Speculatively Exploiting Cross-Invocation Parallelism

Jialu Huang[†] Prakash Prabhu[†] Thomas B. Jablin[⊤] Soumyadeep Ghosh

Sotiris Apostolakis Jae W. Lee[‡] David I. August

Princeton University
{soumyade, august, sa8}@princeton.edu

[†]Google Inc.
{jialuh, prakashp}@google.com

[⊤]UIUC
jablin@illinois.edu

[‡]Sungkyunkwan University
jaewlee@skku.edu

ABSTRACT

Automatic parallelization has shown promise in producing scalable multi-threaded programs for multi-core architectures. Most existing automatic techniques parallelize independent loops and insert global synchronization between loop invocations. For programs with many loop invocations, frequent synchronization often becomes the performance bottleneck. Some techniques exploit cross-invocation parallelism to overcome this problem. Using static analysis, they partition iterations among threads to avoid cross-thread dependences. However, this approach may fail if dependence pattern information is not available at compile time. To address this limitation, this work proposes SpecCross–the first automatic parallelization technique to exploit cross-invocation parallelism using speculation. With speculation, iterations from different loop invocations can execute concurrently, and the program synchronizes only on misspeculation. This allows SpecCross to adapt to dependence patterns that only manifest on particular inputs at runtime. Evaluation on eight programs shows that SpecCross achieves a geomean speedup of 3.43× over parallel execution without cross-invocation parallelization.

Keywords

Automatic parallelization; Barrier speculation; Code optimization

1. INTRODUCTION

Scalable parallel programs are necessary to harness the performance potential of multi-core processors. Automatic parallelization techniques are a promising approach for producing well-performing parallel programs. Most existing automatic parallelization techniques exploit loop level parallelism [1, 11, 28, 36, 38, 39, 40, 42, 48]. These techniques parallelize loops and globally synchronize at the end of each loop invocation. As a result, programs with many loop invocations will experience frequent synchronizations.

For such programs, most existing automatic parallelization techniques do not deliver scalable performance because synchronization forces all threads to wait for the last thread to finish an invocation [30]. At high thread counts, threads spend more time idling at synchronization points than doing useful computation. However, there is an opportunity to improve performance: iterations from different loop invocations can often execute concurrently without violating program semantics. Instead of waiting at synchronization points, threads can execute iterations from subsequent loop invocations. This additional cross-invocation parallelism can improve the processor utilization and help the parallel programs achieve much better scalability.

Prior work has presented some automatic parallelization techniques that exploit cross-invocation parallelism [15, 33, 49, 51]. These techniques respect cross-invocation dependences without inserting coarse-grained global synchronizations such as barriers. Some of these techniques [15, 51] combine several small loops into a single larger loop. This approach side-steps the problem of exploiting cross-invocation parallelism by converting it into cross-iteration parallelism. Other approaches [33, 49] carefully partition the iteration space in each loop invocation so that cross-invocation dependences are never split between threads. However, both types of techniques rely on static analysis, which is inherently conservative. They cannot adapt to runtime dependence patterns manifested by particular inputs. In reality, many statically detected dependences only manifest under specific input conditions. For many programs, these dependences rarely manifest given the most common program inputs. Thus, programs could exploit additional cross-invocation parallelism and achieve greater scalability, if they are adapted to dependence patterns of specific inputs at runtime.

This insight motivates SpecCross, the first automatic parallelization technique designed to aggressively exploit cross-invocation parallelism using high-confidence speculation. SpecCross parallelizes independent loops and replaces the barrier synchronization between two loop invocations with its speculative counterpart. Unlike non-speculative barriers which pessimistically synchronize to enforce dependences, speculative techniques allow threads to execute past barriers without stalling. Speculation allows programs to optimistically execute potentially dependent instructions and later check for misspeculation. If misspeculation occurs, the program recovers using checkpointed non-speculative state. Speculative barriers improve performance by synchronizing only on misspeculation.

SpecCross consists of three components: a parallelizing

PACT '16, September 11-15, 2016, Haifa, Israel

DOI: http://dx.doi.org/10.1145/2967938.2967959

compiler, a profiling library and a runtime library. The parallelizing compiler automatically detects and parallelizes a code region with many loop invocations. The profiling library determines how aggressively to speculate and is the key to achieving high confidence speculation. The runtime library provides support for speculative execution, misspeculation detection, and recovery.

SpecCross uses profiling to find program regions where speculation will be beneficial. Profiling identifies access patterns conducive to barrier speculation. For instance, scanning data access pattern is highly amenable to barrier speculation. In this pattern, when two loop invocations separated by a barrier scan through memory, the last few iterations before the barrier will not interfere with the first few iterations of the loop after the barrier. By determining which regions to speculate, profiling keeps misspeculation rates low.

The SpecCross runtime system allows threads to execute speculatively without synchronization. To support recovery in case of misspeculation, non-speculative program state is checkpointed at regular intervals. Periodically, the speculative threads compute and send signatures encoding summaries of their memory accesses to a checker thread. The checker thread uses these signatures to determine if any cross-thread dependences are violated. If so, the checker thread restores the program to the most recent checkpoint. Restoring a checkpoint allows non-speculative replay.

The contributions of this paper are:

- A novel automatic parallelization technique to exploit cross-invocation parallelism using high-confidence speculation;
- Design and implementation of the first software-only speculative barrier;
- Design and implementation of the first profiler targeting speculative cross-invocation parallelization.

Evaluation over eight benchmark applications shows that SpecCross achieves a geomean speedup of 3.43× over codes without cross-invocation parallelization and 4.59× over the best sequential execution on a 24-core machine.

2. MOTIVATION

2.1 Limitations of Static Analysis-based Parallelization

Figure 1(a) shows a code example before parallelization. In this example, loop `L1` updates array elements in array `A` while loop `L2` reads elements from array `A` and uses those values to update array `B`. The whole process is repeated `STEP` times. Both `L1` and `L2` can be individually parallelized using DOALL [1]. However, dependences between `L1` and `L2` prevent the outer loop from being parallelized.

Ideally, we should only synchronize inner loop iterations that depend on each other, without stalling the execution of independent iterations. If static analysis [33, 49, 53] could prove that each thread accesses a separate section of arrays `A` and `B`, the loops can be naïvely parallelized without any synchronization between two adjacent inner loop invocations. However, since arrays `A` and `B` are accessed in an irregular manner (through index arrays `C` and `D`), static analysis cannot determine the dependence pattern between `L1` and `L2` precisely. As a result, this naïve parallelization may lead to incorrect runtime behavior.

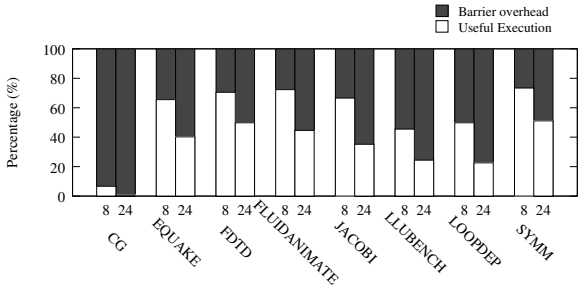

Figure 2: Overhead of barrier synchronizations for programs parallelized with 8 and 24 threads

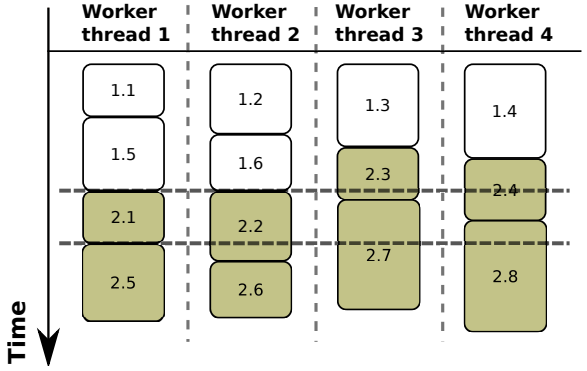

Figure 3: Execution plan for TM-style speculation: each block $A.B$ stands for the B_{th} iteration in the A_{th} loop invocation: iteration 2.1 overlaps with iterations $2.2, 2.3, 2.4, 2.7, 2.8$, thus its memory accesses need to be compared with theirs even though all these iterations come from the same loop invocation and are guaranteed to be independent.

Alternatively, if static analysis could determine a dependence pattern between iterations from two invocations, e.g., iteration `1` from `L2` always depends on iteration `2` from `L1`, then fine-grained synchronization can be used to synchronize only those iterations. But this requires accurate analysis about the dependence pattern, which is in turn, limited by the conservative nature of static analysis. Instead, barrier synchronization is used to globally synchronize all threads (Figure 1(b)). Barrier synchronization conservatively assumes dependences between any pair of iterations from two different loop invocations. All threads are forced to stall at barriers after each parallel invocation, which greatly diminishes effective parallelism. Figure 2 shows the overhead introduced by Pthreads barrier synchronizations on eight programs parallelized with 8 and 24 threads. Barrier overhead refers to the total amount of time threads sit idle waiting for the slowest thread to reach the barrier. For most of these programs, barrier overhead accounts for more than 30% of the parallel execution time and increases with the number of threads. This overhead translates into an Amdahl's Law limit of 3.33× maximum program speedup.

2.2 Speculative Cross-Invocation Parallelization

The conservativeness of barrier synchronization prevents any cross-invocation parallelization and significantly limits performance gain. Alternatively, the optimistic approach unlocks potential opportunities for cross-invocation parallelization. It allows the threads to execute across the invo-

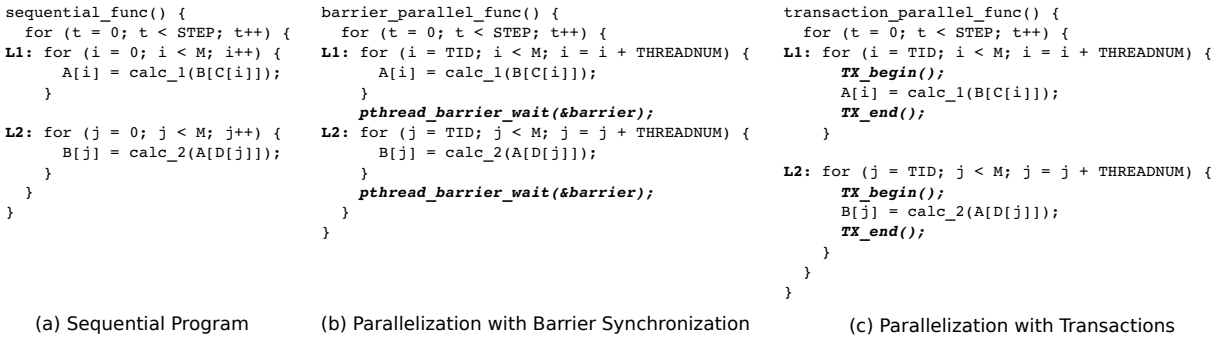

Figure 1: Example of parallelizing a program with different techniques

cation boundary under the assumption that the dependences rarely manifest at runtime. Two major optimistic solutions have been proposed in the literature: one relies on transactional memory and the other uses speculative barriers.

Figure 1(c) demonstrates the basic idea of speculative parallelization using transactional memory systems (TM) [21]. In this example, each inner loop iteration is treated as a separate transaction. The commit algorithms proposed in Grace [2] and TCC [19] allow transactions within the same inner loop invocation to commit out of order but guarantee transactions from later invocations should commit after those from earlier ones. However, this approach assumes that every transaction may conflict with another transaction and must be compared against each other for violation detection. It ignores the important fact that for most programs which could benefit from cross-invocation parallelization, all iterations from the same invocation are often guaranteed to be independent at compile time and wrapping them in separate transactions introduces unnecessary runtime checking and commit overhead. For example, in Figure 3, the execution of the first iteration of the second loop invocation (annotated as `2.1`) overlaps with that of iterations `2.2`, `2.3`, `2.4`, `2.7` and `2.8`. Even though they come from the same loop invocation, the TM framework needs to check them for access violations before `2.1` can commit. More coarse-grained transactions can reduce this checking overhead, but they also increase the possibility of misspeculation between two transactions.

Speculative barrier synchronization, on the other hand, preserves the DOALL property of each loop invocation while still allowing threads to execute past the barrier without stalling. Since all existing speculative barrier synchronization techniques [23, 26, 30] require specialized hardware support, we refer to these techniques collectively as hardware-based barrier speculation (HWBS). Compared to execution models supported by TM, HWBS distinguishes speculative and non-speculative threads to avoid unnecessary value buffering and violation checking. Non-speculative worker threads can commit their writes concurrently without waiting. As a result, HWBS is regarded as a better solution for a program pattern where iterations in the same loop invocation are independent and iterations from different invocations may depend on each other.

Despite its effectiveness, HWBS requires specialized hardware to detect misspeculation and recover. Programs parallelized for commodity hardware cannot benefit from it. This limitation motivates us to design and implement the first software-only speculative barrier for SpecCross, which aims at generating scalable parallel programs for commodity multicore machines.

3. SPECCROSS SYSTEM OVERVIEW

SpecCross enables automatic cross-invocation parallelization with software-only speculative barriers. Figure 4 gives an overview of the SpecCross system, which consists of the following three components: parallelizing compiler, runtime library, and profiler. The parallelizing compiler takes a sequential program as input and detects a SpecCross code region as the transformation target (§3.1). A SpecCross code region is often an outermost loop composed of consecutive parallelizable inner loops. The compiler parallelizes each independent inner loop and then inserts SpecCross runtime library function calls to enable speculative cross-invocation parallelization. To avoid high penalty for misspeculation, SpecCross uses a profiler to determine the *speculative range*, which controls how aggressively to speculate (§3.2). Choosing the correct limit for the speculative range keeps misspeculation rates low.

SpecCross's runtime library implements a software-only speculative barrier, which works as a light-weight substitution for HWBS on commodity hardware (§4). This runtime library provides efficient misspeculation detection (§4.1). At runtime, the original process spawns multiple worker threads and a checker thread. All worker threads execute their code sections and periodically send memory access signatures to the checker thread. These signatures approximately summarize all the speculative memory accesses. The checker thread uses these signatures to verify that speculative execution respects all memory dependences.

If the checker thread detects misspeculation, it signals a separate checkpoint process for misspeculation recovery (§4.2). The checkpoint process is periodically forked from the original process. Once recovery is completed by squashing all speculative workers, execution is resumed from the last checkpoint with non-speculative barriers.

3.1 SpecCross Parallelizing Compiler

The SpecCross parallelizing compiler automatically parallelizes the sequential program and applies software-only speculative barriers to enable cross-invocation parallelization. Non-speculative barriers guarantee that all code before a barrier executes before any code after the barrier. Speculative barriers such as the one applied by the SpecCross compiler, violate this synchronization guarantee to achieve higher processor utilization and thus, higher performance.

SpecCross targets a code region whose original parallel performance is limited by frequent global synchronizations.

Often, such a code region is an outermost loop which contains multiple parallelizable inner loops. To locate these code regions, we analyze all hot loops within the sequential program. A hot loop should account for at least 10% of the overall execution time. A hot loop is a candidate for SpecCross if it satisfies three conditions: (1) the outermost loop itself cannot be successfully parallelized by any automatic parallelization technique implemented in the existing parallelizing compiler infrastructure (including DOALL [1], LOCALWRITE [20] and DSWP [34]); (2) each inner loop can be independently parallelized by a non-speculative and non-partition based parallelization technique such as DOALL and LOCALWRITE; and (3) the sequential code between two inner loops can be privatized and duplicated among all worker threads.

After locating the candidate loops, the SpecCross transformation is applied to each of them. The transformation itself consists of two major steps. First, the compiler parallelizes the hot loop by applying DOALL or LOCALWRITE to each inner loop. Second, the compiler inserts SpecCross library function calls in the parallel program to enable barrier speculation. The function calls that comprise the SpecCross runtime library interface are described in §4.3.

3.2 SpecCross Profiler

To determine when speculation is beneficial, SpecCross uses profiling to determine the *speculative range*, which controls how aggressively to speculate (Figure 4). SpecCross assigns each thread an *epoch number* and a *task number*, which are then used to determine the speculative range and also to determine when speculative memory accesses must be checked at runtime.

An *epoch* is defined as the code region between two consecutive barriers, and the *epoch number* counts how many speculative barriers a thread has passed. Non-speculative barriers ensure that all threads have the same epoch number, whereas speculative barriers allow each thread's epoch number to increase independently. When the difference of two threads' epoch numbers is zero, misspeculation is impossible, since nothing is speculated. When the difference is high, misspeculation is more likely, since more intervening memory accesses are speculated. A *task* is the smallest unit of work that can be independently assigned to a thread, and the *task number* counts how many tasks a thread has executed since the last barrier. For many parallel programs, a task is one inner loop iteration. Note that epoch number-task number pairs are not unique across threads as each thread counts epochs and tasks independently.

In profiling mode, all barriers are non-speculative and thus, threads synchronize at every barrier. The profiler compares the memory access signature (summary of memory accesses) of every task to signatures of tasks from earlier epochs. If two tasks conflict, the distance between them is recorded. Note that this dependence distance between tasks refers to the number of intervening inner loop iterations (or *tasks* in our terminology). At the end of the profiling, the smallest dependence distance is set as the speculative range. If the minimum dependence distance is smaller than a threshold value, speculation will not be done. By default, the threshold value is set to be equal to the number of worker threads.

If the dependence distance is large, it means the program has an access pattern suitable for barrier speculation. To reduce the possibility of misspeculation, the minimum dependence distance is passed as an input parameter to the speculation runtime library (§4.3). At runtime, the leading thread stalls if it executes beyond this distance.

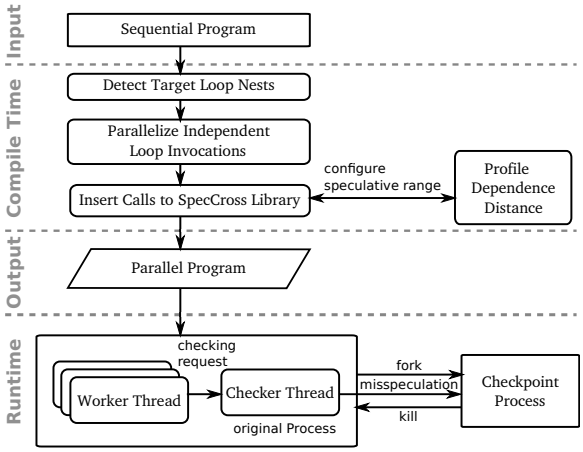

Figure 4: Overview of SpecCross: At compile time, the SpecCross compiler detects code regions composed of consecutive parallel loop invocations, parallelizes the code region and inserts SpecCross library function calls to enable barrier speculation. At runtime, the whole program is first executed speculatively without barriers. Once misspeculation occurs, the checkpoint process is woken up. It kills the original child process and spawns new worker threads. The worker threads will re-execute the misspeculated epochs with non-speculative barriers.

4. SPECCROSS RUNTIME SYSTEM

The runtime system of SpecCross implements a software-only speculative barrier. To maintain the same semantics as non-speculative barriers, speculative barriers check for runtime dependence violations. Upon detecting a violation, the runtime system triggers a misspeculation (§4.1) and rolls back to non-speculative state using the most recent checkpoint (§4.2).

4.1 Misspeculation Detection

Signature-based violation detection. SpecCross uses signature-based violation detection [9, 16, 28, 44, 52] to detect misspeculation. A signature is an approximate summary of memory accesses and provides a customizable tradeoff between signature size and false positive rates. By default, SpecCross defines a thread's signature as the *range of memory addresses speculatively accessed by that thread.* There are two advantages to using this scheme. First, it incurs low runtime overhead, as computing the memory access signature involves keeping the minimum and maximum memory address accessed, instead of logging every speculative memory access. Second, the correctness of this scheme can be proved easily: Threads are deemed independent if the signatures do not overlap.

However, adopting range-based signatures may result in an increase in the number of misspeculations detected by the SpecCross runtime system (*false positives*). To enable users to achieve the right balance between false positives and complexity of checking, SpecCross also provides an API to plug in user-provided signature generation and checking

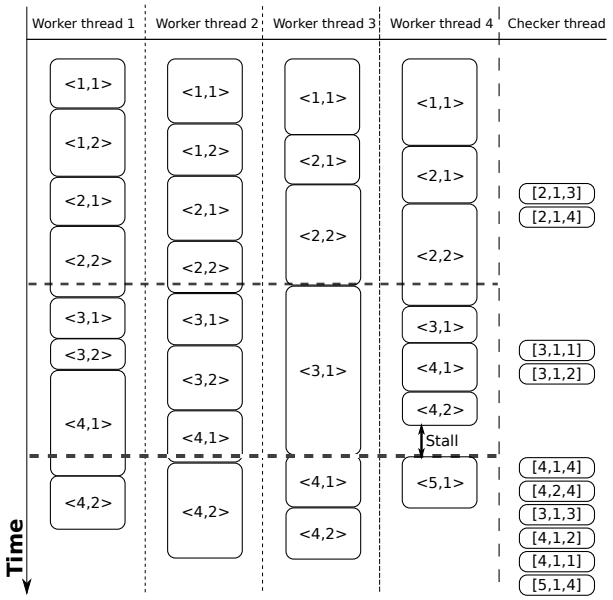

Figure 5: Timing diagram for SpecCross showing epoch and task numbers. Label $<A, B>$: The thread updates its epoch number to A and task number to B when the task starts executing. Label $[A, B, C]$ in the checker thread: Check memory access signature of task with epoch number A, task number B, and executing on worker thread C against the memory access signatures of tasks indicated in Table 1.

Checked tasks	Compared against
[2,1,3]	[1,2,1], [1,2,2]
[2,1,4]	[1,2,1], [1,2,2]
[3,1,1]	[2,2,4]
[3,1,2]	[2,2,4]
[4,1,4]	[3,2,1], [3,1,2], [3,1,3]
[4,2,4]	[3,2,2], [3,1,3]
[3,1,3]	[2,2,1], [2,2,2], [2,2,4]
[4,1,2]	[3,1,3]
[4,1,1]	[3,2,2], [3,1,3]
[5,1,4]	[4,1,1], [4,1,2], [4,1,3]
[5,2,4]	[4,2,1], [4,2,2], [4,2,3]

Table 1: Comparisons of memory access signatures performed by the checker thread for the timeline in Figure 5. Label $[A, B, C]$ refers to thread C executing task number B in epoch number A.

schemes. For instance, range-based signature schemes work well when memory accesses are clustered. For random access patterns, a Bloom filter-based signature can offer lower false positive rates. Our experiments have shown that the simple range-based checking scheme enables effective and efficient misspeculation detection; in fact, there were no false positives in any of the benchmarks evaluated in Section 5.

Violation detection scheme. To determine when the speculative barrier assumptions need to be checked, SpecCross assigns each thread an *epoch number* and a *task number* (defined in §3.2). Worker threads collaborate with the checker thread to detect violations. When a worker thread begins a new task, it updates its own epoch and task numbers and then records the current epoch and task numbers for all the other threads. Next, the worker executes the task itself. During execution, the worker computes the memory access signature for that task by maintaining the minimum and maximum values of memory addresses accessed. Once the task finishes execution, the worker thread saves its signature in a global signature log. On completion of each task, the worker thread sends the following information to the checker thread to enable asynchronous misspeculation detection: (1) the worker thread's current epoch and task numbers, (2) the memory access signature for the task, and (3) the recorded epoch and task numbers of all other threads that were running when the current task began. The checker thread compares the task's memory access signature to all signatures from epochs earlier than the current task's epoch, but at least as recent as the epoch-task number pair recorded when the task began. Signatures from the same epoch are safely ignored as they are not separated by a barrier.

If the checker thread detects misspeculation, it signals a separate checkpoint process for misspeculation recovery (explained in §4.2). By default, SpecCross checkpoints program state once every thousand epochs. Therefore, each worker thread has one thousand entries in the signature log. Worker threads synchronize at the checkpoint and signature log entries can be re-used after checkpointing. Each entry of the signature log contains a pointer to an array used for saving signatures within a single epoch. The size of the array is initialized to store 1024 signatures and is increased dynamically, if the number of tasks exceeds the array size.

Example. Figure 5 is a timing diagram for speculative barrier execution. When worker thread 3 starts task `<3,1>`, the other worker threads are still executing task `<2,2>` in the second epoch. The checker thread must check the access signatures of these three tasks against that of `[3,1,3]` (Table 1). Before task `<3,1>` finishes in worker thread 3, all other threads have already begun the fourth epoch. As a result, the checker thread must check the memory access signatures of task `<4,1>` in threads 1, 2, and 4 and task `<4,2>` in thread 4 against `[3,1,3]` (Table 1).

After sending data to the checker thread, a worker thread may stall to wait for other worker threads if continuing execution would exceed the speculative range. In Figure 5, we assume that the profiler calculates a speculative range limit of two. When thread 4 tries to start task `<5,1>`, tasks `<3,1>`, `<4,1>`, and `<4,2>` have already finished executing in thread 4, whereas thread 3 is still executing task `<3,1>`. Thus, the *dependence distance* between the two threads (defined as the difference in task numbers of the two threads) is three, one more than the limit specified by the profiler. Therefore, thread 4 stalls after executing task `<4,2>`. Note that the example is simplified as in real programs, the speculative range is always at least the number of worker threads and usually much larger.

Overhead of dependence detection. The checker thread forms the actual bottleneck for the dependence detection process. It needs to compare the signature of every task with the signatures of tasks being executed by all other threads at the time when the former task began execution. The checker runs in parallel to the worker threads and can keep up with the worker threads, especially if the number of workers is not high. However, when the worker thread count increases, the checker may not be able to compare signatures fast enough, which would lead to stalls whenever the program reaches a checkpoint. During these stalls, the worker threads wait for the checker to finish all the outstanding signature comparisons to determine if a misspeculation occurred. Once the checker detects no misspeculation, a new checkpoint is then created.

As discussed earlier, the runtime overhead for the worker threads is minimal as the range is defined only by the minimum and maximum address speculatively accessed. While there are potential false positives due to the use of range-based signatures, such false positives were not found in any of the evaluated benchmarks. Further, as the memory accesses of entire tasks are summarized by the range, using range-based signatures also keeps the overhead of checking signatures against each other minimal. The cost of checking is further mitigated by minimizing the number of signature comparisons. Checking is only necessary for two threads separated by a barrier when the two threads execute out-of-order with respect to the non-speculative execution. Thus, signatures from the same epoch are safely ignored as they are not separated by a barrier. Furthermore, if compiler analysis guarantees tasks to be independent, speculative checking is unnecessary and thus, skipped.

Memory consistency issues. There are two subtle memory consistency issues with the checking scheme described above. First, the checking scheme assumes that updates to the epoch and task numbers will be globally visible after all other stores in the previous task. If the memory consistency model allows the architecture to reorder stores, this assumption will be false. In other words, the checking methodology assumes a Total Store Order (TSO) architecture. Modern TSO architectures include: x86, x86-64, SPARC, and the IBM zSeries [27]. For architectures that do not support TSO, such as ARM and POWER, each thread should execute a memory fence before updating the epoch and task numbers. The costs of memory fences may be greater than the costs of speculative barriers when the number of tasks per epoch is high.

Second, the epoch and task numbers must update together atomically. The easiest way to accomplish this is to store these numbers as the high and low bits of a 64-bit word and use an atomic write operation. For x86-64, 64-bit writes are atomic by default, so no special handling is required.

4.2 Checkpointing and Recovery

There are three conditions that trigger misspeculation in SpecCross. First, the checker thread triggers misspeculation if it finds a pair of conflicting signatures. Second, misspeculation occurs if any of the worker threads triggers a segmentation fault. Third, misspeculation can be triggered in response to a pre-defined timeout. Timeouts are necessary, since speculative updates to the shared memory may change the exit condition of a loop and cause infinite execution.

For enabling misspeculation recovery, the worker threads periodically checkpoint their state. Checkpoints act as non-speculative barriers. All worker threads synchronize at the checkpoint, waiting for the checker thread to finish all checking requests before the checkpoint, thereby ensuring the safety of the checkpoint's state. During checkpointing, SpecCross first saves the register state of each thread using the C standard library's `setjmp` function. After the register state is saved, the process forks, duplicating the memory state of the entire process. The newly forked child sleeps until the checker thread detects misspeculation. Upon misspeculation, the child spawns new worker threads and each newly spawned thread executes `longjmp` to inherit the program state of an original thread.

As SpecCross is based on speculating barriers, checkpointing based on epochs is much easier to reason about than checkpointing based on time. Additionally, SpecCross explicitly allocates stacks of worker threads to ensure they are not deallocated by `fork`. This is necessary because the POSIX standard allows processes to deallocate the stacks of associated threads after forking. Explicitly allocating the worker threads' stacks ensures that `longjmp` will restore a valid stack.

Checkpointing Overhead. Our experiments show that recovering from misspeculation requires about one millisecond. The execution time for recovering non-speculative state is dominated by the `kill` and `clone` system calls. The checker thread invokes `kill` to asynchronously terminate misspeculating threads and awaken the checkpoint process. It then invokes `clone` (called internally by `pthread_create`) to create new worker threads. The number of syscalls scales linearly with the number of worker threads, but performance is not affected by either the program's memory footprint or the size of the speculative state.

Checkpointing Frequency. SpecCross's misspeculation recovery mechanism presents a fine tradeoff between the frequency of checkpointing and the overhead of the runtime system. Infrequent checkpointing reduces runtime overhead but increases the cost of misspeculation. SpecCross's profiling library enables very low rates of misspeculation, thus infrequent checkpointing is efficient in practice. By default, SpecCross checkpoints at every thousandth speculative barrier, though it can be re-configured based on program characteristics.

Handling Irreversible Operations. Some epochs contain irreversible operations (for example I/O) which must be executed non-speculatively. Before entering an irreversible epoch, all worker and checker threads synchronize. Just like checkpointing, synchronizing all threads ensures non-speculative execution. After exiting the irreversible region, the program checkpoints. Otherwise, later misspeculation could cause the irreversible region to be repeated.

4.3 Runtime Interface

Table 2 lists the interface functions exposed by SpecCross, along with a description of each function's semantics. Figure 6 shows an instantiation of these functions in a parallel program. In the example, each inner loop invocation is treated as an epoch, and each inner loop iteration as a task. SpecCross provides the same interface functions for both profiling and speculation purposes. As a result, the compiler must only insert the function calls once, for both profiling and speculative execution. Whether to profile or speculate is decided by defining the environment variable `MODE`. The `MODE` value can also be set to `NON-SPECULATIVE`. In non-speculative mode, most interface functions do nothing and speculative barriers are replaced with non-speculative ones. This non-speculative mode is enabled automatically when re-executing the misspeculated epochs after recovering from misspeculation. The following paragraphs describe how the compiler automatically inserts some of these function calls into the program.

A code region between two consecutive `enter_barrier` functions is considered as an epoch. For simplicity, each inner loop invocation is usually treated as an epoch. As a result, the SpecCross compiler inserts `enter_barrier` function calls at the beginning of each inner loop's preheader basic block.

Operation	Description
Functions for Both Profiling and Speculation	
init()	Initialize data structures used for barrier profiling or speculative execution. If in speculation mode, checkpoint the program before beginning the parallel execution.
exit_task(threadID)	Record the signature of the current task in global signature log. Increment the *task number*. If in profiling mode, compare the signature of current task with signatures of tasks belonging to previous epochs and return the minimum dependence distance. If in speculation mode, return value is 0.
spec_access(threadID, callback, addr_list, ...)	Apply callback function to each address in the addr_list to compute the signature.
enter_barrier(threadID, loop_name)	Increment the *epoch number*. If in profiling mode, execute the actual non-speculative barrier operation. If in speculation mode, checkpoint according to the checkpointing frequency.
create_threads(threads, attrs, start_routines, args)	Create worker threads and if in speculation mode, create a checker thread for violation detection.
cleanup()	Wait for worker threads and checker thread to finish. Free data structures allocated for profiling or speculation.
Functions for Speculation Only	
enter_task(threadID, spec_distance)	Collect *epoch number* and *task number* of other worker threads and send them to the checker thread. The parameter spec_distance specifies the speculation distance between two tasks.
send_end_token(threadID)	Send an END_TOKEN to checker thread to inform it of the completion of a worker thread.
sync()	Synchronize all threads before entering the next epoch.
checkpoint()	Checkpoint program state before entering the next epoch.

Table 2: Interface for SpecCross runtime library

Main thread:

```
main() {
  init();
  create_threads(threads, attrs, SpecCross_parallel_func, args);
  cleanup();
}
```

Worker thread:

```
SpecCross_parallel_func(threadID) {
  for (t = 0; t < STEP; t++) {
    enter_barrier(threadID, "L1");
L1: for (i = threadID; i < M; i = i + THREADNUM) {
      enter_task(threadID, minimum_distance_L1);
      spec_access(threadID, callback, &A[i], &B[C[i]], (char*)0);
      A[i] = do_work(B[C[i]]);
      exit_task(threadID);
    }
    enter_barrier(threadID, "L2");
L2: for (j = threadID; j < M; j = j + THREADNUM) {
      enter_task(threadID, minimum_distance_L2);
      spec_access(threadID, callback, &B[j], &A[D[j]], (char*)0);
      B[j] = do_work(A[D[j]]);
      exit_task(threadID);
    }
  }
  send_end_token(threadID);
}
```

Figure 6: Demonstration of using SpecCross runtime library in a parallel program

Function `enter_task` marks the beginning of each task. Since an inner loop iteration is usually treated as a separate task, the SpecCross compiler inserts `enter_task` function calls at the beginning of each inner loop's header.

Function `exit_task` marks the end of a task, i.e., it is invoked either before exiting an inner loop invocation or before entering another inner loop iteration. To locate these insertion points, the SpecCross compiler checks the terminator instruction of each basic block in an inner loop. If a terminator instruction is an unconditional branch which either exits the loop or branches back to the header of the loop, the compiler inserts an `exit_task` function call right before the terminator instruction. If the terminator instruction is a conditional branch and (1) if one of its targets is a basic block outside the loop and the other is the loop header, the compiler also inserts a call to `exit_task` before the terminator instruction; or (2) if one target is a basic block outside the loop and the other is a basic block within the loop except the loop header, the `exit_task` function is invoked only when the execution exits the loop; or (3) if one target is the header of the loop and the other is some other basic block within the loop, an `exit_task` function is invoked only when the execution branches back to the loop header.

Function `spec_access` is used to calculate the access signature for each task. The compiler inserts calls to this function before each memory operation (store or load) that is involved in a cross-invocation dependence.

Barriers not necessary	Existing techniques not profitable
blackscholes, swaptions (Parsec [4])	mesa, art and ammp (SpecFP [45])
pi, mandelbrot (OMPBench [14])	ferret (Parsec [4])
gemm, doitgen, 2mm (PolyBench [37])	mg (NAS [31])

Table 3: Benchmark programs not suitable for evaluation

5. EVALUATION

The implementation of SpecCross parallelization is evaluated on a 24-core shared memory machine. It has four Intel 6-core Xeon X7460 processors running at 2.66 GHz with 24 GB of memory. Its operating system is 64-bit Ubuntu 9.10. All benchmark programs are compiled using clang compiler version 3.0 with -O3.

5.1 Experimental Setup

Twenty programs from six benchmark suites that are parallelizable by SpecCross were examined as potential candidates for evaluation, and eight were finally selected. We have taken programs from multiple benchmark suites to demon-

Benchmark	Suite Source	Coverage of Exec. Time	# of tasks	# of epochs	# of checking requests	Minimum Distance	
						train	ref
CG	NAS [31]	12.8	63000	7000	40609	*	*
EQUAKE	SpecFP [45]	98	66000	3000	55181	*	*
FDTD	PolyBench [37]	99	200600	1200	96180	599	799
FLUIDANIMATE	Parsec [4]	99	1379510	1488	295000	54 / *	54 / *
JACOBI	PolyBench [37]	99	99400	1000	67163	497	997
LLUBENCH	LLVMBench [24]	90.5	110000	2000	81965	*	*
LOOPDEP	OMPBench [14]	99	245000	1000	98251	500	800
SYMM	PolyBench [37]	99	500500	2000	369731	*	*

Table 4: Details of benchmark programs. * indicates no access conflicts are detected in profiling.

strate the effectiveness of SpecCross over various application types, which work best with different optimization techniques. Of the twenty candidates, seven programs have a DOALL-able outermost loop and hence do not require barriers. For five programs, existing techniques are not profitable for either outer loop or inner loop parallelization. Table 3 summarizes these rejected programs. The remaining eight programs are chosen because they share two characteristics: their performance dominating loop nests cannot be profitably parallelized by non-speculative automatic parallelization techniques such as DOALL [1], LOCALWRITE [20] or DSWP [34]. Meanwhile, although these loop nests contain parallelizable inner loops, parallelizing them introduces frequent barrier synchronizations limiting overall scalability. These two characteristics are required for SpecCross to have a potential benefit.

Table 4 characterizes these eight programs. We compared two parallel versions of these programs: (a) inner loop parallelization with non-speculative pthread barriers [6]; and (b) outer loop parallelization with SpecCross. The non-speculative technique imposes synchronization on every barrier assuming that there are dependences between any pair of iterations from two different outer loop invocations. Even with a non-pthread barrier method, the performance would not have differed much. The common trait of any such method is that no cross-invocation parallelism is exploited and barrier synchronization is enforced. For performance measurements, the best sequential execution of the parallelized loops is considered the common baseline for both cases.

For most of these programs, the parallelized loops account for more than 90% of the execution time. When parallelizing using SpecCross, each loop iteration is regarded as a separate task. The range-based signature described in §4.1 is used to track the range of memory locations (or array indices) accessed by each task. Each parallel program is first instrumented using the profiling functions provided by SpecCross. The profiling step finds a minimum dependence distance value for use in speculative barrier execution. All benchmark programs have multiple input sets. We chose the training input set for profiling run. Table 4 shows the minimum dependence distance results for the evaluated programs using two different input sets (a training input set for profiling run and another reference input set for performance run). Four of the eight programs had runtime dependences detected by profiling functions while the rest do not. The minimum dependence distance between two inner loops in program FLUIDANIMATE varies a lot. Some of the loops do not cause any runtime access conflicts while others have a very small minimum dependence distance. For the latter case, SpecCross basically serves as a non-speculative barrier. Regarding program JACOBI, the profiler underestimated the speculative range (497 for training input vs. 997 for reference input), resulting in conservative speculation. This reflects in the performance of the benchmark, as shown in Figure 7(e).

5.2 Performance

Figure 7 compares the speedups achieved by the parallelized loops using pthread barriers and SpecCross. It demonstrates the benefits of enabling cross-invocation parallelization. The best sequential execution time of the parallelized loops is considered as the common baseline for both SpecCross and pthread barriers. The original execution with pthread barriers does not scale well beyond a small number of cores. Among the eight programs in barriers, CG performs the worst since each of its epochs only contains nine iterations. With higher thread counts, the overhead caused by barriers increases without any gains in parallelism.

The speculative barrier solution provided by SpecCross enables all programs to scale to higher thread counts when compared to an equivalent execution with pthread barriers. At lower thread counts, pthread barrier implementation for some programs yields better performance than SpecCross. This happens for two reasons: (a) SpecCross requires an extra thread for violation detection. At lower thread counts, one fewer thread is available to do actual work, which is significant when the total number of threads is small; (b) the overhead of barrier synchronization increases with increasing thread counts. As a result, the effectiveness of SpecCross is more pronounced at higher thread counts.

5.3 Overhead Analysis

In our evaluation, the program state is checkpointed once every 1000 epochs. For the eight programs evaluated, profiling results are accurate enough to result in high-confidence speculation and no misspeculation is recorded at runtime. As a result, the operations that contribute to major runtime overheads include computing access signatures, sending checking requests, detecting dependence violation, and checkpointing.

Table 4 shows for each program, the number of tasks executed, the number of epochs, and the number of checking requests for execution with 24 threads. The performance results (Figure 7) indicate that with higher thread counts, the checker thread may become the bottleneck. In particular, the performance of SpecCross scales up to 18 threads and either flattens or decreases after that. The effects of checker thread in limiting performance can be illustrated by considering the example of LLUBENCH. The number of check-

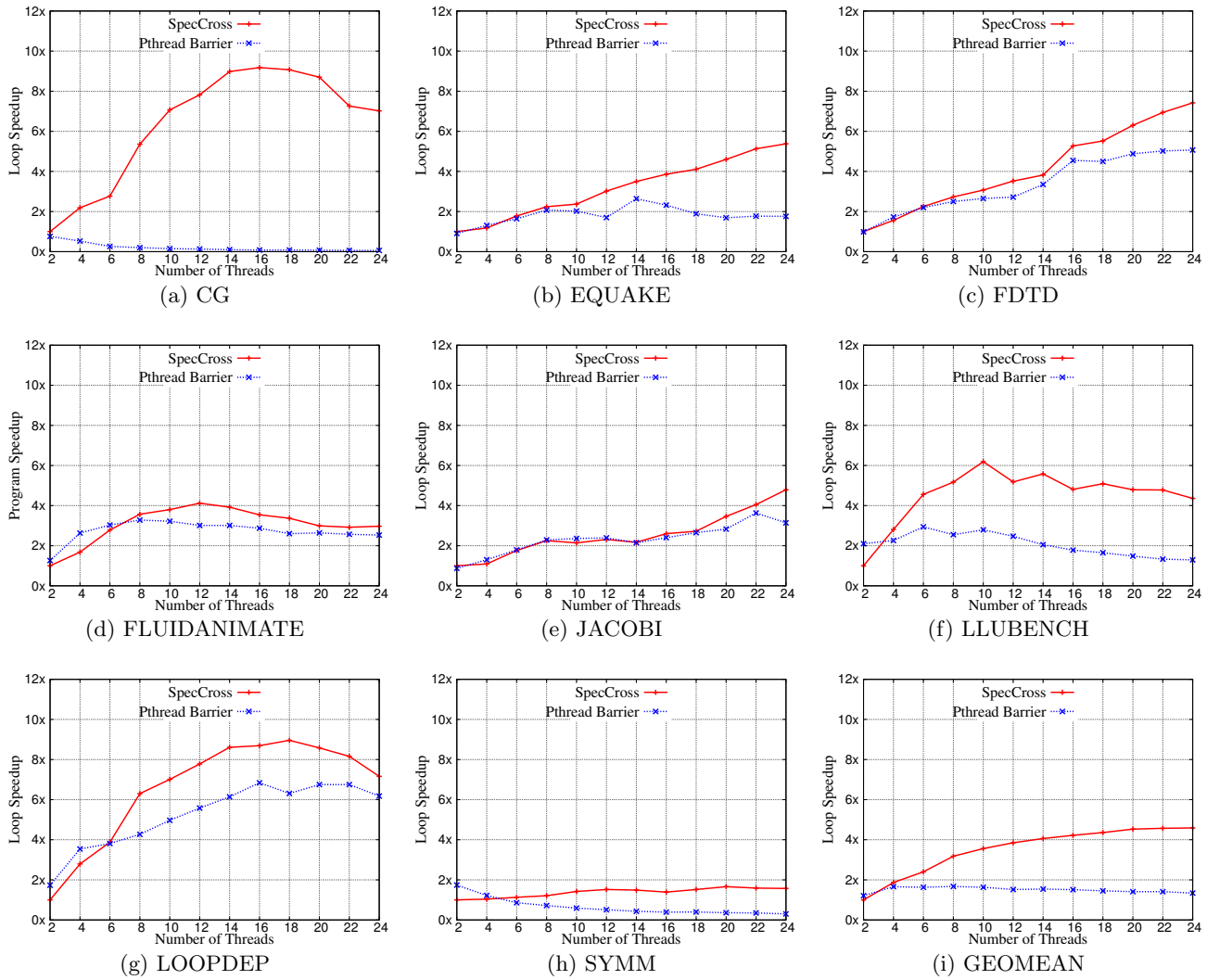

Figure 7: Performance comparison between code parallelized with or without cross-invocation parallelization.

ing requests for LLUBENCH increases by 3.3× when going from 8 threads to 24 threads, with the resulting performance improvements being minimal. Parallelizing dependence violation detection in the checker thread is one option to solve this problem and is part of future work.

Checkpointing is much more expensive than signature calculation or checking operations and hence is done infrequently. For benchmark programs evaluated, there are fewer than 10 checkpoints, since SpecCross by default checkpoints every 1000 epochs. However, frequency of checkpointing can be reconfigured depending on desired performance characteristics. To quantify the impact of checkpointing on performance, Figure 8 shows the geomean speedup results of increasing the number of checkpoints from 2 to 100, for all of the eight benchmark programs. To evaluate the overhead of the whole recovery process, we randomly triggered a misspeculation during the speculative parallel execution. Evaluation results are shown in Figure 8. As can be seen, more checkpoints increases the overhead at runtime, however also reduce the time spent in re-execution once misspeculation happens. Finding an optimal configuration for them is important and will be part of the future work.

The time required by worker threads to compute memory access signatures is another source of overhead for the SpecCross system. As mentioned in Section 4.1, the range-based memory access signatures incur a low overhead as it does not need to log every speculative memory access. This is borne out by the 1.04% geomean overhead (for 24 threads) per task in computing these access signatures. The highest overhead was for `CG`, where computing signatures resulted in an average overhead of 6.7% per task. This was because of a relatively large number of speculative memory accesses in every task. On the other hand, for `LLUBENCH` and `EQUAKE`, the number of range computations could be minimized due to compiler analysis, resulting in overheads as low as 0.26% and 0.13% per task.

5.4 Comparison of SpecCross and Previous Work

Figure 9 compares speedups achieved by SpecCross and previous work using the same programs [4, 7, 14, 17, 22]. All comparisons were done on our evaluation platform. We have omitted EQUAKE from Figure 9 as it was parallelized by Helix [7] whose source code is not accessible. For all other programs, SpecCross achieves better performance.

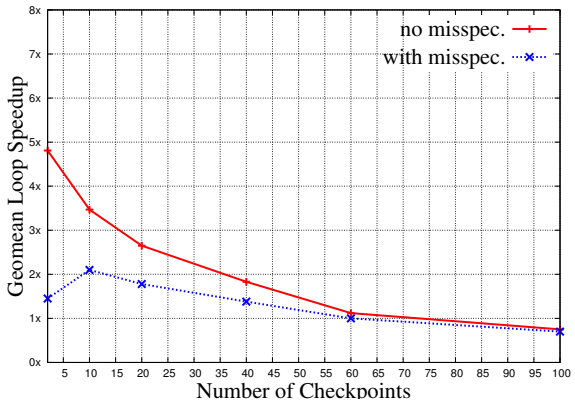

Figure 8: Loop speedup with and without misspeculation for execution with 24 threads: the number of checkpoints varies from 2 to 100. A misspeculation is randomly triggered during the speculative execution. With more checkpoints, overhead in checkpointing increases; however, overhead in re-execution after misspeculation decreases.

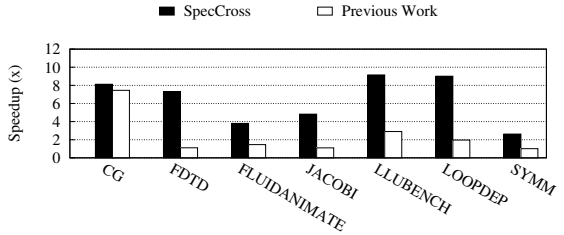

Figure 9: Best performance comparison between SpecCross and previous works on the same platform.

Programs `JACOBI`, `FDTD` and `SYMM` were originally designed for Polyhedral optimizations [17]. They can be automatically parallelized by the Polly optimizer in LLVM compiler infrastructure. Polly uses an abstract mathematical representation to analyze the memory access pattern of a program and automatically exploits thread-level and SIMD parallelism. Polly successfully achieves promising speedup for 16 out of 30 Polyhedral benchmark programs. However, it fails to extract enough parallelism from `JACOBI`, `FDTD` or `SYMM` due to the irregular memory access patterns of the outermost loop. SpecCross applies DOALL to the inner loops and exploits cross-invocation parallelism using speculative barriers, and therefore achieves much better performance.

`CG` was manually parallelized by DSWP+ technique [22]. DSWP+ performs as a manual equivalence of SpecCross parallelization. As shown in the graph, SpecCross is able to achieve close performance gain as the manual parallelization.

Both `LOOPDEP` and `FLUIDANMIATE` have a parallel implementation in their benchmark suites. We compare the best performance results of their parallel versions with the best performance achieved by SpecCross. Helix automatically parallelized `EQUAKE` and its performance gain was presented in [7]. We compare the best performance reported with that achieved by SpecCross. These previous works parallelize the three programs in a similar way: each inner loop within the outermost loop is independently parallelized by DOALL or DOACROSS and non-speculative barrier synchronizations are inserted between inner loops to respect cross-invocation dependences. Those synchronizations limit the scalability of the performance.

We cannot find any existing parallelization result for `LLUBENCH`, so in Figure 9, best performance result of SpecCross is compared against the best performance achieved by inner loop DOALL with pthread barrier synchronizations.

6. RELATED WORK

Cross-invocation Parallelization: Existing automatic cross-invocation parallelization techniques all rely on static analysis. Zhao et al. [54] transform an inner DOALL loop to an outer DOALL loop before parallelization in order to avoid barriers between two consecutive parallel loop invocations. Ferrero et al. [15] aggregate small parallel loops into large ones for the same purpose. Tseng et al. [49] apply flow analysis to carefully partition the iteration space so that cross-invocation dependences do not flow across threads. Compared to these techniques, SpecCross is not limited by the conservativeness of static analysis, and therefore can adapt to irregular dependence patterns manifested by particular inputs.

Comparison to Other Dynamic Parallelization Systems: Tian et al. [48] proposed Copy-Or-Discard (CoD) execution model for speculative parallelization, which achieves excellent speedup on six benchmarks. CorD's execution model focuses on parallelizing loops, where different iterations of the same loop can be speculatively executed in parallel. By contrast, SpecCross aims to exploit cross-invocation parallelism, where tasks from different loop invocations may be speculatively executed in parallel. The other contrast lies in the way the two techniques detect conflicts: CorD uses memory versioning while SpecCross uses range-based memory access signatures.

BOP [13] offers a promising method for speculative parallelization by allowing specification of potential for parallel execution of code regions. The specification can be made either via programmer-inserted annotations or by certain profilers. If speculation does not prove profitable, execution time is the same as the sequential execution time. In BOP's execution model, each speculative region is run in a separate process; thus, it incurs extra overhead due to merging of modified pages by different processes (this cost may be partly hidden by copy-on-write mechanisms). By contrast, SpecCross speculates barriers which leads to tasks executing in different parallel threads. As a result, speculative accesses once checked, do not require separate copying or merging. Additionally, compared to BOP's page-based monitoring, SpecCross monitors only inter-invocation dependences, which may lead to lower checking overheads.

Alternative Synchronizations Fuzzy Barriers [18] specifies a synchronization range rather than a specific synchronization point. However, it also relies on static analysis to identify instructions that can be safely executed while a thread waits for other threads to reach the barrier. Fuzzy barriers cannot reduce the number of barriers; it can only reschedule barriers for more efficient execution.

Some techniques apply speculative barriers between loop invocations to enable cross-invocation parallelization. Nagarajan and Gupta [30] speculatively execute parallel programs past barriers and detect conflicts by re-designing the Itanium processor's *Advanced Load Address Table* (ALAT) hardware. Martínez and Torrellas [26] apply *thread level speculation* [32, 46] to speculatively remove barriers. A speculative synchronization unit (SSU) is designed to detect conflict and recover from misspeculation at runtime.

ECMon [29] exposes cache events to software so that barrier speculation could efficiently detect violations at runtime. These techniques are not limited by complicated dependence patterns; however, they all require special hardware support. By contrast, SpecCross implements a software-based technique which can run on commodity multicore machines.

Ziarek et al. [55] propose language extensions for speculative barriers and formulate safety properties for correct speculation. However, they only apply it to a toy language and provide no concrete implementation or performance results. SpecCross provides the design, implementation and evaluation of a real system that automatically parallelizes programs using speculative barriers.

Transactional Memory Supported Cross-invocation Parallelization: Transactional Memory (TM) [21, 43] was first introduced as a technique providing a way to implement lock-free data structures and operations. The key idea behind TM is to make a sequence of memory reads and writes appear as a single transaction; all intermediate steps are hidden from the rest of the program. A log of all memory accesses is kept during the transaction. If a transaction reads memory that has been altered between the start and end of the transaction, execution of the transaction is restarted. This continues until the transaction is able to complete successfully, and its changes are committed to memory.

Original TM systems do not support cross-invocation speculation since they do not enforce the commit order between transactions from different loop invocations. Grace [2] and TCC [19] extend existing TM systems to support cross-invocation speculation. Grace automatically wraps code between fork and join points into transactions, removing barrier synchronizations at the join points. Each transaction commits according to its order in the sequential version of code. TCC [19] relies on programmers to wrap concurrent tasks into transactions. It requires special hardware support for violation checking and transaction numbering, which ensures the correct commit order of each transaction. As discussed in Section 2, both systems ignore the important fact that a large number of independent tasks do not need to be checked for access violations. Instead, SpecCross is customized for this program pattern, thus it avoids unnecessary overhead in checking and committing. Besides, neither Grace or TCC is an automatic parallelization technique. They are either applied to already parallelized programs or manually inserted as annotations in sequential programs. Instead, SpecCross is an automatic parallelization technique which does not require any manual work.

Load Balancing Techniques: Work stealing techniques, implemented in parallel subsystems like Cilk [5], Intel TBB [41], and X10 [10], balance load amongst parallel threads by allowing one thread to steal work from another thread's work queue. Balancing workloads in turn reduces the impact of barrier synchronization on program performance. However, existing work stealing implementations only allow workers to steal work from the same epoch at any given time. This is because these implementations do not leverage knowledge about program dependences to steal work from multiple epochs (across barriers) at the same time. As a result, programs that have a limited number of tasks in a single epoch (for example, CG in our evaluation) do not benefit from work stealing techniques. In contrast, SpecCross allows tasks from different epochs to overlap and achieve better load balance across epochs. Other load balancing techniques such as guided self-scheduling [35], affinity-based scheduling [25], and trapezoidal scheduling [50] also suffer from the same limitations as the work stealing techniques.

Synchronization via scheduling [3] is a method of load balancing that employs static and dynamic analyses to capture runtime dependences between tasks in a task graph. The task graph is exposed to a scheduler that schedules the tasks onto threads in a way so as to minimize the idling time of each thread. While being able to handle more general dependence patterns than working stealing, this technique still does not overlap tasks from successive executions of the same task graph in parallel.

Multithreaded Program Checkpointing: Several prior techniques implement multithreaded program checkpointing. Dieter et al. [12] propose a user-level checkpointing system for multi-threaded applications. Carothers et al. [8] implement a system call to transparently checkpoint multi-threaded applications. SpecCross checkpoints the multi-threaded programs for misspeculation recovery. These checkpointing techniques could be merged into the SpecCross framework.

Dependence Distance Analysis: If dependence distance manifests in a regular manner, programs can still be parallelized by assigning dependent tasks to the same worker thread. Dependence distance analysis [47] has been proposed to serve that purpose. Like other static analyses these techniques tend to be conservative and cannot handle irregular dependence patterns. SpecCross takes advantage of profiling information to get a minimum distance and speculates it holds for other input sets to further reduce the misspeculation rate. Since this information comes from profiling, it applies to programs with irregular dependence patterns as well.

7. CONCLUSION

This paper presented SpecCross, the first automatic parallelization technique to aggressively exploit cross-invocation parallelism using high-confidence speculation. Unlike prior techniques which are all limited by conservative static analysis, SpecCross can adapt to irregular dependence patterns determined by input data sets. The SpecCross system consists of three major components: a parallelizing compiler, a runtime library and a profiler. The profiler enables high-confidence speculation, and the runtime library implements a software-only speculative barrier. The parallelizing compiler runs the profiler and automatically transforms a sequential program into a scalable parallel program targeting the runtime library. Evaluation on eight programs demonstrates that SpecCross achieves a geomean speedup of 4.59× over the best sequential execution on a 24-core machine. This demonstrates the effectiveness of SpecCross in realizing well-performing cross-invocation parallelization.

8. ACKNOWLEDGMENTS

We thank the Liberty Research Group for their support and feedback during this work. We also thank Xipeng Shen and the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Science Foundation (NSF) through Grants OCI-1047879, CCF-1439085, and CNS-0964328. All opinions, findings, conclusions, and recommendations expressed in this paper are those of the Liberty Research Group and do not necessarily reflect the views of the NSF. This work was carried out when the authors were working at Princeton University.

9. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[2] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA*, 2009.

[3] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *PLDI*, 2011.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT*, 2008.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP*, 1995.

[6] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[7] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. Helix: automatic parallelization of irregular programs for chip multiprocessing. In *CGO*, 2012.

[8] C. D. Carothers and B. K. Szymanski. Checkpointing multithreaded programs. *Dr. Dobbs*, August 2002.

[9] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, 2006.

[10] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, 2008.

[11] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *ICPP*, 1986.

[12] W. R. Dieter and J. E. Lumpp Jr. User-level checkpointing for linuxthreads programs. In *USENIX, FREENIX Track*, 2001.

[13] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI*, 2007.

[14] A. J. Dorta, C. Rodriguez, F. de Sande, and A. Gonzalez-Escribano. The OpenMP source code repository. In *PDP*, 2005.

[15] R. Ferrer, A. Duran, X. Martorell, and E. Ayguadé. Unrolling loops containing task parallelism. In *LCPC*, 2009.

[16] S. Ghosh, Y. Park, and A. Raman. Enabling efficient alias speculation. In *LCTES '15*.

[17] T. C. Grosser. Enabling polyhedral optimizations in llvm. Diploma thesis, Department of Informatics and Mathematics, University of Passau, Germany, April 2011.

[18] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS*, 1989.

[19] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[20] H. Han and C.-W. Tseng. Improving compiler and run-time support for irregular reductions using local writes. In *LCPC*, 1999.

[21] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[22] J. Huang, A. Raman, Y. Zhang, T. B. Jablin, T.-H. Hung, and D. I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *CGO*, 2010.

[23] K. Z. Ibrahim and G. T. Byrd. On the exploitation of value predication and producer identification to reduce barrier synchronization time. In *IPDPS*, 2001.

[24] LLVM Test Suite Guide. http://llvm.org/docs/TestingGuide.html.

[25] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *SC*, 1992.

[26] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS*, 2002.

[27] P. E. Mckenney. Memory barriers: a hardware view for software hackers, 2009.

[28] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI*, 2009.

[29] V. Nagarajan and R. Gupta. ECMon: exposing cache events for monitoring. In *ISCA*, 2009.

[30] V. Nagarajan and R. Gupta. Speculative optimizations for parallel programs on multicores. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, 2010.

[31] NAS Parallel Benchmarks 3. http://www.nas.nasa.gov/Resources/Software/npb.html.

[32] C. E. Oancea and A. Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE*, 2008.

[33] M. F. P. O'Boyle, L. Kervella, and F. Bodin. Synchronization minimization in a SPMD execution model. *J. Parallel Distrib. Comput.*, 29, September 1995.

[34] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO*, 2005.

[35] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12), December 1987.

[36] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *SC*, 1993.

[37] L.-N. Pouchet. PolyBench: the Polyhedral Benchmark suite. http://www-roc.inria.fr/ pouchet/software/polybench/download.

[38] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *PLDI*, 2011.

[39] L. Rauchwerger, N. M. Amato, and D. A. Padua. A scalable method for run-time loop parallelization.

International Journal of Parallel Programming (IJPP), 26:537–576, 1995.

[40] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE TPDS*, 1999.

[41] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *IPDPS*, 2008.

[42] J. Saltz, R. Mirchandaney, and R. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40, 1991.

[43] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.

[44] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA*, 2008.

[45] Standard Performance Evaluation Corporation. http://www.spec.org.

[46] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 2005.

[47] P. Swamy and C. Vipin. Minimum dependence distance tiling of nested loops with non-uniform dependences. In *IPDPS*, 1994.

[48] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO*, 2008.

[49] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *PPoPP*, 1995.

[50] T. Tzen and L. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1), January 1993.

[51] M. J. Wolfe. *Optimizing Compilers for Supercomputers*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, October 1982.

[52] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: decoupling hardware transactional memory from caches. In *HPCA*, 2007.

[53] N. Yonezawa, K. Wada, and T. Aida. Barrier elimination based on access dependency analysis for openmp. In *Parallel and Distributed Processing and Applications*. Springer Berlin / Heidelberg, 2006.

[54] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar. Reducing task creation and termination overhead in explicitly parallel programs. In *PACT*, 2010.

[55] L. Ziarek, S. Jagannathan, M. Fluet, and U. A. Acar. Speculative n-way barriers. In *DAMP*, 2008.