ABSTRACT

RAMASESHAN, RAVI. Traced Based Dependence Analysis for Speculative Loop Optimizations. (Under the direction of Associate Professor Dr. Frank Mueller).

Thread level speculation (TLS) is a powerful technique that can harness, in part, the large computing potential of multi-core / chip multiprocessors. The performance of a TLS system is limited by the number of rollbacks performed, and thus the number of dependence violations detected at run-time. Hence, the decomposition of a serial program into threads that have a low probability of causing dependence violations is imperative.

In this thesis, we develop a framework that calculates a dynamic dependence graph of a program originating from an execution under a training input. We are investigating our hypothesis that by generating such a dependence graph, we are able to parallelize the program beyond the capability of a static compiler while limiting the number of required rollbacks. In our approach, we evaluated two techniques for calculating dependence graphs to perform our dependence analysis: power regular segment descriptors and shadow maps. After calculating dependence graphs that aid loop nest optimizations and after determining program performance after parallelization, we assess results obtained with our framework and then discuss future directions of this research.

We observed the most improvement in performance for two benchmarks, while the others showed either no improvement or degradation in performance or in one case even a slow-down with our analysis.

Traced Based Dependence Analysis for Speculative Loop Optimizations

by

Ravi Ramaseshan

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science in

Computer Science

Raleigh, NC

2007

Approved By:

Dr. Xiaosong Ma	Dr. Thomas Conte

Dr. Frank Mueller
Chair of Advisory Committee

Biography

Ravi Ramaseshan was born on February 28, 1982. He received his Bachelor of Engineering in Computer Engineering from the University of Pune in July of 2004. With the defense of this thesis, he will receive his Master's of Science in Computer Science from North Carolina State University in June of 2007.

Acknowledgements

I would like to acknowledge the following people for their support in completing my thesis: My adviser, Dr. Frank Mueller, my thesis committee, Dr. Xiaosong Ma, and Dr. Thomas Conte and all the graduate students in the systems lab. Their guidance and support made this thesis possible. A special thanks to Dr. Jaydeep Marathe, whose insights and help were invaluable throughout my Masters. I would also like to thank my parents, my sister and other friends who have always supported me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thread Level Speculation

Thread-level speculation is a technique that enables parallel execution of sequential applications on a speculative basis at run-time with either software or hardware support for roll-back[3]. Despite the use of profilers and parallelizing compilers, automatic parallelization has proven to be a very difficult problem . While successful for certain scientific applications, automated parallelization has typically provided poor parallel performance in the absence of or for weak inter-procedural analysis and for indirect memory accesses. To alleviate this problem, speculative run-time systems have been proposed to exploit the parallelism implicit in a sequential application. Since dependence, and hence correctness, is enforced by the speculative run-time, the compiler can improve performance by speculating on ambiguous or low probability dependences without absolute guarantees of independence or correctness. To enforce correctness, the run-time employs data-dependence tracking mechanisms, buffers data from speculative threads in temporary storage, rolls back incorrect executions, and commits data to the memory system only when speculative threads do not violate data dependencies [5]. Figure 1.1 shows how a speculative run-time can be used to parallelize a sequential program to detect data dependence violations between speculative threads. The sequential program is decomposed into thread i and the speculative thread $i + 1$. If the addresses being read to and written from are unknown at compile time, conservative dependence analysis would place a dependence edge between the write in the first thread and the read in the second. Hence, static compiler analysis would not be able to parallelize the two threads without any synchronization. However, with a spec-

ulative run-time system, the compiler can ignore this dependence, parallelize the program and defer the governance of correct program semantics to the run-time. In this example, the read in thread $i+1$ reads a stale value of X before the write from thread i writes to X. The run-time detects this dependence violation, flushes any work done and restarts the speculative thread $i+1$. Note that even though there is a dependence from the write in the first thread to the very next read, the dependence does not cross the thread boundary, it is not a dependence violation.

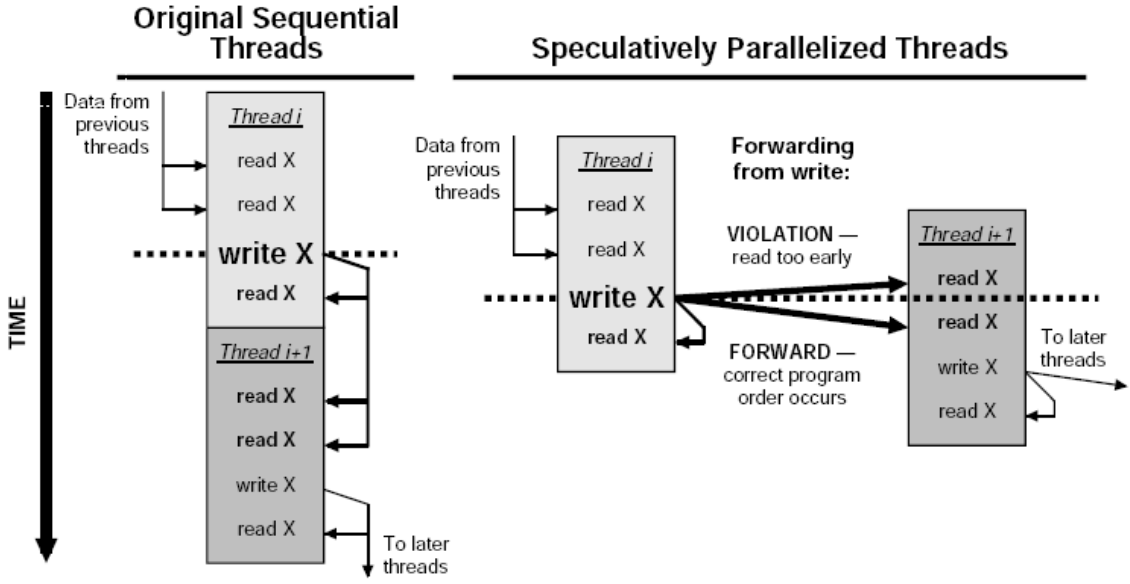

Figure 1.1: Thread Level Speculation: An example of speculative threads with data dependences [3]

Conventionally, the speculation run-time provides a mechanism to run serial code in parallel. The high cost of the squash-rollback-re-execute approach on a dependence violation makes prudent selection of speculatively parallel threads the key to the success of the approach. Ideally, no data dependence should cross a thread boundary to avoid dependence synchronization delays and dependence violation rollbacks. Also, the threads should be large enough to amortize the cost of their dispatch. The compiler is responsible for decomposing a sequential program into speculatively parallel threads while considering performance overheads related to data dependence violations.

Finding optimum program decompositions in general is NP-complete [16]. However, researchers have mainly targeted two constructs for extracting parallelism — subrou-

tines/functions and loops [3][12]. Loops are attractive candidates for parallelization because programs spend most of their time executing within loops. Also, several loop iterations can be combined into a single "chunk" to construct larger and more load balanced threads.

```
void vectCopy (int A[N], int B[N]) {
  for (i = 1; i < N; i++) {
    A[i] = B[i];
  }
}
```

Figure 1.2: Loop Parallelization Example

Consider the example in Figure 1.2. The compiler would not be able to parallelize the above loop since it would not be able to prove that the two arrays A and B do not overlap. However, it is possible to speculatively parallelize this loop by assuming each iteration to be independent of the other if A and B do not overlap for most of the invocations of the function. In the event that A and B do overlap, the speculative run-time would detect the dependence violation and execute the loop serially. Figure 1.3 graphically illustrates the speculative parallelization of different iterations of a loop. In the figure, we combined three iterations of the loop while decomposing the loop into threads. The thread on CPU1 is the speculative thread. The shaded region denotes the dispatch overhead. Note that the master (CPU0) thread suffers more dispatch overhead than the worker threads (CPU1). This is because, besides the normal thread start-up overhead, the master thread is responsible for dividing the work among all the worker threads.

1.2 Dependence Analysis

Data dependence analysis is the basic step in detecting loop level parallelism [8]. Traditionally, there have been many sophisticated approaches to calculate exact dependences in a program statically. However, due to incomplete or obscured dependence information, the compiler may be conservatively forced to assume a dependence between two memory access points since it must maintain program correctness. If this dependence does not actually exist, we call this a false dependence. False dependencies can arise due to many causes:

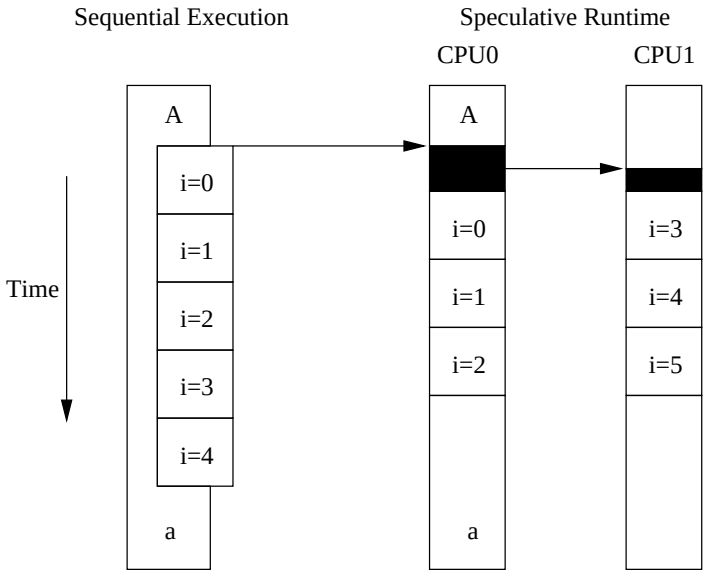

Figure 1.3: Example of a speculatively executed loop

1. **Imprecise Alias Analysis**: This is a problem that occurs more frequently in C/C++ programs. If the alias classification scheme is imprecise (as any moderate-cost alias analysis is), then conservative aliasing between pointers will have to be assumed.

2. **Indirect Accesses**: These occur in codes that use index arrays. Consider the access `A[B[i]]`. In this case, the compiler has no information about which parts of the array A are accessed unless the values in the entire array B are known. This is not usually the case at compile time. Hence, the compiler must assume that any element of array A may be accessed.

3. **Calls**: A significant fraction of loop nests contain calls in the loop body. Without knowing the memory accessed by the called function, the compiler must assume that all global memory and the references passed as arguments to the called function are modified by the call. Inter-procedural analysis can often provide information about the memory locations accessed by the called function. However, such analysis may be imprecise (e.g., due to symbolic variables) or may not be available for all calls (e.g., library functions).

4. **Symbolic Terms**: The presence of symbolic terms in array subscripts, loop bounds and condition predicates may make it impossible to precisely determine if dependencies

exist.

The most straightforward approach for the compiler to parallelize a loop is to statically prove that there are no loop-carried dependencies between iterations of that loop. If true, then all iterations of the loop can be conceptually executed in parallel. Such a loop is called a DOALL loop [6]. Considering dependences introduced due to the above mentioned imprecision in dependence analysis, this constraint prevents a large number of loop nests from being parallelized.

In this thesis, we explore the use of dynamic dependence graphs for decomposing a program into threads. We respect all dependences that occur in the dynamic dependence graph while decomposing the program. Such a decomposition is not safe for all input but since we use a training input characteristic of the common input to the program, the dependences that occur in our analysis are those that are the most likely to occur for any input. Hence, it is our hypothesis that such a thread decomposition would be profitable for coarse-grain speculative parallelization. Though dependence graphs, to guide speculative optimization, have been dynamically created in the past, we employ a novel approach of using *Power Regular Section Descriptors* for regular accesses and *Largest Common Sub-range Descriptors* for irregular accesses that makes our analysis scalable. We also evaluate the precision of our analysis technique versus that of a shadow-map based non-scalable approach.

The remainder of this thesis is organized as follows. Chapter 2 describes our trace-based dependence analysis and optimization framework. Chapter 3 discusses the results that we obtained. Chapter 4 describes related work. Chapter 5 provides our conclusions.

Chapter 2

Tracing and Dependence Analysis Framework

2.1 Overview

The objective of this framework is to build a dependence graph to guide speculative loop nest optimizations. Figure 2.1 illustrates a high-level view of the framework. The program is first traced with a training input, and the tracing and dependence analysis framework calculates a dependence graph of the program. The Loop Nest Optimizer (LNO) phase in the Open Research Compiler (ORC) then generates a speculatively parallel program using this dependence graph. The program is speculatively parallel because the dependence graph used in the optimizations may not respect every dependence that may exist in the program for different inputs but only those that were exercised by the inputs in the tracing run.

The entire framework operates in four distinct serial phases:

1. **Instrumentation** :— The framework, implemented within the LNO component of ORC, instruments the program by inserting calls to a tracing library at certain points in the program.

2. **Tracing** :— The instrumented program is linked with a tracing library and run with a training input to generate a trace of the program execution representative of the common case behavior of the program.

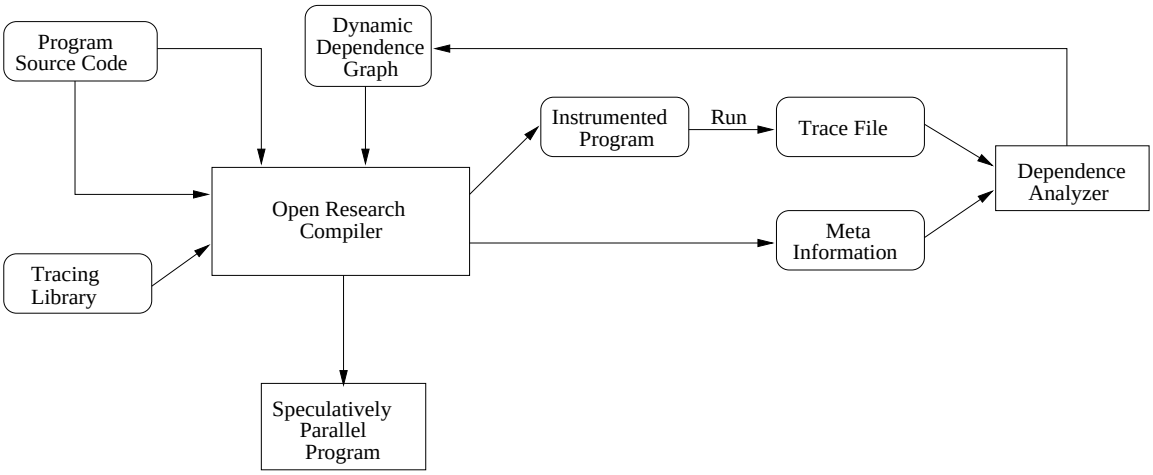

Figure 2.1: Framework Overview

3. **Dependence Analysis** — The generated trace is processed to calculate a dynamic dependence graph of the program.

4. **Speculative Parallelization** — Using the dynamic dependence graph of the program, aggressive loop optimizations are applied.

Each of these phases shall be described in detail later. In the following sections we shall briefly describe ORC and its internal representation — WHIRL (*Winning High Level Intermediate Representation Language*).

2.1.1 Open Research Compiler

The following description has been taken from ORC's website.

"The Open Research Compiler (ORC) [13] project provides a leading open source Itanium^{TM}Processor Family (IA-64) C/C++ and Fortran90 compiler infrastructure to the compiler and architecture research community. This project is a collaboration between Intel Corp. and Chinese Academy of Sciences. It is based on the Pro64 (Open64) open source compiler from SGI [2] [10]. It has the following major back-end components:

- *Inter-procedural Analysis and Optimizations (IPA)*
- *Loop Nest Optimizations (LNO)*
- *Scalar Global Optimizations (WOPT)*

- *Code Generation (CG)"*

This research is focused on speculative loop nest optimizations. Hence the component in back-end of interest here is the LNO. It implements a large number of loop nest optimizations, *e.g,* loop fission, unrolling, tiling, interchange, reduction, privatization and auto-parallelization. LNO uses two analysis frameworks that are relevant to this research — array dependence graphs and array region analysis. Array dependence graphs are primarily used for most loop optimizations while array region analysis is used for privatization.

2.1.2 WHIRL

The following description of WHIRL has been taken from [19].

"Every compiler uses an internal representation (IR) to bridge the semantic gap between the source language and machine instructions. This enables the compiler to handle many languages and many architectures with the same infrastructure. In general, a compiler has multiple levels of IR. The higher levels of IR contain more structural information about the program while the lower levels lose this information."

"WHIRL is the IR used by ORC and it has five levels. Each WHIRL level is semantically the same, however, the structural information about the program decreases in the lower levels. This multi-level approach to the IR enables optimizations to work at the most appropriate levels. Figure 2.2 shows the different components of ORC and the levels of the IR each component works with. Compilation can be viewed as a process of gradual transition, called lowering*, from high level language constructs to low level machine instructions. During a typical compilation, the source code is compiled into VH (Very-High) WHIRL and the IR is progressively lowered into L (Low) WHIRL and finally into machine instructions."*

LNO uses H (High) WHIRL in which side effects can only occur at statement boundaries and control flows are fixed. In H WHIRL, high level control flow constructs are represented using operators such as `DO_LOOP`, `DO_WHILE`, `WHILE_DO` and `IF`. Also array accesses are represented using the `ARRAY` operator. Scalar loads and stores are represented as `LDID`s and `STID`s while indirect loads and stores are represented as `ILOAD`s and `ISTORE`s, respectively. Each WHIRL *statement* is organized as a tree and a *program unit* contains a list of WHIRL statements.

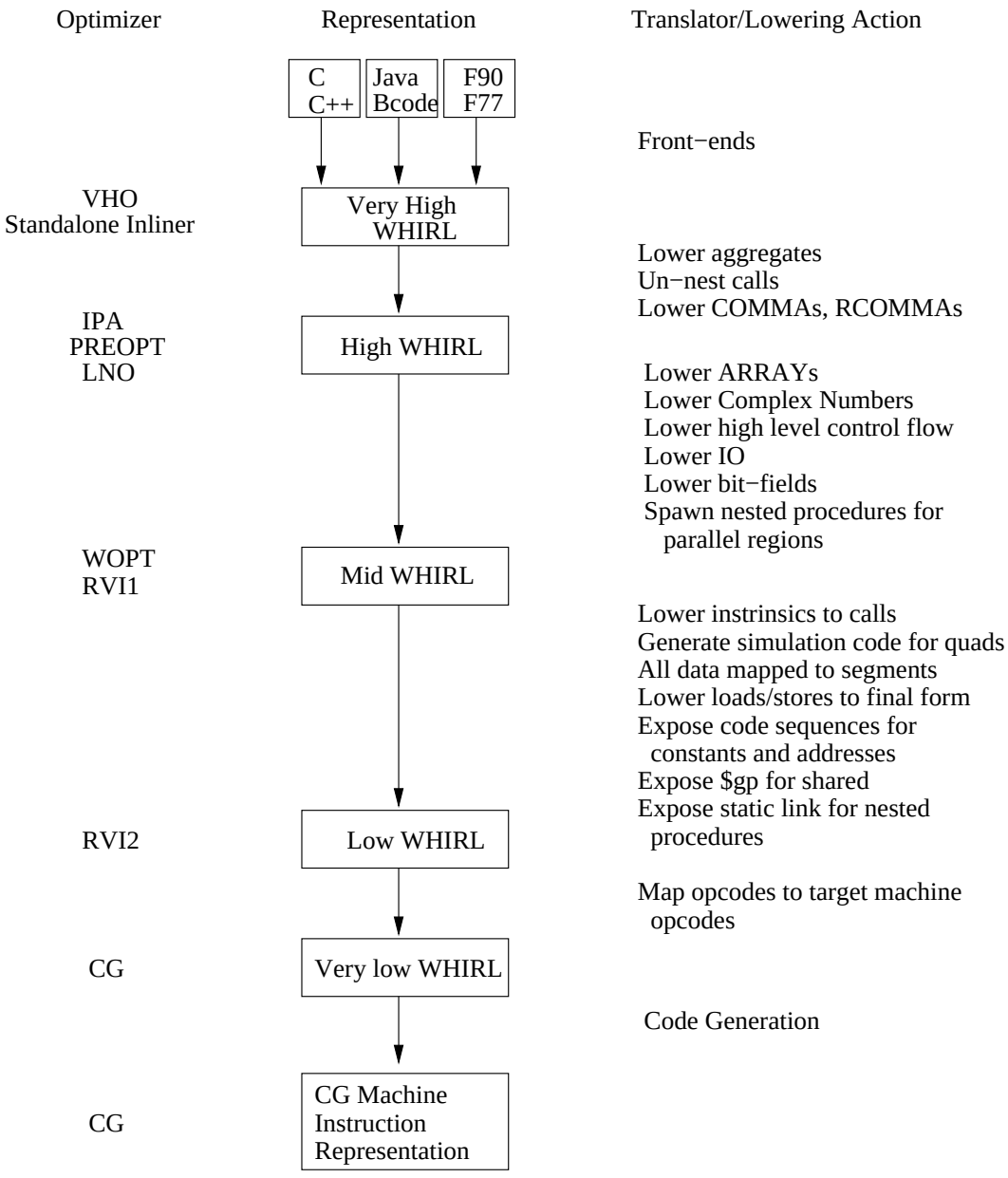

Figure 2.2: Continuous Lowering in the SGI Pro64 Compiler [19]

2.2 Instrumentation Framework

The instrumentation framework is responsible for placing calls to a tracing library in the program at points corresponding to possible nodes in the dependence graph. It also places instrumentation for points corresponding to structured control flow constructs. The

objective of the framework is to create a program that generates a trace of addresses for memory accesses and control flow transitions for loops, calls and functions.

We shall now describe the statements in the IR that we need to instrument.

"LNO builds a graph of all the array statements in a function which it calls the array dependence graph*. Each array load (*`ILOAD`*) and store (*`ISTORE`*) is mapped to a vertex in the graph and so are* `CALLS`*. Non array loads and stores (*`LDID`*s and* `STID`*s) may also appear in the graph only if they exhibit a dependence with some array load/store. A good* `DO` *loop is one which satisfies certain criteria which include not containing any unsummarized calls* [1]*, bad memory references* [2] *and early exits. Edges only exist between loads and stores that share at least one common* good `DO` *loop. The edges in the dependence graph have a list of lexicographically positive dependence vectors associated with them. The dependence vectors may either be distance or direction vectors. Any statement in the IR which is not mapped to a vertex in the graph must be assumed to be dependent on everything."* [3]

We instrument array and scalar loads and stores, function entry and exits and call sites. `DO_LOOP`s are instrumented so that we can distinguish between different iterations of each loop.

Each instrumentation point is associated with a unique integer identifier, which we call a *reference identifier*. We maintain the *type* of the statement that was instrumented, and in the case of loads and stores, the *size* of the memory access as *meta information* per instrumentation point. The instrumentation passes the reference identifier of the instrumentation point and, in the case of loads and stores, the memory address accessed to the instrumentation library. This technique simplifies the instrumentation and also reduces the amount of trace data generated.

For loads, stores and calls, the instrumentation for each statement is added in post-order so that the trace generated correctly reflects the execution of the program. We place two instrumentation points per loop. The first marks the beginning of the loop body and the second marks the completion of all iterations of the loop. The sequence of markers generated by this instrumentation is adequate to distinguish between two iterations of a loop and removes one instrumentation point from the loop body. This reduces tracing overhead in terms of time as well as stable storage space.

[1] Unsummarized calls are call statements in the WHIRL without any IPA information inside the `DO` loop

[2] Bad memory references are loads and stores that are not mapped in ORC's dependence graph

[3] The description of array dependence graphs is taken from comments in the source code file `dep_graph.h` in the ORC source tree

2.3 Tracing Phase

In the tracing phase, the instrumented program is linked with the tracing library and executed to generate a faithful trace of the execution of the program through some training input.

The tracing library writes the trace information onto stable storage. Each record written by the tracing library is a 10 byte tuple of $(reference\ identifier, address)$ where the reference identifier is 2 bytes and the memory referenced is 64 bits.

2.4 Dependence Analysis Framework

The dependence analysis framework uses the trace file and the meta-information to calculate a dynamic dependence graph whose edges are associated with dependence vectors.

2.4.1 Dependence Analysis

The following summary of some of the key concepts in dependence analysis has been taken from [6] as background.

"Dependences can be characterized by several different properties. The type *of a dependence* - true, anti- *and* output - *tells whether it corresponds to a write before a read, read before a write or a write before a write. Dependences in loops have special properties."* An iteration vector of an access in a loop is a vector of normalized iteration numbers of the loop nest in which the access is seen. *"A dependence* direction vector *describes the relationship (<, =, or >) between the values of the loop indices for the nest at the source and the sink of the dependences. The* distance vector *gives the number of iterations crossed by the dependence for each index in the loop nest. A dependence is said to be* loop independent *if its direction vector entries are all equals. Otherwise it is a loop carried dependence. The level of a loop-carried dependence is the nesting level of the loop that corresponds to the leftmost non-equals direction in the direction vector. A transformation that reorders the iterations of a level-k loop, without making any other changes, is valid of the loop carries no dependence."*

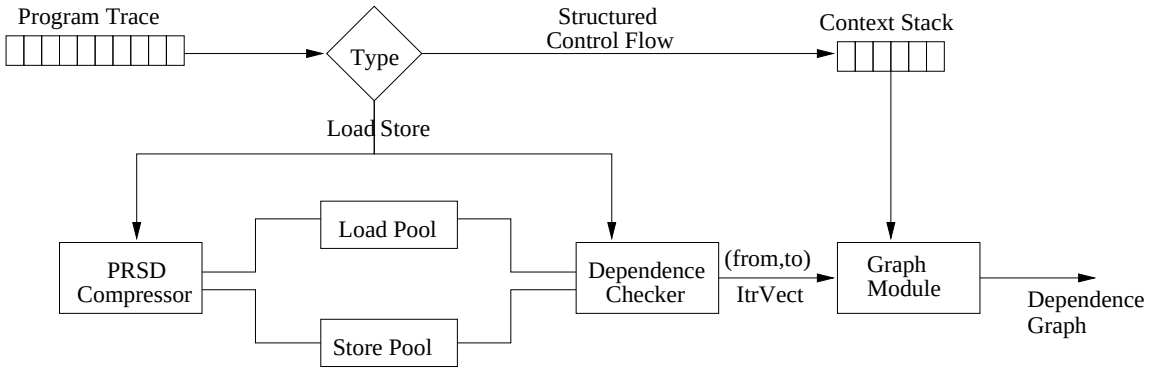

Figure 2.3: General design of the Dependence Analyzer

2.4.2 Overall Structure

The basic technique used in the dependence analyzer is to maintain sets of memory accesses and to check whether the current memory access exhibits a dependence with any past memory accesses. Dependence vectors are determined by calculating the difference between the iteration vectors of the two dependent accesses. The dependence analyzer has four major parts — maintaining the current context, maintaining the access pools, detecting dependences and calculating dependence vectors.

Figure 2.3 shows the overall structure of our dependence analyzer framework. The trace file is read in and each is processed through the dependence analysis framework. Loops, call sites and function entry and exits are records in the trace file that we classify as *structured control flow* (SCF) records. These records are used by the dependence analyzer to maintain the current context of the program and to associate an iteration vector with every memory access. The remainder of this section describes the core of the trace based dependence analyzer in detail.

2.4.3 Extended Context Stack

We define an extended context of the program to contain not only the call stack but also loop nest information. A *loop context* encodes the loop identifier and the iteration number of the current iteration. A *calling context* encodes the call site and also the function being called. Besides pushing and popping calling contexts onto and from the extended context stack while processing functions, we also push and pop loop contexts as the loop body is entered and exited per iteration. We assume a regular loop structure where loops

may either be disjoint or completely nested. Due to this property, we only require loop iteration begin markers without their corresponding loop iteration end markers. When we process a loop iteration-begin marker, the current context stack is checked to see if this loop is the current context. Otherwise, we push this loop context with an iteration number of zero onto the context stack. If the current context is already this loop then we simply increment the iteration number of the loop context. In this way, we maintain normalized iteration vectors for each loop nest independent of the actual induction variable. The trace file also contains records for call sites and function entry and exits. Using this information, we maintain a call stack as a part of our extended context stack.

2.4.4 Access Pools

An access pool is defined as a an iteration-space ordered or unordered set of memory accesses stored as a tuple of reference identifier, address and possibly the iteration vector of the access. Access pools are required to keep track of past memory accesses. In order to calculate dependence vectors and dependences due to calls, we save the extended context stack as an *extended iteration vector* along with each access in the access pool. Keeping track of past accesses becomes a hard problem for programs that perform large amounts of memory accesses. In this thesis, we focus our research on scientific programs that typically have a large number of memory accesses though most of these accesses are array references. Marathe et al. [7] performed a detailed study of memory access patterns and showed that most accesses for scientific programs are regular or slow changing. They also proposed a compressed representation of a group of memory accesses per instruction pointer called a Power Regular Section Descriptor (PRSD). Beyond this work, we establish that the regular access region of a PRSD also makes it possible to test for a dependence in time linear in the depth of a PRSD. Hence, we use a PRSD-like mechanism to maintain our access pools.

Our PRSDs encode rectangular regions using a *base address* and a vector of *dimensions* and are maintained per memory access instrumentation point. Each dimension describes a *stride*, which is the distance each memory access is apart from its preceding access in this dimension, and *length*, which is the number of accesses in this dimension. Initially, an access is a zero-dimension PRSD and is then combined using a compatibility function to form larger dimension PRSDs. The test that determines if two PRSDs P and Q are compatible is works as follows:

1. If the two PRSDs are both of dimension N then they are compatible if the strides and lengths of each of their N dimensions are identical.

2. If P has dimension $N-1$ and Q has dimension N, then P is said to be compatible with Q if all the dimensions of P are identical to the lower $N-1$ dimensions of Q and the base address of P is predictable by the stride of the N^{th} dimension of Q.

3. All other PRSDs are considered incompatible.

We shall illustrate the process in which PRSDs are combined to form larger PRSDs in the following example:

1.	`&A[0][0]`	`&A[0][0]`	Trivial Zero-Dimension PRSD
2.	`&A[0][1]`	`&A[0][1]`	Trivial Zero-Dimension PRSD
3.		`&A[0][0], (+4, 2)`	Combine 1 and 2 (Rule 1)
4.	`&A[0][2]`	`&A[0][2]`	Trivial Zero-Dimension PRSD
5.		`&A[0][0], (+4, 3)`	Combine 3 and 4 (Rule 2)
6.	`&A[1][0]`	`&A[1][0]`	Trivial Zero-Dimension PRSD
7.	`&A[1][1]`	`&A[1][1]`	Trivial Zero-Dimension PRSD
8.		`&A[1][0], (+4, 2)`	Combine 7 and 8 (Rule 1)
9.	`&A[1][2]`	`&A[1][2]`	Trivial Zero-Dimension PRSD
10.		`&A[1][0], (+4, 3)`	Combine 8 and 9 (Rule 2)
11.		`&A[0][0], (+4, 3), (+40, 2)`	Combine 5 and 10 (Rule 1)
12.	`&A[2][0]`	`&A[2][0]`	Trivial Zero-Dimension PRSD
13.	`&A[2][1]`	`&A[2][1]`	Trivial Zero-Dimension PRSD
14.		`&A[2][0], (+4, 2)`	Combine 12 and 13 (Rule 1)
15.	`&A[2][2]`	`&A[2][2]`	Trivial Zero-Dimension PRSD
16.		`&A[2][0], (+4, 3)`	Combine 14 and 15 (Rule 2)
17.		`&A[0][0], (+4, 3), (+40, 3)`	Combine 11 and 16 (Rule 1)

Searching for an access to a particular memory address involves a search in the memory region defined by the PRSD. This search in the PRSD address space can be solved using integer linear programming. For rectangular region PRSDs, the compatibility test has a complexity linear in the number of dimensions of the PRSD.

The list of PRSDs generated for a particular instrumentation point is divided into *active* and *dormant sets*. The PRSDs from the active set are tested for compatibility during PRSD compression while the PRSDs from the dormant set are maintained only for dependence checking. This heuristic may reduce the efficiency of the PRSD mechanism but the loss in compression makes the process scalable to a large number of PRSDs, which is often encountered for irregular accesses.

2.4.5 Dependence Testing

Our dependence testing strategy compares the current access with past accesses in the access pools. Maintaining two separate access pools, one for loads and one for stores, makes dependence checking simpler and faster by reducing the number of accesses that need to be consulted for detecting dependences. If the current access is a load, then all accesses in the store pool must be checked to see if there was a store to the same address. If one is detected then we determined a read-after-write or a true dependence. The iteration vectors for the store and the current load are extracted and then used to calculate a dependence vector. Similarly, if the current access is a store, we detect anti- and output dependences, respectively by checking with the load and store pools.

The compressed representation of all the accesses traced by an instrumentation point avoids pairwise testing of each access. However, for a large number of PRSDs in each pool, this technique becomes infeasible. Hence we maintain an auxiliary data structure that we call a *super range access pool* along with our load and store pools in which we divide the memory address space defined by the PRSDs into disjoint regions based on the bounds of the PRSD. It is called a super range access pool because it calculates super-sets of overlapping memory regions. Every PRSD belongs to exactly one region. These regions are formed as follows:

If P is the new PRSD and R is the set of regions for an access pool then the set R, which is initially empty, is generated according to the following RULES:

1. If the region defined by P does not intersect with any $R_i \in R$, then make a new region, containing P and with the same bounds as P and add it to R.

2. If the region defined by P is completely enclosed within the region defined by $R_i \in R$, then add P to R_i's set of PRSDs.

3. If the region defined by P intersects with $R_i \in R$, and possibly $R_i + 1 \in R$ then make a new region containing all PRSDs in regions P, R_i and possibly $R_i + 1$ whose bounds enclose the regions defined by R_i, $R_i + 1$ and P.

Figure 2.4 shows a snapshot of the super range access pool. As can be seen from the figure, it divides PRSDs into maximal disjoint regions of overlapping PRSDs. All PRSDs are maintained as a part of linked lists in the PRSD table per reference identifier. Since

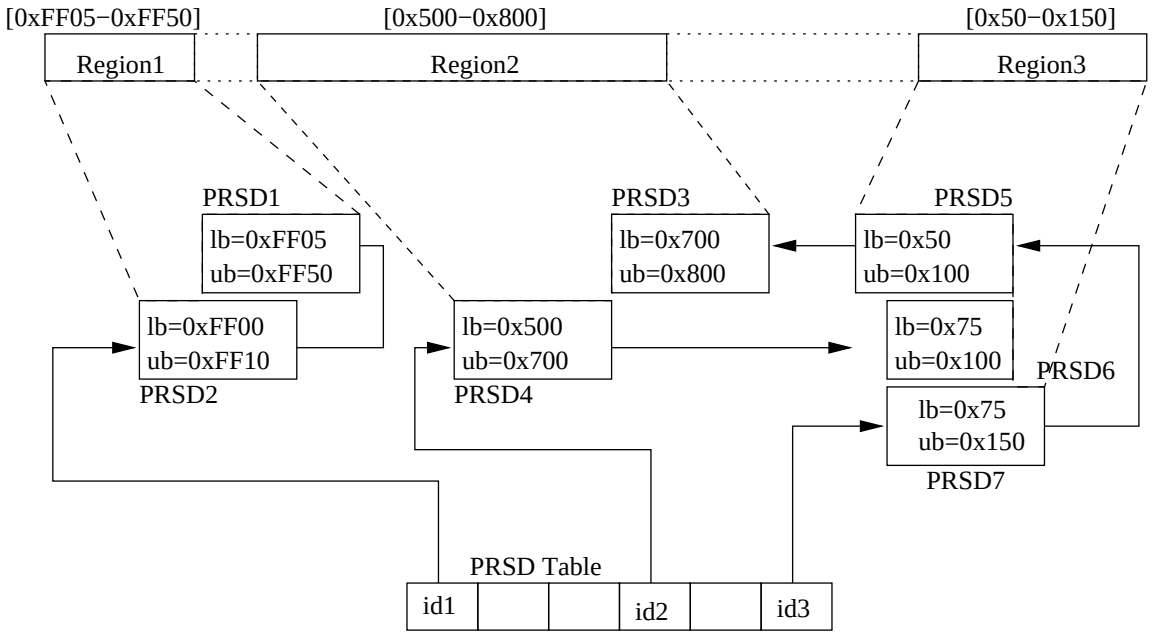

Figure 2.4: Super Range Access Pool

$PRSD1$ and $PRSD2$ overlap in region $0xFF05 to 0xFF10$, we create region $Region1$ for the two overlapping PRSDs. Similarly, region $Region2$ consists of $PRSD3$ and $PRSD4$ due to the overlapping access $0x700$. Region $Region3$ consists of three PRSDs — $PRSD5$, $PRSD6$ and $PRSD7$. Thus, any access that belongs to region $Region1$ needs to be tested for dependence only with $PRSD1$ and $PRSD2$. Maintaining this auxiliary region map makes testing for a dependence much faster on average by distributing PRSDs into disjoint regions. Now, a PRSD may only need to be tested for a dependence against other PRSDs in its region.

2.4.6 Calculating Dependence Vectors

After identifying the two memory accesses due to which the dependence was manifested and extracting their corresponding extended iteration vectors, we a) identify nodes between which the dependence edge must be placed in the per-function dependence graph, b) calculate dependence vectors for the edge with respect to the loop nest and c) collect dependence edges in dependence graphs per function. We scope this project to only calculating direction vectors for an edge. Although we identified the memory access due to which a dependence is caused, the presence of calls in the program complicates the placement of

the dependence edge in the correct function. This is because the accesses may be in different invocations of the function at different call sites.

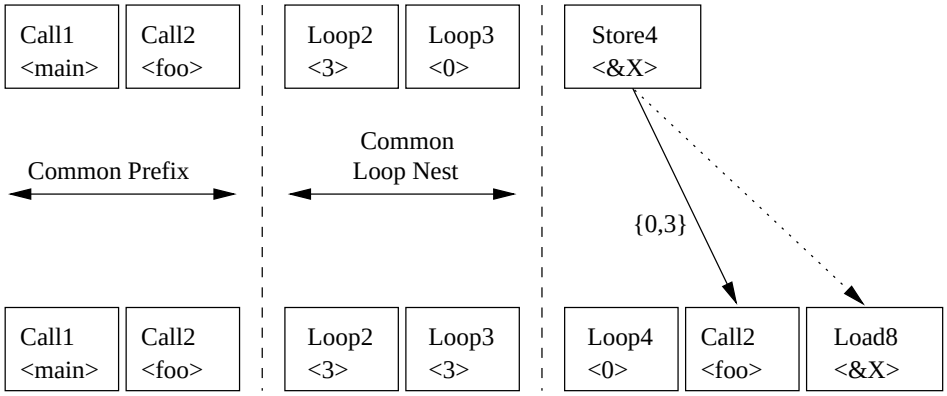

Figure 2.5: Placement of Dependence Edges

We shall describe the process of calculating the dependence vector an edge using the example illustrated in Figure 2.5. We use the two extended iteration vectors shown in the figure to calculate the points between which the dependence needs to be placed and also the dependence vector for the edge. A dependence was detected from $Store4$ to $Load8$ and is represented by the dotted arrow. The common prefix of both the iteration vectors is skipped and the last calling context is recorded. In this case, the last calling context was $Call2 < foo >$. Hence, foo is the function in which the dependence edge needs to be placed. Next, the common loop nest is identified and the difference between the iteration numbers for each context gives the dependence vector for the edge. Then the edge is placed between the next call in the extended iteration vector or the memory access itself depending on which one exists. In this case, the the edge was placed from $Store4$ to $Call2$. In this way, we create per-function dependence graphs for the entire program.

2.5 Speculative Parallelization Phase

This phase reads in the dynamic dependence graph computed by the dependence analysis phase, described in Section 2.4, combines it with ORC's static dependence graph and builds a dependence graph that potentially more accurately reflects opportunities of successful TLS than either graphs.

We first read in our dependence graph and establish a mapping between the nodes

and access points in the IR. A node in the dependence graph corresponds to an instrumentation point that is uniquely identified by its reference identifier. By traversing the IR in the same manner as was done while instrumenting the program, we can establish a mapping between points in the IR and nodes in our graph. The dependence graph calculated by the compiler using static analysis is imprecise due to the reasons described in Section 1.2. Depending on the precision of sequencing information maintained by the PRSDs the dynamic dependence graph computed by the trace-based analyzer might also be imprecise. This is discussed in detail in Section 2.6. Let G_s be the dependence graph calculated by the compiler using static analysis and G_d be the dynamic dependence graph calculated using our dependence analyzer. We build the new dependence graph G_n by iterating over each of the edges of both G_s and G_d as follows:

1. If an edge exists in the G_s and not in G_d, then we do not add the edge in the new dependence graph G_n.

2. If an edge does not exist in G_s but exists in G_d, then we add the edge to G_n if one or both of the end points of this edge belongs to $V(G_s)$.

3. If the edge being added to the dependence graph G_n is present in both G_s and G_d, then theoretically we would want to create a set of dependence vectors that tightly constrains the two dependence vector sets. However calculating such a set is hard. Hence, we select the more constrained of the two sets of dependence vectors based on the number of dimensions and whether they are distance or direction vectors.

ORC uses the new dependence graph to parallelize loop nests. By using a more opportunistic dependence graph in ORC's auto-parallelization framework, loops that were earlier not auto-parallelizable by ORC may now be parallelized. The program generated is a speculatively parallel program since the set of dependences represented in the new dependence graph were only those that were manifested with the training input. They may not be the complete set of dependences. Particularly, some other input to the program might exercise dependences that were not honored by our analysis and cause incorrect program behavior. Our research aims to use the dynamic dependence graph as a guide to speculatively parallelize a program. We assume that the training input is characteristic of the program and, hence, dependences not represented in the dynamic dependence graph would be exercised infrequently during a regular run. We believe that this would reduce

the number of roll-backs in the speculative run-time environment and thus improve the performance of a TLS system.

2.6 Enhancements to the Framework

In this section, we describe techniques that we explored to improve our framework in terms of analysis time, memory requirements as well as precision of the dependence graph generated.

2.6.1 Sequencing Information and Dependence Analysis

Sequencing information is the information required to maintain a complete order among all the accesses in the system. It plays an important role in the precise calculation of dependences. Absence of, or limited ordering information may result in false dependences between points in the program and a conservative set of dependence vectors for an edge.

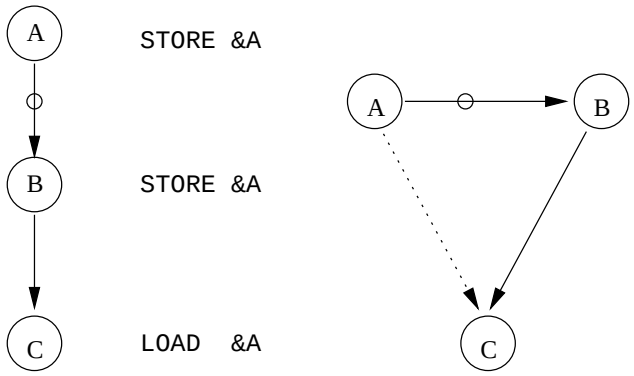

Figure 2.6: False Dependence Edge

Figure 2.6 illustrates how the lack of ordering information can lead to false dependence edges being added to the dependence graph of the program. The code snippet has a sequence of two stores followed by a load. There is one output dependence from A to B and one true dependence from B to C. Let us assume that we are currently at access C and the stores A and B are maintained without ordering. Since there is no way to know which was the last store, we have to conservatively assume that it could have either been A or B. Thus, we have the extra dependence from A to C, which is not a real dependence. In general we may introduce dependences that do not exist in the original program for any

input or that do not exist for our training input but might exist for some other input. In either case, the dynamic dependence graph should not contain this dependence.

Figure 2.7 illustrates how the lack of ordering information can cause imprecise dependence vectors for edges in the graphs. Vectors enclosed within angular brackets are iteration vectors for each memory access, and the vectors enclosed in braces are the dependence vectors for each edge. The left-most figure shows the dependence graph of the code sequence — a store followed by a load referencing the same address within a loop. The second figure splits the nodes of the first dependence graph per iteration and the third recombines the split vertices. In the absence of sequencing information, the ordering among the accesses in different iterations of the loop is lost and for the same reason discussed in the previous example, we need to place a flow dependence from every store to every lexicographically positive load referencing the same address. The same holds true for anti and output dependences. In this example, the imprecision is seen in the edge $A \rightarrow B$ where the distance vector $\{1\}$ is summarized as a direction vector of $\{<\}$. In future iterations of the loop, the other dependences would also become imprecise.

Hence, without sequencing information, the only safe set of direction vectors that we can place on an edge are the set of all lexicographically positive direction vectors for every level of the common loop nest. Due to this, even loops that exhibit only loop independent dependences cannot be parallelized.

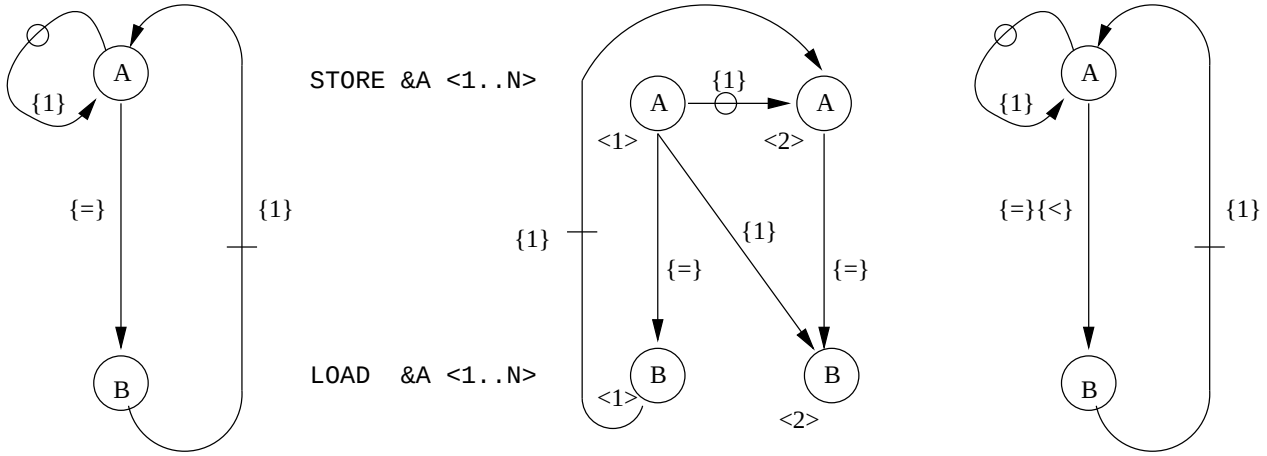

Figure 2.7: Imprecise Dependence Vector

Thus, maintaining sequencing information with every access is highly imperative

to identify parallelism. However, maintaining this information with every access is difficult for a large number of accesses. In the next two sections, we describe possible techniques that we explored to maintain enough sequencing information to calculate dependences at a relatively low overhead.

2.6.2 Access Pools using Shadow Maps

Chen et al. [1] used a shadow memory structure to calculate dependences between loads and stores. Besides the PRSD-based access pools, we also explored a shadow memory structure to calculate dependences in a program. We maintain separate shadow memories — one for loads and one for stores. The load pools are different from the store pools in that they maintain a list of accesses per address as opposed to store pools where only the latest access to that address is maintained. Our shadow memory structure is indexed using the address of the load or store. We update these shadow memories as follows:

for $Access \in AccessStream$ **do**
  if $Access$ is a load **then**
    Add flow dependence edge: $StoreMap[Access_{Address}] \rightarrow Access$
    $List(LoadMap[Access_{Address}])+ = (Access_{Id}, Access_{IterationVector})$
  else {$Access$ is a store}
    Add anti-dependence edges for all: $LoadMap[Access_{Address}] \rightarrow Access$
    Add output dependence edge: $StoreMap[Access_{Address}] \rightarrow Access$
    $StoreMap[Access_{Address}]) = (Access_{Id}, Access_{IterationVector})$
  end if
end for

The advantage of this technique is that it maintains perfect sequencing information and, hence lets us calculate dependences very precisely. It discards old accesses that should no longer be used for testing dependence and, hence, does not store all past accesses, but only those accesses that are relevant to dependence analysis. However it suffers from the disadvantage that its memory usage scales linearly with the memory foot-print of the analyzed program.

2.6.3 Power Sequenced Regular Section Descriptors

PRSDs are effective structures in compressing large regular or slow-changing accesses. However, as pointed out in Section 2.6.1, the absence of sequencing information in PRSDs introduces a lot of imprecision in the dependence graph. In order to improve the precision, we maintain the address and extended iteration vector of the last access described by this PRSD. This heuristic was based on the observation that programs tend to have a large number of loop-independent updates. By keeping information about the latest access described by this PRSD, we are able to identify such dependences as loop independent and reduce the imprecision.

A more general solution to the problem, however, would be to maintain complete sequencing information along with the PRSD itself by compressing the extended iteration vectors in a way similar to how PRSDs are compressed. Our new PRSD, which encodes sequencing information in the form of compressed iteration vectors as a vector of the following tuples, is structured as follows:

- *Base Address*: The address of the first access in this dimension.
- *Address Stride*: The difference between consecutive accesses in this dimension.
- *Base Iteration Number*: The first iteration number generating an access for this dimension.
- *Iteration Stride*: The difference between consecutive iterations generating an access for this dimension.
- *Length*: The number of accesses described in this dimension.

The following example examines the execution trace generated by the program depicted in Figure 2.8 and demonstrates how the access at point X in the loop nest can be completely described:

Thus at the end of the process, we generate the sequenced PRSD {`&A[0][0], +40, 0, +1, 3`} {`&A[0][0], +4, 0, +1, 3`} that describes each access and also the iteration vector corresponding to each access. The advantage of this technique is that it can describe and compress a large number of regular accesses, thus making the process potentially very scalable. However, this technique cannot efficiently handle addresses and control flow of

```
  int A[10][10];
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
X:    A[i][j] = ...
```

Figure 2.8: Control Flow and Address Regular Access Compression

```
    &A[0][0]  (&A[0][0],  +0, 0, +0, 1), (&A[0][0], +0, 0, +0, 1)  Trivial
    &A[0][1]  (&A[0][1],  +0, 0, +0, 1), (&A[0][1], +0, 1, +0, 1)  Trivial
              (&A[0][0],  +0, 0, +0, 1), (&A[0][0], +4, 0, +1, 2)  Combine
    &A[0][2]  (&A[0][2],  +0, 0, +0, 1), (&A[0][2], +0, 2, +0, 1)  Trivial
              (&A[0][0],  +0, 0, +0, 1), (&A[0][0], +4, 0, +2, 3)  Combine
    &A[1][0]  (&A[1][0],  +0, 1, +0, 1), (&A[1][0], +0, 0, +0, 1)  Trivial
    &A[1][1]  (&A[1][1],  +0, 1, +0, 1), (&A[1][1], +0, 1, +0, 1)  Trivial
              (&A[1][0],  +0, 1, +0, 1), (&A[1][0], +4, 0, +1, 2)  Combine
    &A[1][2]  (&A[1][2],  +0, 1, +0, 1), (&A[1][2], +0, 2, +0, 1)  Trivial
              (&A[1][0],  +0, 1, +0, 1), (&A[1][0], +4, 0, +1, 3)  Combine
              (&A[0][0], +40, 0, +1, 2), (&A[0][0], +4, 0, +1, 3)  Combine
    &A[2][0]  (&A[2][0],  +0, 1, +0, 1), (&A[2][0], +0, 0, +0, 1)  Trivial
    &A[2][1]  (&A[2][1],  +0, 1, +0, 1), (&A[2][1], +0, 1, +0, 1)  Trivial
              (&A[2][0],  +0, 1, +0, 1), (&A[2][0], +4, 0, +1, 2)  Combine
    &A[2][2]  (&A[2][2],  +0, 1, +0, 1), (&A[2][2], +0, 2, +0, 1)  Trivial
              (&A[2][0],  +0, 1, +0, 1), (&A[2][0], +4, 0, +1, 3)  Combine
              (&A[0][0], +40, 0, +1, 3), (&A[0][0], +4, 0, +1, 3)  Combine
```

Figure 2.9: Sequenced PRSD Compression Simulation Example

irregular accesses. Instead, the technique would generate a large number of sequenced PRSDs, which also makes dependence testing inefficient.

2.6.4 Handling Irregular Accesses

One technique we explored during our research for handling irregular accesses was founded on the observation that large data structures are usually defined by regular regions of memory. For most optimized algorithms, all information maintained in data structures is used, hence, addresses of irregular accesses would eventually define a regular memory region that can be represented as a PRSD. This can be done by buffering all irregular accesses that occurred for a certain length of time into the past, sorting the accesses and applying PRSD-

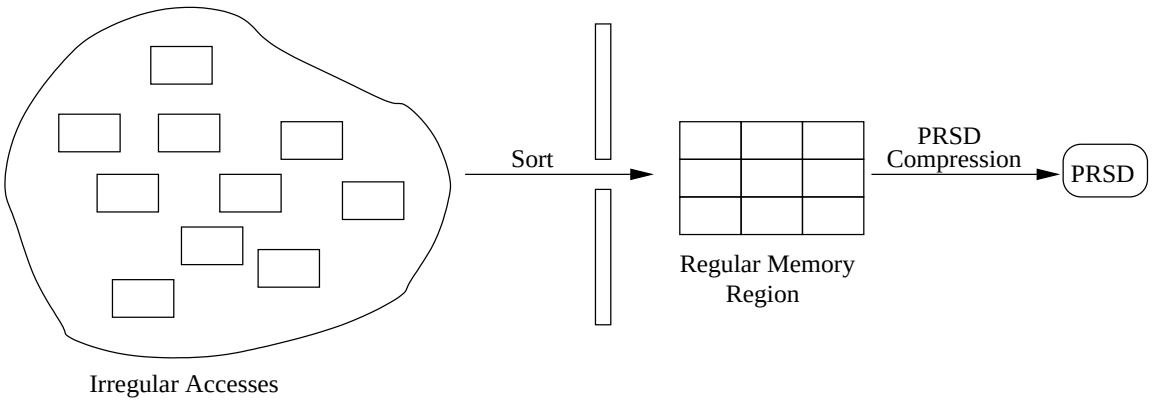

Figure 2.10: Imprecise Dependence Vector

like compression on the new access stream. Although this technique has the advantage of being able to compress irregular accesses, dependence analysis may be adversely affected by the loss of sequencing information. Hence, we currently do not use this method in our dependence analysis.

We also propose *largest common sub-range descriptors* (LCSDs) for compressing irregular accesses by expanding regions defined by PRSDs in a hierarchical manner to build up larger regions. The objective of this scheme would be to describe the largest region of memory by introducing the fewest pseudo-references in the access stream. This scheme would introduce false-dependences in the analysis but would make the process of dependence analysis more scalable. The handling of irregular references is subject to future work.

2.6.5 Split Range Access Pools

The auxiliary memory maps that we maintain along with our PRSD access pools are important in reducing the complexity of dependence analysis. These maps divide the address range to limit the number of PRSDs that a particular access needs to be checked against for dependence. These maps are organized as balanced binary search trees making searching for an appropriate region extremely fast. Hence, we refer to the set of regions as the region-tree.

A region R of PRSDs P_k in the *split range access pool*, described in Section 2.4, is defined as follows:

$$R = \bigcup\{P_i | \forall P_i \in P | R = \phi \vee \exists P_j \in R \wedge P_i \cap P_j \neq \phi\}$$

Intuitively, it coalesces all address ranges of overlapping PRSDs into a single memory region. The idea behind forming such ranges is that PRSDs of different ranges are independent of each other and this fact makes dependence analysis faster in the average case. However, the super-range access pool does have the disadvantage of causing more dependence checks for PRSDs. Consider an example where only regions R_1 and R_2 and regions R_1 and R_3 overlap. An artifact of the union of the three sets into a single region is that an access belonging to region R_2 would also be checked for dependence with PRSDs from region R_3. This is unnecessary and increases the time required for dependence analysis.

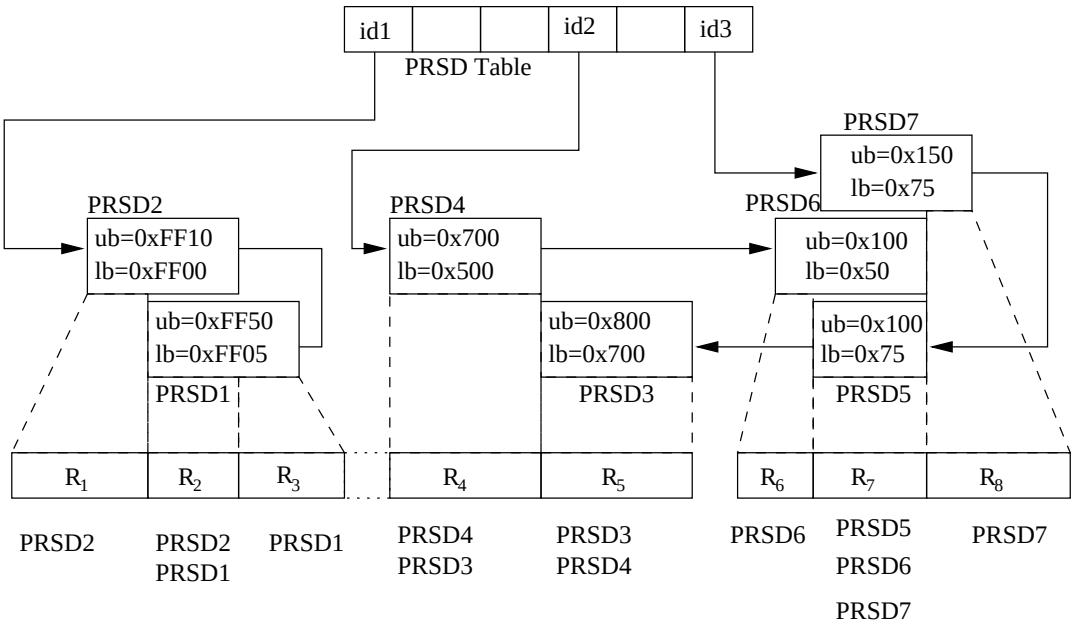

Figure 2.11: Split Range Access Pool

We explored another data structure we call the *split range access pool* shown in Figure 2.11. In this technique, if two ranges P and R overlap, then we create three new ranges R_1, R_2 and R_3, defined as follows:

$R_1 = P - (P \cap R)$ $R_2 = (P \cap R)$ $R_3 = Q - (P \cap R)$

Simply put, if two ranges overlap, we form three new regions: one corresponding to the overlapping region and at most two more regions for the remaining parts of the original two regions. Consider PRSDs $PRSD1$ and $PRSD2$ in Figure 2.11. The regions formed are regions $R1$, $R2$ and $R3$. $R1$ and $R3$ only contain accesses from $PRSD1$ and $PRSD2$, respectively while $R2$ contains accesses from both $PRSD1$ and $PRSD2$. Thus,

any access that belongs to region $R1$ only needs to be checked for dependence with $PRSD1$ and not with $PRSD2$, which does not belong to $R1$. Only an access that belongs to region $R2$ needs to be checked with both $PRSD1$ and $PRSD2$ for dependence. As can be seen from the figure, the pessimistic approach of unification in the super-range access pool case only applies to the intersection region, which can be significantly smaller than the union of both regions. Thus this structure eliminates false dependences among PRSDs, and the PRSDs in a particular region are the minimal set of PRSDs that need to be checked for dependence. Hence, we believed that this structure makes dependence checking much faster. A key point to note here is that in this case, each PRSD may appear in multiple regions in the region tree. If a PRSD needs to be deleted, multiple nodes in the region tree would have to be visited, and the PRSD would subsequently be deleted from each of them. **Our experiments showed that in the case of non-row major accesses, the region tree suffered from heavy fragmentation and, thus, resulted in it having as many vertices as the number of elements accessed.** This caused an explosion in the number of references to a PRSD, making deletion of a PRSD from the access pool very costly. Hence, in spite of the favorable properties of the split range access pool, we chose the super-range access pool.

Chapter 3

Experimental Results

This chapter describes the experiments to measure the correctness of our framework and performance of the programs generated through our framework.

3.1 Experimental Setup

Our experiments were conducted on a dual processor 900Mhz Itanium^{TM}2 machine running Linux and configured our benchmarks to be compiled for two threads. We used ORC version 2.1 for our framework.

We used a modified version of the NAS OpenMP C benchmark set (version 2.3) for our experiments. The source code of each benchmark was modified by removing all OpenMP directives from them. In addition, we also replaced the complex number implementation used in the `FT` benchmark to use that provided by the C Standard Library. This modification was required to circumvent a deficiency in our branch of ORC that performs reduction analysis on structure elements incorrectly. The benchmarks were compiled at the highest level of optimization (`-O3`). Our research focuses on do-all rather than do-across auto-parallelization, hence we used the do-all `-apo` flag and turned do-across auto-parallelization off. We also switched inter-procedural analysis and binary dead-code elimination off to bypass secondary bugs in our branch of ORC. The benchmark speedup numbers reported in this chapter are that for the highest class that our branch of ORC could compile them for. The compilation for some of the benchmarks failed for the highest class due to issues in the compiler back-end that could not be fixed due to time constraints.

In the following sections, we first describe our methodology for verifying the cor-

rectness of our framework. We then evaluate the results of our framework using speedups and number of loops parallelized as our metrics.

3.2 Validating Correctness of the Framework

We manually verified correctness of each stage of the framework for micro-benchmarks by examining the trace file and the dependence graphs generated and relating them to the original source code. The NAS benchmarks compare the results computed with pre-computed results stored as static tables to verify them. Our tracing and optimization phases passed the verification tests successfully thus validating the correctness of our framework. We could also run any benchmark-class combination through our framework that our unmodified branch of ORC could compile.

3.3 Speed-up Measurements

In this section, we present experimental results for the speed-up experiments that we performed for the NAS benchmark set. The speedup of the program generated using our framework is measured with respect to a program parallelized using only static analysis. The following tables show the results for our PRSD compression-based technique as well as the shadow map-based technique.

We used class `S` as our training input set for generating traces. The dependence graphs generated using this training input set were used to optimize the benchmarks for all other classes, too. While dynamic analysis has the drawback that the optimizations are only valid for the input set for which the original analysis was performed, we assume that the training data set for our analysis is representative of the inputs to the program in general.

We report our performance results in Table 3.1. `BT`, `FT` and `MG` were compiled for class B while `EP`, `LU` and `SP` were compiled for class C. Interestingly, all the benchmarks successfully passed their verification tests even though their dependence graphs were generated for class S. We see the most improvement in performance for `FT` and `LU` benchmarks. `BT`, `EP` and `SP` do not show any improvement or degradation in performance. However, the `MG` benchmark shows a slow-down with our dynamic dependence graph. When we combine the dependence graphs obtained from static and dynamic analysis, we observed that the

Table 3.1: Performance Results with NAS Parallel Benchmarks

NPB	Wall-clock time (seconds)			Speedup	
	Static Graph	Dynamic Graph			
		PRSD	Shadow Map	PRSD	Shadow Map
BT	4222.80	4235.66	4228.72	-0.30%	-0.14%
EP	1956.20	1956.69	1949.99	-0.03%	0.32%
FT	680.48	604.23	602.14	11.20%	11.51%
LU	7547.34	7261.83	6979.08	3.78%	7.53%
MG	96.26	97.51	98.91	-1.30%	-2.75%
SP	6420.07	6391.06	6414.79	0.45%	0.08%

Table 3.2: Analysis Runtime Overhead Comparison

NPB	Wall-clock time (seconds)		Overhead
	PRSD	Shadow Map	
BT	21176.291	11057.15	47.79%
EP	4514.791	3008.81	33.36%
FT	3613.289	3545.50	1.88%
LU	4360.125	3721.64	14.64%
MG	251.062	244.19	2.74%
SP	1653.741	1511.70	8.59%

results from our PRSD compression technique are comparable to those from the shadow-map based technique. We also measured the analysis time required by the PRSD-based technique versus that of the shadow map-based technique. As expected, the shadow map based tecnhique requires lesser time to analyze a benchmark compared to the PRSD-based technique. However, the low memory requirement is what makes the PRSD-based technique more attractive.

3.4 Discussion

We anticipated larger speedups for most of the benchmarks when optimized with the dynamic dependence graph obtained from our framework. However, our results do not reflect the potential parallelism that exists in these benchmarks. In this section, we shall present some details of our analysis of ORC and the dependence analysis framework. Through this analysis, we provide insights as to why we observe such small speed-ups or, in some cases, even slow-downs.

3.4.1 Array Privatization

If every use of the array has its corresponding definition in the same iteration (or chunk of iterations for a processor) then the compiler can assign separate copies of the array for each processor. This compiler transformation is called array privatization and it is a powerful technique that aids automatic parallelization. The privatization transformation also needs to know how to initialize the private copy of the array and how to determine the final values of the original array [9], [18]. ORC handles the final value problem by peeling-off the last iteration of the loop and executing the remaining iterations in parallel. This way after $N-1$ iterations of the loop have been executed in parallel, the last iteration writes the final values into the original array. Consider the following example adapted from `psinv` function in the `MG` benchmark:

Exmaple 1 `MG`: Array Privatization Opportunity

```
for i3 = 1 to n3-1 do
  for i2 = 1 to n2-1 do
    for i1 = 0 to n1 do
      r1[i1] = ...
    end for
    for i1 = 1 to n1-1 do
      u[i2][i1] = r1[i-1] + r1[i+1] + ...
    end for
  end for
end for
```

Our dependence analysis framework reports that there is no dependence carried by loop $i2$. This implies that every use of the array $r1$ has its corresponding definition within the same iteration of the $i2$ loop. Hence, it is possible to privatize $r1$ based on our dependence graph. ORC uses an analysis framework called *Array Region Analysis* (ARA) for privatization. This analysis does not seem to be using the array dependence graph for privatization and, hence, does not privatize $r1$. Hence, it does not parallelize this loop nest. We believe that by modifying the ARA analysis to use our array dependence graph, we would be able to improve our results.

3.4.2 Parallelization Costs

Even though a loop may be proven to be parallelizable, it may not always be profitable to run different sets of iterations in parallel. ORC evaluates the profitability of loop parallelization using the following cost function: $T_c + P \times T_p + N \times W \div P$ where:

- T_c is the constant overhead for initializing a parallel region. If the loop contains any reductions, we also add in an estimated cost for combining the partial reduction results computed by each processor.
- T_p is the per-processor overhead for initializing a parallel region.
- P is the number of processors.
- W is an estimate of the serial work (number of cycles) per iteration of the loop, based on a combination of the machine operation and cache miss cost estimates for the loop.
- N is the number of iterations of the loop.

ORC places a guard condition before the parallel region in order to dynamically control whether a particular loop nest is to be parallelized or not. If the bounds of the loop can be statically determined, then the guard condition is not required since the decision of whether a loop should be parallelized or not can be made at compile time. The guard condition evaluates the following inequality:

$$T_c + P \times T_p + N \times W \div P \leq N \times W$$

Simply put, this inequality evaluates whether the cost of parallelization exceeds its benefit. If this inequality is not satisfied, then the loop should not be parallelized. We observe slow-downs in some cases due to sub-optimal values for the T_c and T_p parameters. We believe that with better tuned values of these parameters for our implementation of the OpenMP library, ORC would be able to guard against profitable parallelization and, hence, improve our results.

3.4.3 Non-`DO` Loops

A `DO` in WHIRL has the same semantics as a FORTRAN do loop. Usually, `DO` loops correspond to C `for` loops. The induction variable, bounds and step of the `DO` may never be changed inside the loop body. However, if they do change, then ORC converts those loops to `DO_WHILE` and `WHILE_DO` before the IR reaches the LNO phase.

Consider the following example adapted from the FT benchmark. In this program, the array d is passed as a parameter to this function. Due to limited inter-procedural analysis, the compiler assumes that d may be modified across the call to function $cfftz$. Hence, ORC converts both i and j loops to DO_LOOPs whereas the jj and k loops are converted into WHILE_DOs. Inspite of our framework's ability to show the independence between the call to $cfftz$ and the array d, ORC does not consider non-DO_LOOPs for parallelization. As is in this case, outer loops may not be parallelized and all the parallelism that exists in the benchmark may not be exploited. We believe that this is another reason why we are not achieving higher speedups for our benchmarks.

Exmaple 2 FT: Non-DO Loops

```
for k = 0 ; k < d[2] ; k++ do
  for jj = 0 ; jj <= d[1] - fftblock ; jj += fftblock do
    for j = 0 ; j < fftblock ; j++ do
      for i = 0 ; i < d[0] ; i++ do
        y0[i][j] = x[k][j+jj][i]
      end for
    end for
    cfftz (is, logd[0], d[0], y0, y1)
    for j = 0 ; j < fftblock ; j++ do
      for i = 0 ; i < d[0] ; i++ do
        xout[k][j+jj][i] = y0[i][j]
      end for
    end for
  end for
end for
```

Hence, we conclude that translating opportunities for parallelism exposed by our dynamic dependence analysis into performance improvements requires stronger support within the compiler.

Chapter 4

Related Work

RSDs have been used to describe data references in a loop [4]. The idea of PRSDs originates from memory trace compression performed on-the-fly [7]. While their work introduced the general concepts and an algorithm for compressing regular data references, our work uses PRSDs to calculate dependences in a program.

Rus et al. proposed hybrid analysis [15] to combine static and dynamic analysis of memory references using *Linear Memory Access Descriptors*. They place run-time dependence tests for loops in order to perform automatic parallelization. While their work is not targeted for speculative optimizations, their approach of aggregating memory accesses into a single descriptor to ease analysis is similar to our PRSD-based approach to dependence analysis.

Oplinger et al. developed a trace-driven tool called `Memdeps` [11] to select the best loop for speculative parallelization in a dynamic loop nest. `Memdeps` operates on traces generated by the `Atom`™ tool[17] which annotates program binaries and generates user level traces that include events of interest: loads and stores, procedure entries and exits, and iteration advances. While our tracing tool is similar in the events traced, we operate at the IR level within the compiler as opposed to directly on the binary. Also, they target integer application while we target scientific applications. Hence, their work targets finer grained parallelism than ours, *i.e,* ours can exploit larger iteration spaces on arrays and matrices. `Memdeps` detects dependences at run-time and characterizes these dependences with the *minimum execution delay* of the edge, which is the number of instructions between the two end-points of the dependence edge. Our dependence analysis framework is different in that we calculate dependence vectors for each edge.

Maydan et al. provides a methodology to evaluate the effectiveness of data dependence analyzers [8]. They use Larus' dynamic dependence analysis system `llpp` to evaluate the effectiveness of their own symbolic dependence analysis scheme. They conclude that memory disambiguation is the major cause of failure for dependence analysis. They observed that array privatization is an important technique to parallelize loops, which is one of the conclusions that we, too, were able to derive from our analysis.

Rauchwerger et al. proposed the LRPD test [14] for detecting dependence violations at run-time in speculatively parallelized loops. Our dependence analysis framework bears close resemblance to their dependence violation detection scheme but differs in that we use more efficient compressed representations for accesses while they use shadow-map like bit-arrays. Also, we can detect dependencies at any level in a loop nest, while the LRPD test can only detect dependencies for the speculatively parallel region as a whole.

The profiling and speculative optimization framework developed by Chen et al. for ORC [1] is the most closely related work to ours. Their framework used shadow maps for dependence analysis and was targeted for speculative PRE and speculative code scheduling optimizations. They instrument the IR within ORC after the WOPT phase. In contrast, our instrumentation is done at the LNO phase and we target loop nest optimizations, such as auto-parallelization, loop interchange, privatization and reduction optimizations. Compared to their dependence graph, we do not maintain input dependences. Instead, we have to maintain a list of all the loads from a particular address without an intervening store to the same address. Since they maintain input dependences in their dependence graph, they only maintain the latest load from a particular address. This makes dependence testing for anti-dependences slightly faster but entails transitively converting input dependences into anti-dependences.

Chapter 5

Conclusions and Future Work

This thesis describes a novel application of PRSDs for dependence analysis aiding speculative loop optimizations. Our contributions are an instrumentation framework within ORC and a dependence analysis framework that produces dynamic array dependence graphs for each function in the program. In addition, we provide an interface within ORC to read and transparently use our dynamic dependence graph to perform loop nest optimizations. We explored different techniques to make the dependence analysis both efficient and scalable. We present a technique to deal with irregular accesses. We also present an enhanced PRSD that compresses not only the address trace but also the iteration vectors of each access. We also evaluate the ability of our PRSD-based dependence analysis to accurately reflect TLS opportunities with a shadow-map based technique. We observe up to 11% improvement in performance and conclude that the compiler needs to be enhanced to take full advantage of our relaxed dependence information in order to yield larger benefits from speculative parallelism.

The dependence analysis framework represents the first step towards a completely automated speculation environment. Initially, we plan on adding sequencing information as discussed in Section 2.6.3 and handle irregular accesses using LCSDs as discussed in Section 2.6.4 to enhance our dependence analysis framework. We also plan to modify the compiler to use our relaxed dependence information in correctly generating `DO` loops and performing array privatization. We also plan on making the parallelization cost calculations more accurate by deriving correct values for parallelization overhead parameters and obtaining run-time estimates for the cost of calls. These enhancements would strengthen the optimization framework and capitalize on the true power of speculative loop optimiza-

tions. While presently we do not provide any guarantees for correctness of the speculative program, our next step would be to build a complete speculation run-time system that detects dependence violations and, in case of mis-speculation, rolls-back computation. Using PRSDs we plan to make enhancements to the LRPD work.

Bibliography

[1] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. Data dependence profiling for speculative optimizations. In Evelyn Duesterwald, editor, *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2004.

[2] Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors. *Languages and Compilers for High Performance Computing, 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers*, volume 3602 of *Lecture Notes in Computer Science*. Springer, 2005.

[3] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, New York, NY, USA, 1998. ACM Press.

[4] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[5] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 59–70, New York, NY, USA, 2004. ACM Press.

[6] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[7] Jaydeep Marathe, Frank Mueller, Tushar Mohan, Bronis R. de Supinski, Sally A.

McKee, and Andy Yoo. METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *CGO*, pages 289–300. IEEE Computer Society, 2003.

[8] D. Maydan, J. Hennessy, and M. Lam. Effectiveness of data dependence analysis. In *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*, 1992.

[9] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 2–15, New York, NY, USA, 1993. ACM Press.

[10] Open64 - the open research compiler.

[11] Jeffrey Oplinger, David Heine, Shih Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical report, Stanford, CA, USA, 1997.

[12] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303, Washington, DC, USA, 1999. IEEE Computer Society.

[13] Open research compiler for itanium processor family, Sept 2005.

[14] Lawrence Rauchwerger and David Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 218–232, New York, NY, USA, 1995. ACM Press.

[15] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.*, 31(4):251–283, 2003.

[16] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 202–211, New York, NY, USA, 1986. ACM Press.

[17] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.

[18] Peng Tu and David A. Padua. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, London, UK, 1994. Springer-Verlag.

[19] *WHIRL Intermediate Language Specification*, October 2000.