Automatic Speculative DOALL for Clusters

Hanjun Kim Nick P. Johnson Jae W. Lee[†] Scott A. Mahlke[‡] David I. August

Princeton University, Princeton, NJ, USA
[†]SungKyunKwan University, Suwon, Korea
[‡]University of Michigan, Ann Arbor, MI, USA

ABSTRACT

Automatic parallelization for clusters is a promising alternative to time-consuming, error-prone manual parallelization. However, automatic parallelization is frequently limited by the imprecision of static analysis. Moreover, due to the inherent fragility of static analysis, small changes to the source code can significantly undermine performance. By replacing static analysis with speculation and profiling, automatic parallelization becomes more robust and applicable. A naïve automatic speculative parallelization does not scale for distributed memory clusters, due to the high bandwidth required to validate speculation. This work is the first automatic speculative DOALL (Spec-DOALL) parallelization system for clusters. We have implemented a prototype automatic parallelization system, called Cluster Spec-DOALL, which consists of a Spec-DOALL parallelizing compiler and a speculative runtime for clusters. Since the compiler optimizes communication patterns, and the runtime is optimized for the cases in which speculation succeeds, Cluster Spec-DOALL minimizes the communication and validation overheads of the speculative runtime. Across 8 benchmarks, Cluster Spec-DOALL achieves a geomean speedup of 43.8× on a 120-core cluster, whereas DOALL without speculation achieves only 4.5× speedup. This demonstrates that speculation makes scalable fully-automatic parallelization for clusters possible.

1. INTRODUCTION

Clusters of commodity servers and switches are the most popular form of large-scale parallel computers to speed up the execution of programs that require large computation power. While clusters provide scalable hardware resources such as processor cores, memory, and I/O bandwidth, programs need to be parallelized to efficiently utilize these parallel hardware resources. As a result, clusters are primarily used for scientific programs or web services, which consist of units of work that are mostly independent.

However, extracting scalable parallelism from sequential programs on clusters is challenging for two main reasons. First, commodity clusters do not provide shared memory. This requires the parallelizer (programmer or compiler) to identify shared data and explicitly insert communication primitives between producers and consumers. Second, clusters have high inter-node communication latency. Without careful communication optimization, the inter-node communication cost easily becomes a performance bottleneck.

There are two main strategies for scalable, efficient parallelization on clusters: parallel programming methods and automatic parallelization methods. Explicit parallel programming using a message passing protocol (e.g., MPI) is one potential solution to the problem, but it can severely limit the programmer's productivity by requiring a deep knowledge of concurrency, domain expertise, and platform-specific performance tuning. Parallelization APIs such as Cluster OpenMP [7] can help programmers parallelize sequential programs on clusters. As the programmers annotate what and how to parallelize the sequential programs, the compiler generates parallel codes. However, the programmers still need to analyze the data and control dependences of the program to find effective parallelization strategies.

Automatic parallelization research has a rich history, especially in the scientific computing community. Automatic parallelizing compilers such as SUIF [1, 19] and Polaris [3] parallelize a sequential program without programmer's intervention. These compilers automatically find loops that can be parallelized using static analyses, and transform the loops for parallel execution. However, their applicability has been limited mostly to array-based scientific applications that have well-analyzable, regular data access patterns mainly because of imprecise static analysis. Moreover, alias analysis is too fragile to achieve stable performance with small changes in the source code. More sophisticated, robust memory dependence analysis can mitigate this problem, but there are still many applications that are hard to parallelize automatically including those with memory accesses through pointers or indirect arrays, interprocedural dependences, and so on.

Automatic speculative parallelization [11, 13, 16, 20, 25] can overcome the limitations of static compiler analysis. These compilers speculatively remove memory or control dependences among instructions, and optimistically parallelize loops. However, these proposals assume the availability of specialized hardware or cache-coherent shared memory, and their scalability has not been demonstrated beyond 8 cores.

This paper is the first to demonstrate automatic speculative DOALL (Spec-DOALL) parallelization for clusters, addressing the problems of limited applicability and lack of performance stability. We have implemented a prototype automatic parallelization system, called Cluster Spec-DOALL, by combining a Spec-DOALL compiler with a speculative runtime system.

Copyright©ACM, 2012. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in 2012 International Symposium on Code Generation and Optimization (CGO 2012).

The Cluster Spec-DOALL compiler automatically identifies DOALL-able or speculatively DOALL-able loops in a program via dynamic profiling runs and static dependence analysis at compile-time. The compiler speculatively removes data and control dependences that prevent parallelization of the loops, guided by a set of profilers. A code generator transforms the loop and inserts communication primitives for flow dependences across parallel contexts. A communication optimizer aggressively promotes and batches communication calls in inner loops to reduce the amount of communication from worker processes to the validation and commit processes, which easily become a performance bottleneck on clusters.

The parallelized programs are executed on top of the Cluster Spec-DOALL runtime system. As the program is executed, the Cluster Spec-DOALL runtime checks if misspeculation occurs. If a speculatively removed dependence manifests at run-time, the Cluster Spec-DOALL runtime rolls back to a previous non-speculative program state, executes the misspeculated iteration sequentially, and resumes speculative parallel execution of the following iterations.

The primary contributions of this paper are:

- The first fully-automatic speculative parallelization system targeting commodity clusters (called Cluster Spec-DOALL)
- Highly effective communication optimizations which reduce the communication and validation overhead, enabling scalable performance for clusters
- An in-depth evaluation of Cluster Spec-DOALL on a 120-core cluster using 13 benchmarks from PolyBench and PARSEC benchmark suites

2. BACKGROUND AND MOTIVATION

Automatic parallelization has achieved limited success in parallelizing sequential programs mainly because of imprecise and fragile static analysis. Section 2.1 identifies the limitations of conventional analysis-based approaches to automatic parallelization. Section 2.2 motivates speculative parallelization to overcome these limitations and communication optimization to achieve scalable performance.

2.1 Analysis-based Approaches in Automatic Parallelization

Automatic parallelization is an ideal solution which frees programmers from difficulties of parallel programming and platform-specific performance tuning. Parallelizing compilers can automatically parallelize affine loops [1, 3]. `Loop_A` in Figure 1 shows such an example code. If a compiler proves that all memory variables in the body of the function `foo` do not alias the array `regular` via inter-procedural analysis, the loop is parallelized. Therefore, the utility of an automatic parallelizing compiler is largely determined by the quality of its memory dependence analysis.

In some cases, static analysis may be imprecise. For example, within the function `foo`, assume that there is a read from or write to the array element `regular[i+M]`, (and the size of the array is greater than (`M`+`N`)), where `M` is an input from the user. In this case, `Loop_A` may not be DOALL-able depending on the value of `M`. If `M` is greater than `N`, the loop is DOALL-able; otherwise, it is not. Some research compilers such as SUIF [1] and Polaris [3, 18] integrate low-cost run-time analysis capabilities to insert a small

```
1: Loop_A:                        5: Loop_B:
2: for (int i=0; i<N; i++)        6: for (int i=0; i<N; i++) {
3:  regular[i] += foo(i);         7:   irregular[idx[i]] += foo(i);
4:                                8:   if (irregular[idx[i]] > error)
                                  9:     printf("I/O operation!");
                                 10: }
```

Figure 1: Sequential Code with Two Loops

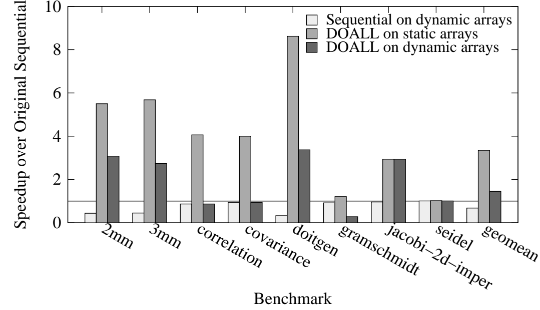

Figure 2: Performance sensitivity due to memory analysis on a shared-memory machine

test code to check the value of `M` at run-time to select either a sequential or parallel version of the loop accordingly. However, the coverage of these techniques is mostly restricted to the cases when a predicate can be extracted outside the analyzed loop and a low cost run-time test can be generated [18]. They cannot be applied to `Loop_B` in Figure 1, for example, where an index array is used to access the array `irregular` and a simple predicate cannot be extracted outside the loop to test the `if` condition within the loop body.

Another issue with automatic parallelization is the fragility of static analysis. Figure 2 illustrates how fragile static analysis can be with a small change in the program. In this example, the automatic parallelizer can easily parallelize the unmodified PolyBench benchmarks [15] using static arrays. However, if we replace the static arrays with dynamically allocated arrays, it not only suppresses some of the optimizations previously applied but also blocks parallelization for several benchmarks since heap objects are generally more difficult to analyze. This leads to run-time performance that is highly sensitive to the implementation style.

Therefore, analysis-based approaches, both static and dynamic, are not sufficient for parallelization of even array-based applications, let alone pointer-based ones, having irregular memory accesses and complex control flows. Moreover, recursive data structures, dynamic memory allocation, and frequent accesses to shared variables pose additional challenges. Imprecise, fragile static analysis has severely limited the applicability of conventional automatic parallelization.

2.2 Spec-DOALL Parallelization on Clusters

A viable strategy to overcome the limitations of static analysis is to exploit optimistic parallelism via data and control speculation. For example, the compiler can apply Spec-DOALL parallelization to `Loop_B` in Figure 1, speculating that no cross-iteration dependence violation occurs via concurrent array accesses and that the error condition in Line 8 does not happen at run-time. This approach requires runtime support for misspeculation detection and recovery in either hardware or software to ensure correctness.

System	Fully Automatic	Supports Spec-DOALL	Requires HW Support	Targets Clusters	Number of Cores Used for Evaluation
Cluster OpenMP [7]	No	No	No	Yes	-
SUIF [1]	Yes	No	No	Yes	32
Polaris [3, 18]	Yes	No	No	No	8, 16
POSH [11]	Yes	Yes	Yes	No	4
STMlite [13]	Yes	Yes	No	No	8
Cluster Spec-DOALL [This paper]	Yes	Yes	No	Yes	120

Table 1: Comparison of automatic parallelization systems

Runtime support for speculative parallelism has been an active area of research, and there are a number of proposals including Transactional Memories (TM) and Thread Level Speculation (TLS) memory systems. The runtime system tracks every speculative memory operation within a transaction or task (i.e., region of code executed speculatively) to determine if any atomicity violation (in TM) or dependence violation (in TLS) occurs at commit time. Proposals for TM or TLS memory systems can be divided into two classes: hardware-based approaches [21, 24, 23] and software-only approaches [4, 6, 8, 9, 12, 13, 14, 17, 22]. Software-only approaches can be further divided depending on whether they require cache-coherent shared memory or not. Most existing proposals for software-only speculative runtimes target only small-scale shared-memory computers with tens of cores at most [13, 14, 17, 22].

There are research compilers which parallelize applications using speculation [11, 13, 16, 25]. However, these compilers assume the availability of specialized hardware or cache-coherent shared memory, and their performance is evaluated using a small number of cores (typically fewer than 8). Software transactional memory systems have suffered from large validation overhead [5], consequently they may not scale to a large number of cores. To achieve scalable performance on a large number of cores, it is crucial to optimize communication because the commit bandwidth easily becomes a performance bottleneck.

There have been proposals for TM and TLS memory systems on clusters [4, 6, 8, 9, 12], but only Cluster-STM [4] and DSMTX [8] have demonstrated their scalability on platforms with over 100 cores. In addition, among the proposals, there is no known automatic speculative parallelization system targeting them. Cluster Spec-DOALL is the first fully-automatic speculative parallelization system that scales to hundreds of cores without requiring hardware support or cache-coherent shared memory. Table 1 compares this work with other existing automatic parallelization systems.

3. OVERVIEW OF Cluster Spec-DOALL

Figure 3 illustrates the overall structure of the Cluster Spec-DOALL system. Cluster Spec-DOALL consists of a parallelizing compiler including a set of profilers and a runtime supporting speculative execution. The compiler finds and parallelizes DOALL-able or Spec-DOALL-able loops by using both static alias analysis and dynamic profiling results. The runtime executes these parallelized loops safely and efficiently on clusters.

3.1 Cluster Spec-DOALL Compiler

The Cluster Spec-DOALL compiler takes sequential C/C++ source code as input to generate parallelized code targeting the Cluster Spec-DOALL runtime. The compiler framework is composed of the following components: dependence analyzer, DOALL parallelizer, speculator, Spec-DOALL parallelizer and communication optimizer. The rest of this section briefly explains the functionality of each component, which will be discussed in greater detail in Section 4.

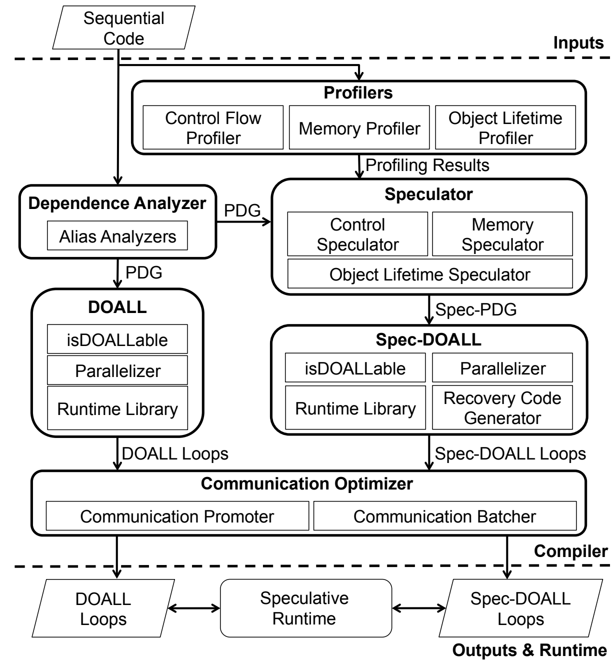

Figure 3: Overall Cluster Spec-DOALL System

Profilers: The profilers gather dynamic information by executing the sequential code with training input sets. More specifically, Cluster Spec-DOALL uses three profiles: control flow profile, memory dependence profile and object lifetime profile. The speculator uses these profiling results to target control and data speculation.

Dependence Analyzer: The dependence analyzer creates a program dependence graph (PDG) via static analysis, which includes both data and control dependences. Since the Cluster Spec-DOALL runtime employs private memory space for each worker process, loop-carried anti- and output-dependences are ignored in the parallel region. Although the compiler exploits profiling information to further refine the program dependence structure, better static analysis is always helpful to generate more efficient parallel codes with fewer speculated dependences leading to lower validation cost.

DOALL Parallelizer: Whenever applicable, the compiler parallelizes a loop without speculation using the classical DOALL transformation. Specifically, loops without loop-carried dependence can be parallelized with DOALL. The DOALL parallelizer adds calls to the Cluster Spec-DOALL runtime for process management, and live-in and live-out handling.

Speculator: When DOALL parallelization is not applicable, the compiler uses profiling information to speculatively remove dependence edges from the PDG. The refined PDG is called *Spec-PDG*.

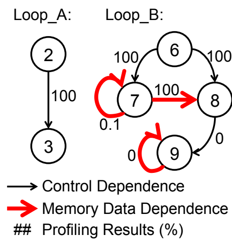

Figure 4: PDG + Profiling

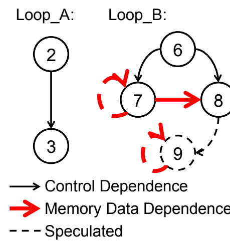

Figure 5: Spec-PDG

The Spec-PDG is fed into the Spec-DOALL parallelizer for speculative parallelization.

Spec-DOALL Parallelizer: If there is no loop-carried dependence in the Spec-PDG, the Spec-DOALL parallelizer transforms the sequential loop for parallel execution as the DOALL parallelizer does. Additionally, the Spec-DOALL parallelizer inserts runtime function calls to detect and recover misspeculated execution. It also generates recovery code to re-execute a misspeculated iteration, which is invoked after rolling back to the correct program state.

Communication Optimizer: It is important to reduce the amount of communication since clusters have high communication cost. For live-out and speculated memory accesses, the communication optimizer aggressively attempts to hoist them out of inner loops by promotion and batching.

3.2 Cluster Spec-DOALL Runtime

The parallelized loops with DOALL and Spec-DOALL are executed on the Cluster Spec-DOALL runtime, which supports speculative memory accesses, misspeculation detection and recovery, live-in and live-out handling, and process management. As a distributed transactional memory system, the runtime validates memory speculation, and manages rollbacks in the event of misspeculation. To efficiently pass live-in values on a cluster, the runtime adopts copy-on-access [8]. Section 5 presents more details of the runtime.

4. Cluster Spec-DOALL COMPILER

The Cluster Spec-DOALL compiler parallelizes sequential C/C++ source code with the following components: dependence analyzer, DOALL parallelizer, speculator, Spec-DOALL parallelizer, recovery code generator, and communication optimizer. This section describes each component in detail.

4.1 Speculator

The speculator removes dependence edges from the PDG that are unlikely to manifest at run-time, to produce a Spec-PDG. The three types of speculation supported by Cluster Spec-DOALL is guided by three different profilers as follows:

Control Speculation: A profiler for control speculation collects the traversal count of every edge in the control flow graph. For each control flow edge, it computes the ratio of the number of times the edge is taken to total loop iterations. When this ratio is smaller than a static threshold, the speculator marks that edge as *speculated*, and all basic blocks which are dominated by that control flow edge as *speculatively dead*. Control speculation does not require any inter-node communication except for misspeculation recovery, so Cluster Spec-DOALL preferentially applies control speculation over the other forms of speculation.

```
Loop_A:
// Master Process
beginInvocation(DOALL);
produceLiveIns();
consumeLiveOuts();
endInvocation();
```

```
// Worker Process
beginInvocation(DOALL);
consumeLiveIns();
for(int i=0; i<N; i++) {
  if(i%NP==tid) {
    regular[i]+=foo(i);
    produce(&regular[i]);
} }
endInvocation();
```

Figure 6: DOALL (Loop_A)

Memory Flow Speculation: Memory flow speculation relies on a memory profiler which observes the flow of values from stores to loads. This information is stronger than alias information because two memory operations may alias even when there is no flow between them. The speculator identifies loop-carried memory flow dependences which occur less frequently than a static threshold, and marks them as *speculated*. Inter-node communication must be inserted to detect a memory flow misspeculation at run-time, so memory flow speculation has higher overhead than control speculation.

Object Lifetime Speculation: Object lifetime speculation is guided by a profiler to identify dynamic objects which are private to a single loop iteration. The profiler reports allocation sites whose object is not freed in the same iteration of a loop, and deallocation sites whose object is not allocated in the same iteration. Updates to iteration-private objects are independent across iterations, and are not live-out of the loop, reducing the amount of inter-node communication. Specialized versions of `malloc` and `free` automatically test for misspeculation without inter-node communication.

Figure 4 shows a PDG with profiling results for the example code in Figure 1, where the node number corresponds to the line number. Since the value of `idx[i]` is irregular, static alias analysis conservatively inserts a loop-carried memory dependence on `node 7`. Since I/O operations must be executed in program order, a loop-carried self-dependence exists on `node 9`. Figure 5 shows how the speculator generates a Spec-PDG by speculatively removing loop-carried dependences based on the profiling results in Figure 4. According to the profiles, the branch from `node 8` to `node 9` never occurs during profiling, hence the control speculator marks `node 9` as speculatively removed. The loop-carried memory dependence on `node 7` rarely occurs, so the memory speculator also removes that dependence.

4.2 DOALL Parallelizer

The DOALL transform is applied whenever a loop can be parallelized non-speculatively. For example, `Loop_A` in Figure 1 can be parallelized with DOALL assuming that static alias analysis proves the absence of loop-carried dependences. Figure 6 shows how the DOALL parallelizer transforms the example code in Figure 1. The loop is wrapped by calls to `beginInvocation` and `endInvocation`, which initialize and finalize the runtime library for parallel execution. Register live-ins are explicitly transferred using inter-node communication queues. Memory live-ins are handled transparently by the copy-on-access mechanism provided by the Cluster Spec-DOALL runtime. Register and memory live-outs are also transferred explicitly via inter-node communication queues.

```
Loop_B:
// Master Process
beginInvocation(Spec-DOALL);
produceLiveIns();
commitProcess(recoveryFcn);
consumeLiveOuts();
endInvocation();

recoveryFcn:
recoveryFcn() {
  int i=loadLV(idx_i);
  irregular[idx[i]]+=foo(i);
  if(irregular[idx[i]]>error)
    printf("I/O operation");
  i++;
  storeLV(idx_i, i);
}
```

```
// Worker Process
beginInvocation(Spec-DOALL);
consumeLiveIns();
executeForLoop();
return;

recoveryBB:
  waitRuntimeRecoverMemory();
  i = loadLV(idx_i);
  goto header;
```

```
executeForLoop(){
  for(int i=0; i<N; i++) {
  header:
    if(TXBoundary()==isMisspec)
      goto recoveryBB;
    if(i%NP==tid) {
      storeLV(idx_i, i);
      specLoad(&irregular[idx[i]]);
      irregular[idx[i]]+=foo(i);
      specStore(&irregular[idx[i]]);
      if(irregular[idx[i]] > error){
        misspec();
        goto recoveryBB;
  } } }
  if(endInvocation()==isMisspec)
    goto recoveryBB;
}:
```

Figure 7: Spec-DOALL (Loop_B)

4.3 Spec-DOALL Parallelizer

Algorithm 1: *Spec-DOALL Parallelizer*

Data: *loop* is a target loop, *specLoopPDG* is PDG with speculation information
Result: generate a speculatively parallelized loop

```
1  let header = getLoopHeader(loop);
2  DOALLTransform(loop);
3  let recoveryBB = insertRecoveryBB();
4  insert(waitRuntimeRecoverMemory, recoveryBB);
5  foreach lv ∈ loop_carried_local_variables do
6  |   insert(loadLV, getLVIdx(lv), lv, recoveryBB) ;
7  |   let newLV = insert(storeLV, getLVIdx(lv), header) ;
8  end
9  let isMisspec = insert(TXBoundary, header) ;
10 insert("if(isMisspec) goto recoveryBB", header) ;
11 foreach BasicBlock exitBB ∈ LoopExitBBs do
12 |   let isMisspec = insert(endInvocation, exitBB) ;
13 |   insert("if(isMisspec) goto recoveryBB", exitBB) ;
14 end
15 foreach branchInfo ∈ getControlSpeculated(specLoopPDG) do
16 |   let branch = getBranchInst(branchInfo);
17 |   let branchOutBB = getUnlikelyBranchedBB(branchInfo);
18 |   let misspecBB = createBB();
19 |   redirectControl(branch, branchOutBB, misspecBB) ;
20 |   insert(misspec, misspecBB) ;
21 |   insert("goto recoveryBB", misspecBB) ;
22 |   removeBBs(getDominatedBBs(branch, branchOutBB));
23 end
24 foreach edge ∈ getMemorySpeculated(specLoopPDG) do
25 |   let StoreInst st = getSrcInst(edge);
26 |   let LoadInst ld = getDstInst(edge);
27 |   insert(specLoad, getPointerAddr(ld), before(ld)) ;
28 |   insert(specStore, getPointerAddr(st), after(st)) ;
29 end
```

If DOALL is not applicable, but there is no loop-carried dependence in the *Spec-PDG*, the Spec-DOALL parallelizer transforms the sequential loop to speculatively parallelized code. The output must include codes to detect and recover misspeculated execution. Figure 7 shows how the Spec-DOALL parallelizer transforms `Loop_B` from Figure 1.

Algorithm 1 shows a procedure that performs this transformation. The Spec-DOALL parallelizer begins by transforming the sequential loop in the same way as the DOALL parallelizer (Line 2). It then creates a basic block named `recoverBB` (Lines 3–4). If control reaches the recovery block, the process begins local misspeculation recovery.

Lines 9–14 isolate each loop iteration as a separate transaction by inserting calls to `TXBoundary` at the loop header and every loop exit. These check whether the master process has sent a misspeculation signal. If so, they initiate local recovery by branching to recovery code. Since other workers may misspeculate after the worker finishes its execution, `endInvocation` blocks until all workers finish parallel execution.

Lines 5–8 insert codes to support recovery in the case of misspeculation. Calls to `storeLV` and `loadLV` pass local variables in each transaction for the runtime to restore locals in the event of misspeculation. It handles only loop-carried local variables; other local variables are either unchanged or unused across iterations, and do not need recovery support.

For control speculation, lines 15–23 redirect speculated branches to *misspecBB*. For memory speculation, lines 24–29 instrument relevant memory operations by inserting `specLoad` and `specStore` calls. These calls collect a transaction log, which is used by the runtime system to detect misspeculation.

4.4 Recovery Code Generator

If misspeculation occurs, all the following speculative iterations must be squashed, and the misspeculated iteration should be executed again honoring the semantics of the original program. The misspeculated iteration will be executed on the master process with the committed program state. The Cluster Spec-DOALL compiler creates a recovery function which performs one iteration of the loop. The compiler redirects back edges to a loop exit block to execute the recovery code only for the misspeculated iteration. To restore register state, the Cluster Spec-DOALL compiler inserts code to restore local variables.

4.5 Communication Optimization

When scaling transactional memories to a large number of cores, the limited commit bandwidth becomes a bottleneck for the whole system. In many programs, memory operations within inner loops of a parallelized loop claim the largest portion of the commit bandwidth to handle live-out and speculative memory accesses. To reduce the amount of communication generated by the inner loops, the Cluster Spec-DOALL compiler performs two optimizations: promotion and batching.

When validating a memory access in a speculative iteration, the validator uses `specStore` to reflect the memory update to the validator's memory version, and `specLoad` to check if the mem-

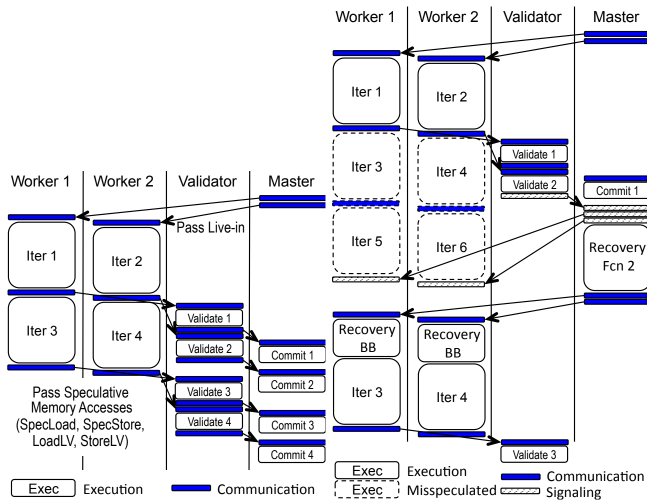

Figure 8: Spec-DOALL **Figure 9: Recovery**

ory access reads the correct version. Within a single transaction, multiple accesses to the same address cause redundant communication; only the first load from and the last store to that address affect validation. Exploiting this, the communication promoter hoists loads from and stores to a loop-invariant address out of the inner loop to the loop preheader and loop exits, respectively. Unlike conventional store/load promotion, this optimization is insensitive to the existence of other instructions that may overwrite or reload the same address, because only the first load and the last store within a transaction matter for validation. Similarly, calls to `produce` for live-outs can be moved to the inner loop's exits.

The service bandwidth is limited not only by the communication volume in bytes but also by the number of messages. Batching is an optimization which gathers dense reads from or writes to a chunk of memory into a single jumbo read or write. The batcher is applicable to memory operations in a counted inner loop whose pointer operands are induction variables of that loop. If applicable, it removes the calls to `specStore` (`specLoad`) from the inner loop, and places calls to `specStoreRange`(`specLoadRange`) after (before) the inner loop. In this way, the Cluster Spec-DOALL runtime can deliver the same number of bytes in fewer messages. A batched function call may be further promoted higher in a loop nest in a way analogous to the promotion of a speculative load or store. In other words, batching not only reduces the number of messages but also exposes hidden opportunities for communication optimization by transforming loop-variant `specStore` and `specLoad` into loop-invariant `specStoreRange` and `specLoadRange`.

5. Cluster Spec-DOALL RUNTIME SYSTEM

The Cluster Spec-DOALL runtime executes parallelized loops supporting speculative memory accesses, misspeculation detection and recovery, live-in and live-out handling, and process management. This section describes how the runtime executes the parallelized loops and detects misspeculation.

5.1 The Execution Model

The Cluster Spec-DOALL runtime orchestrates the execution of parallel codes generated by the Cluster Spec-DOALL compiler on clusters. To exploit parallel hardware for misspeculation detection and recovery, the Cluster Spec-DOALL runtime offloads the task of validating speculative memory accesses to a separate process, called *validator* process, and keeps the committed, non-speculative memory state in the master process as in [8].

Figure 8 shows how the Cluster Spec-DOALL runtime interacts with a Spec-DOALL loop. When a parallel region starts, the master process should pass all live-in values to the worker processes because only the master process keeps the entire non-speculative program state. A conservative approach can be to broadcast all memory pages that are potentially live-in, but this would be prohibitively expensive. Instead, the Cluster Spec-DOALL runtime addresses this using a lazy, copy-on-access (CoA) scheme, which delivers live-in memory pages on demand as in [8]. Each worker installs a custom page fault handler. Whenever a worker process attempts to access a live-in memory address which has not yet been transferred to the worker's memory space, a page fault occurs because the address has not been mapped yet. Then the page fault handler contacts the master process to retrieve a copy of the page being accessed, maps a new page at the same base address in the worker's address space, and copies the contents of the retrieved page.

Worker processes execute each iteration independently. They pass the address, size, and value of both live-out memory store and speculative load and store to the validator process as they execute the parallel loop. Once the validator verifies the correctness of all speculative memory accesses, all live-out values are committed into the master process. When the execution of the parallel loop is finished, sequential execution resumes in the master process. Since all live-out values are passed from the worker processes to the master process, the master process can immediately start executing the following sequential region without extra communication.

If a misspeculation is detected, the recovery procedure begins as shown in Figure 9. The master process sends a misspeculation signal to all workers. The workers unmap all pages which they received via the copy-on-access mechanism. The master process then executes a single misspeculated iteration of the parallel loop with its non-speculative memory state. Register values which are live across loop iterations are communicated to the worker processes, and these processes receive a signal from the master process to resume parallel execution. The workers recover their working memory state via the copy-on-access mechanism.

5.2 Misspeculation Detection

The Cluster Spec-DOALL runtime extends conventional distributed TM systems [4, 8, 9, 12] to support three types of speculation. The Cluster Spec-DOALL runtime supports memory flow speculation via memory versioning, control speculation by allowing the program to explicitly trigger misspeculation recovery, and object lifetime speculation by tracking memory allocations and deallocations.

When memory speculation is employed, one process is dedicated as a validator process. The validator tracks memory accesses and checks memory versions. When a transaction (i.e., single iteration) is finished without misspeculation, the validator forwards all speculated stores to the master process which keeps the committed program state.

The Cluster Spec-DOALL runtime exposes a `misspec` function interface to invoke a recovery process from control misspeculation. The Cluster Spec-DOALL compiler inserts calls to this function along all speculated control flow edges. If a speculated control

flow occurs at run-time, the current and following transactions will be squashed and rolled back for non-speculative re-execution.

Object lifetime speculation is applied to allocation and deallocation sites whose object is likely to be private to one iteration of the loop. The Cluster Spec-DOALL compiler replaces calls to `malloc` and `free` with `specMalloc` and `specFree`. The runtime system records a list of speculatively local objects that have been allocated. When a transaction terminates, the runtime system checks whether the list is empty. If any speculatively local object was not freed by the end of the transaction, misspeculation is signaled. Note that this additional bookkeeping occurs locally at each worker node, so inter-node communication is unnecessary to detect misspeculation. Speculatively local objects are allocated in the private memory space of each worker and considered thread-local, hence reducing overhead for validation and live-out communication.

6. EVALUATION

Cluster Spec-DOALL is evaluated on a 120-core cluster (10 nodes $\times$ 12 cores). Each node has two Intel 6-core Westmere X5650 processors running at 2.67 GHz with 48 GB of memory. It runs 64-bit RedHat Enterprise Linux v5. The inter-node communication link is Mellanox ConnectX Infiniband x4 QDR. OpenMPI (v1.4.1 with gcc v4.1.2, -O2) is used as the underlying communication layer. The Cluster Spec-DOALL compiler builds on the LLVM compiler infrastructure [10].

Cluster Spec-DOALL is evaluated with benchmarks from PolyBench [15] and PARSEC [2] written in C as listed in Table 2. To use different inputs for profiling and evaluation, and to accept a problem size from the command line, we changed statically allocated fixed size arrays to dynamically allocated variable size arrays in the PolyBench benchmarks. For `swaptions`, we removed control predicated I/O operations from a callee of the hottest loop because inter-procedural control speculation is not yet implemented. Memory and object lifetime speculations are inter-procedural.

The sequential programs are profiled less than one minute with profiling inputs. The evaluation inputs are chosen for the original sequential programs to run longer than one hour to observe performance scalability on a large number of cores. Five benchmarks from PolyBench are not used for evaluation because their execution time is too short to be parallelized even with large input sets.

6.1 Parallelization Statistics and Results

Table 2 shows how many dependences are speculated in each benchmark. Although speculation is not necessary to manually parallelize PolyBench benchmarks, Cluster Spec-DOALL employs speculation to avoid relying on strong static alias analysis. While Cluster Spec-DOALL can parallelize benchmarks such as `2mm`, `3mm`, `jacobi-2d-imper`, and `seidel` without any speculation, it requires speculation for the other benchmarks.

`blackscholes`, an option-pricing program from PARSEC, requires control speculation if it is compiled with error checking enabled. The hottest loop prints error messages if a computed price is different from its reference price. The print operation blocks DOALL parallelization, but profile results show that it rarely occurs. Cluster Spec-DOALL speculates that the print condition will not occur, and parallelizes the loop with Spec-DOALL.

Cluster Spec-DOALL parallelizes many loops in `swaptions` with memory speculation. In addition, Cluster Spec-DOALL applies object lifetime speculation for `swaptions` unlike other benchmarks. The outermost loop allocates and frees objects which live only for one iteration. Using object lifetime profile information, Cluster Spec-DOALL speculates that the allocated memory is private to each iteration.

Figure 10 shows the program speedup. Base is the execution time of the original sequential program. In this graph, the horizontal axis shows the number of cores, and the vertical axis shows full application speedups. All execution times were averaged over five runs. The evaluated benchmarks are categorized into two groups; 8 scalable benchmarks and 5 slowdown benchmarks.

6.2 The Eight Scalable Benchmarks

Cluster Spec-DOALL achieves scalable performance on the 8 scalable benchmarks due to a synergistic combination of three design choices.

First, **speculation makes scalable fully-automatic parallelization possible.** Static analysis can always be better, but it is never good enough. Imprecision of static analysis limits classical automatic parallelization. Figure 12 shows performance speedups on 120 cores for DOALL parallelization with only static analysis, Cluster Spec-DOALL, and DOALL parallelization with the aid of an oracle for dependence analysis. Speculation allows the compiler to parallelize outer loops in some programs that DOALL fails to parallelize, and leads to 43.8$\times$ geomean performance speedup for 8 scalable programs, while non-speculative DOALL parallelization achieves only 4.5$\times$ geomean performance speedup. Cluster Spec-DOALL achieves speedup within 7% of the maximum oracle speedup, 46.4$\times$.

Second, **communication optimization realizes the scalability potential in Cluster Spec-DOALL.** Some programs have high ratio of memory accesses to computation. For example, each iteration in `2mm`, a matrix multiplication benchmark, requires two loads and one store to execute only one floating-point multiplication. This high rate of memory accesses requires a large amount of communication, degrading performance. When applicable based on the communication pattern, Cluster Spec-DOALL optimizes communication by promoting and batching communication function calls such as `produce`, `specLoad` and `specStore`. Table 2 shows the number of optimized function calls and over 99% of the communication is optimized away. Small input sets are used for this result because the unoptimized versions explode execution time.

In addition, Cluster Spec-DOALL privatizes dynamically allocated memory objects if the objects are speculated to be iteration-local. Its performance impact is evaluated using `swaption` benchmark and two versions of the runtime with and without privatization on 12 cores. A small input set is used because the execution time explodes without privatization. With privatization, the volume of communication in bytes decreases by 99.6%, and the performance speedup increases from 0.1$\times$ to 3.8$\times$.

Third, **static analysis and a separate commit process reduces validation overhead.** Software transactional memory systems have suffered from large validation overhead [5]. Table 2 shows the validation overhead measured by comparing performance speedups for Spec-DOALL programs with and without validation on 120 cores. Except `lu`, there is no significant performance difference. It is because the task of validation is effectively offloaded to a separate process to overlap validation with computation in the worker pro-

Benchmark	Benchmark Suite	Loops		P'llized Loops		Speculation		Coverage of P'llized Loops	Communication Optimization			Validation Overhead
		Total	P'llizable	DOALL	Spec-DOALL	Mem	Ctrl					
									P	B	Reduction	
2mm	PolyBench	20	14	7	0	0	0	>99.99%	0	9	99.76%	NA
3mm	PolyBench	27	20	10	0	0	0	>99.99%	0	13	99.76%	NA
correlation	PolyBench	13	8	4	1	1	0	>99.99%	3	2	99.01%	3.62%
covariance	PolyBench	11	7	3	1	1	0	>99.99%	1	3	99.13%	6.87%
doitgen	PolyBench	18	14	4	1	1	0	>99.99%	1	7	99.03%	29.85%
gemm	PolyBench	13	8	3	1	1	0	>99.99%	0	6	99.81%	1.85%
gramschmidt	PolyBench	10	5	2	1	2	0	99.97%	1	1	18.10%	34.79%
jacobi-2d-imper	PolyBench	9	6	3	0	0	0	>99.99%	0	4	24.01%	NA
lu	PolyBench	8	5	1	2	5	0	99.96%	2	4	45.79%	2194.14%
ludcmp	PolyBench	12	4	1	2	7	0	99.96%	4	3	34.63%	66.49%
seidel	PolyBench	7	2	1	0	0	0	0.04%	0	1	33.26%	NA
blackscholes	PARSEC	5	2	0	1	0	1	>99.99%	1	0	99.99%	NA
swaptions	PARSEC	87	57	8	16	36	0	>99.99%	5	13	2.41%	8.24%

Table 2: Benchmark Details: Total and P'llizable show the numbers of loops in each benchmark and loops which Cluster Spec-DOALL can parallelize. DOALL and Spec-DOALL show the number of loops which Cluster Spec-DOALL acutally parallelizes. The P'llized loops number can be different from P'llizable because nested loops are not parallelized if their outer loop is parallelized. Mem and Ctrl show the number of memory flows and control flow edge speculations. Coverage shows execution time ratio of parallelized loops over the entire program. In communication optimization, P and B stand for the number of Promoted and Batched function calls, and reduction stands for the percent reduction in inter-process communication in bytes.

cesses. However, as shown in `lu`, once the validation cost exceeds a certain threshold relative to the execution time of a loop iteration, the validation process becomes a performance bottleneck. Therefore, it is still important to reduce the amount of speculative memory accesses.

6.3 The Five Slowdown Benchmarks

There are five benchmarks that experience slowdown: `lu`, `ludcmp`, `gramschmidt`, `jacobi-2d-imper`, and `seidel`. Based on quantitative bottleneck analysis, these benchmarks are divided into two classes.

For the first class, which is exemplified by `seidel`, the performance speedup is limited by Amdahl's Law. As shown in Table 2, the parallelized loop accounts for a very small fraction of the entire program execution because only an initialization loop is parallelized. The hottest loop cannot be parallelized profitably, even speculatively, because of frequent loop-carried data dependences. To parallelize `seidel`, other parallelization techniques need to be applied. In other words, DOALL and Spec-DOALL parallelization is not suitable for this kind of programs.

For the second class of benchmarks such as `gramschmidt`, `lu`, `ludcmp`, and `jacobi-2d-imper`, inter-node communication bandwidth limits performance. Figure 13 explains why these benchmarks suffer slowdown, showing their communication bandwidth after communication optimization. Since outermost loops in these benchmarks have frequent loop-carried data dependences, Cluster Spec-DOALL parallelizes their inner loops, so they are less amenable to communication optimization than the other benchmarks. As a result, they require many hundreds of megabytes per second of communication bandwidth for misspeculation checking, which is orders of magnitude greater than that of the other benchmarks. The high communication bandwidth explains why these benchmarks show performance slowdown even with oracle analysis.

6.4 Misspeculation Analysis

Figure 11 shows how different misspeculation rates affect the performance speedup of `blackscholes` on different numbers of cores. The input files are modified to cause misspeculation with varying rates from 0.01% to 0.64%. The other benchmarks are not evaluated because they do not have input-dependent misspeculation. Higher misspeculation rate and more cores generally lead to greater performance penalties. The misspeculation overhead is more sensitive to the misspeculation rate than the number of cores because additional misspeculation causes a new recovery operation while synchronization overhead from additional cores is overlapped with the existing one. Due to the high recovery overhead, the Cluster Spec-DOALL compiler should speculate dependences only with high confidence to achieve good parallel performance.

7. RELATED WORK

Automatic Parallelization System for Clusters: Intel's Cluster OpenMP [7] extends OpenMP, a parallel programming API for shared-memory multiprocessors, to clusters with distributed memory systems. Although the Cluster OpenMP compiler transforms sequential programs to parallel codes automatically, programmers are still required to specify what and how to parallelize them with programmer annotations. SUIF [1, 19] parallelizes a sequential program without any programmer annotation. However, the applicability of SUIF is limited to array-based scientific applications, and SUIF relies on programmer hints to decompose shared data across multiple nodes on a cluster [1]. In contrast, Cluster Spec-DOALL does not require any programmer annotation for shared data since the Cluster Spec-DOALL runtime handles this via copy-on-access and unified virtual address space, effectively hiding platform details.

Techniques to Overcome the Limitations of Static Compiler Analysis: Rus et al. proposes Hybrid Analysis (HA) which exploits runtime support for dependence analysis in statically indeterminate cases [18]. Although their system potentially improves the applicability of automatic parallelization, heavyweight run-time analysis can slow down program execution significantly since there is no overlap between the analysis and the execution phases. Like Cluster Spec-DOALL, STMlite [13] is a speculative parallelization system for loop parallelization, which consists of an automatic parallelizing compiler and a low-cost software transactional runtime. However, both the HA system and STMlite are implemented and evaluated on small-scale shared-memory machines with 4 and 8 cores, respectively, and their scalability with a large number of

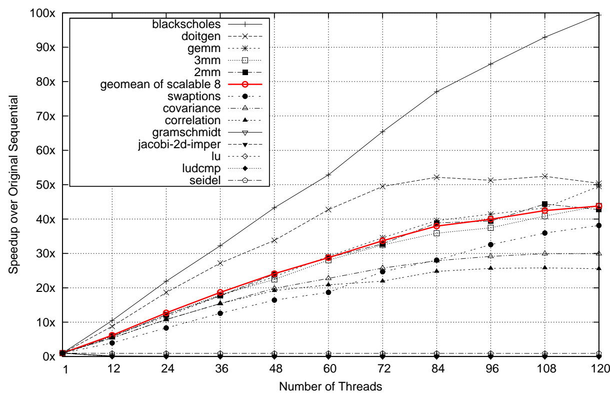

Figure 10: Overall speedup (Benchmarks in the legend are ordered from highest to lowest speedup)

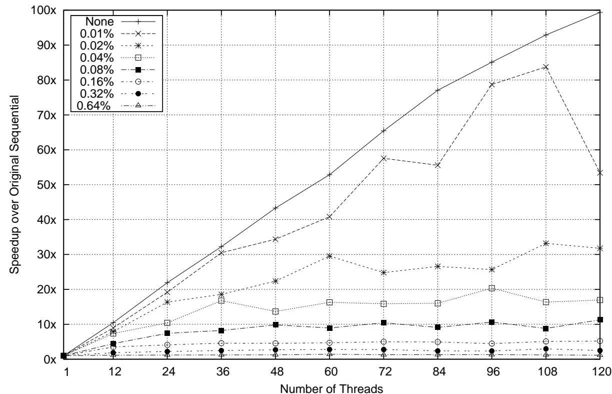

Figure 11: Speedup of `blackscholes` with varying misspeculation rates

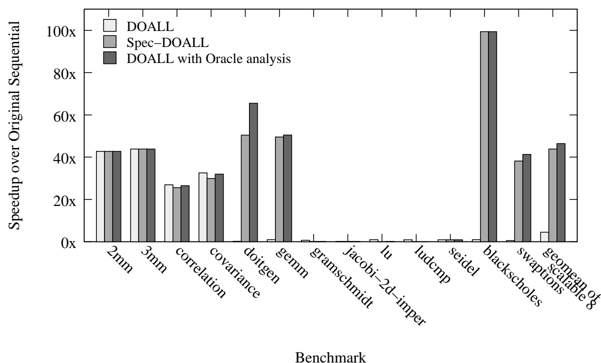

Figure 12: Performance comparison on 120 cores among DOALL-only, Spec-DOALL, and DOALL with oracle analysis

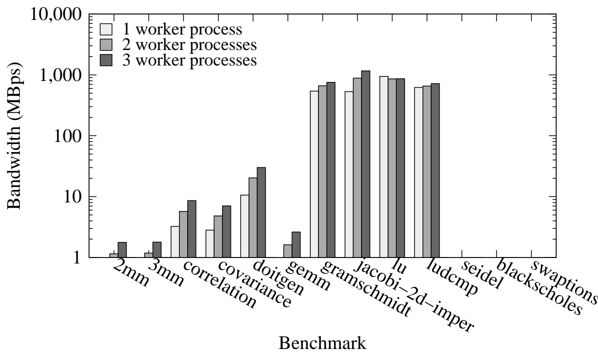

Figure 13: Communication bandwidth

cores has not been demonstrated. Cluster Spec-DOALL makes fewer assumptions on the target memory system, hence it is generally more applicable than these two systems. The POSH compiler [11] is capable of automatically parallelizing complex, general-purpose programs, but it requires TLS hardware support, hence it cannot be used on a commodity cluster.

Speculative Runtime Systems for Clusters: The required runtime support from Cluster Spec-DOALL can be easily provided by existing transactional runtime systems for clusters. The Cluster Spec-DOALL runtime is inspired by DSMTX [8] and STMlite [13] to have a separate commit process. DSMTX supports speculative pipeline parallelism as well as speculative DOALL parallelism on clusters but there is no known automatic parallelizing compiler targeting DSMTX. Distributed Multiversioning (DMV) [12] modifies a software distributed shared memory system to support transactions at the page granularity. Like the Cluster Spec-DOALL runtime, the DMV runtime handles decomposition of shared data across different nodes without programmer hints. Cluster-STM [4] is a software TM (STM) system for large-scale clusters. Cluster-STM uses special memory allocation functions such as `stm_alloc`, `stm_all_alloc`, and `stm_free`, so programmers and compilers should replace the memory allocation function calls to those provided by Cluster-STM. DiSTM [9] is a distributed STM system on Java Remote Method Invocation (RMI).

8. CONCLUSION

Cluster Spec-DOALL is the first fully automatic Spec-DOALL parallelization system for clusters. Cluster Spec-DOALL optimizes communication and validation to improve scalability on clusters. For 8 scalable benchmarks out of 13 PolyBench and PARSEC benchmarks, Cluster Spec-DOALL achieves a geomean speedup of 43.8× over the original sequential programs on a 120-core cluster, whereas DOALL-only parallelization achieves only 4.5× geomean speedup. This speedup is within 7% of the maximum oracle speedup, 46.4×.

9. ACKNOWLEDGMENTS

We thank the Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. The evaluation presented in this paper was performed on computational resources supported by the PICSciE-OIT High Performance Computing Center and Visualization Laboratory. This material is based upon work supported by the National Science Foundation under Grant CCS-0964328 and OCI-1047879, DARPA through contract FA8750-10-2-0253, and United States Air Force Contract FA8650-09-C-7918. All opinions, findings, conclusions, and recommendations expressed throughout this paper are those of the Liberty Research Group and do not necessarily reflect the views of our supporters.

10. REFERENCES

[1] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 1993.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[3] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1994.

[4] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[5] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[6] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. *Pacific Rim International Symposium on Dependable Computing, IEEE*, 0:307–313, 2009.

[7] J. P. Hoeflinger. Extending OpenMP to clusters. *White Paper Intel Corporation*, 2006.

[8] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August. Scalable speculative parallelization on commodity clusters. In *In Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*, 2010.

[9] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, 2008.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[11] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[12] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006.

[13] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[14] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE '08: Proceedings of the 1st International Workshop on Multicore Software Engineering*, 2008.

[15] L.-N. Pouchet. PolyBench: The Polyhedral Benchmark suite. http://www–roc.inria.fr/∼pouchet/software/polybench.

[16] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[17] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[18] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.*, 31:251–283, August 2003.

[19] Stanford Compiler Group. SUIF: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Stanford University, Computer Systems Laboratory, 1994.

[20] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, February 2005.

[21] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[22] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO '08: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[23] R. M. Yoo and H.-H. S. Lee. Helper transactions: Enabling thread-level speculation via a transactional memory system. In *PESPMA '08: Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, 2008.

[24] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors. In *The 5TH International Symposium on High-Performance Computer Architecture*, 1999.

[25] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA '08: Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.