Automatically Exploiting Cross-Invocation Parallelism Using Runtime Information

Jialu Huang Thomas B. Jablin Stephen R. Beard Nick P. Johnson David I. August

Princeton University, Princeton, NJ
{jialuh, tjablin, sbeard, npjohnso, august}@princeton.edu

Abstract

Automatic parallelization is a promising approach to producing scalable multi-threaded programs for multicore architectures. Many existing automatic techniques only parallelize iterations within a loop invocation and synchronize threads at the end of each loop invocation. When parallel code contains many loop invocations, synchronization can easily become a performance bottleneck. Some automatic techniques address this problem by exploiting cross-invocation parallelism. These techniques use static analysis to partition iterations among threads to avoid cross-thread dependences. However, this partitioning is not always achievable at compile-time, because program input determines dependence patterns at run-time. By contrast, this paper proposes DOMORE, the first automatic parallelization technique that uses runtime information to exploit additional cross-invocation parallelism. Instead of partitioning iterations statically, DOMORE dynamically detects cross-thread dependences and synchronizes only when necessary. DOMORE consists of a compiler and a runtime library. At compile time, DOMORE automatically parallelizes loops and inserts a custom runtime engine into programs. At runtime, the engine observes dependences and synchronizes iterations only when necessary. For six programs, DOMORE achieves a geomean loop speedup of 2.1× over parallel execution without cross-invocation parallelization and of 3.2× over sequential execution on eight cores.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors—Code Generation, Compilers, Optimization

General Terms Performance, Design, Experimentation

CGO'13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

1. Introduction

Harnessing the performance potential of multicore processors requires scalable parallel programs. Automatic parallelization techniques are a promising approach for producing well-performing parallel programs. Most existing parallelization techniques exploit loop level parallelism [1, 6, 19, 27, 28, 30–32]. They parallelize loops and globally synchronize at the end of each loop invocation. Consequently, programs with many loop invocations will synchronize frequently.

These parallelization techniques fail to deliver scalable performance because synchronization forces all threads to wait for the last thread to finish an invocation [21]. At high thread counts, threads spend more time idling at synchronization points than doing useful computation. There is an opportunity to improve the performance by exploiting additional parallelism. Often, iterations from different loop invocations can execute concurrently without violating program semantics. Instead of waiting, threads begin iterations from subsequent invocations.

A few automatic parallelization techniques exploit cross-invocation parallelism [9, 25, 35, 37]. Cross-invocation parallelization requires techniques for respecting cross-invocation dependences without resorting to coarse-grained barrier synchronization. Some techniques [9, 37] respect dependences by combining 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 [25, 35] carefully partition the iteration space in each loop invocation so that cross-invocation dependences are never split between threads. However, both techniques rely on static analyses. Consequently, they cannot adapt to the dependence patterns manifested by particular inputs at runtime. Many statically detected dependences may only manifest under certain input conditions. For many programs, these dependences rarely manifest given the most common program inputs. By adapting to the dependence patterns of specific inputs at runtime, programs can exploit additional cross-invocation parallelism and achieve greater scalability.

This work presents DOMORE, the first automatic parallelization technique to capture dynamic cross-invocation parallelism. Unlike existing techniques, DOMORE gathers cross-invocation dependence information at runtime. Even for programs with irregular dependence patterns, DOMORE precisely synchronizes iterations which depend on each other and allows iterations without dependences to execute concurrently. As a result, DOMORE is able to enable more cross-invocation parallelization and achieves more scalable performance.

DOMORE first identifies the code region containing the targeted loop invocations, and then transforms the program by dividing the region into a scheduler thread and several worker threads. At runtime, the scheduler thread detects which iterations access common memory locations, and forwards synchronization conditions to the worker thread using lock-free queues. Each worker thread executes an iteration when its synchronization condition is satisfied. Consequently, only threads waiting on the synchronization conditions must stall, and iterations from consecutive loop invocations may execute in parallel.

The automatic compiler implementation in LLVM [16] provides significant performance gains over both sequential code and parallel code with barriers. Evaluation on six benchmark programs shows a loop speedup of 2.1× over codes without cross-invocation parallelization and 3.2× over the original sequential performance on eight cores.

2. Motivation and Overview

To motivate the DOMORE technique, we present an example using the program CG from the NAS suite [24]. Figure 1(a) shows a simplified version of CG's performance dominating loop nest. The outer loop computes the loop bounds of the inner loop, and the inner loop calls the `update` function, updating values in array `C`. For the outer loop, aside from induction variables, the only cross-iteration dependence is between calls to the `update` function. Profiling reveals that this dependence manifests across 72.4% of outer loop iterations. The inner loop has no cross-iteration dependence since no two iterations in the same invocation update the same element in array `C`.

The `update` dependence prevents DOALL parallelization [1] of the outer loop. Spec-DOALL [31] can parallelize the outer loop by speculating that the `update` dependence does not occur. However, speculating the outer loop dependence is not profitable, since the `update` dependence frequently manifests across outer loop iterations. As a result, DOALL will parallelize the inner loop and insert barrier synchronizations between inner loop invocations to ensure the dependence is respected between invocations.

Figure 2(a) shows the execution plan for a DOALL parallelization of CG. Iterations in the same inner loop invocation execute in parallel. After each inner loop invocation, threads synchronize at the barrier. Typically, threads do not reach barriers at the same time for a variety of reasons. For instance, each thread may be assigned a different number of tasks and the execution time of each task may vary. Threads that finish the inner loop early may not execute past the barrier, resulting in very poor scalability.

Figure 2(b) shows the execution plan after DOMORE's partitioning phase (Section 4.1). The first thread executes code in the outer loop (statements A to D) and serves as the scheduler. The other threads execute `update` code in the inner loop concurrently and serve as workers. Overlapping the execution of scheduler and worker threads improves the performance, however, without enabling cross-invocation parallelism, clock cycles are still wasted at the synchronization points.

Figure 2(c) shows the execution plan after the DOMORE transformation completes and enables cross-invocation parallelization. The scheduler sends synchronization information to the worker threads and worker threads only stall when a dynamic dependence is detected. In the example, most of CG's iterations are independent and may run concurrently without synchronization. However, iteration 1.5 (i.e. when i=1, j=5) updates memory locations which are later accessed by iteration 2.2. At runtime, the scheduler discovers this dependence and signals thread one to wait for iteration 1.5. After synchronization, thread one proceeds to iteration 2.2. As shown in Figure 7(a), DOMORE enables scalable loop speedup for CG up to eight threads on an eight-core machine.

Figure 3 shows a high-level overview of the DOMORE transformation and runtime synchronization scheme. DOMORE accepts a sequential program as input and targets hot loop nests within the program. At compile-time, DOMORE first partitions the outer loop into a scheduler thread and several worker threads (Section 4.1). The scheduler thread contains the sequential outer loop while worker threads contain the parallel inner loop. Based on the parallelization technique used to parallelize the inner loop, the code generation algorithm generates a multi-threaded program with cross-invocation parallelization (Section 4.5). At runtime, the scheduler thread checks for dynamic dependences, schedules inner loop iterations, and forwards synchronization conditions to worker threads (Section 3). Worker threads use the synchronization conditions to determine when they are ready to execute.

3. Runtime Synchronization

DOMORE's runtime synchronization system consists of three parts: detection of dependences at runtime, generation of synchronization conditions, and synchronization of iterations across threads. The pseudo-code for the scheduler and worker threads appear in Algorithms 1 and 2.

3.1 Detecting Dependences

DOMORE's scheduler thread detects dependences which manifest at runtime. Shadow memory is employed for deter-

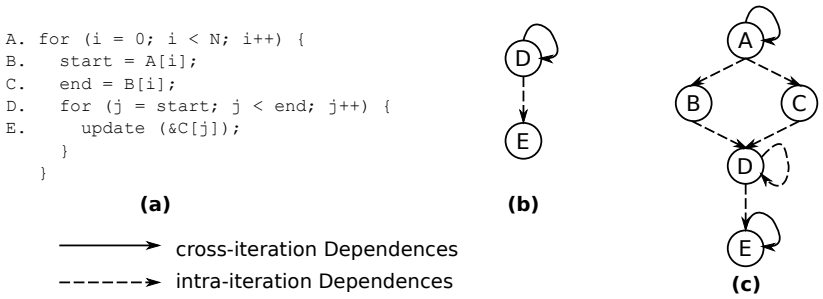


Figure 1: Example program: (a) Simplified code for a nested loop in CG (b) PDG for inner loop. The dependence pattern allows DOALL parallelization. (c) PDG for outer loop. Cross-iteration dependence deriving from E to itself has manifest rate 72.4%.

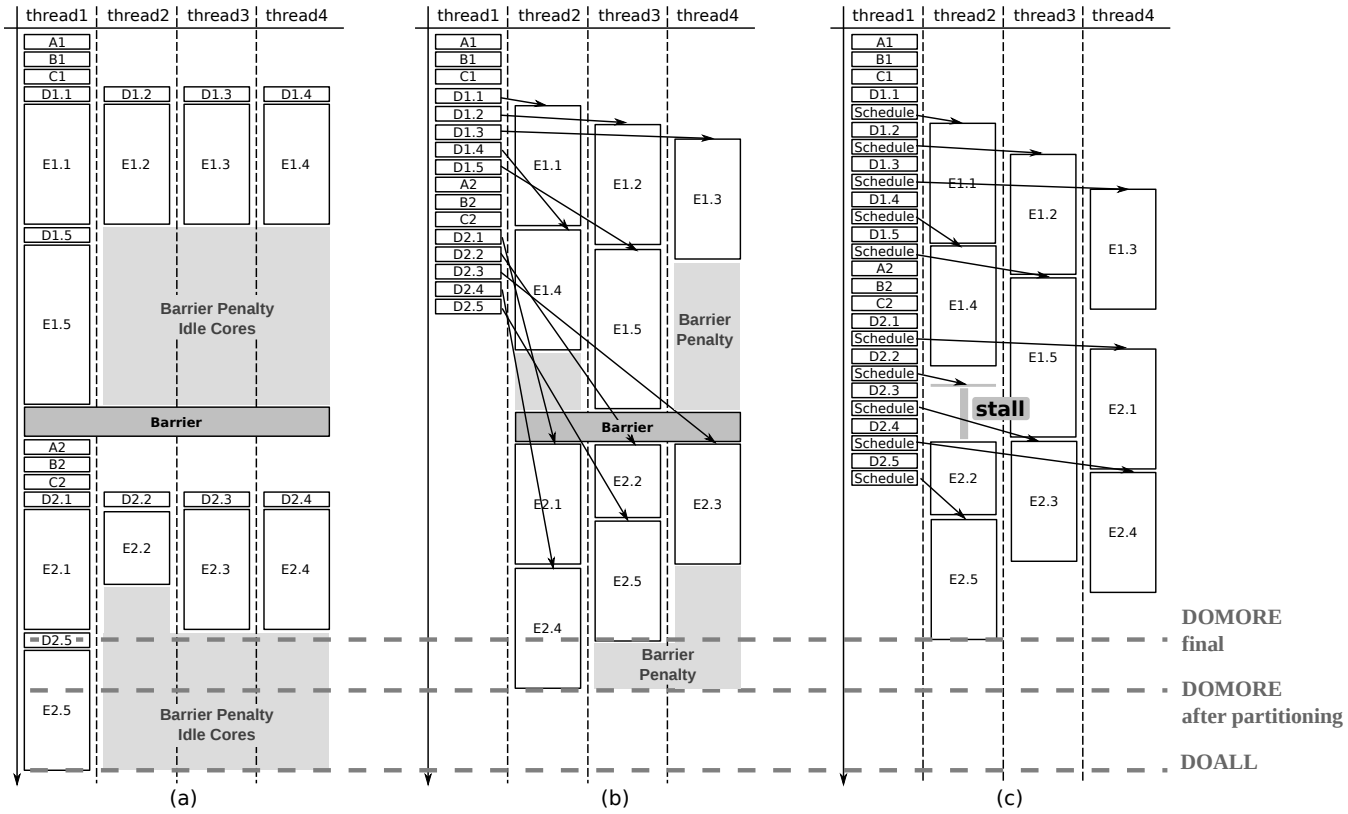


Figure 2: Comparison of performance with and without cross-invocation parallelization : (a) DOALL is applied to the inner loop. Frequent barrier synchronization occurs between the boundary of the inner and outer loops. (b) After the partitioning phase, DOMORE has partitioned the code without inserting the runtime engine. A scheduler and three workers execute concurrently, but worker threads still synchronize after each invocation. (c) DOMORE finalizes by inserting the runtime engine to exploit cross-invocation parallelism. Assuming iteration 2 from invocation 2 (2.2) depends on iteration 5 from invocation 1 (1.5). Scheduler detects the dependence and synchronizes those two iterations.

mining memory dependences. Each entry in shadow memory contains a tuple consisting of a thread ID (`tid`) and an iteration number (`iterNum`). For each iteration, the scheduler determines which memory addresses the worker will access using `computeAddr` function. The `computeAddr` function collects these addresses by redundantly executing related instructions duplicated from the inner loop. Details of automatic generation of `computeAddr` can be found in Section 4.4. The scheduler maps each of these address to a shadow memory entry and updates that entry to indicate that the most recent access to the respective memory location is by worker thread `tid` in iteration `iterNum`. When an address is accessed by two different threads, the scheduler synchronizes the affected threads.

Although the use of shadow memory increases memory overhead, our experiments demonstrate it is an efficient method for detecting dynamic dependences. However, a more space efficient conflict detecting scheme can also be used by DOMORE. For example, Mehrara et al. [19] propose a lightweight memory signature scheme to detect memory conflicts. The best time-space trade-off depends on end-user requirements.

3.2 Generating Synchronization Conditions

If two iterations dynamically depend on each other, worker threads assigned to execute them must synchronize. This requires collaboration between scheduler and worker threads.

The scheduler constructs synchronization conditions and sends them to the scheduled worker thread. A synchronization condition is a tuple also consisting of a thread ID (`depId`) and an iteration number (`depIterNum`). A synchronization condition tells a worker thread to wait for another worker (`depId`) to finish a particular iteration (`depIterNum`).

To indicate that a worker thread is ready to execute a particular iteration, the scheduling thread sends the worker thread a special tuple. The first element is a token (`NO_SYNC`) indicating no further synchronization is necessary to execute the iteration specified by the second element (`iterNum`).

Suppose a dependence is detected while scheduling iteration `i` to worker thread `T1`. `T1` accesses the memory location `ADDR` in iteration `i`. Shadow array (`shadow[ADDR]`) records that the same memory location is most recently accessed by worker thread `T2` in iteration `j`. The scheduler thread will send `(T2,j)` to thread `T1`. When the scheduler thread finds no additional dependences, it will send `(NO_SYNC,i)` to thread `T1`.

3.3 Synchronizing Iterations

Workers receive synchronization conditions and coordinate with each other to respect dynamic dependences. A status array is used to assist this: `latestFinished` records the latest iteration finished by each thread. A worker thread waiting on synchronization condition `(depId, depIterNum)` will stall until `latestFinished[depId]` $\geq$ `depIterNum`. After each worker thread finishes an iteration, it needs to update its status in `latestFinished` to allow threads waiting on it to continue executing.

Synchronization conditions are forwarded using `produce` and `consume` primitives provided by a lock-free queue design [14], which provides an efficient way to communicate information between scheduler and worker threads.

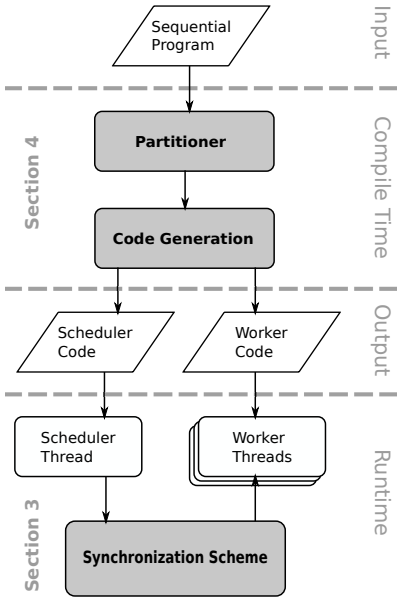


Figure 3: Overview of DOMORE compile-time transformation and runtime synchronization

Algorithm 1: Pseudo-code for scheduler synchronization

```
Input: iterNum : global iteration number
addrSet ← computeAddr(iterNum)
tid ← schedule(iterNum, addrSet)
foreach addr ∈ addrSet do
    < depTid, depIterNum > ← shadow[addr]
    if depIterNum ≠ −1 then
        if depTid ≠ tid then
            produce(tid, < depTid, depIterNum >)
    shadow[addr] ← < tid, iterNum >
produce(tid, < NO_SYNC, iterNum >)
```

Algorithm 2: Pseudo-code for worker

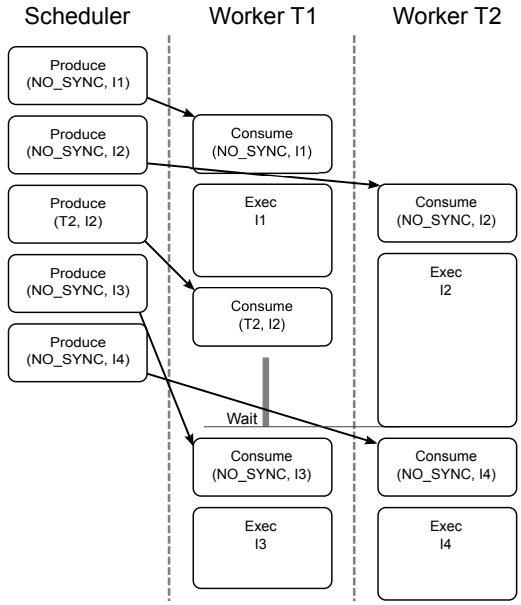
```
< depTid, depIterNum > ← consume()
while depTid ≠ NO_SYNC do
    while latestFinished[depTid] < depIterNum do
        sleep()
    < depTid, depIterNum > ← consume()
doWork(depIterNum)
latestFinished[getTid()] ← depIterNum
```

3.4 Walkthrough Example

The program CG illustrates DOMORE's synchronization scheme. Figure 4(a) shows the access pattern (value *j*) in each iteration for two invocations. Iterations are scheduled to two worker threads in round-robin order. Figure 4(b) shows the change of the helper data structures throughout the execution.

Original				Generated	
Invoc.	Iter.	Access	Sched.	Combined Iter.	shadow
-	-	-	-	initialize	$\langle\perp,\perp\rangle, \langle\perp,\perp\rangle, \langle\perp,\perp\rangle, \langle\perp,\perp\rangle$
1	1	A1	T1	I1	$\langle\perp,\perp\rangle, \langle T1, I1\rangle, \langle\perp,\perp\rangle, \langle\perp,\perp\rangle$
1	2	A3	T2	I2	$\langle\perp,\perp\rangle, \langle T1, I1\rangle, \langle\perp,\perp\rangle, \langle T2, I2\rangle$
2	1	A3	T1	I3	$\langle\perp,\perp\rangle, \langle T1, I1\rangle, \langle\perp,\perp\rangle, \langle T1, I3\rangle$
2	2	A2	T2	I4	$\langle\perp,\perp\rangle, \langle T1, I1\rangle, \langle T2, I4\rangle, \langle T1, I3\rangle$

(a)



(b)

Figure 4: Scheduler scheme running example: (a) Table showing original invocation/iteration, array element accessed in iteration, thread the iteration is scheduled to, combined iteration number, and helper data structure values (b) Execution of the example.

Iteration I1 accesses array element A1, the scheduler finds `shadow[A1] = <⊥,⊥>` meaning no dependence exists. It constructs a synchronization condition `(NO_SYNC,I1)` and produces it to worker thread T1. It then updates `shadow[A1]` to be `<T1,I1>`, implying thread T1 has accessed array element A1 in iteration I1. Worker thread T1 consumes the condition and executes iteration I1 without waiting. After it finishes, it updates `latestFinished[T1]` to be I1. Iteration I2 accesses array element A3 and no dependence is detected. A synchronization condition `(NO_SYNC,I2)` is produced to worker thread T2, which consumes the condition and executes iteration I2 immediately. Iteration I1 in the second invocation accesses element A3 again. Since `shadow[A3] = <T2,I2>`, a dependence is detected. So the scheduler produces `(T2,I2)` and `(NO_SYNC,I3)` to worker thread T1. Worker thread T1 then waits for worker thread T2 to finish iteration I2 (wait until `latestFinished[T2]` $\geq$ `I2`).

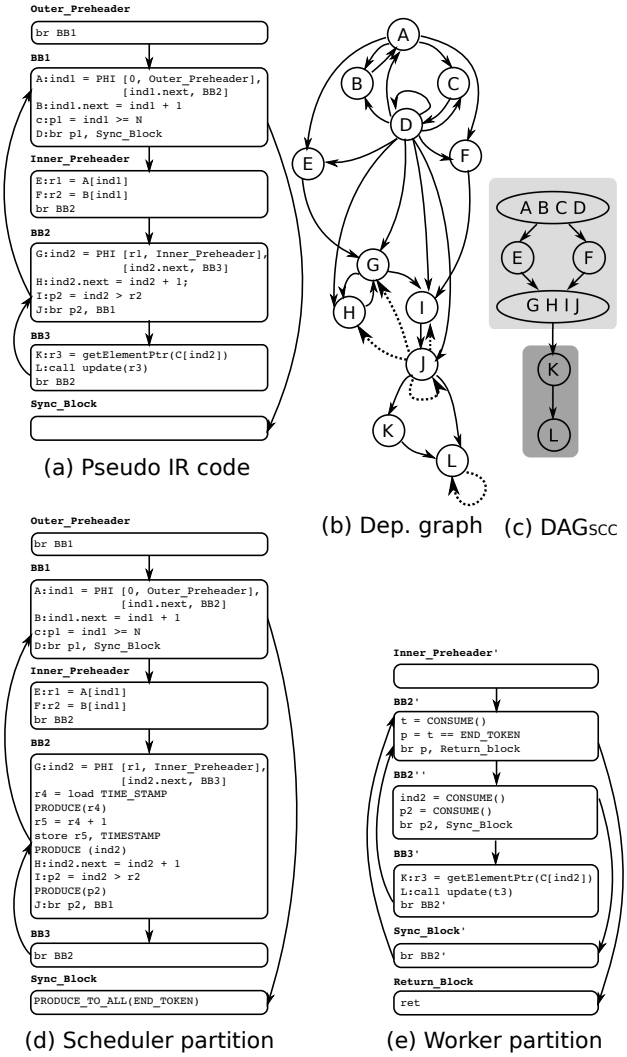


Figure 5: Running example for DOMORE code generation: (a) Pseudo IR for CG code; (b) PDG for example code. Dashed lines represent cross-iteration and cross-invocation dependences for inner loop. Solid lines represent other dependences between inner loop instructions and outer loop instructions. (c) DAG_{SCC} for example code. DAG_{SCC} nodes are partitioned into scheduler and worker threads. (d) and (e) are code generated by DOMORE MTCG algorithm (4.2).

Worker thread T1 then consumes the `(NO_SYNC,I3)` and begins execution of iteration I3.

Using this synchronization scheme, instead of stalling both threads to wait for first invocation to finish, only thread T1 needs to synchronize while thread T2 can move on to execute iterations from the second invocation.

4. Compiler Implementation

The DOMORE compiler generates scalable parallel programs by exploiting both intra- and inter-invocation parallelism. DOMORE first detects a candidate code region which contains a large number of loop invocations. DOMORE currently targets loop nests whose outer loop cannot be efficiently parallelized because of frequent runtime dependences, and whose inner loop is invoked many times and can be parallelized easily. For each candidate loop nest, DOMORE generates parallel code for the scheduler and worker threads. This section uses the example loop from CG (Figure 1) to demonstrate each step of the code transformation. Figure 5(a) gives the pseudo IR code of the CG example.

4.1 Partitioning Scheduler and Worker

DOMORE allows threads to execute iterations from consecutive parallel invocations. However, two parallel invocations do not necessarily execute consecutively; typically a sequential region exists between them. In CG's loop, statement A, B and C belong to the sequential region. After removing the barriers, threads must execute these sequential regions before starting the iterations from next parallel invocation.

DOMORE executes the sequential code in the scheduler thread. This provides a general solution to handle the sequential code enclosed by the outer loop. After partitioning, only the scheduler thread executes the code. There is no redundant computation and no need for special handling of side-effecting operations. If a data flow dependence exists between the scheduler and worker threads, the value can be forwarded to worker threads by the same queues used to communicate synchronization conditions.

The rule for partitioning code into worker and scheduler threads is straightforward. The inner loop body is partitioned into two sections. The loop-traversal instructions belong to the scheduler thread, and the inner loop body belongs to the worker thread. Instructions outside the inner loop but enclosed by the outer loop are treated as sequential code and thus belong to the scheduler.

To decouple the execution of the scheduler and worker threads for better latency tolerance, they should communicate in a pipelined manner. Values are forwarded in one direction, from the scheduler thread to the worker threads.

The initial partition may not satisfy this pipeline requirement. To address this problem, DOMORE first builds a program dependence graph (PDG) for the target loop nest (Figure 5(b)), including both cross-iteration and cross-invocation dependences for the inner loop. Then DOMORE groups the PDG nodes into strongly connected components (SCC) and creates a DAG_{SCC} (Figure 5(c)) which is a directed acyclic graph for those SCCs.

DOMORE goes through each SCC in DAG_{SCC}: (1) If an SCC contains any instruction that has been scheduled to the scheduler, all instructions in that SCC should be scheduled to the scheduler partition. Otherwise, all instructions in that SCC are scheduled to the worker partition; (2) If an SCC belonging to the worker partition causes a backedge towards any SCC belonging to the scheduler partition, that SCC should be re-partitioned to the scheduler. Step (2) is repeated until both partitions converge.

4.2 Generating Scheduler and Worker Functions

After computing the instruction partition for scheduler and worker, DOMORE generates code for scheduler and worker threads. Multi-Threaded Code Generation algorithm (MTCG) used by DOMORE builds upon the algorithm proposed in [26]. The major difference is that [26] can only assign a whole inner loop invocation to one thread while DOMORE can distribute iterations in the same invocation to different worker threads. The following description about DOMORE's MTCG highlights the differences:

1. Compute the set of relevant basic blocks (BBs) for scheduler (T_s) and worker (T_w) threads. According to algorithm in [26]: A basic block is relevant to a thread T_i if it contains either: (a) an instruction scheduled to T_i; or (b) an instruction on which any of T_i's instruction depends; or (c) a branch instruction that controls a relevant BB to T_i. DOMORE's MTCG follows these three rules, and additionally requires that: (d) a BB is relevant to T_w only if it belongs to the original inner loop; and (e) inner loop header is always relevant to both T_s and T_w. Rule (d) simplifies the control flow of code generated for T_w. However, since `produce` and `consume` instructions are placed at the point where dependent values are defined, worker thread may not contain the corresponding BB because of rule (d). Rule (e) guarantees that any value that is defined in BBs which are not duplicated in T_w can be communicated at the beginning of the duplicated inner loop headers.

2. Create the BBs for each partition. Place instructions assigned to the partition in the corresponding BB, maintaining their original relative order within the BB. Add a loop preheader BB and a loop return BB to T_w.

3. Fix branch targets. In cases where the original target does not have a corresponding BB in the same thread, the new target is set to be the BB corresponding to the closest relevant post-dominator BB of the original target. Insert a sync BB to T_w, which serves as the closest post-dominator BB for BBs which do not have a relevant post-dominator BB in T_w. Branch the sync BB in T_w to the loop header.

4. Insert `produce` and `consume` instructions. For Loop flow dependences, `produce` and `consume` instructions are inserted in the BB where the value is defined, if that BB is duplicated in both threads. Since inner loop live-in values are used but not defined inside the inner loop, the respective BB are not duplicated in T_w. To reduce the amount of communications, live-in values which are outer loop invariants are communicated at the end of the inner loop preheader. And the other live-ins will be communicated at the beginning of inner loop header. According to the partition rules, since instructions generating inner loop live-outs to

Algorithm 3: Pseudo-code for generating the computeAddr function from the worker function

Input: worker : worker function IR
Input: pdg : program dependence graph
Output: computeAddr : computeAddr function IR
depInsts ← getCrossMemDepInsts(pdg)
depAddr ← getMemOperands(depInsts)
computeAddr ←
reverseProgramSlice(worker, depAddr)

the outer loop are partitioned to the scheduler thread, DOMORE does not need to handle those live-out values. Finally, a timestamp is communicated at the beginning of the inner loop header. This timestamp value gives a global order for iterations from all invocations and will be used for scheduling and synchronizing iterations.

5. Finalize the communication. To control when each worker thread should return, an `END_TOKEN` is broadcasted when exiting the outer loop in T_s. That value will be captured by the first consume instruction in T_w's duplicated loop header. Two instructions are inserted to decide when to return from T_w: (1) a comparison instruction to check whether that value is an `END_TOKEN`; (2) a branch instruction targeting the return BB if the comparison instruction generates true value.

Up to this point, DOMORE has generated the initial code for scheduler thread and worker thread (Figure 5(d) and (e)). Later steps generate scheduling code, computeAddr code which will be inserted into the scheduler function and workerSync code which will be inserted into the worker function.

4.3 Scheduling Iterations

DOMORE currently supports two scheduling strategies, round-robin and memory partition based scheduling. Round-robin is used by many parallelization techniques. Memory partitioning (LOCALWRITE [12]) divides the memory space into disjoint chunks and assigns each chunk to a different worker thread, forming a one-to-one mapping. Iterations are scheduled to threads that own the memory locations being touched by that iteration. If multiple threads own the memory locations, that iteration is scheduled to any of them. Later, the scheduler will detect the conflicts between those threads and enforce synchronization correctly. DOMORE allows for the easy integration of other "smarter" scheduling techniques. Integration of a work stealing scheduler similar to Cilk [5] is planned as future work.

4.4 Generating the computeAddr function

The scheduler thread uses the `computeAddr` function to determine which addresses will be accessed by worker threads. DOMORE automatically generates the `computeAddr` function from the worker thread function

Algorithm 4: Final Code Generation

Input: program : original program IR
Input: partition : Partition of scheduler and worker code
Input: parallelPlan : parallelization plan for inner loop
Input: pdg : program dependence graph
Output: multi − threaded scheduler and worker program
scheduler, worker ← MTCG(program, partition)
scheduler ← generateSchedule(parallelPlan)
computeAddr ← generateComputeAddr(worker, pdg)
scheduler ← generateSchedulerSync()
worker ← generateWorkerSync()

using Algorithm 3. The algorithm takes as input the worker thread's IR in SSA form and a program dependence graph (PDG) describing the dependences in the original loop nest. The compiler uses the PDG to find all instructions with memory dependences across the inner loop iterations or invocations. These instructions will consist of loads and stores. In the worker thread, program slicing [36] is performed to create the set of instructions required to generate the address of the memory being accessed. Presently, the DOMORE transformation does not handle `computeAddr` functions with side-effects. If program slicing duplicates instructions with side-effects, the DOMORE transformation aborts. After the transformation, a performance guard compares the weights of the `computeAddr` function and the original worker thread. If the `computeAddr` function is too heavy relative to the original worker, the scheduler would be a bottleneck for the parallel execution, so the performance guard reports DOMORE is inapplicable.

4.5 Putting It Together

Algorithm 4 ties together all the pieces of DOMORE's code-generation. The major steps in the transformation are:

1. The Multi-Threaded Code Generation algorithm (MTCG) discussed in Section 4.2 generates the initial scheduler and worker threads based on the partition from Section 4.1.
2. The appropriate `schedule` function (Section 4.3) is inserted into the scheduler based upon the parallelization plan for the inner loop.
3. Create and insert the `computeAddr` (Algorithm 3) `schedulerSync` (Algorithm 1), `workerSync` (Algorithm 2), functions into the appropriate thread to handle dependence checking and synchronizing.

Figure 6 shows the final code generated for CG.

5. Evaluation

5.1 Evaluation Results

We evaluated DOMORE on 6 programs to demonstrate the potential performance gain. Table 1 gives their details. These

Scheduler Function

```
1   void scheduler () {
2     iternum = 0;
3     for (i = 0; i < N; i++) {
4       start = A[i];
5       end = B[i];
6       for (j = start; j < end; j++) {
7         addr_set = computeAddr(iternum);
8         tid = schedule(iternum, addr_set);
9         tid_queue = getQueue(tid);
10        schedulerSync(iternum, tid, tid_queue, addr_set);
11        produce(&C[j], tid_queue);
12        iternum++;
        }
      }
    }
13  produce_to_all(END_TOKEN);
```

SchedulerSync Function

```
1   void schedulerSync(iternum, tid, queue, addr_set) {
2     while (addr = get_next(addr_set)) {
3       depTid = getTid(shadow[addr]);
4       depIterNum = getIterNum(shadow[addr]);
5       if (depTid != tid && depIterNum != -1) {
6         produce(depTid, queue);
7         produce(depIterNum, queue);
        }
8       shadow[addr] = (tid, iternum);
      }
9     produce(NO_SYNC, queue);
10    produce(iternum, queue);
    }
```

Worker Function

```
1     void worker() {
2       while (1) {
3         depTid = consume();
4         if (depTid == END_TOKEN)
5           return;
6         if (depTID == NO_SYNC) {
7           doWork();
        }
8         else
9           workerSync(depTid);
      }
    }
```

doWork Function

```
1     void doWork() {
2       iternum = consume();
3       tid = getTid();
4       addr = consume();
5       update(addr);
6       latestFinished[tid] = iternum;
    }
```

workerSync Function

```
1     void workerSync(depTid) {
2       iternum = consume();
3       while (latestFinished[depTid] < iternum)
4         sleep();
    }
```

Figure 6: Generated code for example loop in CG. Non-highlighted code represents initial code for scheduler and worker functions generated by DOMORE's MTCG. Code in grey is generated in later steps for iteration scheduling and synchronization.

programs were chosen because their performance dominating loop nests contained parallelizable inner loops, and because inner loop parallelization introduces frequent barrier synchronizations that limits overall scalability. These two characteristics are required for DOMORE to have a potential benefit. The inner loops of these programs can be parallelized using different compiler techniques including DOALL, Spec-DOALL [31] and LOCALWRITE [12]. Using these programs, we show that DOMORE parallelization, by enabling additional cross-invocation parallelization, can deliver much more scalable parallel programs.

DOMORE is evaluated on an 8-core shared memory machine. It has two Intel 4-core Xeon E5310 processors running at 1.60GHz with 8GB of memory. Its operating system is 64-bit Linux 2.6.24. Sequential versions are compiled using LLVM Clang 2.9 with -O3.

Figure 7 shows the evaluation results for the outer loop speedup relative to the original sequential execution. For the original parallelized version with barriers between inner loop invocations, none scale beyond a small number of cores. DOMORE shows scalable performance improvements for CG, LLUBENCHMARK and BLACKSCHOLES because their scheduler threads are quite small compared to the worker threads ($< 5\%$ runtime) and processor utilization is high. ECLAT, FLUIDANIMATE and MGRID do not show as much improvement. The following section provides details about those programs.

5.2 Case Studies

ECLAT from MineBench [22] is a data mining program using a vertical database format. The target loop is a two-level nested-loop. The outer loop traverses a graph of nodes. The inner loop traverses a list of items in each node and appends each item to corresponding list(s) in the database based upon on the item's transaction number. Since two items might share the same transaction number, and the transaction number is calculated non-linearly, static analysis cannot determine the dependence pattern. Profiling information shows that there is no dynamic dependence in the inner loop. For the outer loop, the same dependence manifests in each iteration (99%). As a result, Spec-DOALL is chosen to parallelize the inner loop and a barrier is inserted after each invocation. Spec-DOALL achieves its peak speedup at 3 cores. For DOMORE, a relatively large scheduler thread (12.5% scheduler/worker ratio) limits scalability. As we can see in Figure 7, DOMORE achieves scalable performance up to 5 processors. After that, the sequential code becomes the bottleneck and no more speedup is achieved.

FLUIDANIMATE from the PARSEC [4] benchmark suite uses an extension of the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. The target loop is a six-level nested-loop. The outer loop goes through each particle while an inner loop goes through the nearest neighbors of that particle. The inner loop calculates influences between the particle and its neighbors and updates all of their statuses. One particle can be neighbor to multiple particles, resulting in statically unanalyzable update patterns. LOCALWRITE

Benchmark Program	Source Suite	Function	% of Execution Time	Parallelization Plan	% of Scheduler/ Worker
CG	NAS [24]	sparse	12.2	DOALL	4.1
MGRID	SPEC CFP2000 [34]	psinv	29.6	DOALL	1.5
LLUBENCH	llvmbench [17]	main	100	DOALL	1.7
BLACKSCHOLES	PARSEC [4]	bs_thread	100	DOALL	4.5
ECLAT	MineBench [22]	process_inverti	24.5	Spec-DOALL	12.5
FLUIDANIMATE	PARSEC	ComputeForce	50.0	LOCALWRITE	21.5

Table 1: Details about evaluated benchmark programs

chooses to parallelize the inner loop to attempt to reduce some of the computational redundancy. Performance results show that parallelizing the inner loop does not provide any performance gain. Redundant computation and barrier synchronizations negate the benefits of parallelism. DOMORE is applied to the outermost loop, generating a parallel program with the redundant code in the scheduler thread and each inner loop iteration is scheduled only to the appropriate owner thread. Although DOMORE reduces the overhead of redundant computation, partitioning the redundant code to the scheduler increases the size of the sequential region, which becomes the major factor limiting the scalability.

MGRID from the SPECFP2000 [34] suite demonstrates the capabilities of a very simple multi-grid solver in computing a three dimensional potential field. The target loop is a three-level nested-loop. DOALL applicable to the innermost loop. As shown in the results, even after DOMORE optimization, the scalability of MGRID is poor. The major cause is that the execution time of each inner loop invocation only takes about 4,000 clock cycles. With increasing number of threads, the overhead involved in multi-threading outweighs all performance gain.

6. Related Work

Cross-invocation Parallelization

Loop fusion techniques [9, 37] aggregate small loop invocations into a large loop invocation, converting the problem of cross-invocation parallelization into the problem of cross-iteration parallelization. The applicability of these techniques is limited to mainly affine loops due to their reliance upon static dependence analysis. Since DOMORE is a runtime technique, it is able to handle programs with input-dependent dynamic dependences. Tseng [35] partitions iterations within the same loop invocation so that cross-invocation dependences flow within the same working thread. Compared to DOMORE, this technique is much more conservative. DOMORE allows dependences to manifest between threads and synchronizations are enforced only when real conflicts are detected at runtime.

While manually parallelizing a sequential program, programmers can use annotations provided by BOP [8] or TCC [11] systems to specify the potential concurrent code regions. Those code regions will be speculatively executed in parallel at runtime. Both techniques can be applied to exploit cross-invocation parallelism. However, they require manual annotation or parallelization by programmers while DOMORE is a fully automatic parallelization technique.

Synchronization Optimizations

Optimization techniques are proposed to improve the performance of parallel programs with excessive synchronizations (e.g, locks, flags and barriers).

Fuzzy Barrier [10] specifies a synchronization range rather than a specific synchronization point. Instead of waiting, threads can execute some instructions beyond the synchronization point. Speculative Lock Elision [29] and speculative synchronizations [18] design hardware units to allow threads to speculatively execute across synchronizations. Grace [3] wraps code between fork and join points into transactions, removing barrier synchronizations at the join points and uses a software-only transactional memory system to detect runtime conflicts and do recovery.

These techniques are designed to optimize already parallelized programs. DOMORE, instead, takes a sequential program as input and automatically transforms it into a scalable parallel program. DOMORE's runtime engine synchronizes two iterations only when necessary, and thus, does not require further optimization for synchronizations.

Runtime Dependence Analysis

Within the category of runtime dependence analysis, there are techniques which perform preprocessing of loops to identify dependences (i.e. scheduling based) and those which identify dependences in parallel with execution of the loop (i.e. speculative techniques such as transactional memory [13, 15, 33] and the LRPD family of tests [7, 31]). DOMORE is a scheduling based technique.

Generally, scheduling techniques have a non-negligible fixed overhead that changes very little based upon the number of data dependences in the program. For DOMORE, this is the overhead introduced by the scheduler. Speculative techniques typically have a small amount of fixed overhead with a highly variable amount of dynamic overhead based upon the number of data dependences, which translate to misspeculation, in a program. Therefore, for programs

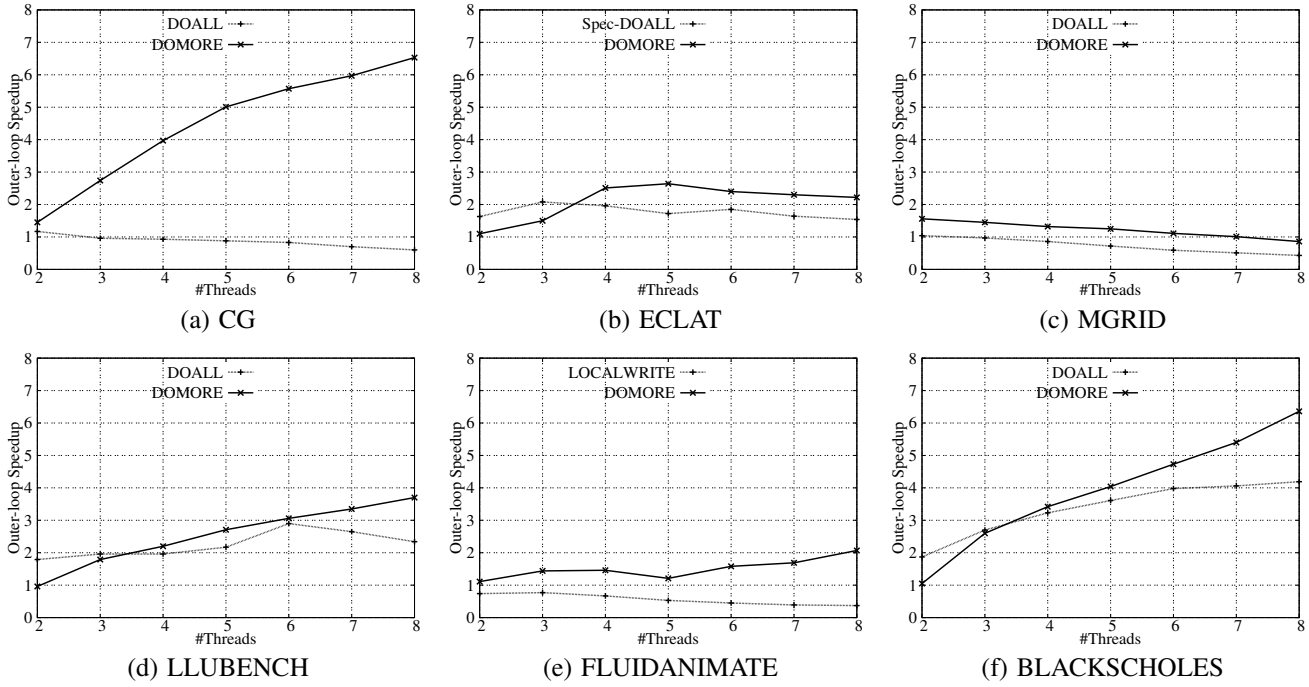


Figure 7: Performance Comparison between parallel code with and without cross-invocation parallelism.

with a small number of dynamic dependences, speculative techniques will typically see better performance improvements. However, for programs that have more than some small number of dynamic dependences, the fixed scheduling overhead can prove to be much less than the overhead of mis-speculation recovery.

DOMORE instruments the program to detect dynamic dependences between iterations at runtime. A similar idea has been used to exploit parallelism by the Inspector-executor (IE) model [27, 30, 32], which was first proposed by Saltz et al. IE consists of three phases: inspection, scheduling, and execution. A complete dependence graph is built for all iterations during the inspecting process. By topological sorting the dependence graph, each iteration is assigned to a wavefront number for later scheduling. There are two important differences between DOMORE and IE. First, DOMORE is able to exploit cross-invocation parallelism while IE is a parallelization technique limited to iterations from the same invocation. Second, IE's inspection process is serialized with the scheduling process. DOMORE overlaps the inspecting and scheduling processes for efficiency.

Cilk [5] uses a work stealing scheduler to increase load balance among processors. DOMORE can use a similar work stealing technique as an alternative scheduling policy. Baskaran et al. [2] proposed a technique that uses an idea similar to IE to remove barriers from automatically parallelized polyhedral code by creating a DAG of dependences at runtime time and using it to self-schedule code. This technique can only be used for regular affine codes whose dependences are known at compile-time while DOMORE is designed for irregular codes with dependences that cannot be determined statically. However, the DAG scheduling technique could also be integrated into DOMORE as another potential scheduling choice.

Predicate-based techniques resolve dependences at runtime by checking simple conditions. Moon et al. [20] inserts predicates before the potential parallel region. If the predicates succeed, the parallel version is executed. If they fail, the sequential version will be used instead. The same idea is used by Nicolau et al. [23] to remove synchronizations between threads. This predicate-based dependence analysis can be used by DOMORE as an efficient way to detect conflict between two iterations.

7. Conclusion

Exploiting the performance of multicore processors requires scalable parallel programs. Most automatic parallelization techniques parallelize iterations within the same loop invocation and synchronize threads at the end of each parallel invocation. Unfortunately, frequent synchronization limits the scalability of many codes. In practice, iterations in different invocations of a parallel loop are frequently independent. DOMORE exploits this cross-invocation parallelism by observing cross-invocation dependences at runtime and only synchronizing iterations when necessary. For six benchmarks, DOMORE achieves a geomean loop speedup of 2.1× over parallel execution without cross-invocation parallelization and of 3.2× over sequential execution on eight cores.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This material is based on work supported by National Science Foundation Grants 0964328 and 1047879, and DARPA contract FA8750-10-2-0253. All opinions, findings, conclusions, and recommendations expressed throughout this work are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

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

[2] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *PPOPP*, 2009.

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

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

[7] F. H. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. In *IPDPS*, 2002.

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

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

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

[11] 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.

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

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

[14] T. B. Jablin, Y. Zhang, J. A. Jablin, J. Huang, H. Kim, and D. I. August. Liberty Queues for EPIC Architectures. In *EPIC*, 2010.

[15] T. Knight. An architecture for mostly functional languages. In *LFP*, 1986.

[16] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

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

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

[19] 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.

[20] S. Moon, B. So, M. W. Hall, and B. R. Murphy. A case for combining compile-time and run-time parallelization. In *LCR*, 1998.

[21] V. Nagarajan and R. Gupta. Speculative optimizations for parallel programs on multicores. In *LCPC*, 2009.

[22] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. 2006.

[23] A. Nicolau, G. Li, A. V. Veidenbaum, and A. Kejariwal. Synchronization optimizations for efficient execution on multicores. In *ICS*, 2009.

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

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

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

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

[28] 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.

[29] R. Rajwar and J. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO*, 2001.

[30] 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.

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

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

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

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

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

[36] M. Weiser. Program slicing. In *ICSE*, 1981.

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