependence AND the source is labeled idempotent.*

Example

Figure 4 shows a serial loop, `BUTS_do1` in `APPLU`, which includes many nested small loops. The outermost loop is defined as our region and is parallelized speculatively by selecting each iteration (`k`) as a segment. The loop contains only one shared variable, `v`. Both references to `v` in statement S2 are dependent on the three references in S1. All of these three references are dependence sources only and hence can be labeled as idempotent by Theorem 2. Since the references in S2 are dependence sinks they must remain speculative.

```
      do k = nz-1, 2, -1
       do j = ny-1, 2, -1
        do i = nx-1, 2, -1
         do m = 1, 5
          ......
          do l = 1, 5
S1:          ... = v(l,i,j,k+1) + v(l,i,j+1,k)
     &            + v(l,i+1,j,k)
          end do
         end do
          ......
         do m = 1, 5
S2:        v(m,i,j,k)=v(m,i,j,k) - ...
         end do
        end do
       end do
      end do
```

Figure 4: Idempotent and speculative references in `APPLU BUTS_do1`.

5. EVALUATION

In this section, we empirically quantify execution overhead under HOSE, evaluate opportunity for labeling idempotent references in non-parallelizable code, and present performance results on applying our labeling algorithm on a selected group of segments. In the following, we first present an overview of our compiler infrastructure and experimental methodology.

We have developed a preliminary version of our algorithm on top of the Multiplex [13] compiler. Multiplex is a proposal for a a chip multiprocessor supporting both conventional and speculative execution of threads (i.e., segments). The Multiplex compiler integrates Polaris [1] and the Multiscalar compiler [14] into a single infrastructure for generating conventional and speculative threaded code.

We execute the code on a cycle-accurate simulator of Multiplex. In the rest of this paper, we assume Multiplex chips with four processors. Multiplex provides per-processor speculative storage, which is backed up by a full memory hierarchy serving as non-speculative storage. The compiler communicates reference idempotency labels for memory instructions to the hardware, to allow bypassing the speculative storage and placing the data directly in the non-speculative storage. As in conventional multiprocessors, the runtime system allocates a private stack for every segment. The compiler transforms and places the private variables into these per-segment private stacks.

5.1 Speculative Storage Overflow in HOSE

We have argued that execution under HOSE incurs significant speculative storage overflow. Figure 5 quantifies this problem. It shows the overheads incurred by a number of test programs taken from the SPEC CPU95 and the Perfect benchmark suites. We use small data sets (*test* or *train* for the SPEC benchmarks) in favor of reduced simulation time. The figure compares and contrasts execution overhead for Multiplex chips with a 4K-entry per-processor speculative storage, representative of practical implementations, and a 1M-entry per-processor speculative storage, serving as a reference point to gauge opportunity to reduce overhead. Each storage entry corresponds to a byte of data which is the smallest data unit for which Multiplex tracks and enforces dependence. The numbers correspond to optimal selection

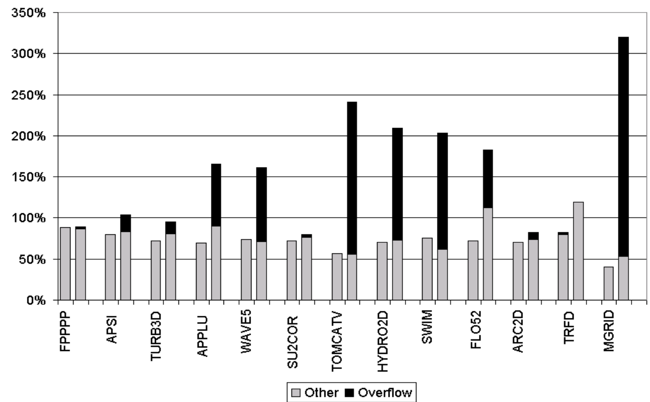

Figure 5: Execution overheads in speculative executions. The y-axis illustrates overhead using 1M-entry (left) and 4K-entry (right) speculative storage respectively. Both the 4K-entry and the 1M-entry numbers are measured overheads in cycles normalized to total execution time using 1M-entry storage. The figure shows overhead break down of speculative storage overflow and the sum of all other execution overheads (e.g., memory and pipeline stalls, runtime libraries, etc.).

of segment sizes to extract maximum parallelism from execution.

The figure indicates that the limited capacity of speculative storage introduces significant execution overhead and prevents the system from extracting parallelism. The actual magnitude of the overhead also varies across applications, with applications using larger segments to extract a high degree of parallelism (e.g., `TOMCATV, HYDRO2D`, `SWIM`, and `MGRID`) incurring higher speculative storage overhead. The figure also indicates that speculative storage overflow is completely absent in the large storage runs. As a result, a large opportunity for labeling idempotent references would reduce the pressure on the speculative storage and thereby reduce the likelihood for overflow.

5.2 Labeling Idempotent References

In this section, we first evaluate the opportunity for labeling idempotent references in all of our benchmarks. Next, we present performance results on removing idempotent references from speculative storage for selected groups of non-parallelizable loops. Each group of loops exhibits large opportunity for labeling a specific category of idempotent references. In the interests of simulation time and due to the difficulty of timing individual code sections, we only present results on individual loops. The loops, however, are representative of the rest of the non-parallelizable code sections.

A key question in this work is what fraction of the total references our algorithm can identify as idempotent in non-parallelizable code sections. To answer this question we have extracted from all the benchmarks the code sections that could not be automatically parallelized by our compiler. Note, that the parallelizable code sections are "uninteresting" from the point of view of this paper, because *all* data references can be marked idempotent (shown in Lemma 7). Figure 6 shows the fraction of total references in non-parallelizable code sections that our analysis

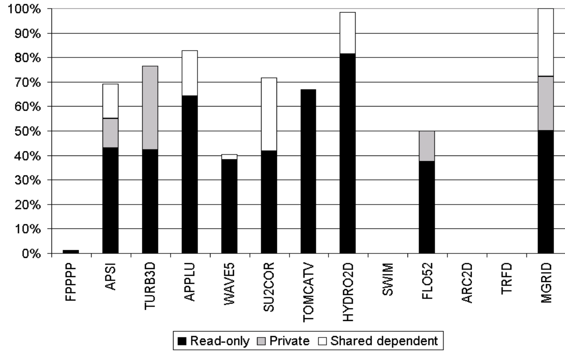

Figure 6: Fraction of idempotent references in code sections that cannot be detected as parallel. It shows idempotent references in the categories read-only, private, and shared-dependent.

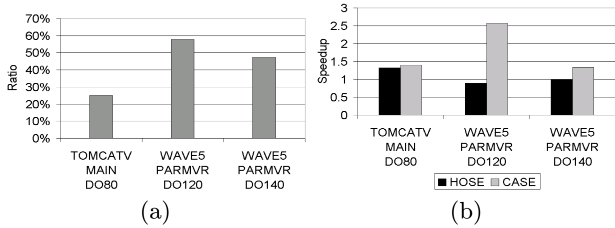

Figure 7: Examples of loops for idempotency category *read-only references*: (a) ratio of read-only references to total memory references, and (b) loop speedups before and after reference labeling.

was able to detect as idempotent. In 7 out of the 13 benchmarks more than 60% of these references are idempotent. The largest fraction is read-only idempotent variables. In four programs there is a substantial fraction of private idempotent variables. Most important is that the category of shared-dependent idempotent variables is a significant fraction is 5 benchmarks. The benchmarks with few or no idempotent variables fall into two opposite categories. `SWIM`, `TRFD` and `ARC2D` are fully-parallel programs, while `FPPPP` is known to be highly unstructured and difficult to analyze.

Figure 7 shows a selection of loops, `MAIN_do80` in `TOMCATV`, `PARMVR_do120` and `PARMVR_do140` in `WAVE5`, that have idempotent references in the read-only category. The figure shows the distribution of the read-only references with respect to the total memory references under CASE. The figure also shows loop speedups relative to a uniprocessor. Labeling the idempotent references in these loops reduces the pressure on the speculative storage, allowing for significant reductions in execution time. While array reduction can make the two loops in `WAVE5` fully independent, it would introduce significant execution overheads, offsetting the gains from the transformation.

Figure 8 shows the fraction of references and speedups under CASE in two loops, `DRCFT_do2` in `TURB3D` and `SETBV_do2` in `APPLU` that have idempotent references in the private category. In `SETBV_do2`, a significant fraction (about half) of the total memory references are private. Moving the private variables to per-segment stacks adds a large number of instructions to each segment for setting up private address spaces. Nevertheless, there are small speedup gains under

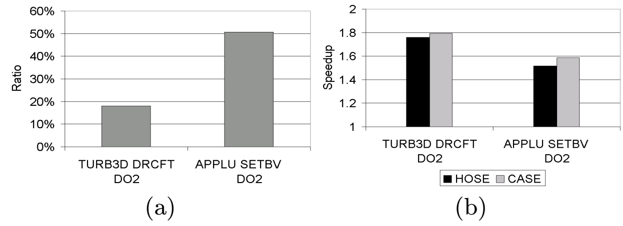

Figure 8: Examples of loops for idempotency category *private references*: (a) ratio of private read and write references to total memory references, and (b) loop speedups before and after reference labeling.

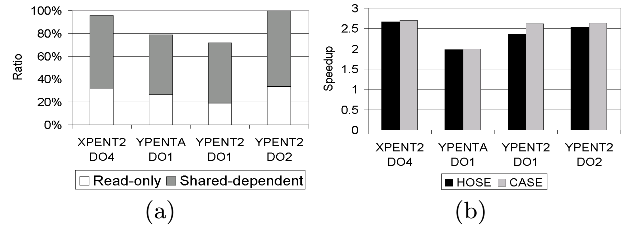

Figure 9: Examples of loops for idempotency category *shared dependent*: (a) ratio of idempotent references to the total memory references, and (b) loop speedups before and after reference labeling.

CASE as compared to HOSE.

Figure 9 shows loops including idempotent references in the shared-dependent and read-only category. The figure shows idempotent references as a fraction of the total number of references, and the corresponding loop speedups after labeling under HOSE and CASE. The ability to remove shared-dependent references from speculative storage is one of the most advanced qualities of the presented compiler techniques. The fact that there are program sections with more than 50% idempotent shared dependent references is an important result. Note, that these loops are not independent and thus cannot be parallelized by our current compiler technology.

Figure 10 includes all references in fully independent regions in three major loops of the program `MGRID`. This category applies to do loops with fully independent iterations. CASE improves the performance significantly over HOSE, which incurs significant speculative storage overflow. Figure 10 (b) shows that read-only references represent the major category of idempotent references in `RESID_do600` and `PSINV_do600`, and write shared references represent the major category in `ZRAN3_do400`.

6. CONCLUSIONS

We have discovered a new program property called reference idempotency to alleviate speculative storage overflow, a critical limitation of speculative execution. Idempotent references' key feature is that they do not cause any data-dependence violations on their own, although they may propagate incorrect values. Because the initial incorrect values are eventually corrected and propagated, idempotent references need not be tracked in speculative storage, though the reference is temporarily incorrect. Idempotent references, even from non-parallelizable program sections,

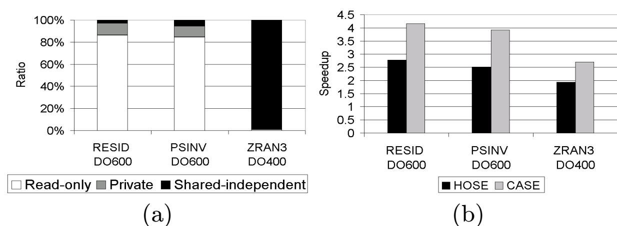

Figure 10: Examples of loops for idempotency category *fully independent regions*: (a) ratio of idempotent references, and (b) loop speedups before and after reference labeling.

can directly access non-speculative storage. By filtering out idempotent references, we reduce the demand for speculative storage space in large threads, uncovering more parallelism without incurring much overflow.

We defined a formal framework for idempotency and presented a novel compiler-assisted speculative execution model. We proved the necessary and sufficient conditions for reference idempotency under our model. We also presented a compiler algorithm to label idempotent memory references for the hardware. Experimental results show that for our benchmarks, over 60% of the references in non-parallelizable code sections are idempotent.

Reference idempotency enables compilers to deal with code sections that are unanalyzable by classical compiler techniques. The current generation of compilers is most capable of optimizing program sections for which the absence of data dependences can be proven. While such analysis applies to many regular programs, a large number of programs are irregular in nature. Reference idempotency applies to these very programs. With architectural support – in the form of the proposed compiler-assisted speculative execution model – it enables new optimizations where conventional compiler techniques face hard limits.

7. REFERENCES

[1] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu, "Parallel programming with Polaris," *IEEE Computer*, pp. 78–82, Dec. 1996.

[2] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, pp. 84–89, Dec. 1996.

[3] Lawrence Rauchwerger and David Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[4] Manish Gupta, "Techniques for speculative run-time parallelization of loops," in *International Conference on Supercomputing (ICS'98)*, 1998.

[5] Lance Hammond, Mark Willey, and Kunle Olukotun, "Data speculation support for a chip multiprocessors," in *The Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, October 1998.

[6] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry, "A scalable approach to thread-level speculation," in *The 27th Annual International Symposium on Computer Architecture (ISCA-27)*, June 2000.

[7] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *The 22th International Symposium on Computer Architecture (ISCA-22)*, June 1995, pp. 414–425.

[8] Sun Microsystems, "MAJC architecture tutorial," *White Paper*, September 1999.

[9] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas, "Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors," in *The Fifth International Symposium on High-Performance Computer Architecture (HPCA-5)*, January 1999.

[10] Sridhar Gopal, T.N. Vijaykumar, James E. Smith, and Gurindar S. Sohi, "Speculative versioning cache," in *The Fourth IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, Jan. 1998, pp. 195–205.

[11] Utpal Banerjee, *Dependence Analysis for Supercomputing*, Kluwer. Boston, MA, 1988.

[12] Peng Tu and David Padua, "Automatic array privatization," in *Proceedings of Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, Eds., August 1993, vol. 768, pp. 500–521.

[13] Seon Wook Kim, Chong-Liang Ooi, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar, "Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor," Tech. Rep. TR-ECE 00-13, School of ECE, Purdue University, 2000.

[14] T. N. Vijaykumar and Gurindar S. Sohi, "Task selection for a multiscalar processor," in *The 31st International Symposium on Microarchitecture (MICRO-31)*, December 1998.