CilkSpec: Optimistic Concurrency for Cilk

Shaizeen Aga
University of Michigan
Ann Arbor
shaizeen@umich.edu

Sriram Krishnamoorthy
Pacific Northwest
National Laboratory
sriram@pnnl.gov

Satish Narayanasamy
University of Michigan
Ann Arbor
nsatish@umich.edu

ABSTRACT

Recursive parallel programming models such as Cilk strive to simplify the task of parallel programming by enabling a simple divide-and-conquer programming model. This model is effective in recursively partitioning work into smaller parts and combining their results. However, recursive work partitioning can impose additional constraints on concurrency than is implied by the true dependencies in a program. In this paper, we present a speculation-based approach to alleviate the concurrency constraints imposed by such recursive parallel programs. We design a runtime infrastructure that supports speculative execution and a predictor to accurately learn and identify opportunities to relax extraneous concurrency constraints. Experimental evaluation demonstrates that speculative relaxation of concurrency constraints can deliver gains of up to $1.6\times$ on 30 cores over baseline Cilk.

Categories and Subject Descriptors

D.1.3 [Software]: Concurrent Programming—*Parallel programming*

Keywords

Parallel Programming, Cilk, Speculation

1. INTRODUCTION

The advent of large multi-core systems has renewed interest in programming models that simplify parallel programming. Recursive parallel programming models, as exemplified by Cilk [4], put forth a simple concurrency model. A programmer divides the given work into smaller work units whose results are recursively combined to provide the final result. This divide-and-conquer strategy simplifies programmer's effort to specifying concurrency. Often, a concurrency-annotated version of a sequential recursive program (a principle referred to as "serial elision" in Cilk).

However, this divide-and-conquer strategy can impose additional constraints on concurrency in the program. Specif-

ically, all work represented by two work units are ordered if any operation in one unit depends on any operation in the other. In general, Cilk-like programming models represent the computation as a series-parallel directed acyclic graph (dag). Computations whose dependence structure is modeled by a more general dag need to be embedded into an SP-dag by introducing additional edges. This can significantly limit available parallelism and thus scalability.

In this paper, we try to answer the question: *is it possible to achieve the best performance for arbitrary computation dags while retaining Cilk's simplicity?* We present an approach to speculatively recover the concurrency lost due to a recursive parallel program specification. We exploit several characteristics of the structured expression of concurrency in Cilk programs. In many Cilk computations, the dag structure is independent of the problem size, enabling us to capture data-independent concurrency constraints efficiently¹. Unlike speculation for loop programs, relaxing such concurrency constraints in Cilk exposes significant additional work and concurrency therein, which, in turn, greatly increases the benefits achievable from optimistic parallelization. Finally, scalable Cilk programs employ coarse-grained base cases that can be efficiently annotated to track data accesses and detect conflicts.

We design a runtime system that enables speculative execution of Cilk programs. We present schemes that employ increasing degrees of speculation to explore additional opportunities for relaxing concurrency constraints. These include single-task, multi-task (a.k.a. deep), and parallel speculation. We develop a predictor to efficiently and accurately identify opportunities for speculation while considerably reducing the number of mis-speculations. We design a user-level API to annotate Cilk programs and achieve efficient data versioning and conflict detection under speculation.

We evaluate the various schemes and demonstrate that speculation can significantly reduce idle times induced by concurrency constraints and improve upon original Cilk programs. We demonstrate that the speculation framework incurs low space and time overheads. In addition, we show that speculation incurs low overheads in the absence of profitable speculation opportunities. Evaluation of the speculation predictor shows it can precisely identify speculation opportunities and keep mis-speculations low.

The primary contributions of the paper are:

- A speculation approach to overcome concurrency constraints in recursive parallel programs

¹We do not address data-dependent computations (e.g., parallel breadth-first search) in this work.

ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC '15, November 15-20, 2015, Austin, TX, USA

© 2015 ACM. ISBN 978-1-4503-3723-6/15/11...\$15.00

DOI: <http://dx.doi.org/10.1145/2807591.2807597>

- An efficient speculation predictor to identify opportunities to relax concurrency constraints
- Design and implementation of efficient support for speculation and conflict detection
- Experimental evaluation of the speculation overheads and accuracy.

2. BACKGROUND AND MOTIVATION

Consider the example program in Figure 1(a) that computes array A. The corresponding recursive version (Figure 1(a)) divides the work into four quadrants and computes them recursively. The spawn and sync keywords in the recursive program represent the concurrency and synchronization annotations in a Cilk program. The dependences (a.k.a. the spawn and sync annotations) in the recursive program are induced by dependences in the loop program. The quadrant A00 begins execution of a given task. A01 and A10 depend on A00 and are separated from the processing of A00 by a `sync`. A01 and A10 can themselves be processed concurrently and are invoked with a spawn keyword without any intervening synchronization. A11 depends on both A01 and A10 and is ordered appropriately.

Recursive programs exhibit many desirable properties. The recursive structure often mirrors the dependence structure in the element-level program specification, simplifying programming. Besides the ease of programming, such a structure accesses recursively smaller portions of the array. As a result, recursive structure is often employed in designing cache-oblivious algorithms [3].

While recursive programs alleviate the burden on the programmer, their structures can prevent them from achieving good speedups on large multi-core systems. On occasion, recursive programming can incur the cost of increased critical path length. For example, the optimal critical path length for the example in Figure 1(a) is $\Theta(n)$, whereas the critical path length of the recursive version is $\Theta(n^{\log_2 3})$ for a $n \times n$ matrix. The cause for this suboptimal critical path length is the inherently coarser dependencies required by the recursive expression. In Figure 1(a), not all of the quadrant A01 (or A10) must wait for all of quadrant A00 to be computed. In Figures 1(b) and 2(a), the dependence-induced dags and corresponding critical paths are illustrated for a problem size of 4 (ihi-ilo=jhi-jlo=4). For example, given a problem size of 4, the dependence-induced dag for the loop version in Figure 1(a) has a critical path length of 7, compared to 9 for the recursive Cilk version.

In general, mapping a computation dag onto a Cilk-like recursive parallel model can increase the critical path length and result in reduced scalability due to decreased available parallelism. Decreased available parallelism caused by coarser dependencies, together with large core counts, can lead to poor scalability. We aim to preserve the simplicity of recursive programming by recovering the lost concurrency in such overly constrained recursive programs.

3. CilkSpec OVERVIEW

As identified in Section 2, certain dependencies in recursive parallel programs are extraneous. To overcome the constraints they impose, we need a mechanism that at runtime determines which of the expressed dependencies are true and which ones are artificial, merely an artifact of recursive expression. To this end, a classic technique employed

is speculation that enables optimistic elision of an expressed dependency while relying on precise data access tracking to rollback in the event of a conflict. In our approach (referred to as *CilkSpec*), we too employ speculation to explore the expressed dependencies and derive newer, more relaxed dependencies to recover the concurrency lost due to recursive program specification. While employed in varied contexts, such as loops, speculation for recursive parallel programs presents some interesting opportunities and challenges.

One key challenge in optimistic concurrency involves identifying valid speculation opportunities. While this challenge is inherent in any speculation approach, the structured nature of recursive programs aids in designing a speculation predictor, which not only can quickly learn the profitable speculation opportunities, but also accomplish this by tracking only a subset of all tasks in the system. The parent-child relationship present in recursive programs implies that any speculation in the parent could depend only on tasks present in the child computations, which are incomplete. We exploit this opportunity to design an efficient predictor with learns fast, tracks less, and has low space overhead.

Another key challenge in speculation is maintaining correctness, which requires precise data access tracking to detect when a true data dependency is violated or a conflict is detected. The overhead for such tracking largely depends on the granularity at which this tracking is done. Again, the regular structure of recursive programs aids in keeping this overhead in check. Recursive programs typically have a base case that works on the smallest division of work present in the computation. Instead of tracking individual accesses to memory, we track memory accesses at the granularity of access done by this base task. We design a library that works at this higher granularity, which programmers can use to easily annotate data accesses in their programs while keeping data access tracking overhead fairly low.

A challenge unique to speculation for recursive programs is that eliding a synchronization constraint possibly could expose huge chunks of work, depending on the recursion depth at which this constraint is elided. While large chunks of speculative work will help amortize speculation overhead, it could also dramatically increase mis-speculation penalties if aggressively pursued. To overcome this challenge, we design several flavors of speculation that iteratively explore deeper dependencies and augment our predictor design to additionally track depth of speculation while still keeping space overheads low. While large chunks of speculative work complicate predictor design, they also expose the opportunity for parallel speculation. Using already present programmer annotations, we divide speculative work among multiple threads to further expose more work under speculation and keep more idle workers busy.

In the remaining sections, we present our approach to overcome the following challenges:

- Design of an accurate and efficient speculation predictor
- Infrastructure to support efficient speculation, deeper speculation, and parallelization of speculative work
- Low overhead data versioning, data access tracking, and conflict detection.

4. CilkSpec DESIGN

Execution in Cilk corresponds to a depth-first traversal of the computation dag. We use the term `sync` to denote both a statement in the program (Figure 1(a)) and a runtime in-

Figure 1: (a) Iterative and corresponding Cilk kernel; (b) Dependence-induced dag for the loop kernel. Label $[xy]$ denotes the task to compute $A[x][y]$. A critical path is highlighted in red.

stance of a synchronization constraint we encounter. A *sync* is said to be unsatisfied in Cilk if computation preceding the *sync* is ongoing. The computation preceding a *sync* is termed as a *child* computation, while the computation following the *sync* is termed the *parent* computation. On encountering such an unsatisfied *sync*, a worker thread suspends the *sync*, i.e., abandons the execution of that depth-first traversal and seeks another starting point for its execution. In CilkSpec, we speculatively continue execution past an unsatisfied *sync*, backing up the relevant state in the process. When a suspended *sync* can continue normal execution, CilkSpec checks the correctness of the speculative execution. A speculative execution is deemed valid if no true data dependency is violated. An incorrect speculation causes the speculative traversal to be quashed, and the execution gets rolled back to the backed-up state.

4.1 When to speculate

Speculation predictor. In a specification that combines true dependencies and artificial concurrency constraints, speculating every time we encounter an unsatisfied *sync* can lead to wasteful work that is discarded. Therefore, we build a speculation predictor that teases apart true and artificial dependencies. To explore all possible speculation opportunities, it is paramount that the predictor identify computation points in the program that are delayed at runtime because of artificial concurrency constraints. We refer to such a computation point as a *specPoint*. To ensure successful speculation at this *specPoint*, the predictor needs to learn the extent of progress required of the computation preceding the *specPoint*. We refer to this as the *progress boundary* required for a *specPoint*. Figure 2(b) depicts three possible *progress boundary* values for speculation at identified *sync*. The *parent-child* construct in a recursive computation implies that a suspended *parent* computation is data dependent only on *child* computations. As shown in Section 5, the structured form of Cilk computations and their dependencies helps in compact specification of both the *specPoint* and the *progress boundary* and allows efficient predictor queries while continuing to expose ample speculation opportunities.

Figure 2: (a) Dependence-induced dag for the Cilk kernel. Label $[ab:cd]$ denotes a task that recursively computes $A[a:b][c:d]$. A critical path is highlighted in red. (b) Flavors of CilkSpec.

Delayed speculation. As execution progresses, a *specPoint* not ready for speculation at one point in time might become a good candidate. For example, in Figure 2(b), the identified *sync* is not ready to be speculated when only task [00] is completed. On the other hand, it is ready to be speculated when both task [00] and task [01] are done. Depth-first schedulers drop an unsatisfied *sync* onto the heap and inspect it only when one of its pending dependencies is complete. This limits the number of times a *specPoint* can be inspected by the predictor. We address this limitation with a more aggressive strategy, referred to as *delayed speculation*, wherein we track a *specPoint* that has not been speculated on. Idle workers then opportunistically inspect these speculation points and predict whether or not they can now be successfully speculated upon. To be profitable, we need to balance the cost of such lookups with the benefits accrued from identifying additional speculation opportunities.

4.2 How much to speculate

While an accurate predictor is essential to minimize mis-speculation penalties, the amount of work performed speculatively at each *specPoint* also needs to be determined. A challenge inherent to speculation in recursive programs is that the smaller the recursion depth at which a synchronization constraint is elided, the larger the chunk of work exposed for speculation. Therefore, an aggressive speculation strategy may be susceptible to huge mis-speculation penalties. A more careful exploration of speculation depth is warranted, especially when synchronization constraints stem from true data dependencies. To this end, we design the most basic flavor of speculation, termed *pred*, which explores only the task immediately following a *sync*. This is depicted in Figure 2(b).

Deep speculation. The more we speculate at a *specPoint*, the less time workers spend idling in the system. In addition, performing non-trivial amounts of work helps amortize the overheads of the speculation infrastructure: data backup, resumption, and checking. As such, we develop *deep speculation* to allow arbitrary amounts of speculation. This is depicted in Figure 2(b), where we speculatively execute mul-

multiple tasks following a *sync*. In the *pred* scheme, the predictor tracks a single *progress boundary* per *specPoint*. Such a strategy for *deep* speculation would lead to far stricter progress boundaries and reduce speculation opportunities. To address this, we augment our predictor to make it aware of speculation depth by identifying work units nested within a single root *specPoint* and tracking *progress boundary* for every such nested work unit. In addition to the backup taken at the beginning of speculation, we devise a mechanism to take multiple backups during *deep* speculation to avoid throwing away large chunks of computation because of a mis-speculation.

Parallel speculation. The elision of *sync* speculated on in the dependence-induced dag in Figure 2(b) exposes two chunks of work (tasks [02:13] and [20:31]) that can be speculatively executed in parallel. While these work units can also be explored sequentially, such an exploration will forgo additional performance benefits and fail to address the lack of concurrency present in the program. To address this, we propose and implement parallel speculation, where we let multiple workers share, through work stealing, the speculative work past a *specPoint*. This further exposes speculation opportunities while exposing a greater degree of concurrency. However, parallel speculation can worsen the costs associated with speculation, both in terms of the cost of mis-speculation and the additional inter-worker synchronization to allow such parallelized speculation. We design an efficient implementation of parallel speculation that we term as *para*.

4.3 Correctness under speculation

Preserving correctness while eliding synchronization constraints requires versioning and precise data access tracking to detect mis-speculation. We employ versioning to roll back updates under mis-speculation. Together with versioning, data access tracking helps identify conflicts and trigger a mis-speculation. In general, versioning and tracking can incur prohibitive overheads. The regular structure of recursive programs assists in designing a library that versions and tracks data at a higher granularity than individual memory accesses. Data and their accesses annotated with this library are transparently versioned and tracked to ensure correctness under speculation.

5. SPECULATION PREDICTOR

The speculation predictor must identify unique speculation configurations and predict the potential for successful speculation. This requires a compact representation of speculation points and progress boundaries. The speculation prediction problem can be defined as follows:

```
specPoint p : tuple {spawn, recursion_level}
p.child_spawn_set : spawns that p depends on per cilk
specification

for all specPoints p in a cilk procedure:
  for all spawns s in p.child_spawn_set:
    is p dependent on s?
```

Each *spawn* and *sync* statement in a program is labeled, starting with 0 at the start of the function and incrementing the counter for each subsequent *spawn* or *sync* statement. Each task, an instance of a function, can be uniquely identi-

fied by the list of labels, identifying the stack calling context, from the root of the computation to the task of interest. In extreme cases, each read and write operation to a data block can be annotated with this label list. This can be used to construct the true dependence graph for the execution with the task reading a data block dependent on the last preceding task that wrote that block. While precise, the space overhead of such a scheme is extremely high. In addition, each speculation configuration is unique, effectively allowing no generalization opportunities. Therefore, we approximate the *specPoint* in terms of the *spawn/sync* label and the recursion level.

A precise characterization of the *progress boundary* can be expensive and imprecise because of the inherently parallel execution of the speculator and the outstanding children. We instead encapsulate the progress boundary by the number of spawns immediately enclosed in the outstanding children (referred to as *child spawn set*) that have been completed. This information, referred to as the *progress metric*, is a simple counter that is tracked by the outstanding children and looked up when the predictor is being queried.

The predictor determines whether or not to speculate based on the *specPoint* and progress metric. Listing 1 shows the algorithm employed by the predictor. For each *specPoint*, the predictor tracks and learns the progress metric for each spawn in the *child spawn set*. The progress metric of all spawns in the *child spawn set* of a speculation point is collectively considered its *progress boundary*.

```
//Our heuristic for work progress
progress(spawn s) = #spawns completed in a runtime
instance of s

p.boundary = map{key=spawn s, value=progress(spawn s)}
where keys = child_spawn_set(p)

progressBoundary: an observed snapshot of p.boundary

p.failureBoundary: a snapshot of p.boundary at which
speculation was attempted but resulted in a failure

p.lastSpecBoundary: a snapshot of p.boundary at which
the most recent decision was made to speculate

//pb is the current snapshot of p.boundary being queried
for speculation potential
can_speculate(specPoint p, progressBoundary pb):
  for all spawns s in pb.keys():
    //current progress less than progress at which
    speculation failed
    if pb[s] < p.failureBoundary[s]: return NO
  p.lastSpecBoundary = pb
  return YES

on_spec_failure(specPoint p):
  if p.failureBoundary == NULL:
    p.failureBoundary = p.lastSpecBoundary
  else:
    //Find s such that for all s1 in p.lastSpecBoundary.
    keys(), p.lastSpecBoundary[s] <= p.
    lastSpecBoundary[s1]
    p.failureBoundary[s] = p.lastSpecBoundary[s]
  p.lastSpecBoundary = NULL
```

Listing 1: Predictor decision given a progress boundary

Once a speculation is successful for a given *progress boundary*, we observe that the data dependences are deemed satisfied, and any additional execution progress by the out-

standing children can be ignored with respect to that dependence. Therefore, we attempt to determine the smallest portion of execution in the outstanding children, encoded as the smallest progress values, that leads to successful speculation. Once a successful speculation is observed for a progress boundary among the outstanding children, speculation under further progress in the boundary can also be predicted as possibly successful. The query to decide regarding speculation then becomes a simple check if all spawns in *child spawn set* have progressed just enough, i.e., their *progress* values are greater than those for which we observed a failure. If a failure has not been observed thus far, the predictor optimistically decides to speculate.

Therefore, for each *specPoint*, this algorithm requires us to track what we term as the *failure boundary*, which is the *progress* value per spawn in *child spawn set* for which we have observed a speculation failure. The *failure boundary* is updated every time we encounter a mis-speculation. As there can be multiple spawns in a *child spawn set*, it is crucial to decide which or all of their progress values require updating. Rather than update all spawns, we employ a conservative strategy: our heuristic updates the spawn that has made the least progress, i.e., we assume that the slowest-progressing spawn is more likely to have caused a mis-speculation.

5.1 Predictor for deep speculation

Thus far, we have assumed that the predictor makes a decision once per *specPoint*. However, the spawn associated with a *specPoint* could have many spawns enclosed in it. Making a single decision for all enclosing spawns exposes us to the same problems of baseline Cilk, i.e., recognizing coarser dependencies as each one of the enclosed spawns might depend, to a different degree, on the spawns in *child spawn set* of the root *specPoint*.

To overcome this, we differentiate spawns enclosed within a single root *specPoint* from each other and track them individually. A naive way to add this to the existing controller would be track the path from the root spawn of a *specPoint* to each enclosed spawn and use it to query the predictor. Instead, we mark task boundaries (using the API in Section 7) within each Cilk procedure, use them to differentiate between work units within a *specPoint*, and track them.

```

specPoint p : tuple {spawn, recursion_level,
    task_count_from_root_of_speculation}

p.successBoundary: a snapshot of p.boundary at which
speculation was successful

on_spec_success(specPoint p):
  for all spawns s in p.lastSpecBoundary.keys():
    if p.lastSpecBoundary[s] < p.successBoundary[s]:
      p.successBoundary[s] = p.lastSpecBoundary[s]
  p.lastSpecBoundary = NULL

//pb is the current snapshot of p.boundary being queried
for speculation potential
can_deep_speculate (specPoint p, progressBoundary pb):
  for all spawns s in pb.keys():
    if p.successBoundary[s] > pb[s]: return NO
  p.lastSpecBoundary = pb
  return YES

```

Listing 2: Predictor for deep speculation

The CilkSpec runtime takes a backup at the beginning of a speculation and relies on the predictor to communicate to it

Figure 3: Activation frame state transition diagram

when additional backups need to be taken under *deep* speculation. To make this decision, we track (per *specPoint*) a *success boundary*—the *progress* value per spawn in *child spawn set* where we have observed a speculation success—and we update it by observing successes as depicted in Listing 2. If the current *progress boundary* is greater than *failure boundary* but less than the known *success boundary*, the predictor informs the runtime to take a backup.

6. CilkSpec IMPLEMENTATION

Cilk activation frame state transitions. The life cycle of a task in Cilk can be described in terms of its activation frame’s state transitions, shown in Figure 3. The solid arrows and ellipses correspond to a frame’s state transitions in Cilk, while the dashed arrows and ellipses denote the states and transitions introduced by our speculation framework. A spawn invocation of a task (a function) results in the creation of an activation frame in the Running state. When the task terminates by returning to its invoking function, the frame transitions to the Returning state and is subsequently destroyed. When a task’s execution reaches a *sync* statement, the executing worker checks if any task spawned by the task has not yet returned. In the absence of work stealing, there would be no outstanding spawns because the depth-first execution order would execute all spawned tasks before encountering any subsequent statements. When workers steal portions of work, one or more spawned tasks may still be under execution by other workers. A worker cannot execute past such a *sync*. Therefore, the task activation frame is transitioned into a Suspended state with the worker moving onto other non-suspended tasks that can be stolen and executed. When a worker completes executing a task, it returns to the task that spawned the given task (referred to as the task’s parent). If the parent frame is suspended and this task is the last child to return, the parent frame can now continue its execution. The worker performs a *provably good steal* of the parent frame, transitions the parent frame into Running state, and proceeds to execute it.

6.1 CilkSpec activation frame states

Our speculation framework adds three new activation frame states: Speculating, Spec Suspended, and Adjudicate Ready. When a worker encounters a *sync* with outstanding children in the presence of speculation support, it determines if it is

a promising candidate for speculation. If the predictor determines that this `sync` is a viable speculation candidate in the present execution state, rather than suspend the frame, the worker backs up the current frame (*backup_frame*) and proceeds with execution. When the worker finishes the speculation phase and all outstanding children have returned, the frame enters the Adjudicate Ready state. When conflicts are detected, the speculation is discarded, and the execution resumes at the statement immediately following the `sync` statement. If no conflicts are detected, the speculation is treated as being valid, and the worker resumes execution of the `sync` from the instruction that follows the last instruction executed in the speculative state. In either scenario, the frame transitions to the Running state and is processed as a normal, non-speculative frame.

At the end of the speculation phase, we could have a stack of activation frames saved along with other speculation metadata, such as reads and writes associated with this speculation. In addition, it is possible that the speculator does the last unit of work in a frame. We do not allow such a frame to return to the parent frame. We will relax this in the *para* flavor of CilkSpec. A suspended `sync` traditionally resumes by the last child frame that completes. Under CilkSpec, either the last child frame or the speculation on the frame might complete later. If speculation is still underway when the last child frame completes, we mark the frame as Adjudicate Ready to signal the speculating thread to stop speculation. The last returning child or the speculator, whichever completes last, adjudicates the speculation and resumes executing the frame. Adjudication involves checking if any data read while speculating was modified by another thread (Section 7). If the speculation is successful, backup frame and speculation metadata are destroyed. The speculation stack is rewound, and frame is resumed from where it was left off at end of speculation. However, if a mis-speculation is detected, the speculation stack, along with speculation metadata, is destroyed, and the backup frame is restored. Child frames update the parent frame as they complete. Under CilkSpec, these updates need to be made to the backup frame as well. Note that frame backups preserve correctness for Cilk runtime data only. Any application data embedded in Cilk frames will have to be versioned and tracked using our Speculation API described in Section 7.1.

6.2 Delayed speculation

In baseline Cilk, besides the thief, a `sync` is inspected only by the child frames it depends on. This severely limits the ability to speculate, especially for frames waiting at a `sync` with a single outstanding child frame. Completion of such a child frame marks completion of `sync` and precludes any speculation. If the `sync` is not ready to be speculated on at any of these inspection points, the opportunity to speculate is lost. If the computation the predictor is waiting on has not progressed enough (Listing 1), the predictor may deem a `sync` not ready for speculation.

To overcome this limitation, we propose *delayed* speculation. Under this optimization, if a `sync` is found to be not-speculatable, we add it to the *delayed queue*. We maintain one such queue per worker thread. Idle workers first try to perform steals to look for normal work, and, after a user-defined threshold of steal failures, they look for `syncs` in *delayed queue* and check if they are ready for speculation. If

they find a speculatable `sync`, they begin speculation on it. The last completing child frame removes it from the *delayed queue* before resuming it. We observe that this optimization substantially increases the amount of work done speculatively as it tracks a suspended `sync` throughout its lifetime. Currently, we wait for a number of workers' worth of steal failures before inspecting the *delayed queue*. Intuitively, our heuristic looks at all other workers to find non-speculative work before seeking speculative work.

6.3 Deep speculation

Speculation for recursive programs can expose huge chunks of work to be done speculatively. While the single-step speculation already described keeps the complexity of predictor and speculation machinery in check, the amount of speculation it delivers, in practice, does not amortize the overhead associated with speculation. To address this, we propose *deep* speculation, where speculation continues until either the predictor decides the speculator should stop, the dependencies of the frame complete, or the speculation machinery cannot support further speculation.

Recall that we backup a frame when beginning a speculation phase. If even under *deep* speculation we took a single backup, we will have to throw away the speculation in its entirety on a mis-speculation. As we do deeper speculation, the predictor is more likely to explore paths it has not seen before. In these cases, the predictor tries to optimistically speculate. Not to hinder the predictor, but, at the same time, not to throw away useful work, we take multiple backups in *deep* speculation. We rely on predictor feedback to inform us when a speculation decision is exploratory. In such cases, we backup the frame at the top of the current Cilk stack. Thus, at the end of speculation, we could have a collection of speculation phases, each with a corresponding backup frame and associated speculation metadata (reads, writes, etc.), together with multiple distinct restart points for this computation, corresponding to the different backup frames.

The adjudication protocol is modified to account for multiple backup frames. The speculation phases are adjudicated one after another in the order of their creation. A mis-speculation in the first phase causes the subsequent phases to be discarded as well. On the other hand, a mis-speculation only in the second phase will preserve the work done in the first phase, resuming computation at the second backup frame. As described in Section 8, *deep* speculation significantly boosts the amount of work done speculatively.

6.4 Parallel speculation

While *deep* speculation enables a far greater extent of work to be done speculatively, all of this work is done in series, limiting the performance benefit that can be achieved. To overcome this effect, we allow steals from speculative workers in a flavor of CilkSpec we term *para*. Cilk already has the mechanism to allow steals from workers, which we use and augment to take necessary actions for speculation. A worker stealing from a speculator also enters speculation mode, backing up the stolen frame before continuing execution. A `sync` encountered with incomplete speculative child frames transitions to the Spec Suspended state and suspends its execution. All speculative child frames return to a speculative parent before resuming the parent frame's execution. In baseline Cilk, returning child frames are destroyed, but

under *para*, the child frames and all associated speculation metadata are retained until they are adjudicated.

Adjudication protocol in *para* is more involved than in *deep* because at the end of speculation we have a speculative steal tree to adjudicate. We do a post-order traversal of the speculative steal tree at adjudication. Post-order traversal visits frames in their sequential order, which is required as a mis-speculation in a frame might require subsequently created frames to be discarded. This also requires us to maintain an order among speculative child frames as they must be visited in their sequential order of creation as well. Mis-speculation in a frame also will cause all activation frames that causally follow it in terms of dependences in the original Cilk program to be discarded. An activation frame *f2* following another *f1* is causally dependent on *f1* if *f1* and *f2* are separated by synchronization (a *sync* statement).

6.5 CilkSpec execution example

Figure 4 illustrates adjudication of a speculative steal tree. On the left, we depict four steal events that lead to the speculative steal tree on the right. A frame (*F1*, *F2*, etc.) is depicted with both how far it has progressed (dark gray) and its independent sub-units that can be stolen (*F1a*, *F1b*, etc.). Each worker has a stack of frames ((*F1*, *F2*) in *W0*, (*F1*, *F3*, *F5*) in *W1*, etc.) associated with it. Worker *W1* steals frame *F1* from *W0*. Because this frame is at a *sync*, *W1* backs up this frame before beginning its speculation. In turn, worker *W1* creates frames *F3* and *F5*, of which it also backs up frame *F3*. Recall that under *deep* speculation, a worker can backup the frame at the top of its stack per feedback from the speculation predictor. Subsequently, worker *W2* steals frame *F1* from *W1*, worker *W3* steals frame *F3* from *W1*, and, finally, worker *W4* steals frame *F3* from *W3*. At each steal, the speculators backup the stolen frame before continuing. In addition, they could optionally backup the top frame of their stack based on predictor feedback.

On completion of normal worker *W0*, this speculative steal tree is ready to be adjudicated. All backed-up frames are preserved until adjudication, when they are either restored to become normal frames or destroyed. On the right of Figure 4, we depict the post-order traversal of this tree. Notice that the speculation by *W3* is deemed incorrect, causing speculation by worker *W4* to also be discarded. This is because the restoration of *F3* to its second backup causes any further speculation in this frame to be discarded (i.e., work by *W3*). On the other hand, only part of speculation done by *W2* is deemed incorrect, requiring only the restoration of frame *F4*'s backup, which is still present in *W2*'s local stack.

7. VERSIONING AND CONFLICTS

In this section, we present our data versioning and conflict detection strategy that aid the speculation schemes.

7.1 Speculation API

We employ a user-level API that exposes the read and write accesses to shared data structures. This API allows the runtime to track access to data blocks and transparently manage pointers to support various speculation actions. Recursive programs typically employ a coarser base case, and we instrument the program at this granularity. This approach is similar to API support for optimistic concurrency employed in other contexts [7].

Tasks request pointers to the shared data structures using the following data management API:

- `createRegion(id, size)`: Create a block of size elements identified by `id`.
- `readAccess(id)`: Obtain a pointer to a block identified by `id` for reading.
- `writeAccessBegin(id)`: Obtain a pointer to a block identified by `id` for writing.
- `writeAccessEnd(id)`: Mark the end of writes to a block identified by `id` by this task.
- `taskEnd()`: Mark this task's completion (or return).

7.2 Detecting and managing conflicts

A speculation phase conflicts with the rest of the concurrent computation when both the speculation phase and concurrent computation access the same data block, and one of them writes to that block. Tracking these read and write access sets can be expensive. We exploit certain key characteristics of recursive parallel programs to optimize the cost of tracking read and write accesses.

OBSERVATION 1. *In a data-race free recursively parallel program, speculation on a `sync` statement in a given task *A* cannot conflict with other tasks non-speculatively executing and in parallel with *A*.*

By construction, two tasks, *A* and *B*, executing in parallel are concurrent. In a data-race free program, these concurrent tasks cannot have conflicting data accesses. Consider speculation performed within a task *A*. This speculated work is concurrent with any work underway in task *B*, and thus cannot conflict with any computation in task *B*.

```

Block b: { void *normal, void *speculative, int version, int
id}
on speculative read access to block b:
  if b.speculative exists:
    rb = b.speculative
  else:
    rb = b.normal
  readSet.add((b,rb, b.version))
  return rb

on speculative write access to block b:
  new_b = create new Block
  cur = speculative read access to block b
  copy cur to new_b
  readSet.add((b,cur,b.version))
  writeMap[b] = new_b
  return new_b

on end of speculative write access to block b:
  b.speculative = writeMap[b]

on normal read access to block b:
  return b.normal

on normal write access to block b:
  return b.normal

on end of normal write access to block b:
  b.version += 1

```

Listing 3: Version management in CilkSpec

Therefore, conflicts only need to be detected between the speculative work following a *sync* and any concurrent work immediately preceding the *sync*. In a recursive parallel program, the set of data accesses (read and write) by a task

Figure 4: Creation and adjudication of a speculative steal tree, BF1[1] = First backup of frame F1

is a union of the accesses within its task body and the accesses performed by its spawned tasks. This information can be gathered as part of the task return step. However, this metadata collection can be extremely inefficient and introduce significant overheads during normal execution.

We employ a versioning scheme to track non-speculative updates to data blocks. Each data block starts with version 0. Every task writing to a data block increments its version number. In a data-race free program, writes to a data block are serialized. Therefore, this version number update can be performed without expensive locks or atomic actions.

A speculative thread performs its updates on a shadow version of the data blocks and tracks the version numbers of any data blocks it reads or writes. When all computations preceding the sync being speculated—the task’s outstanding children—complete, the current version numbers are compared with those accessed in the speculation phase. A difference in the version numbers indicates a conflict.

```

is in conflict:
for all (b, ptr, version) in readSet:
    if not (b.normal == ptr and b.version == version):
        return true
    return false

accept speculation:
for all (b) in writeMap:
    tmp = b.normal
    b.normal = writeMap[b]
    recycle(tmp) /*free for later use*/

discard speculation:
for all (b) in writeMap:
    recycle(writeMap[b]) /*free for later use*/

```

Listing 4: Conflict detection in CilkSpec

Listing 3 shows the actions on the data blocks. For each data block, we track the current version number and pointers to the normal and speculative buffers. Non-speculative reads and writes to a data block directly access the pointer and incur little overhead. Speculative reads track the pointer and version number of data block being accessed. Speculative writes create a private buffer for the current speculation phase. Upon completion of a task’s write, the speculative pointer is updated to forward this buffer for subsequent use in the same speculation phase. While each speculative writer gets a separate version to write to, the versioning scheme only tracks the latest speculative version to be supplied to any speculative readers.

Benchmark	Problem size	Block size	Task count	Serial time (secs)
LCS	256K	256	1,000K	240
RNA	8K	16	131K	236
Wave2d	65K	256	1,000K	60
Seidel2d	65K	256	1,000K	69
Wave3d	2K	32	256K	114

Table 1: Benchmarks

Listing 4 depicts the conflict detection algorithm. On becoming adjudicate ready, i.e., when all dependencies of a speculated sync are satisfied, conflict detection is performed. Recall that a speculation is incorrect if any read data have been modified by some other thread. A conflict is detected if the read pointer and version do not match the normal pointer and current version of the data block. A change in the normal pointer indicates that a speculator wrote to this data block, and the speculation was accepted. On the other hand, a change in version implies that a normal writer wrote to this data block. On a mis-speculation, the speculative pointer is recycled. However, on a successful speculation, the normal pointer is replaced with the speculative pointer, and the old normal pointer is recycled.

8. EXPERIMENTAL EVALUATION

We evaluate our approach on a 32-core Intel Xeon system with four eight-core X7560 CPUs. We implemented our speculation scheme in MIT Cilk version 5.4.6. All benchmarks were compiled with GNU gcc version 4.1.2 and optimization ‘-O2’. Table 1 lists the benchmarks used along with their problem sizes and single-threaded execution time. To balance the amount of concurrency exposed and the performance achieved by the sequential kernel, recursive programs typically switch from a parallel to an optimized serial implementation once a particular problem size cut-off (base case) is reached. We implement efficient tiled versions of the benchmarks. For each benchmark, we choose the tile/block size that achieved the best performance for Cilk. Read/write instrumentation is added at the tile granularity to reduce instrumentation overhead. We observe no additional overhead due to instrumentation. Speedups are plotted against this instrumented baseline Cilk version.

As seen in Table 1, LCS computes the longest common subsequence. RNA solves the RNA folding problem. Seidel2d implements the two-dimensional iterative Gauss-Seidel algorithm. Wave2d is representative of a typical wavefront

algorithm. These benchmarks are candidates that exhibit constrained expression of concurrency in recursive-style programming. To evaluate the overhead of various speculation schemes in the presence of abundant concurrency, we select a three-dimensional wavefront example (Wave3d) with a sufficiently large problem size.

For each benchmark, we compare the performance of the baseline Cilk implementation with four speculation schemes: (1) *always*, a scheme that always speculates when a *sync* is about to be suspended without relying on the predictor; (2) *pred*, the single-step speculator based on the predictor; (3) *deep*, the deep speculation scheme; and (4) *para*, the parallel speculation scheme. The latter three schemes also include the delayed speculation optimization. We benchmark each scheme separately. Note that the schemes themselves are inclusive (though not necessarily additive in their effect): *deep* subsumes *pred*; *para* includes *deep*.

8.1 Work inflation

In a non-uniform memory access (NUMA) system, despite improved cache locality enabled by recursive execution, random load balancing in Cilk can lead to significant NUMA-remote accesses. This can increase work time compared to NUMA-aware execution. This *work inflation* can lead to performance degradation despite load balance and low idle times. Figure 5 depicts the work inflation in baseline Cilk for all benchmarks with increasing core counts. In general, work inflation increases with core count. Some benchmarks, such as Wave2d and Seidel2d, suffer high work inflation. Work inflation limits the speedups Cilk obtains, which, in turn, limits the scalability feasible with speculation.

As they primarily strive to expose available work sooner, the speculation strategies presented cannot address work inflation. Work inflation can be addressed using NUMA-aware scheduling strategies [9]. The scalability achieved by all schemes—unmodified Cilk and the speculation schemes—will improve with lower work inflation. Our work is complementary to these strategies in that we optimize idle times.

8.2 Suboptimal work discovery

Figure 11 shows idle time as a percentage of work time for baseline Cilk (labeled *cilk*). We observe up to 200% idle time. Wave3d is an exception with very little idle time as intended by its choice of large problem size exhibiting significant concurrency. While the majority of the idle time in the remaining benchmarks is due to stricter dependencies imposed by recursive-style programming, we observe another significant contributing factor.

Cilk’s random work stealing policy delivers its provable time guarantees as long as there is sufficient concurrency. Intuitively, in the presence of significant amounts of parallelism, all workers will quickly find another active worker to steal from. This is not necessarily true in a concurrency-constrained environment. When only a few workers have work, locating them through a random search could be expensive. In practice, we observe several unsuccessful steals that make locating work expensive. To address this scalability bottleneck, we incorporate an optimization to improve the time it takes to look for work. An idle worker inspects the state of all workers in a lock-free fashion before attempting an actual steal. This cost, incurred only when a worker attempts to steal, is often cheaper than locking a worker only to find it has no work. In Figure 11, we observe that Cilk

with this optimization, labeled *cilk-wd*, improves idle time compared to Cilk baseline with an amplified difference as the core counts increase. While useful in most cases, such a work discovery aid is not guaranteed to improve performance as evident with regards to RNA.

8.3 CilkSpec efficacy: a bird’s eye view

Figure 11 depicts how various flavors of CilkSpec consistently bring down the idle time compared to the baseline. Although the *always* flavor helps reduce idle time by speculating on every *sync* to be suspended, predictor guidance helps *pred* bring down idle time far more than *always*. The *deep* flavor helps reduce the idle time further with *para* bringing down the idle time the most.

Bringing down idle time has a direct impact on performance. The speedups obtained are shown in Figures 6 to 10. With the exception of Wave3d, we see that predictor-guided speculation schemes scale much better than *cilk-wd*. For LCS, while *cilk* gets about 13× speedup on 30 cores, *para* gets more than 19× speedup. For some of these benchmarks, both *cilk* and *cilk-wd* show negative scaling beyond 16 cores. However, *para*—and, in some cases, *pred* and *deep*—scales, at least, up to 24 cores. Beyond 24 cores, high work inflation also limits the scalability achieved by the speculation schemes. As mentioned earlier, the Wave3d benchmark has little idle time, and CilkSpec neither helps nor hurts its performance.

Despite reducing idle time, the *always* flavor performs worse than *cilk-wd*. The *always* scheme speculates on every suspended *sync*. While this reduces idle time, the injudicious choice of *when* to speculate leads to high discard rates, making its performance worse than other schemes that exhibit very low discard rates.

Note that baseline Cilk and *cilk-wd* show similar scaling behavior at lower core counts. With increasing core counts, the system gets more work constrained, and Cilk scales poorly compared to *cilk-wd*.

8.4 CilkSpec detailed analysis

In this section, we present a detailed analysis of CilkSpec, shedding more light on the reasons for overall performance observed in Section 8.3. In particular, after highlighting the effect speculation has on idle time, we analyze its effect on work time and the overheads introduced. Note that together work time, idle time, and overheads constitute the overall application execution time.

Figure 12 shows the increase in work time (normal and speculatively accepted work) compared to *cilk-wd* for various speculation schemes. Unlike the data in Figure 5, which uses single-threaded Cilk as baseline, the comparison here is against the *cilk-wd* work time for corresponding core count. This figure intends to gauge if any additional work inflation is added to the baseline Cilk by the speculation schemes, where part of the work is done speculatively and adds versioning overhead. As the figure demonstrates, the CilkSpec infrastructure does not inflate work any more than baseline Cilk. Two competing effects influence work inflation. First, speculating past a *sync* improves cache locality. Second, speculative work costs more than normal work as it involves versioning overhead. In general, with increasing core counts, we observe that better locality triumphs versioning overhead, reducing work inflation. In the worst case, speculation results in a roughly 4% increase in work time.

Figure 5: Work time inflation (all threads)

Figure 6: LCS speedup

Figure 7: RNA speedup

Figure 8: Wave2d speedup

Figure 9: Seidel2d speedup

Figure 10: Wave3d speedup

Figure 11: Idle time (all threads) as a percentage of work time.

Figure 12: Work time (normal work and speculatively accepted work) inflation (all threads) for CilkSpec flavors

Figure 13: Overhead added (all threads) by CilkSpec flavors

nproc	LCS			RNA			Wave2d			Seidel2d			Wave3d		
	Pred	Deep	Para	Pred	Deep	Para	Pred	Deep	Para	Pred	Deep	Para	Pred	Deep	Para
16	5.9	13.9	17.9	10.8	20.6	32.7	6.5	14.3	18.7	6.9	15.4	18.7	0.2	1.3	3.2
24	8.5	19.5	22.0	13.3	25.9	38.1	9.2	19.4	22.8	10.1	20.6	24.0	0.3	1.5	3.9
30	10.1	22.3	26.1	14.8	27.3	39.6	10.5	20.1	25.2	11.5	22.3	26.7	0.3	1.7	5.1

Table 2: Percentage of work done under speculation

Table 2 depicts the percentage of work done speculatively under different CilkSpec flavors. The general trend here is that *deep* exposes more work sooner compared to *pred*, while *para* exposes a significant chunk of work ahead of time, as high as 40%. Also, as the processor count increases and the system is more work constrained, more and more work gets exposed speculatively.

While speculation helps reduce idle time by exposing work ahead of time, it does have concomitant overheads that limit the improvements it delivers. We plot Figure 13 to show the overheads added by speculation. This figure depicts three sources of overhead as percentages of *cilk-wd* work time for a particular core count. These three sources are namely: *decide*, *discard*, and *other*. The time spent doing speculative work that is discarded due to misprediction is depicted as *discard*. All of the speculation setup cost, including adjudication and saving and resuming of speculative work, is clubbed under the bucket *other*. The time spent deciding whether or not to speculate is included under *decide*.

In general, the setup cost for speculation is very little as depicted by the *other* bucket. The *discard* cost is high for the *always* scheme, which does not selectively choose *when* to speculate. This limits the performance delivered by the *always* scheme. This cost is completely optimized by the CilkSpec predictor. The amount of time spent under *decide* is the only overhead that shows up considerably in all predictor-guided speculation schemes, which can be up to 12% in the case of *para*.

8.5 Predictor analysis

Figure 14 depicts the percentage of decisions made by the speculation predictor that are mispredictions. The misprediction rate for *pred* is negligible. This shoots up for *deep* as the predictor explores more speculation opportunities. The misprediction rate is still less than about 10%, except in Wave3d. Recall that, in Wave3d, we barely spend any time speculating. As such, the absolute number of tasks speculated is really small. In the initial execution phase, the predictor encounters mispredictions as it explores the de-

Figure 14: CilkSpec predictor accuracy

Figure 15: CilkSpec predictor entries

pendencies. These get amortized with subsequent correct decisions. In Wave3d, we barely speculate and do not see this effect. The misprediction rate further drops in *para* because we do more work speculatively, which reduces the overall share of mispredictions.

8.6 Space overhead

Recall from Section 7 that CilkSpec creates copies of data to support writes under speculation. Table 3 shows the amount of data allocated to support this feature as a percentage of total application data footprint for 30 cores. The table only shows the footprint for data buffers and not for

Benchmark	Pred	Deep	Para
LCS	0.21	1.39	1.64
RNA	0.39	1.00	1.67
Wave2d	0.17	1.08	1.29
Seidel2d	0.19	1.04	1.38
Wave3d	0.09	0.61	1.41

Table 3: Data footprint of CilkSpec for 30 cores (percentage of application data footprint)

backup frames, which are far smaller in size. Furthermore, we present data only for 30 cores, the core count at which we see maximum speculation. We see that the amount of space needed by CilkSpec is small. As we speculate more from *pred* to *para*, the space needs increase but are still less than 2%. Judicious speculation (preferring normal work) and recycling buffers keep the footprint manageable. Note that space usage can be controlled by a user-provided bound.

In Figure 15, we depict the total entries allocated in our predictor for various CilkSpec flavors across different processor counts. The number of entries allocated increases as the predictor explores more dependencies from *pred* to *para*. For a given problem size, the number of entries is largely independent of the processor core count. Thus, increased speculation at 30 processors does not add to the number of entries created by the speculation predictor.

9. RELATED WORK

Cilk [4] is an exemplar of a large number of parallel programming models that support recursive-parallel programming (e.g., OpenMP [2], Java Concurrency Utilities [8], Intel Thread Building Blocks [10], and X10 [11]). These models have been used in the design of cache-oblivious algorithms (e.g., [3]). Work-first and help-first schedulers explore different strategies in executing the computation graph [4, 6]. These choices differ on the order in which the tasks are executed and exposed for stealing. Neither impacts nor addresses synchronization constraints or critical path lengths. The task graph structures supported by Cilk-like programming models can be modeled as series-parallel dags [14]. Task graph schedulers, such as data-driven futures [13] and Nabbit [1], support a more general task graphs, but they do not support Cilk’s provable space bound and lose the benefits of recursive execution, such as cache-obliviousness.

The cache-oblivious wavefront scheme [12] addresses concurrency constraints in recursive parallel programs by exploiting application information to eagerly enable ready tasks. In the absence of application information, we cannot make forward-looking decisions about ready tasks. We employ speculation in an application-agnostic fashion to automatically identify such ready tasks. Speculation support for recursive programs designed by Gao et al. [5] employs helper threads to execute past synchronization points but does not focus on the design of an efficient and accurate predictor.

10. CONCLUSION

The primary impediment to wide acceptance of parallelism and harnessing of the now ubiquitous multi-cores is the inherent difficulty in writing parallel programs. While we believe that multi-threaded runtime systems such as Cilk ease this difficulty by providing useful abstractions to express parallelism, we observe how the dependencies expressed in Cilk can be overly constrained and lead to poor scalability.

To overcome this performance roadblock without burdening the programmer, we propose and evaluate optimistic concurrency in the Cilk infrastructure. Armed with a gamut of optimizations and an accurate predictor, we show that optimistic concurrency is not only viable but delivers gains of up to 1.6 $\times$ on 30 cores over baseline Cilk.

11. ACKNOWLEDGMENTS

We would like to thank the reviewers for their helpful suggestions and comments. This material is based on work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under contract number 63823 and by the National Science Foundation under the CAREER-1149773 award.

12. REFERENCES

- [1] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work-stealing. In *IPDPS*, pages 1–12, 2010.
- [2] E. Ayguadé, N. Copt, A. Duran, J. Hoeftinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of OpenMP tasks. *TPDS*, 20(3):404–418, 2009.
- [3] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.
- [4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, 1998.
- [5] L. Gao, L. Li, J. Xue, and T.-F. Ngai. Exploiting speculative tlp in recursive programs by dynamic thread prediction. In *Compiler Construction*, pages 78–93, 2009.
- [6] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS*, pages 1–12, 2009.
- [7] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.
- [8] D. Lea et al. Java specification request 166: Concurrency utilities, 2004.
- [9] J. Lifflander, S. Krishnamoorthy, and L. V. Kalé. Optimizing data locality for fork/join programs using constrained work stealing. In *SC*, pages 857–868, 2014.
- [10] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O’Reilly Media, 2007.
- [11] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *PPoPP*, pages 271–271, 2007.
- [12] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *PPoPP*, pages 205–214, 2015.
- [13] S. Tasirlar and V. Sarkar. Data-driven tasks and their implementation. In *ICPP*, pages 652–661, 2011.
- [14] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. In *STOC*, pages 1–12, 1979.