Exploiting Reference Idempotency to Reduce Speculative Storage Overflow

SEON WOOK KIM
Korea University
CHONG-LIANG OOI and RUDOLF EIGENMANN
Purdue University
BABAK FALSAFI
Carnegie Mellon University
and
T. N. VIJAYKUMAR
Purdue University

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 article, 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 nonparallelizable program sections, need not be tracked in the speculative storage, and instead can directly

This work was performed while the authors were at Purdue University. A short version of the article appears in the *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, Snowbird, UT, June 18–21, 2001.
This material is based upon work supported in part by the National Science Foundation under Grant no. 9703180, 9975275, 9986020, and 9974976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.
Authors' addresses: S. W. Kim, Department of Electronics and Computer Engineering, Korea University, Anam-dong Seongbuk-Gu, Seoul, 136-701 Korea; C.-L. Ooi, R. Eigenmann (contact author), and T. N. Vijaykumar, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907; email: eigenman@ecn.purdue.edu; B. Falsafi, Computer Architecture Laboratory, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213-3891.

access nonspeculative 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 nonparallelizable program sections are idempotent.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compiler*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Multiple-instruction stream, multiple-data-stream processors (MIMD)*

General Terms: Algorithm, Design, Experimentation, Performance, Theory

Additional Key Words and Phrases: Idempotent references, speculation, compiler-assisted speculative execution

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 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 which are unknown at compile time and therefore not entirely parallelizable [Blume et al. 1996; Hall et al. 1996]. Runtime data dependence tests can parallelize certain unanalyzable code sections [Rauchwerger and Padua 1995; Gupta 1998]. 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 [Hammond et al. 1998; Steffan et al. 2000; Sohi et al. 1995; Sun Microsystems 1999]) 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, threads commit the speculative data from speculative storage to *nonspeculative* 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 that is 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—for example custom hardware buffers [Sohi et al. 1995; Zhang et al. 1999] or level-one data caches [Gopal et al. 1998]. 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 access 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 references made by the provably parallel sections in speculative storage and direct such references to nonspeculative storage, avoiding speculative storage overflow for those sections. In nonparallelizable sections, however, the hardware blindly tracks data dependence for *all* memory references, increasing the chances of speculative storage overflow.

In this article, we discover a new program property called *memory reference idempotency*. Idempotent references can be directly placed in nonspeculative instead of speculative storage, even if the references are from nonparallelizable sections. Reference idempotency is based on our fundamental insight that in speculative execution, incorrect values are created due to dependence violations and propagated through subsequent computation. A key feature of idempotent references is that they maintain all data-dependence relationships, although they may propagate incorrect values. Because initial incorrect values are eventually corrected and repropagated, 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), it 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 article 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 them directly in nonspeculative storage.
—We show results in which, for our benchmarks, over 60% of the references in nonparallelizable code sections are idempotent.

This article 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. In Section 4 we introduce reference idempotency, prove its necessary and sufficient conditions, and describe a compiler algorithm for idempotency analysis. Section 5 shows experimental results on the frequency of idempotent references, and Section 6 presents conclusions.

An Introductory Example. Current proposals for speculative multithreaded processors assume a *hardware-only speculative execution* (HOSE) model. In

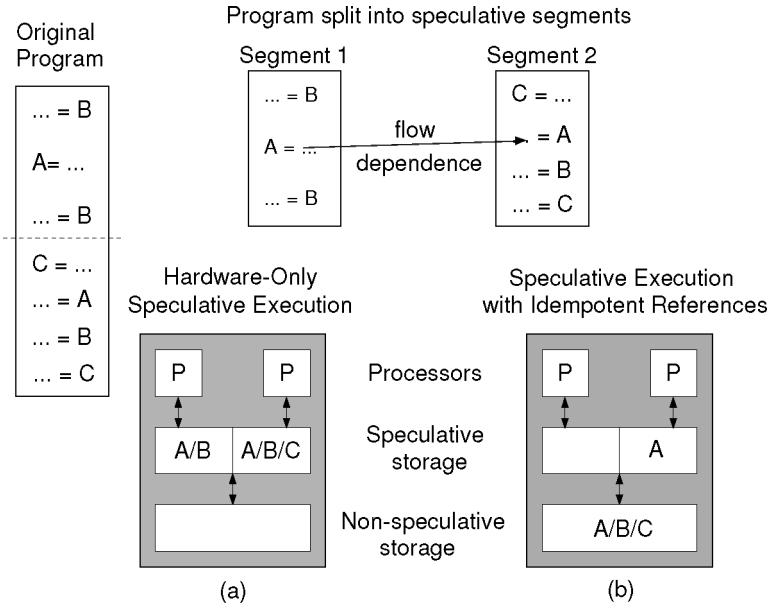

Fig. 1. Basic ideas of labeling idempotent references. (a) In hardware-only speculation, all data is placed in the speculative storage. (b) After labeling idempotent references, these references can go directly to the nonspeculative storage.

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 intersegment 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 that are produced or referenced in speculative storage. The data values remain there 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 speculative storage are transferred, or "committed," to nonspeculative 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 restarting 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, that is, references that do not require buffering in speculative storage and can directly access nonspeculative storage. First, the compiler can identify all references to variable B as 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. However, to enforce dependences, the write reference 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 nonspeculative storage, without occupying any space in speculative storage. Third, variable C in this example is private to segment 2, that is, there are no dependences across segments on this variable and all references to it are idempotent. Although segment 2 may reexecute due to incorrect speculation, the write reference C always occurs first whenever the segment is reexecuted. Hence, even if an incorrect value was written initially, the value of C will be corrected in the final execution of the segment.

Generalizing this example, our intuition for idempotent references is as follows. Read-only variables are always idempotent. Their values are invariant of any control and data speculation. The similar holds for private variables—variables whose values are both created and consumed within a single segment. Although a misspeculated segment may write incorrect values to private variables, the final, correct execution of the segment will write and read the correct values. If the same segment is never reexecuted in a correct form (i.e., its execution is a control misspeculation), the values are also never consumed. However, care must be taken at the boundaries of regions within which the read-only and private properties hold. The most interesting category of idempotent references are those that access truly shared variables. Our key idea is that if we can prove that an incorrect value is eventually corrected and propagated to all consumers, such accesses are also idempotent. An important underlying assumption is that the addresses of idempotent variables are invariant of speculative execution. This is the case for a large number of references in our test programs.

Proving correctness of this intuition is nontrivial. The following sections deliver this proof in a concise form. We begin by precisely defining and proving correctness of the speculative execution mechanisms. Next, we consider different types of data-dependence relationships and show which data references involved in such dependences can be labeled idempotent. Finally, Theorems 4.7 and 4.8 give the necessary and sufficient conditions for labeling data accesses as idempotent.

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 2.1 (*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, which is introduced in Section 3.

Definition 2.2 (*HOSE Mechanism*). A hardware-only speculative execution is an execution mechanism for the programs given in Definition 2.1 with the following properties:

(1) *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.

(2) *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 5) and *rolls-back* incorrect segments. Upon a roll-back, all data generated by the segment is discarded (see Property 4). This process may repeat several times.

(3) *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 is incorrectly started due to control misspeculation, the final execution may execute a different segment or may be empty.

(4) *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 speculative storage. They do not affect nonspeculative storage until the segment is *committed* (see Property 6). If a read reference accesses a location that is not present in the speculative storage, then the value is fetched from the youngest ancestor that contains a value for this location or from nonspeculative storage if no ancestor contains this location. A write reference affects only the segment's own speculative storage.

(5) *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 4) that are committed in sequential order (Property 6).

(6) *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 nonspeculative 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 2.3 (*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 nonspeculative 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 nonspeculative storage have the same value as in a sequential execution of the program.

LEMMA 2.4 (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 $\mathcal{R}_1 \ldots \mathcal{R}_n$ be the segments in region $\mathcal{R}$ from oldest to youngest. To satisfy the correctness criterion of Definition 2.3, 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 nonspeculative 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 1). All data references use the segment's own speculative storage, and this storage cannot be modified by any other segment (HOSE Property 4). Hence, the segments execute and produce the same final values as a sequential program if we can show that the 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 5);

and (b) by HOSE Property 4, 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 nonspeculative storage (where they are correct, given the preceding region's correct execution).

(2) By HOSE Property 6, all segments commit in sequential order. Therefore, all segments' values will be seen in nonspeculative 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 6, 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 2.1. As in HOSE, segments are the primary units of speculative execution. Regions are important for enclosing the code sections in which certain data attributes hold (e.g, read-only or dependence-free). The execution mechanism is defined as follows.

Definition 3.1 (*CASE Mechanism*). A compiler-assisted speculative execution is a program execution mechanism with the basic properties of HOSE, as given in Definition 2.2. 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 nonspeculative 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, their value is then directly placed in nonspeculative storage and no information about the references is kept in the speculative storage.

From the definition of idempotent references, it follows that the references access nonspeculative storage and do not occupy any space in speculative storage. Thus, idempotent references help reduce speculative storage overflow, as motivated in Section 1. Note that 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 reexecution) and a hardware property (the memory reference accesses nonspeculative storage.)

3.2 Correctness of CASE

In CASE, programs contain both speculative and idempotent references. The hardware guarantees correctness for speculative references, as in HOSE.

However, 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 this end, the following labeling conditions must be satisfied by candidate idempotent references.

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, that is, 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 nonspeculative 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 nonspeculative storage. A read reference is correctly labeled as idempotent only if it obtains from nonspeculative storage the value generated by any prior write reference. (If either the source or sink of a flow-dependence is a speculative reference and 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 3.2 (CORRECTNESS OF CASE). *CASE is correct under Definition* 2.3 *if and only if all idempotent references satisfy the three labeling conditions LC*1 *through LC*3.

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

[1]We use the notation v for variables and $\hat{x}$ for memory 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 nonspeculative 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 from the values produced in speculative storage in HOSE is that instructions may consume input values through read references that are 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 either produced or 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 in a program the variable references 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 4.7 and 4.8 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 *recurring first write* will be useful. It is defined as follows.

Definition 4.1 (*Recurring First Write* (*RFW*)). A write reference to the variable v in segment $\mathcal{R}_i$ is an 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 2.2, 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 an 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 2.2.

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 nontrivial in the presence of pointers and subscripted subscripts. The compiler must guarantee that the references in misspeculated and 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 one region at a time. The data-dependences are assumed to have been analyzed for the region on a reference-by-reference basis. Note that this means there are only data-dependences between references to the same variable. Only intraregion dependences are considered.

LEMMA 4.2 (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. Thus, the hardware will not enforce the dependence because of HOSE Property 5. If the sink is a write reference, it directly writes to the nonspeculative storage, violating the dependence. In both cases, the labeling criterion LC2 is not satisfied, which contradicts the assumption. □

LEMMA 4.3 (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 $\hat{y}$. Intrasegment dependences are always satisfied because of the sequential execution of segments. A cross-segment dependence is also satisfied because $\hat{y}$ is labeled speculative by Lemma 4.2. 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 Properties 4 and 6). Hence LC2 is satisfied. LC3 is not applicable because there is no write reference preceding $\hat{x}$. □

LEMMA 4.4 (INDEPENDENT RFW). *A recurring 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 recurring first write. By Definition 4.1, 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, intrasegment 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 by Definition 3.1 as long as the sink is speculative. This is the case by Lemma 4.2. 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 nonspeculative memory upon the commit of the segment containing $\hat{y}$, which is *after* the reference $\hat{x}$ (HOSE Property 6). 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 4, the speculative read reference consumes the value from the nonspeculative 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 4.5 (COVERED READ). *A read reference $\hat{y}$ that is dependent on an idempotent RFW reference $\hat{x}$ within the same segment can be labeled idempotent.*

PROOF. LC2 and LC3 need to be considered. For LC2, all intrasegment dependences are satisfied because of the sequential execution of segments. Write references are only labeled idempotent with Lemma 4.4. 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 4.3. Therefore, LC2 is satisfied. LC3 is also satisfied because an idempotent $\hat{y}$ correctly reads the value generated by an idempotent $\hat{x}$ in nonspeculative storage. □

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

LEMMA 4.6 (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 nonspeculative, 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 when in their final and correct form. Labeling them as idempotent satisfies all three labeling criteria trivially. □

Lemmas 4.2 through 4.5 provide the basis for proving necessary and sufficient conditions for idempotent read and write references in segments that include dependences.

THEOREM 4.7 (IDEMPOTENT WRITE). *A write reference is idempotent if and only if it is a recurring first write and is not the sink of a cross-segment dependence.*

THEOREM 4.8 (IDEMPOTENT READ). *A read reference is idempotent if and only if it is not the sink of any data-dependence or is dependent on an idempotent write reference within the same segment.*

PROOF (IDEMPOTENT WRITE). By Lemma 4.4, an RFW that is not the sink of a 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 an RFW cannot be labeled idempotent. By Lemma 4.2, a cross-segment dependence sink cannot be labeled idempotent. If a reference, $\hat{x}$, to variable v is not an RFW, then after the enclosing

segment rolls-back, execution can take a path that (Case 1) does not reference v or (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 4.3, a read reference that is not the sink of a data-dependence can be labeled idempotent. By Lemma 4.5, 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: (Case 1) The source is in a different segment; and (Case 2) the source within the same segment is labeled speculative. By Lemma 4.2, 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 preceeding 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 code sections containing both dependent and independent references is nontrivial because of the interaction of idempotent and speculative references. This is shown in the proof of Lemma 4.3.

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 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 the 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.

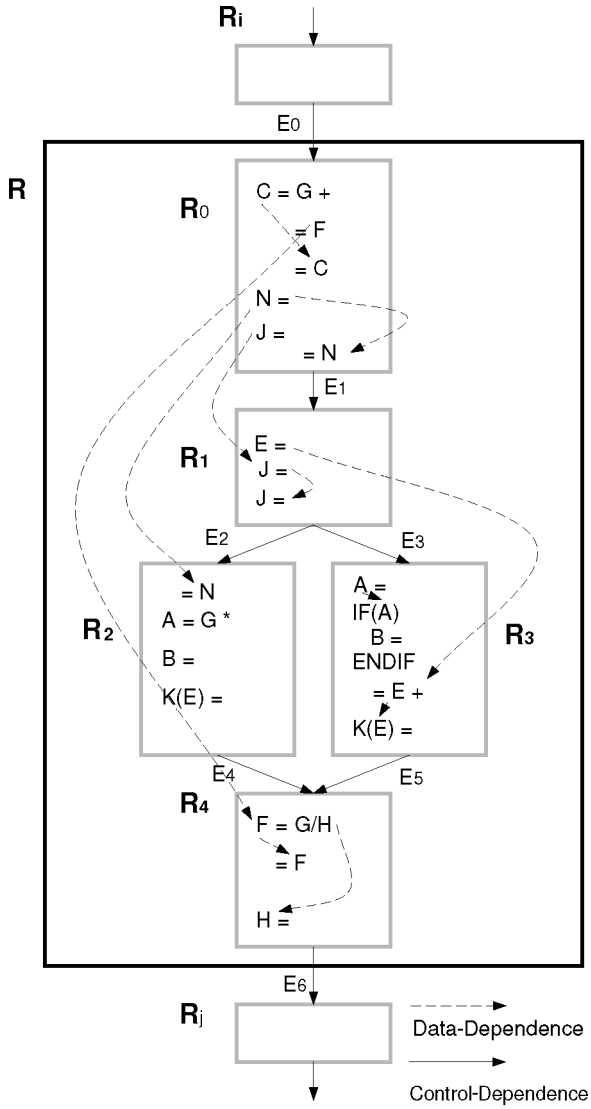

Fig. 2. Example code with control- and data-dependence graphs. The region $\mathcal{R}$ contains five segments, $\mathcal{R}_0, \ldots, \mathcal{R}_4$.

It is important to note that these write references may produce temporarily incorrect values in nonspeculative memory. The idempotency property guarantees that correctness is still ensured.

Examples. Figure 2 shows several examples. By Definition 4.1, $\texttt{RFW}(\mathcal{R}_0)$ = {C, N, J}, $\texttt{RFW}(\mathcal{R}_1)$ = {E, J}, $\texttt{RFW}(\mathcal{R}_2)$ = {A}, $\texttt{RFW}(\mathcal{R}_3)$ = {A}, and $\texttt{RFW}(\mathcal{R}_4)$ = {F}. The reference to B in $\mathcal{R}_2$ is not an RFW because the reference to B is not guaranteed to execute if $\mathcal{R}_2$ is rolled-back. Similarly, the reference to B in $\mathcal{R}_3$ is not an RFW. The write reference to H in $\mathcal{R}_4$ is preceeded by a read. The references

to K(E) in $\mathcal{R}_2$ and $\mathcal{R}_3$ are not RFWs because E is not read-only. All the RFW references are idempotent, except for J in $\mathcal{R}_1$ and F in $\mathcal{R}_4$. These references to J and F are not idempotent by Lemma 4.4 because they are the sink of output and antidependences from $\mathcal{R}_0$.

The read references to N in $\mathcal{R}_2$ and E in $\mathcal{R}_3$, and a write reference to F in $\mathcal{R}_4$ are speculative by Lemma 4.2 because they are sinks of cross-segment dependences. All references to variable G in $\mathcal{R}$, F in $\mathcal{R}_0$, and the read of H in $\mathcal{R}_4$ are idempotent by Lemma 4.3 because they are independent reads. The read references to N and C in $\mathcal{R}_0$, A in $\mathcal{R}_3$, and F in $\mathcal{R}_4$ are idempotent by Lemma 4.5 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 article. In our evaluation, regions are loops and segments are loop iterations. Furthermore, we assume that read-only variables, private variables, and data-dependences have been analyzed with state-of-the-art compiler techniques (e.g., Banerjee [1988], Tu and Padua [1993]). The following algorithm finds recurring first write references.

4.2.2 *Analyzing Recurring First Write References.* Recall that by Definition 4.1, a write reference to a variable, x, in segment $\mathcal{R}_i$ is an RFW if 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-RFWs for a given variable x if any successor has an exposed read reference to x.

Algorithm 4.9. Identifying recurring 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 RFWs (`White`) or none are RFWs (`Black`). The algorithm finds this property.

(1) Initially, for each node v and 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). From each node with reference type `Read`, mark all `Null` successor nodes `Read`.

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

(3) Search G (depth- or breadth-first). From each node v, if v is `Black` or any successor has reference type `Read`, color all successors `Black`.
(4) All write references to x in `White` nodes are recurring first writes.

The complexity of the algorithm is $\mathcal{O}(|V|+|E|)$ for each variable because the steps of Algorithm 4.9 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 4.10 (CORRECTNESS OF ALGORITHM 4.9). *The write references in* `White` *nodes are recurring first writes of a region,* $\mathcal{R}$.

PROOF. Let x be a variable to which Algorithm 4.9 has identified RFW references in segment W_m. We prove that after W_m 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 also colored all nodes W_2 through W_m `Black`. 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 programming language used 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 nonspeculative (this is implemented through proper synchronization). Therefore, our compiler can assume that all array references using affine subscript expressions have correct addresses and thus are candidates for recurring first writes.

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

4.2.3 *Labeling Idempotent References.* Given a region $\mathcal{R}$, the algorithm labels all idempotent references.

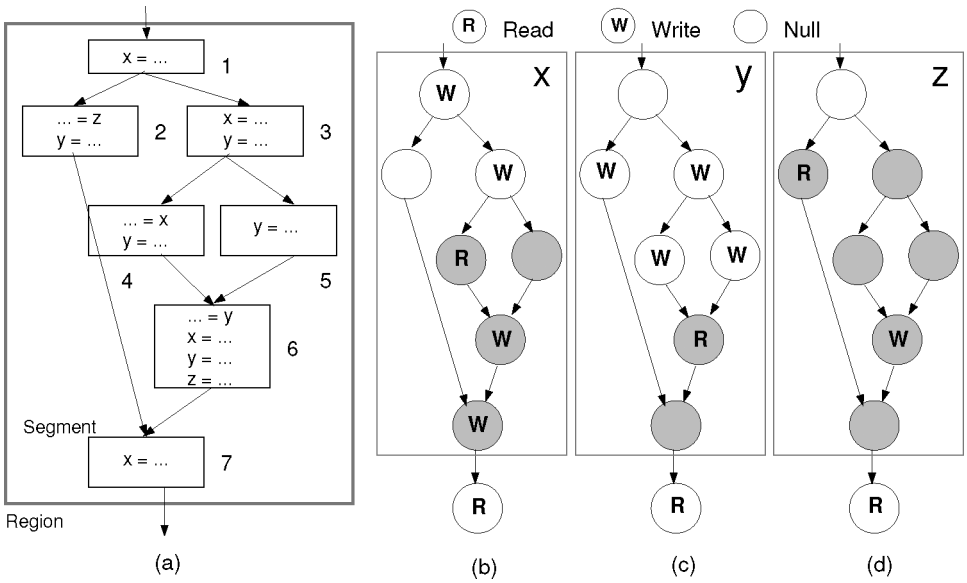

Fig. 3. Example of recurring first writes (a) control flow diagrams of segments (b) graph marked for variable x (c) variable y and (d) variable z.

Algorithm 4.11. Identifying idempotent references in region $\mathcal{R}$. At the beginning, all references are labeled speculative.

(1) Analyze read-only, private, and reference-by-reference dependence of shared variables in $\mathcal{R}$ (using classical analysis).
(2) Analyze first recurring write references (using Algorithm 4.9).
(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 as 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 intrasegment 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 three references are dependence sources only, and hence can be labeled as idempotent by Theorem 4.8. 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)      |   idempotent references
     &           + 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) - ...             |   speculative references
         end do
        end do
       end do
      end do
```

Fig. 4. Idempotent and speculative references in APPLU BUTS_do1.

5. EVALUATION

In this section, we empirically quantify execution overhead under HOSE, evaluate the opportunity for labeling idempotent references in nonparallelizable 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 [Ooi et al. 2001] compiler. Multiplex is a proposal for a chip multiprocessor supporting both conventional and speculative execution of threads (i.e., segments). The Multiplex compiler integrates the Polaris [Blume et al. 1996] and Multiscalar compiler [Vijaykumar and Sohi 1998] 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 article, we assume Multiplex chips with four processors. Multiplex provides per-processor speculative storage which is backed up by a full memory hierarchy serving as nonspeculative storage. The compiler communicates reference idempotency labels for memory instructions to the hardware so as to allow bypassing the speculative storage and placing the data directly in nonspeculative 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 used the *train* data set for the SPEC benchmarks.[3] The

[3]In the interest of reduced simulation time, we have reduced the train data set in SU2COR, TOMCATV, HYDRO2D, and SWIM.

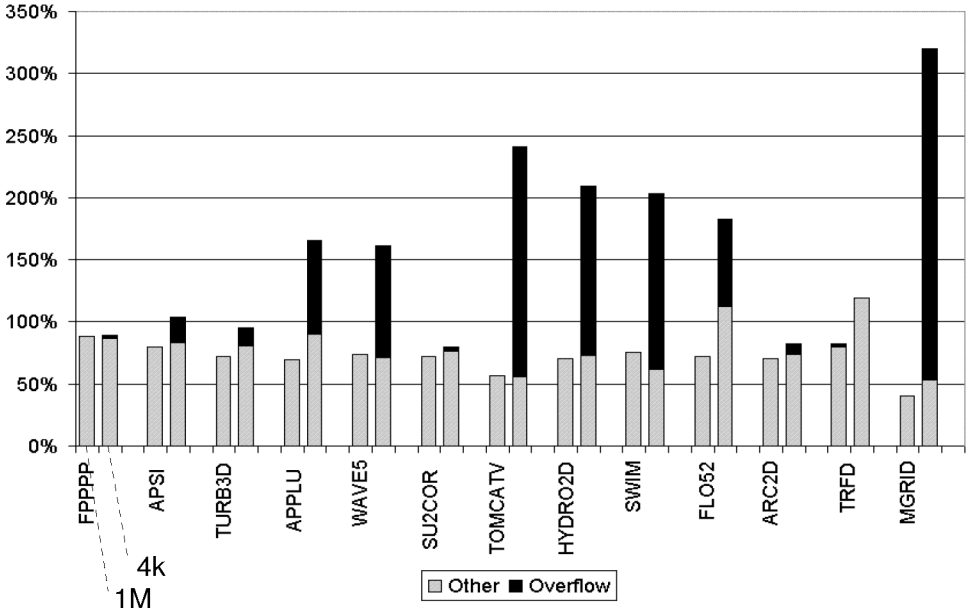

Fig. 5. Execution overheads in speculative executions. The *y*-axis illustrates overhead using 1M-entry (left) and 4k-entry (right) speculative storage, respectively. Both the 4k-entry and 1M-entry numbers are measured overheads in cycles that are normalized to total execution time using 1M-entry storage. The figure shows overhead breakdown of speculative storage overflow and the sum of all other execution overheads (e.g., memory and pipeline stalls, runtime libraries, etc.). The difference between the 1M-bars and 100% represents useful work.

figure compares and contrasts execution overhead for Multiplex chips with a 4k-entry per-processor speculative storage that is representative of practical implementations, and a 1M-entry per-processor speculative storage that serves as a reference point to gauge the opportunity for reducing overhead. Each storage entry corresponds to a byte of data, which is the smallest data unit for which Multiplex tracks and enforces dependences. Segments have been chosen so that maximum program parallelism is extracted.

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 that use 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 the performance results of

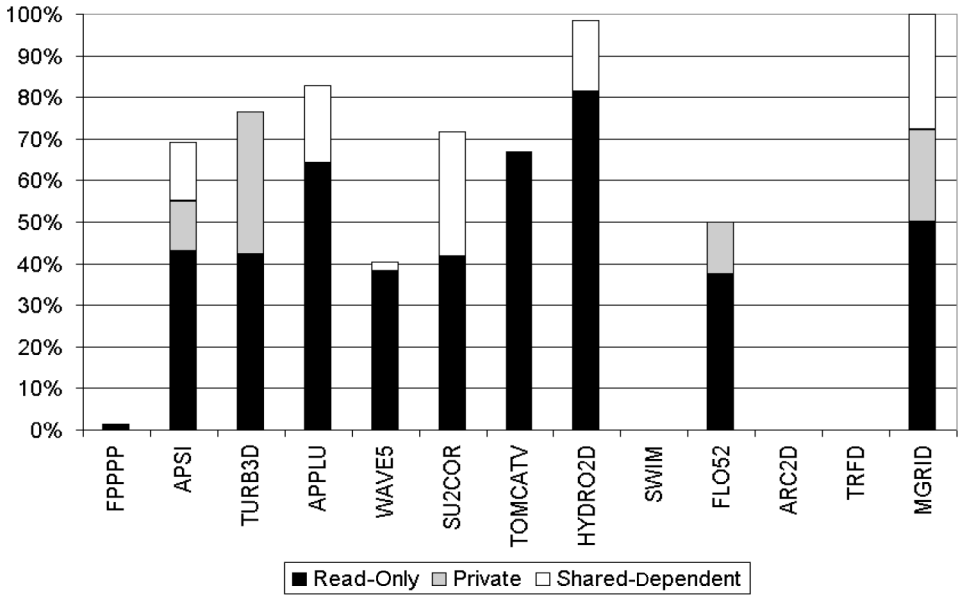

Fig. 6. Fraction of idempotent references in code sections that cannot be detected as parallel. It shows the dynamic counts of idempotent references in the categories read-only, private, and shared-dependent.

removing idempotent references from speculative storage for selected groups of nonparallelizable loops. Each group of loops exhibits a large opportunity for labeling a specific category of idempotent references. The goal of these results is to provide a proof of concept, and not exhaustive evaluation. The loops shown are representative of the nonparallelizable code sections in our benchmarks. As parallel code sections are trivially idempotent, adding these numbers would inflate our results, hence they are excluded.

A key question in this work is what fraction of the total references our algorithm can identify as idempotent in nonparallelizable code sections. To answer this question, we extracted from all 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 article, because *all* data references can be marked idempotent (as shown in Lemma 4.6). Figure 6 shows the fraction of total references in nonparallelizable code sections that our analysis 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 of 5 benchmarks. These benchmarks with few or no idempotent variables fall into two opposing 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`, and `PARMVR_do120` and `PARMVR_do140` in `WAVE5`, that have idempotent references in the read-only

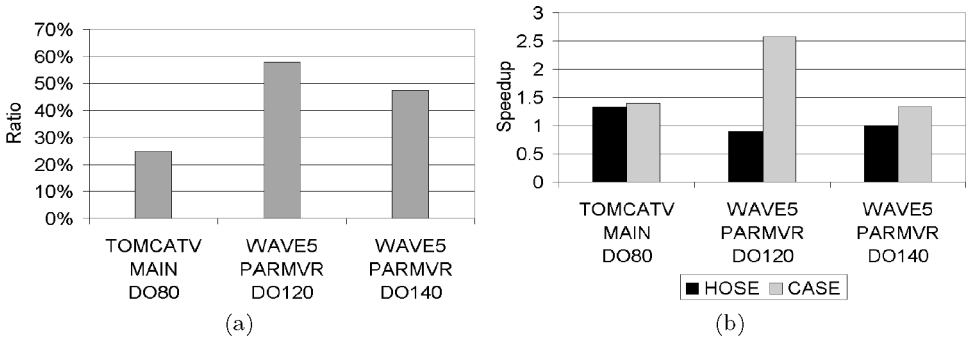

Fig. 7. Examples of loops for idempotency category read-only references: (a) ratio of read-only references to total memory references, and (b) loop speedups before and after reference labeling.

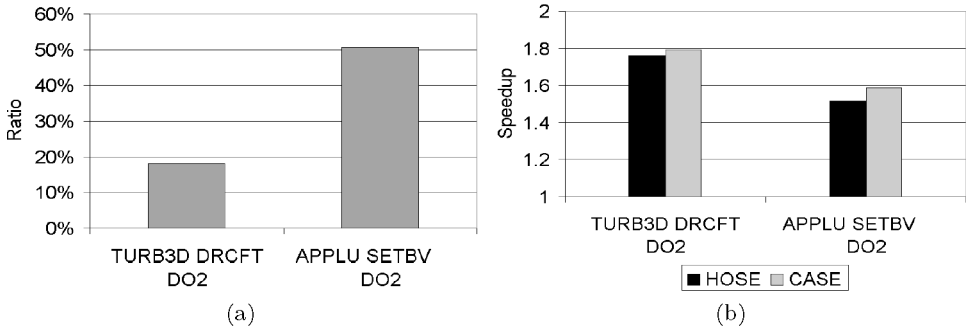

Fig. 8. Examples of loops for idempotency category private references: (a) ratio of private read and write references to total memory references, and (b) loop speedups before and after reference labeling.

category. The figure shows the distribution of read-only references with respect to the total memory references under CASE. It 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 also 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. In our system, private variables are implemented in software on per-segment stacks, the setup of which adds a large number of instructions to each segment. Nevertheless, there are small speedup gains under CASE, as compared to HOSE.

Figure 9 shows loops including idempotent references in the shared-dependent and read-only categories. The figure shows idempotent references

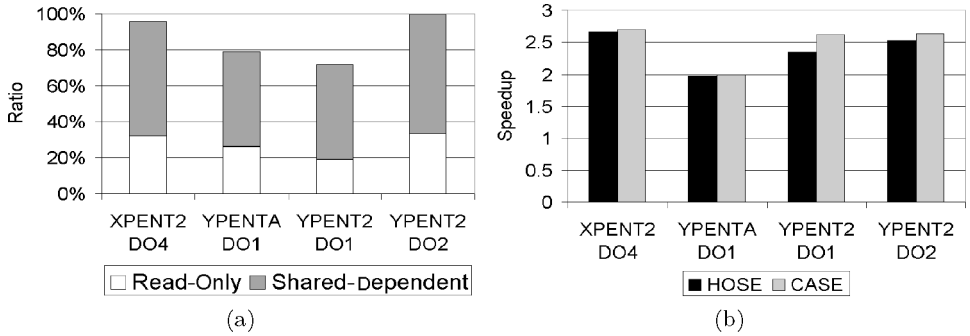

Fig. 9. Examples of loops for idempotency category shared-dependent: (a) ratio of idempotent references to the total memory references, and (b) loop speedups before and after reference labeling.

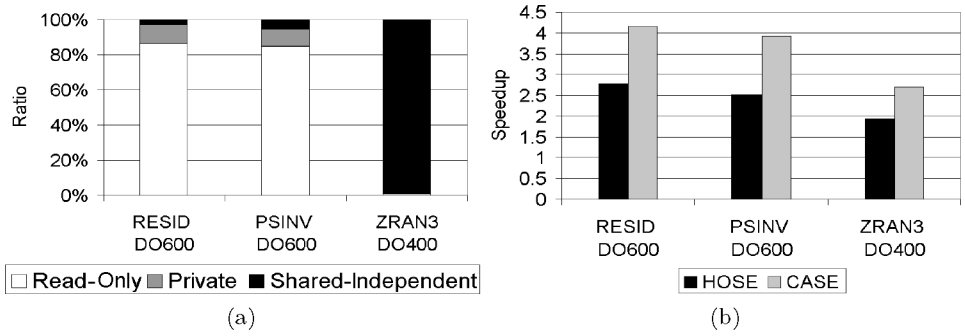

Fig. 10. Examples of loops for idempotency category fully independent regions: (a) ratio of idempotent references, and (b) loop speedups before and after reference labeling.

as a fraction of the total number of references, and the corresponding loop speedups after labeling under both 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 with state-of-the-art 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 as compared to HOSE, which incurs significant speculative storage overflow. Figure10(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`.

Impact of Architecure Parameters. We now briefly discuss the impact of architectural parameters on reference idempotency. Because the results in Figures 6 through 10 show the variable-level behavior of the programs, they are not architecture-dependent. These results do not depend on system granularity

such as cache block size. However, because speculative storage is implemented through the cache hierarchy in many proposals, the parameters of the hierarchy affect the effectiveness of our technique. Needless to say, the smaller or less associative the cache is, the larger the chances of overflow. The L1 and even L2 caches are usually small and not sufficiently highly associative (e.g., 32k two-way set-associative L1 cache) to maintain low access latencies. Even as technology scales to allow more transistors and larger caches, L1 and L2 caches will continue to be small (although more transistors may allow L3 or L4 caches), especially because the clock will also scale to higher speeds. Apart from size and associativity, cache block size also impacts the effectiveness of our technique. If both idempotent and nonidempotent references fall within the same cache block, then the entire cache block has to be buffered in speculative storage and may incur speculative storage overflow. If cache block size increases, then this false-sharing effect will worsen. To mitigate this effect, the compiler could be used to pad memory allocation such that variables with idempotent references do not share a cache block with nonidempotent variables. However, this requirement is no different from what is needed in conventional multiprocessors to avoid false-sharing due to hardware-cache-coherence granularity.

6. CONCLUSIONS

We have discovered a new program property, called reference idempotency, to alleviate speculative storage overflow, hitherto 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 even though the reference is temporarily incorrect. Idempotent references, even from nonparallelizable program sections, can directly access nonspeculative 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 nonparallelizable 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 just where conventional compiler techniques face hard limits.

REFERENCES

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer, Boston, MA.

BLUME, W., DOALLO, R., EIGENMANN, R., GROUT, J., HOEFLINGER, J., LAWRENCE, T., LEE, J., PADUA, D., PAEK, Y., POTTENGER, B., RAUCHWERGER, L., AND TU, P. 1996. Parallel programming with Polaris. *IEEE Comput.* 78–82.

GOPAL, S., VIJAYKUMAR, T., SMITH, J. E., AND SOHI, G. S. 1998. Speculative versioning cache. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*. 195–205.

GUPTA, M. 1998. Techniques for speculative runtime parallelization of loops. In *Proceedings of the International Conference on Supercomputing (ICS)*.

HALL, M. W., ANDERSON, J. M., AMARASINGHE, S. P., MURPHY, B. R., LIAO, S.-W., BUGNION, E., AND LAM, M. S. 1996. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Comput.* 84–89.

HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. 1998. Data speculation support for a chip multiprocessors. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

OOI, C.-L., KIM, S. W., PARK, I., EIGENMANN, R., FALSAFI, B., AND VIJAYKUMAR, T. N. 2001. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the International Conference on Supercomputing (ICS)*. ACM Press, 368–380.

RAUCHWERGER, L. AND PADUA, D. 1995. The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

SOHI, G. S., BREACH, S. E., AND VIJAYKUMAR, T. N. 1995. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA-22)*. 414–425.

STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. 2000. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA-27)*.

SUN MICROSYSTEMS. 1999. MAJC architecture tutorial. *White Paper*.

TU, P. AND PADUA, D. 1993. Automatic array privatization. In *Proceedings of 6th Workshop on Languages and Compilers for Parallel Computing* (Portland, OR), Lecture Notes in Computer Science, U. Banerjee et al., eds. vol. 768, 500–521.

VIJAYKUMAR, T. N. AND SOHI, G. S. 1998. Task selection for a multiscalar processor. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*.

ZHANG, Y., RAUCHWERGER, L., AND TORRELLAS, J. 1999. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA-5)*.

Received January 2005; revised June 2005; accepted October 2005