Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution*

Seon Wook Kim[1], Chong-Liang Ooi[1], Rudolf Eigenmann[1],
Babak Falsafi[2], T. N. Vijaykumar[1]

[1]School of Electrical and Computer Engineering, Purdue University†
[2]Department of Electrical and Computer Engineering, Carnegie Mellon University

mux@ecn.purdue.edu, http://www.ece.purdue.edu/~mux

ABSTRACT

Recent proposals for multithreaded architectures allow threads with unknown dependences to execute speculatively in parallel. These architectures use hardware *speculative storage* to buffer uncertain data, track data dependences and roll back incorrect executions. Because *all* memory references access the speculative storage, current proposals implement this storage using small memory structures for 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 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 need not be tracked in the speculative storage, and instead can directly access non-speculative storage (i.e., the conventional memory hierarchy). Thus, we reduce the demand for speculative storage space. 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.

1. INTRODUCTION

Multithreaded and multiprocessor architectures are emerging as attractive candidates for future high-performance single-chip computers. As in shared-memory multiprocessors, some of these single-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 sections that have dependences unknown at compile time and are therefore not entirely parallelizable [2, 5]. While runtime data dependence tests can parallelize certain unanalyzable code sections [8, 4], these tests are not applicable to general program patterns and/or incur high software overhead.

Alternatively, recent proposals for multithreaded architectures (e.g., the Sun Microsystems' MAJC [11], and the Multiplex [7], Multiscalar [9], Stampede [10], and Hydra [6] chip multiprocessors) employ *speculative execution* to allow threads with unknown dependences to execute speculatively in parallel. To guarantee correct execution and to verify speculation, these architectures provide hardware structures — which we refer to as the *speculative storage* — to temporarily maintain a thread's memory state. Speculative storage records the data a thread produces/consumes and the memory reference information necessary to track dependences across the threads. Once speculation is verified, a thread's produced data is transferred from speculative storage to the conventional memory hierarchy — which we refer to as the *non-speculative storage*.

A key bottleneck of speculative multithreaded architectures is the limited capacity of the speculative storage. The proposed systems require the hardware to track and enforce dependence on *all* memory references in speculative storage without regard to an application's memory access patterns and actual data dependences. To eliminate adverse effects on memory system performance, current proposals also use small (i.e., kilobytes of) hardware structures to allow for fast memory reference and dependence tracking [9, 3]. Unfortunately, when a thread fills up its speculative storage, execution halts until speculation is resolved, significantly reducing parallelism in execution and performance. To avoid speculative storage overflow, current proposals often execute small thread sizes and granularities (e.g., inner loop iterations), potentially limiting the opportunity and degree of extracted parallelism [7].

In this paper, we discover a new program property, called memory *reference idempotency*, which allows a large number of memory references to avoid dependence tracking and placement in speculative storage. Idempotent references can be placed in the non-speculative storage directly, reducing the likelihood of speculative storage overflow and increasing the opportunity for extracting higher degrees of parallelism. Idempotent references span a large spectrum of memory ref-

†Seon Wook Kim is now with Intel Corp., Champaign, IL.

*This work was supported in part by NSF grant #9974976-EIA.

PPOPP'01, June 18-20, 2001, Snowbird, Utah, USA.

erences including not only references to read-only and private variables, but also certain references to shared variables with cross-thread dependences.

Reference idempotency can be explained by the insight that, in speculative execution, incorrect values are created due to dependence violations, and propagated through subsequent computation. Essentially, idempotent references are those that never violate dependences, although they may propagate incorrect values. Because only a speculative reference can originate an incorrect value, hardware speculative mechanisms are guaranteed to correct this value eventually and re-propagate it through subsequent computation. Idempotent references never violate dependences, and therefore need not be tracked in speculative storage.

We focus our experiments in this paper on code sections that are not fully analyzable by the compiler. In a recent paper [7], we presented an architecture that unifies both conventional and speculative multithreaded execution on a single chip. Our architecture allows code sections that are parallelizable to execute as conventional parallel programs (using non-speculative storage) without any speculation overhead. The present paper targets code sections that require hardware speculation support to execute in parallel.

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 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 identified by the compiler and executes them speculatively in parallel. Segments can range in size from a single instruction to entire subroutines.

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 cross-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 and all dependences are satisfied, hence 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), hence the sink must remain in speculative storage. 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 segment.

2. HARDWARE-ONLY SPECULATION

2.1 The 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. Regions and segments are related by age. An*

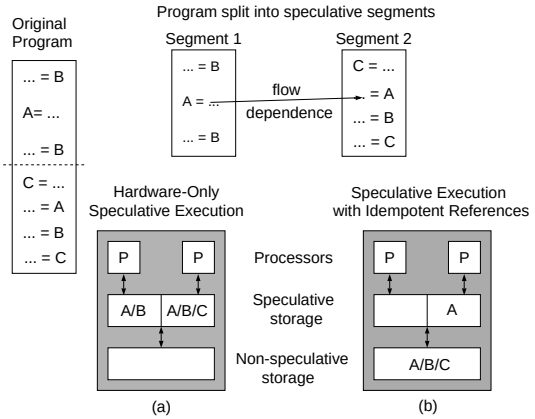


Figure 1: Basic idea of labeling idempotent references. (a) In hardware-only speculative execution, all data is placed in the speculative storage. (b) Idempotent references can go directly to the non-speculative storage (conventional memory hierarchy).

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 fine-grain or entire subroutines in large-grain speculative execution 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:*

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:** *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 are 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 was incorrectly started due to misspeculation, the final execution may execute a different segment or it 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 the speculative storage. They do not affect the non-speculative storage until the segment is* committed *(see Property 6). If a read reference accesses a location not yet 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.*

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), which are committed in sequential order (Property 6).*

6. **Segment Commit:** *When the oldest segment in execution has completed all instructions, it is ready to commit (i.e., conceptually move) its speculative storage to the non-speculative storage. A segment cannot commit until 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.

Essentially, our definition of correctness says that any execution must have the same effect on the memory as a sequential program. It does not specify any order for the program execution. For example, two accesses to different memory locations could be reordered. Also, two write accesses to the same location may be reordered if we can prevent the effect of the originally first access (e.g., by renaming the reference to a different location).

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 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 segment executes and produces the same final values as a sequential program if we can show the following: upon a read reference, a data value that is not yet present in the segment's speculative storage is fetched in a way that yields the same value as in a sequential program. This follows from two facts. (a) By HOSE Property 5, all cross-segment time orderings are satisfied. (b) By HOSE Property 4, values for locations not yet 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 6, 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 $\mathcal{R}$ follows directly from the correctness of the segments in $\mathcal{R}$. All segments write the same values as they would in a sequential execution. Since the segments are committed in sequential order (HOSE Property 6), these values appear in the non-speculative storage in the same order as in a sequential execution of the region. □

3. COMPILER-ASSISTED SPECULATION

3.1 The CASE Model

The Compiler-Assisted Speculative Execution (CASE) model is an extension of the HOSE 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 the rest of the 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 (for prematurely executed speculative loads), 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 the likelihood of speculative storage overflow, as motivated in Section 1. Note, for brevity 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] â *to a variable x in region $\mathcal{R}$ is correctly labeled as idempotent only if it is guaranteed that x will eventually be correct — i.e., an incorrect x 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 irrespective of the control flow path taken.)

LC2: *A reference* â *is correctly labeled as idempotent only if, in the final execution, all time orderings as dictated by data dependences involving* â *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 to x is correctly labeled as idempotent only if any subsequent read reference to x 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 there by any prior write reference.* (If one of the source and 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 that in speculative execution, incorrect values are created due to control and data dependence violations, and

[1]We use the notation x, y for variables and $\hat{a}$, $\hat{b}$ for memory references.

propagated through subsequent computation. LC1 ensures that even if such values are written to the non-speculative storage, they do not persist and are eventually overwritten with the correct values. LC2 and LC3 together imply that idempotent references never violate data dependences. Thus, LC1, LC2, and LC3 together guarantee that idempotent references do not generate incorrect values on their own. However, LC1, LC2, and LC3 do not disallow an idempotent reference from propagating an incorrect value. Because only a speculative reference can originate such an incorrect value, hardware speculative mechanisms are guaranteed to correct this value eventually and re-propagate it through subsequent computation. Therefore, idempotent references need not be tracked in speculative storage, even though they may write temporarily incorrect values.

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 x may be written in a misspeculated segment, LC1 guarantees that, in all final executions of segments referencing x, 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 even if 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. Note, that in some degenerate case even a misspeculated value might not lead to an incorrect execution (e.g., if the program multiplies this value with 0). Our proof does not account for such cases, as do the correctness proofs of most program transformations.

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

4. REFERENCE IDEMPOTENCY

In this section we present the 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 necessary to prove the two theorems. In addition, the term *re-occurring first write* will be used. It is defined as follows.

DEFINITION 5 (RE-OCCURRING FIRST WRITE (RFW)). *A write reference to the variable x in segment $\mathcal{R}_i$ is a RFW if, following any roll-back of $\mathcal{R}_i$, a live x 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 x in $\mathcal{R}_i$ is a RFW if x is first written on all possible control flow paths p, where p is a path from the end of any ancestor of $\mathcal{R}_i$ to the end of $\mathcal{R}$. If x is not live then its value is irrelevant for correctness by Definition 3.

The RFW attribute will allow us to identify a write reference as idempotent, even though it may write an incorrect value as a result of data or control misspeculation. The RFW attribute ensures that a write reference to the same variable x is guaranteed to re-occur with a correct value before the end of $\mathcal{R}$. Hence, x's 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 to x 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.

In the following presentation, we consider one region at a time. This is sufficient, as regions execute sequentially with respect to each other. Data dependences (*may*-dependences) are assumed to have been analyzed for the region on a 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. We will show examples at the end of the subsection.

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 5. 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* â *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 $\hat{a}$ is involved in a dependence with sink $\hat{b}$. Intra-segment dependences are always satisfied because of the sequential execution of segments. A cross-segment dependence is also satisfied because $\hat{b}$ is labeled speculative per Lemma 3. This means that the value of $\hat{b}$ is committed at the end of the final execution of the enclosing segment, which happens *after* $\hat{a}$ (HOSE Properties 4 and 6). Hence LC2 is satisfied. LC3 is not applicable because there is no write reference preceding $\hat{a}$. □

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 $\hat{a}$ is the source of a flow dependence with sink $\hat{b}$. 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 $\hat{a}$ to $\hat{b}$. This dependence is also satisfied. Since $\hat{b}$ is speculative, it will be written to the non-speculative storage upon the commit of the segment containing $\hat{b}$, which is *after* the reference $\hat{a}$ (HOSE Property 6). Hence LC2 is satisfied.

LC3 needs to be considered for the case of a flow dependence from $\hat{a}$ to $\hat{b}$. By HOSE Property 4, the speculative read reference $\hat{b}$ 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 $\hat{a}$ 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* $\hat{\text{b}}$ *that is dependent on an idempotent RFW reference* $\hat{\text{a}}$ *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 $\hat{b}$ cannot be the sink of a cross-segment dependence. On the other hand, $\hat{b}$ 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 $\hat{b}$ correctly reads the value generated by an idempotent $\hat{a}$ 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 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 a RFW cannot be labeled idempotent. By Lemma 3, a cross-segment dependence sink cannot be labeled idempotent. If a reference $\hat{a}$ to variable x is not a RFW then, after the enclosing segment rolls back, execution can take a path that does not write x. Hence, the incorrect value written by $\hat{a}$ will persist. □

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 from x will not consume the value written by a preceding speculative write reference to x. □

Examples

Figure 2 shows several examples. By Definition 5, $\texttt{RFW}(\mathcal{R}_0)$ = {`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 a RFW, because the reference to `B` is not guaranteed to happen after a possible roll-back of $\mathcal{R}_2$. Similarly, the reference to `B` in $\mathcal{R}_3$ is not a RFW. The write reference to `H` in $\mathcal{R}_4$ is preceded by a read. The references to `K`(`E`) in $\mathcal{R}_2$ and $\mathcal{R}_3$ are not RFW because they are not guaranteed to access the same address. All above 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 5 because they are the sink of output and anti dependences 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 3 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 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 6 because they are covered reads.

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 $\mathcal{R}$

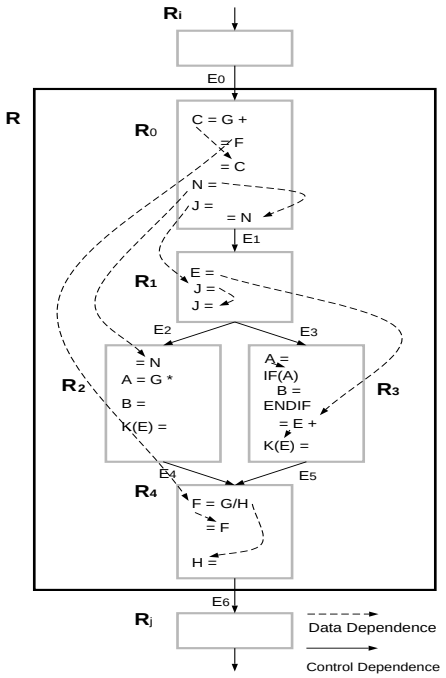


Figure 2: Example code with control and data dependence graphs. The region $\mathcal{R}$ contains five segments, $\mathcal{R}_0, \cdots, \mathcal{R}_4$.

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.

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 thus not live at 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 storage. The idempotency property guarantees that correctness is still ensured.

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., [1, 12]) has analyzed read-only and private variables, and also the data dependences of every reference in each region. Data dependences are *may*-dependences. The following algorithm determines RFW 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 if x is first written on all possible control flow 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 goal of the following graph algorithm is to mark all successors of a segment as non-RFW to 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 $\mathcal{R}$. A node has 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 breadth-first. At each v, if it is* `White`*:*
   - *If v reaches any node marked as* `Read` *through zero or more* `Null` *nodes, then recursively color all* `White` *successors of v* `Black`*.*
3. *All write references to x in* `White` *nodes are re-occurring first writes.*

Note, that the algorithm relies on the compiler's ability to identify references that go to the same address. Two references $\hat{a}$ and $\hat{b}$ 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

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

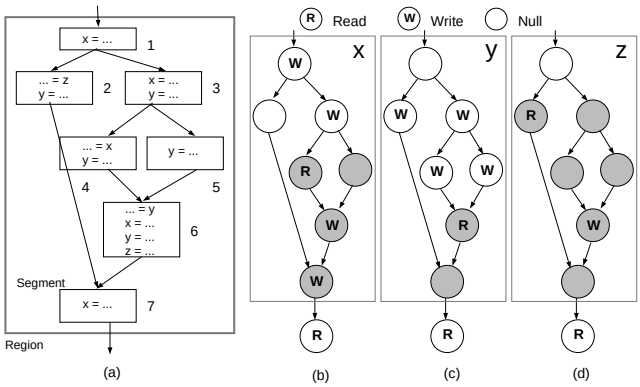


Figure 3: Example of a re-occurring first write analysis. (a) Segment control flow graph. (b) Graph marked for variable x, (c) variable y, and (d) variable z.

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 with affine subscript expressions have correct addresses and are thus candidate RFWs.

Figure 3 shows an example of the analysis performed by Algorithm 1. It shows colored graphs for the three variables x, y, and z in the program region (a). In graph (b), the write references to x in segments 6 and 7 are found not to be RFW because there is an exposed read in segment 4. Similarly, in (d), the write reference to z in segment 6 is not RFW because segment 2 has an exposed read. In (c), all write references to y are RFW.

4.2.3 Labeling Idempotent References

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

ALGORITHM 2. *Identifying idempotent references in region $\mathcal{R}$. The following information is assumed to have been analyzed in $\mathcal{R}$: Read-only and private variables, reference-by-reference data dependences of shared variables, and the segment control flow graph.*

Initially, all references are labeled speculative.

1. *Analyze RFW references with Algorithm 1.*
2. *If $\mathcal{R}$ is fully independent with respect to cross-segment data and control dependences, then*
   - *Label all references in $\mathcal{R}$ as idempotent.*
3. *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.*

```
      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`.

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.

5. EVALUATION

We evaluate opportunities for labeling idempotent references in code sections that the compiler is unable to parallelize automatically. Recall, that speculative storage overflow is a critical limitation of the currently proposed speculative architectures, and these overheads increase further as advanced compilers identify large speculative threads. We have quantified these overheads in prior work [7]. Here we present performance results of applying our labeling algorithm on a selected group of segments.

For our experiments we have developed a preliminary version of our algorithm on top of the Multiplex compiler. Multiplex is a proposal for a chip multiprocessor supporting both conventional and speculative execution of threads (i.e., segments) [7]. The Multiplex compiler integrates Polaris [2] and the Multiscalar compiler [13] 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

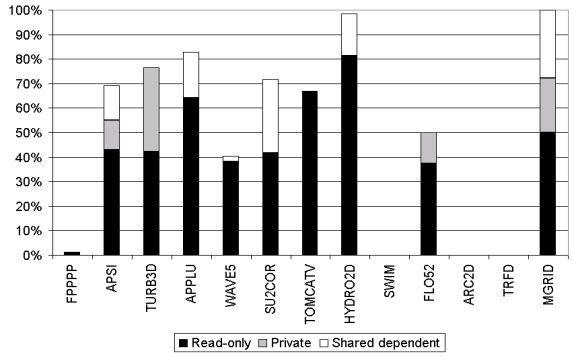


Figure 5: Fraction of idempotent references in code sections that cannot be detected as parallel. It shows idempotent references in the categories read-only, private, and shared-dependent.

system allocates a private stack for every segment. The compiler transforms and places the private variables into these per-segment private stacks.

5.1 Labeling Idempotent References

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. The loops 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. Recall, 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 5 shows the fraction of total references in non-parallelizable code sections that our analysis detects 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. Note, that even a single reference that causes speculative storage overflow will lead to large delays — essentially serializing the execution. Therefore, even small increases in the number of references that do not access speculative storage, can lead to significant performance gains. The benchmarks with few or no idempotent variables fall into two opposite categories. `SWIM, TRFD` and `ARC2D` are fully-parallel programs with no unanalyzable variables, while `FPPPP` is known to be highly unstructured and difficult to analyze.

Figure 6 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

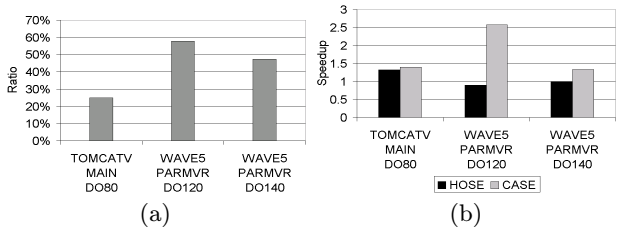


Figure 6: Examples of loops for idempotency category *read-only references*: (a) ratio of read-only references to total memory references, and (b) loop speedups before (HOSE) and after (CASE) reference labeling.

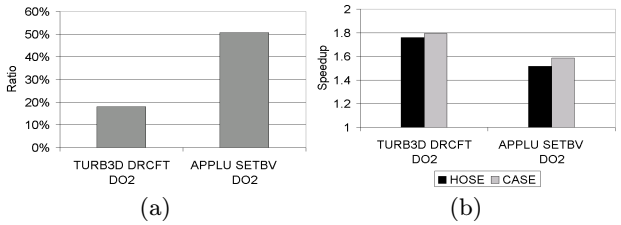


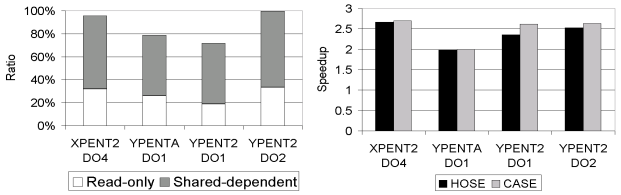
Figure 7: Examples of loops for idempotency category *private references*: (a) ratio of private references to total memory references, and (b) loop speedups before (HOSE) and after (CASE) reference labeling.

pressure on the speculative storage, allowing for significant reductions in execution time.

Figure 7 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. Our implementation sets up per-segments stacks, of which private variables can make use. The stack setup adds a substantial number of instructions. Nevertheless, there are small speedup gains under CASE as compared to HOSE.

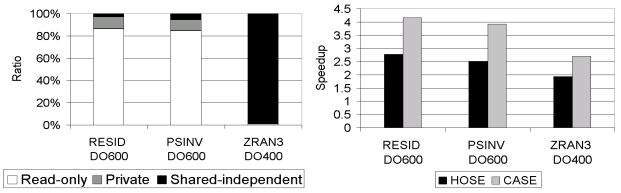
Figure 8 shows loops that include idempotent references in the shared-dependent category. The figure shows idempotent references as a fraction of the total number of references, and the corresponding loop speedups after labeling under HOSE and CASE. The ability to remove shared-dependent references from 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 9 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 9 (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`.



(a) (b)

Figure 8: Examples of loops for idempotency category *shared-dependent* references: (a) ratio of idempotent references to the total memory references, and (b) loop speedups before (HOSE) and after (CASE) reference labeling.



(a) (b)

Figure 9: Examples of loops for idempotency category *fully-independent regions*: (a) ratio of idempotent references, and (b) loop speedups before (HOSE) and after (CASE) reference labeling.

6. CONCLUSIONS

In this paper we have discovered a new program property called memory *reference idempotency*. Idempotent references can access non-speculative storage directly and, hence, alleviate speculative storage overflow, a critical limitation of speculative execution. A key feature of idempotent references is that, during speculative parallel execution, they do not cause any data-dependence violations on their own. References with this property are guaranteed to be eventually corrected, though the references may write temporarily incorrect values during speculative execution. Because speculative execution mechanisms eventually correct and propagate these incorrect values, idempotent references need not be tracked in speculative storage. By filtering out idempotent references, we reduce the demand for speculative storage space. This reduction is especially important for large threads, allowing them to uncover more parallelism without incurring substantial 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 and control 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] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer. Boston, MA, 1988.

[2] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, Dec. 1996.

[3] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *The Fourth IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, pages 195–205, Jan. 1998.

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

[5] 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*, pages 84–89, Dec. 1996.

[6] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessors. *The Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, October 1998.

[7] C.-L. Ooi, S. W. Kim, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the 2001 International Conference on Supercomputing*, 2001.

[8] L. Rauchwerger and D. 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.

[9] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. *The 22th International Symposium on Computer Architecture (ISCA-22)*, pages 414–425, June 1995.

[10] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *The 27th Annual International Symposium on Computer Architecture (ISCA-27)*, June 2000.

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

[12] P. Tu and D. Padua. Automatic array privatization. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 1993.

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