Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism

ALEXANDROS TZANNES, University of Illinois, Urbana-Champaign
GEORGE C. CARAGEA, UZI VISHKIN, and RAJEEV BARUA, University of Maryland, College Park

10

Lazy scheduling is a runtime scheduler for task-parallel codes that effectively coarsens parallelism on load conditions in order to significantly reduce its overheads compared to existing approaches, thus enabling the efficient execution of more fine-grained tasks. Unlike other adaptive dynamic schedulers, lazy scheduling does not maintain any additional state to infer system load and does not make irrevocable serialization decisions. These two features allow it to scale well and to provide excellent load balancing in practice but at a much lower overhead cost compared to work stealing, the golden standard of dynamic schedulers. We evaluate three variants of lazy scheduling on a set of benchmarks on three different platforms and find it to substantially outperform popular work stealing implementations on fine-grained codes. Furthermore, we show that the vast performance gap between manually coarsened and fully parallel code is greatly reduced by lazy scheduling, and that, with minimal static coarsening, lazy scheduling delivers performance very close to that of fully tuned code.

The tedious manual coarsening required by the best existing work stealing schedulers and its damaging effect on performance portability have kept novice and general-purpose programmers from parallelizing their codes. Lazy scheduling offers the foundation for a declarative parallel programming methodology that should attract those programmers by minimizing the need for manual coarsening and by greatly enhancing the performance portability of parallel code.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Runtime environments, optimization*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Algorithms, Performance, Languages

Additional Key Words and Phrases: Work stealing, adaptive scheduling, load balancing, nested parallelism, task-parallel, fine-grained, declarative, performance portability

ACM Reference Format:
Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. 2014. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. ACM Trans. Program. Lang. Syst. 36, 3, Article 10 (August 2014), 51 pages.
DOI: http://dx.doi.org/10.1145/2629643

This research was partially supported by the National Science Foundation under grants CSR-0834373 and CNS-1161857.
Authors' addresses: A. Tzannes, Department of Computer Science, University of Illinois at Urbana-Champaign, Thomas M. Siebel Center for Computer Science, 201 N Goodwin Ave, Urbana, IL 61801; email: atzannes@illinois.edu; G. C. Caragea, Department of Computer Science, University of Maryland, A. V. Williams Building, College Park, MD 20742; U. Vishkin, The University of Maryland Institute for Advanced Computer Studies, College Park, MD 20742; email: vishkin@umd.edu; R. Barua, 1431 A. V. Williams, Department of Electrical & Computer Engineering, University of Maryland, College Park, MD 20742; email: barua@umd.edu.

DOI: http://dx.doi.org/10.1145/2629643

1. INTRODUCTION

For close to a decade now, the clock speeds of general-purpose processors have remained stagnant, and, instead, increased parallelism is the main avenue for improving single-program performance. Nevertheless, parallel programming models are still inadequate for general-purpose programmers who value most ease of programming and performance portability because they improve productivity. Certainly, good speedups are also necessary to justify the effort of parallelizing code.

One of the tenets of general-purpose programming is *modularity*, the ability to combine and reuse code (e.g., from libraries) to write large programs without reinventing the wheel every time. For parallel code, that means supporting efficiently *nested parallelism*, the ability to create more parallelism from an already parallel context. This need gave rise to task-parallel languages and libraries such as Cilk [Frigo et al. 1998], Java Fork/Join [Lea 2000], Intel's TBB [Robison et al. 2008], Microsoft's TPL [Leijen et al. 2009], and many others. In task-parallel programs, the programmer expresses parallelism through high-level constructs that create tasks (e.g., parallel loops, parallel reducers, futures, etc.) and have implicit synchronization and scheduling. Explicit synchronizations, which are hard to reason about, are still possible but less frequently needed. This greatly simplifies parallel programming and to some degree decouples it from the target platform.

In addition to modularity, nested parallelism is also useful when parallelism gradually becomes available during computation and the original parallelism is insufficient or imbalanced: Decomposing outer parallel tasks into subtasks can result in improved load balance and increased parallelism, leading to better performance.

Dynamic scheduling is necessary to reap the benefits of nested parallelism because of its dynamically unfolding nature. *Work stealing* [Blumofe and Leiserson 1999] is currently the golden standard of dynamic scheduling for task-parallel languages because of its efficiency and scalability, at least at the scale of shared memory platforms that we consider in this article. Like all dynamic scheduling, however, work stealing must pay some runtime scheduling overhead per task, which prevents very fine-grained tasks from being executed efficiently. Currently, programmers are responsible for controlling the granularity of tasks to rein in scheduler overheads. This is achieved either by not exposing all available parallelism or by serializing some of the exposed parallelism. We refer to both as *coarsening* of parallelism.

As we elaborate in Tzannes [2012a], coarsening has two goals: *amortizing* scheduler overheads per task and *pruning* parallelism. Intuitively, amortizing is needed to avoid exposing tasks that are too short, and pruning is needed to avoid exposing too many tasks for the target platform. In the first case, the scheduling overheads but also other characteristics of the platform (e.g., communication latency and bandwidth between cores) make the profitable exploitation of parallelism that is too fine-grained impossible. For example, advice such as "create tasks with at least X operations" is aimed at achieving the amortization goal. In the second case (pruning), the scheduling overheads are wasted because most of the exposed parallelism is superfluous and cannot be exploited by the platform at hand.[1] For example, advice such as "create approximately $8 \cdot P$ tasks" aims at pruning excessive parallelism. Amortizing is relatively simple because it only involves coarsening *leaf tasks* (the coarsened tasks do not have nested tasks) to a fixed granularity. It involves detecting very fine-grained tasks and combining a small number of them. Pruning is much harder because it depends on how much parallelism a code exposes, which typically varies with its input data D, on how much parallelism a platform can support in hardware M (e.g., how many cores it has), and

[1]It could, nevertheless, be useful on a larger platform.

on what fraction of the platform is available to execute the code. This last parameter depends on the calling context C of the code among other factors (e.g., it also depends on what other programs are running concurrently, in the case of multiprogrammed systems, which is the norm for general-purpose platforms). If the code is called from a sequential context, perhaps the entire platform is available; but when called from a parallel context, only a couple of cores may be free. In short, pruning depends on three parameters (D, M, C).

Currently, programmers are responsible for coarsening for both pruning and amortizing. This is problematic for two reasons: first, it is tedious, and second, since pruning depends on the three parameters (D, M, C), manual coarsening often results in code that has been overfit for some values of those parameters, thus harming performance portability. Some measure of performance portability is possible with parametric pruning, when the programmer writes his code so that it takes into account the input and the platform. However, parametric pruning is even more tedious than naive tuning, and taking into account the context C means sacrificing modularity, which by definition hides context.

A holy grail for task-parallel programming is efficiently supporting *declarative* parallelism, where programmers are allowed—in fact, encouraged—to expose *all* available parallelism[2] without any coarsening and let the compiler, the runtime, and the platform be responsible for efficiently executing such codes. In addition to relieving programmers from tedious coarsening, declarative programming also benefits performance portability by not hiding any of the available parallelism. Unfortunately, such codes are not supported efficiently by existing languages, runtimes, and platforms, as confirmed by our experimental results, and therefore manual coarsening remains justifiably the law of the land.

Our proposed scheduling technique, *lazy scheduling,* performs pruning dynamically, by adapting to load conditions. Static compiler transformations perform automatic amortization of overheads for many commonly occurring types of codes. For example, it is possible to approximately estimate statically the work per iteration of fine-grained parallel loops (e.g., that do not contain function calls or inner loops) and pick a grain size for the loop. Furthermore, when the number of iterations of a fine-grained loop is statically unknown, the compiler can also generate a sequential version of the loop and pick the appropriate version at runtime, based on the number of iterations [Tzannes 2012a]. Together with those static transformations, lazy scheduling makes programming less tedious by practically eliminating the need for coarsening, and it makes codes more performance portable by automatically pruning parallelism, which allows for the maximum profitable amount of parallelism to remain exposed. We originally presented lazy scheduling in Tzannes et al. [2010], where we adapted it to work stealing and presented an implementation and experimental evaluation on the XMT manycore architecture developed at the University of Maryland [Wen and Vishkin 2008], a platform designed to exploit the theory of PRAM parallel algorithms by accommodating their needs [Vishkin et al. 1998; Naishlos et al. 2001, 2003; Vishkin 2011].

This article extends our prior work [Tzannes et al. 2010] in the following ways:

—We propose two new alternatives for lazy scheduling and show how they scale on three different multicores. We also show that our original method does not scale with some declarative codes.
—We implement and evaluate the three lazy scheduling alternatives just described on commercial multicores and compare them in terms of scalability and performance. We also compare them to existing work stealing approaches, both on declarative

[2]Within the confines of structured high-level parallel constructs.

and coarse codes. On declarative code, lazy scheduling achieves an average speedup between 2× and 3× compared to existing work stealing and matches its performance on coarsened code. We also find that, when parallelism is amortized (either manually or automatically), lazy scheduling achieves a *software performance optimality ratio*[3] above 85% in the worst case among the benchmarks explored and above 92%, on average, whereas work stealing achieves 51% in the worst case and 71%, on average.

—We compare different work stealing algorithms in terms of their space and time bounds when scheduling parallel loops, and we discuss some limitations of lazy scheduling in terms of synthetic worst-case scenarios.

Overall, we believe lazy scheduling constitutes a significant step toward the efficient execution of declarative code, with the simultaneous benefits of less tedious programming and improved performance portability.

2. BACKGROUND

The idea of work stealing is at least as old as Burton and Sleep [1981] and Halstead's [1984]work on functional programming, but it gained popularity with Cilk [Frigo et al. 1998] and is now incorporated in many commercial products [Lea 2000; Robison et al. 2008; Leijen et al. 2009; Leiserson 2009]. In work stealing, each worker (i.e., processor or thread) that encounters parallel work (e.g., a parallel loop or future) starts executing some of that work and places the continuation (the remaining parallel work and the rest of the parent) on a shared work-pool. When a processor runs out of work, it looks for available work on that shared work-pool. The design of the work-pool makes work stealing interesting: It consists of one double-ended queue, called a *deque*, per worker. Deques are "double-ended" because data are accessed from both ends: Each worker treats its own deque as a stack, accessing it from one end, but it treats all other deques as queues, accessing them from the other end, when its own deque is empty. Owning one end of the local deque greatly reduces the need for synchronization for workers accessing their own deque.

A worker pushes parallel tasks it encounters onto its deque and pops tasks when it runs out of work, treating its own deque as a stack. When a worker runs out of work and its deque is empty, it becomes a *thief*: It picks a *victim* worker at random and tries to *steal* a task from its deque. Popping the newest task from the local deque results in *depth-first execution*, and stealing the oldest task from a victim deque results in *breadth-first thefts*.

Four major benefits of work stealing are *(1)* depth-first execution promotes temporal and often spatial locality and *(2)* keeps the stack footprint bounded relative to a sequential execution of the same program; *(3)* breadth-first thefts tend to result in stealing larger chunks of work, thereby resulting in good load-balancing while minimizing the scheduling overheads of thefts, which are expensive; *(4)* the deques can be implemented efficiently with low synchronization overheads. A disadvantage of work stealing is its stealing phase, when idle processors randomly probe deques for work, causing potentially unnecessary interprocessor communication.

Task Descriptors (TDs), also known as work descriptors, are used to describe ranges of tasks coming from parallel loops, reducers, or other parallel constructs that simultaneously create multiple tasks. The specific structure of TDs is implementation-specific, but one possible implementation is the following: The ID of the first task and the number of tasks (or the ID of the last task) can be used to represent the range; a single

[3]The *software performance optimality ratio* is formally defined in Tzannes [2012a]. Informally, it answers the question "how far is the performance of a piece of software from optimal?" because it is the ratio of the best possible execution time (e.g., with any coarsening) over the execution time achieved (e.g., with a given coarsening) for a given problem, input, and execution platform.

pointer to the code to be executed is necessary since all tasks execute the same code using the task ID (iteration ID) as a parameter; a pointer to the stack frame of the parent task is also needed to allow access to its variables and for synchronizing with the parent on completion. Optionally, TDs may contain additional fields such as a *grain size* (the minimum number of tasks the TD should contain), the maximum number of *chunks* into which to split the TD, or a cost estimate of the tasks it contains.

When it comes to scheduling TDs, there are several different ways to treat them. The two main categories include iteratively breaking off constant-sized chunks from the TD and recursively splitting it in half, which we call *Eager Binary-Splitting* (EBS). We cover these alternatives in the next subsections.

2.1. Iterative Chunking

Two commonly used approaches of iterative chunking are *work-first* and *help-first,* described here. Both have a critical path that is linear in the number of tasks N in the TD, so the recursive splitting approaches, which incur a smaller $O(\log N)$ overhead on the critical path, are usually preferred. Nonetheless, work-first can in certain situations be preferable to recursive splitting, as we show in Table II.

2.1.1. Work-First Work Stealing. The *work-first* approach, which we called *Serializing Work Stealing* (SWS) in Tzannes et al. [2010], keeps the first task of a task descriptor (or the *grain* first tasks if a *grain size* is specified) and pushes the rest onto the local deque. The drawback of work-first work stealing is that a task descriptor created by a parallel loop is never split, so its accesses by workers contending for work will be serialized.

In Section 2.3, we present a simple example to illustrate the limitations of work-first and of the other alternatives when scheduling a TD.

2.1.2. Help-First Work Stealing. The *help-first* work stealing approach treats a parallel loop as a sequential loop whose iterations each spawn one parallel task (Algorithm 1). This approach also serializes the creation of the tasks, but it creates a TD per task, allowing parallel access to them, unlike the work-first approach described earlier. By creating a TD per task (or per *grain* tasks when a grain size is provided), help-first work stealing ends up having a potentially unbounded memory footprint relative to the sequential footprint for the same code.

ALGORITHM 1: Parallel loop semantics with help-first work stealing

```
for i ∈ {1, 2, ..., N} do
  | spawn CODE(i);
end
sync;
```

Guo et al. [2010] have implemented a scheduler that adaptively switches between the help-first and work-first work stealing to get the benefits of help-first task creation while keeping the memory footprint bounded. However, the depth (critical path) of scheduling a parallel loop of N tasks is linear in N for both help-first and work-first approaches. This depth is added to the critical path of the application and can overwhelm it in some cases. The EBS approaches described here reduce this depth from linear to logarithmic in the number of tasks. Furthermore, the work by Hendler and Shavit [2002] seems to suggest that recursive splitting benefits performance by spreading tasks around in larger chunks, rather than stealing a single task at a time.

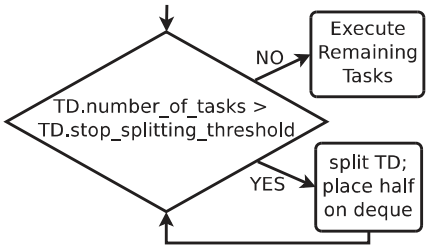

Fig. 1. Processing a task descriptor with simple partitioner.

2.2. Eager Binary Splitting (SP & AP)

Intel's Threading Building Blocks (TBB) [TBB 2008], Cilk++ [Leiserson 2009], and more recently CilkPlus [CilkPlus 2011] implement EBS work stealing schedulers: upon creating, stealing, or popping a TD, a worker splits it into two TDs of approximately equal number of tasks and pushes one on its deque; then, the worker continues iteratively splitting the remaining TD until a (manually or automatically) pre-determined grain size is reached. We call this approach *eager* because splitting proceeds regardless of runtime conditions such as load.

An important performance consideration for EBS is when to stop splitting. Although splitting TDs is helpful for creating enough parallelism and achieving load balance, excessive splitting induces unnecessary overheads, which can severely hurt performance. It can be preferable to coarsen parallelism by stopping the splitting of TDs before they are reduced to a single tasks and execute all the tasks in the coarser TDs sequentially. Finding this *stop-splitting-threshold (sst)* is hard because it depends on several factors, such as the number of available workers, the number of tasks of each parallel loop (which can be a function of the input), and the calling context (e.g., sequential vs. parallel), which is often unknown when programming in a modular way. For example, the author of a library of parallel algorithms does not know in which context they will be called.

TBB offers two options for controlling the splitting of TDs: Simple Partitioner (SP) and Auto-Partitioner (AP).[4] Cilk++ and CilkPlus only implement SP: By default, the grain size is set to $\min(K, N/8P)$ unless manually overridden.[5] This means a loop will be split into at least $8P$ TDs (assuming $N > 8P$), and TDs will have at most K iterations.

Cilk++ also has a mechanism inherited from Cilk [Frigo et al. 1998] for reducing parallelism overheads by creating two versions of functions and choosing at runtime which one to execute: the one for fast local and serialized execution with simplified synchronizations, or the one for true parallel execution that pays the full synchronization cost.[6] This mechanism is orthogonal to our proposed lazy-scheduling, and combining the two approaches would be beneficial.

2.2.1. Simple-Partitioner. Figure 1 shows how SP splits a task descriptor while the number of iterations in its range is above an *sst*, referred to as *grain size* in TBB's manual [TBB 2008]. This eagerness to split may result in an excessive number of TDs being

[4]TBB also provides affinity partitioner, which increments AP with a mechanism for improving locality for codes with consecutive parallel loops over the same data by trying to map tasks that access the same data to the same worker [Acar et al. 2000; Robison et al. 2008]. However, affinity partitioner does not offer a different way for controlling the splitting of TDs.

[5]$K = 512$ for Cilk++ and $K = 2{,}048$ for CilkPlus. N is the number of loop iterations and P the number of processors.

[6]This optimization is not currently available in CilkPlus, but it is planned in future releases.

created, which is why the programmer is expected to define an appropriate *sst* to stop the splitting earlier. The TBB manual [TBB 2008] suggests the following approach to determine the appropriate *sst*[7]:

(1) Set the *sst* parameter of the parallel loop to 10,000. This value is high enough to amortize the scheduler overhead sufficiently for practically all loop bodies but may unnecessarily limit parallelism.
(2) Run your algorithm *on one processor*.
(3) Start halving the threshold parameter and see how much the algorithm slows down as the value decreases.

⇒ A slowdown of about 5% to 10% is a good setting for most purposes.

There are two problems with this approach. *First*, it is extremely tedious. Not only does the programmer have to provide a threshold, he has to run his program several times to find the appropriate threshold. Moreover, if the code has multiple parallel loops, a different threshold has to be determined for each loop, which means more runs. Ideally we would want the 5–10% slowdown to be only compared to the code of the parallel loop, not of the whole application, so the programmer will have to isolate the parallel loops during this tuning process and time them separately. Finally, because the code will run on a single processor, this tuning process will also be very slow. *Second*, another equally serious problem with this approach is that the derived fixed threshold limits the performance portability of the code to different platforms, inputs, and contexts. More evidence that manual coarsening is tedious and harms performance portability can be found in Tzannes [2012a, chapter 3].

In conclusion, EBS with SP is an improvement over work-first and help-first work stealing because splitting task descriptors reduces scheduling overhead on the critical path from linear to logarithmic. However, determining the grain size (*sst*) manually is very tedious, and if it is a fixed constant, as suggested by TBB's tuning procedure, it harms performance portability. Work-first and help-first work stealing also have the same issue of needing a manually determined grain size. AP, described hereafter, attempts to relieve the programmer from picking a grain size (*sst*) but is only partly successful.

2.2.2. Auto-Partitioner. TBB's other option for controlling splitting, AP, splits the tasks of a parallel loop into $K \cdot P$ TDs, assuming the number of iterations in the original parallel loop is at least $K \cdot P$, where P is the number of workers and K is a small implementation-specific constant. AP replaced SP as TBB's default scheduler because it relieves the programmer from manually picking the *sst* and delivers good performance. It has two fixed parameters, K and V, as well as an additional field-per-task descriptor we call *chunks* (called *n* in Robison et al. [2008]). When executing a parallel loop and creating its TD, *chunks* is initialized to $K \cdot P$. Every time the task descriptor is split, *chunks* is also halved (Figure 2), and whenever a TD is stolen, *chunks* is set to be at least V, which gives AP some limited runtime granularity adaptivity. A TD is not split further if *chunks* ≤ 1 or if it is not divisible (i.e., contains a single task or fewer than *grain* tasks). K and V are set to four in Robison et al. [2008].

Instead of coarsening parallelism by combining tasks with the *sst*, AP uses *chunks* to determine into how many pieces to split a TD. This is preferable because it does not require programmer tuning, it allows for some platform and dataset portability by taking into account the total number of workers, and it performs better than SP in most cases. The programmer can still define an *sst* in case more aggressive coarsening

[7]Newer versions of the TBB manual dropped this section as AP, which does not require an *sst*, became the default partitioner.

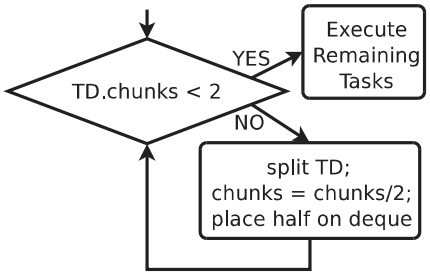

Fig. 2. Processing a task descriptor with auto-partitioner.

is required. For example, if the iterations of a parallel loop are few and fine-grained, AP might still perform excessive splitting without a manually determined *sst*.

Unfortunately, AP is context insensitive and often results in excessive splitting in the presence of nested parallelism. Although splitting iterations into $K \cdot P$ task descriptors for a parallel loop executed from the original sequential context is usually a good heuristic, if that same loop is executed in a nested context, the outer parallelism will likely suffice, and fewer chunks would be preferable. For example, for D levels of nested parallel loops of N iterations each, AP will create $\mathrm{TD}_d = N^{d-1} \cdot (K \cdot P)$ TDs for the loop at nesting depth d over the course of the execution and a total of $\sum_{i=1}^{N} \mathrm{TD}_i$ TDs, which may be excessive.[8] These TDs will not exist simultaneously in memory, but the runtime overheads of creating them over the course of the execution are still substantial. The maximum number of TDs concurrently present in the system will be in the order of $\mathrm{O}(P \cdot (\log K + \log V + (D-1) \cdot \log (K \cdot P)))$, which is $\mathrm{O}(D \cdot P \cdot \log P)$ for constant values of K and V. Reducing K to reduce the number of chunks may result in insufficient parallelism and load imbalance, especially for non-nested loops, so it is not a viable solution. Our lazy scheduling approach is context sensitive by being adaptive to runtime load conditions and overcomes the portability pitfalls of SP and AP and the serialization of parallelism creation of work-first and help-first work stealing without requiring programmer tuning.

Another potential danger with AP, even without nested parallelism, is that, once it starts executing one of the original $K \cdot P$ "fat" chunks, it will execute it to completion, without the possibility of revisiting this coarsening decision of not further splitting the TD. If the tasks of a loop are severely imbalanced, one of the "fat" chunks may contain most of the work, and the performance will suffer of poor load balancing. For that reason, the time bound (presented later) for work stealing schedules does not apply to AP. We give more details in Section 9.3.

Note that even though the default grain size for Cilk++ and CilkPlus splits TDs into $8P$ (or more) TDs, the similarity with AP ends there. The grain size is determined at the onset of the parallel loop and is not affected by thefts, as in the case of AP. Cilk++ and CilkPlus still follow the SP algorithm; they just use a grain size that is parametric in N and P.

2.3. Illustrating Example

This section illustrates how the four different options for scheduling a TD as just described work by means of a simple example. Assume we have two workers, W_A and W_B, and that W_A starts working on a TD with 16 tasks, while W_B is hungry. For

[8]These formulas correct the incorrect (but conservative) formula of $(K \cdot P)^d$ reported in Tzannes et al. [2010].

simplicity, assume that those tasks do not create nested parallelism and that no grain size *sst* was provided by the programmer, making it default to 1.

2.3.1. Work-First. W_A will create a TD with tasks 2 through 16, place it on its deque, and start executing the first task. In the meantime, worker W_B steals the TD from W_A's deque, takes tasks 2, places the remaining TD (tasks 3–16) on its deque, and starts executing task 2. W_A eventually finishes executing task 1 and looks for work on its deque, which it finds empty, so it tries to steal work from W_B; it is successful, takes task 3, places the remaining TD on its deque, and starts executing task 3. And so on.

This example illustrates four shortcomings of work-first work stealing:

(1) If two or more workers end up executing tasks from a TD, they will keep stealing the TD from each other, effectively *serializing accesses to it*.
(2) On modern multicores with private caches, thefts are expensive because they induce coherence traffic by modifying remote deques, which presumably reside in the private cache of the victim worker.
(3) Unless a grain size is provided, each time a worker needs more work, it removes a single task from a TD; this means that TDs (and thus deques) will be accessed as many times as the tasks they have, which introduces significant overheads for fine-grained tasks.
(4) Because of the implicit barrier at the end of parallel loops, tasks need to synchronize upon termination, usually by atomically decreasing a variable representing the number of pending tasks. Unless a grain size is provided, tasks are executed one at a time, and synchronization will also happen individually for each task, possibly inducing significant overheads.

2.3.2. Help-First. Worker W_A starts by pushing 15 TDs on its deque, each containing a single task, thus incurring an overhead on the critical path linear in the number of tasks. During that time, W_B steals a TD and executes that task. When W_A finishes pushing TDs, it executes its remaining task. From that point on, W_A and W_B keep consuming TDs off W_A's deque until all tasks are executed causing nontrivial cache coherence traffic on architectures with private caches.

The shortcomings of help-first work stealing are similar to those of work-first except that, instead of having serialized access to a single TD, it has a serialized TD creation linear in the number of tasks, as well as a corresponding memory overhead.

2.3.3. Simple-Partitioner. W_A splits the TD and pushes a TD with 8 tasks and repeats splitting and pushing TDs with 4, 2, then 1 task, remaining itself with a single task. During that time, worker W_B steals the TD with 8 tasks, and it splits and pushes TDs with 4, 2, and 1 task on its own deque. W_A proceeds to execute its remaining task, then pops the TD with 1 task from its deque and executes it. Then W_A pops the TD with 2 tasks, splits it, pushes half back on to its deque, and executes its remaining task. And so on.

Therefore, SP solves the problem of serialized access to TDs of work-first work stealing and that of serialized TD creation of help-first work stealing, and it reduces the number of thefts that cause unwanted coherence traffic. Nevertheless, it still pays excessive overheads for deque transactions and synchronization, which are linear in the number of tasks, and, although it reduces the time overhead on the critical path from linear to logarithmic, this added logarithmic overhead on the critical path is problematic.

2.3.4. Auto-Partitioner. Assume $K = 2$ and $V = 4$ (while having $V > K$ is not necessarily useful, we choose it here to better illustrate how AP works). W_A sets the number of chunks of its TD of 16 tasks to $K \cdot P = 4$. It splits the TD (and the number of chunks)

Table I. Shortcomings of Existing Work Stealers

Scheduler	# Thefts	Deque Transactions & Synchronizations	Overhead Added to Critical Path	Context Sensitive
Work-First	excessive	O(N)	O(N)	No
Help-First	excessive	O(N)	O(N)	No
Simple-Partitioner	good	O(N)	O(log N)	No
Auto-Partitioner	good	O(P)	O(log P)	No

and pushes half (8 tasks, `chunks` $= 2$) on its deque. It proceeds to push a TD with 4 tasks and `chunks` $= 1$, and it starts executing its remaining 4 tasks sequentially. In the meantime, W_B steals the TD with 8 tasks and `chunks` $= 2$, but it sets `chunks` $= V = 4$ because `chunks` $< V$ and the TD arrived through a theft. Then, W_B splits the TD twice, pushing on its deque TDs with 4 and 2 tasks, and proceeds to execute its 2 remaining tasks. W_A continues by popping its remaining TD, which has 4 tasks and `chunks` $= 1$. Since W_A did not acquire the TD through a theft, it does not update chunks to V, and since `chunks` $= 1$, it proceeds to execute all 4 tasks. W_B proceeds similarly, except its `chunks` will contain 2 tasks.

AP improves on SP by reducing the TD creation overhead on the critical path to logarithmic in the number of workers P and the deque transaction and synchronization to linear in P. However, AP is not context sensitive and incurs these overheads regardless of whether the parallel loop was nested or not, which can be excessive in the presence of nested parallelism.

Table I summarizes the shortcoming of the different work stealing approaches. In Section 3, we present *lazy scheduling*, our proposed solution that has all of the advantages of AP, plus it is effectively context sensitive.

2.4. Theoretical Bounds

Blumofe and Leiserson [1999] helped the adoption of work stealing by proving the good performance of randomized work stealing for fully strict computations[9] using only parallel function calls. The expected time to execute a fully strict computation on P workers using randomized work stealing is $T_1/P + O(T_\infty)$, where T_1 is the minimum sequential execution time (i.e., the total work) and T_∞ the minimum execution time with an infinite number of workers (i.e., the depth of the parallel computation [the length of the critical path]). The stack space required is at most PS_1, where S_1 is the minimum stack space requirement for the sequential execution.

More recent results relax the restriction of fully strict computations but, to the best of our knowledge, still omit including language constructs that introduce multiple tasks simultaneously, such as parallel loops. A notable exception is found in Cormen et al. [2009, chapter 27], which talks about the added log N term on the critical path for SP. In the presence of loops, the just described bounds need to be amended, as we discuss in Section 9.

3. LAZY SCHEDULING

The lack of performance portability in the best existing schedulers (EBS with SP or AP) is a serious issue for general-purpose parallel programming because not only do we want code to run efficiently for different input sets and contexts, we also want it to run faster on a variety of different existing and future parallel platforms with different numbers of cores. Ease of programming is also a crucial consideration: Freeing the programmer from manually coarsening parallelism, which is effectively a

[9]Computations where all join edges from a child task go to its parent. In other words, a parent task cannot complete while it has unfinished child tasks.

full-program manual optimization, will shorten his development cycle and make him more productive. Although AP does not *require* manual tuning, we show that in codes with nested fine-grained parallelism, its performance degrades considerably, and manually coarsening parallelism is necessary.

In this article, we present the concept of *lazy scheduling* and three concrete variations of lazy scheduling based on work stealing. The concept of lazy scheduling is broader than work stealing, however, and it can be applied to other types of scheduling; we show one concrete example of that in Section 8. Lazy scheduling overcomes the drawbacks related to performance portability in SP and AP by not using any statically determined threshold to decide when to stop splitting a TD. Instead, it uses runtime conditions alone in making those decisions. Furthermore, lazy scheduling does not make irrevocable serializing decisions that could harm load balancing, and it does not require maintaining any additional state to monitor load conditions.

3.1. The two Insights of Lazy Scheduling

The first insight of lazy scheduling is that pushing work (e.g., by splitting a task descriptor or by pushing a single task) onto the shared work-pool (the local deque in work stealing) is likely to be a wasted overhead if other workers are busy with other work. In such a situation, it is better for the worker to first execute some work locally, without pushing work onto the work-pool, and then check the system load again to decide whether to make some local work available globally. In this way, unnecessary splitting and work-pool transactions are avoided, but tasks are pushed on the work-pool when other workers are hungry (i.e., looking for work). This insight is also valuable for avoiding work-pool transactions both for parallel-loop TDs and for tasks that are not splittable (e.g., those originating from a parallel function call).

Directly implementing lazy scheduling to follow this insight is not obvious because checking if other workers are hungry for work can be expensive. For example, maintaining global state such as a count of hungry workers will not scale without hardware support, and, on the other hand, querying workers to determine if they are hungry requires expensive remote accesses.

The second insight of lazy scheduling provides a lightweight heuristic for inferring the load of the system during runtime. It involves looking at the size of the shared work-pool or parts thereof. For work stealing, for example, a worker looks at the size of its local deque, and, if it is below a threshold, the worker pushes a TD onto its deque. If the deque size is below a threshold (empty in our original implementation [Tzannes et al. 2010]), this is a strong indication that other workers were hungry and stole work from it. On the other hand, if the local deque is above the threshold, pushing tasks onto it is *postponed*, and the worker executes work locally, checking the size of the deque frequently. This effectively results in dynamic load-based coarsening by avoiding unnecessary shared work-pool operations and by coalescing or skipping synchronization operations.

Unlike other work-pool transactions that have to be atomic, reading its size for inferring load can be done in a racy way to reduce its overheads, as long as the error in the result is reasonable. For example, if the worker queries the size of its deque while a theft is performed, it is acceptable for the check to return any of the two values for the size, before or after the theft. This is because a slightly stale value does not perturb the efficiency of the load heuristic in practice, as long as deque checks are performed frequently.

Lazy scheduling creates a new logical state in which tasks may be in the *postponed* state. Postponed tasks are those that have become available for parallel execution, but the lazy scheduler has detected that the system is under load and has not yet placed them onto the shared work-pool; instead, those tasks reside in memory that is private

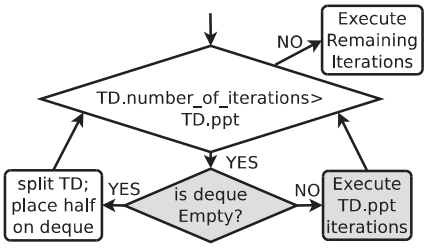

Fig. 3. Processing a task descriptor with Depth-First Lazy Work Stealing (DF-LS).

to the worker that created or stole them (e.g., its stack). A worker starts by working on its most recently postponed tasks; then, in the case of lazy work stealing, it works on the tasks in its deque before trying to steal work.

As we mentioned, lazy scheduling relies on polling the shared work-pool frequently; otherwise, a decrease in system load can go unnoticed, and workers might run out of work and sit idle for nontrivial lengths of time. In this article, our focus is to improve the performance of very fine-grained parallelism, so a reasonable implementation is to check poll the work-pool between tasks. Nevertheless, as we discuss in Section 9.4, when a program includes coarse tasks, additional polling may be necessary.

3.2. Main Benefit of Lazy Scheduling and Our Focus on Parallel Loops

The main benefit of lazy scheduling is its very low overhead in the common case when tasks are executed on the worker that originated them compared to existing approaches. This much lower overhead means that lazy scheduling is much less sensitive to parallelism granularity, allowing programmers to overexpose parallelism with minimal performance degradation. The difference in performance between eager and lazy scheduling is therefore more apparent in applications that have lots of fine-grained parallelism, especially when it is irregular, in which case the over-decomposition of parallelism has the added potential benefit of better load balancing. For that reason, we focus our description and experimental approach on code expressed through parallel loops or reductions. Nevertheless, lazy scheduling is also applicable to other constructs, as we briefly demonstrate in Section 8 (remember: the two insights help decide when to make work shared, which is orthogonal to how parallelism is expressed in the code).

3.3. Depth-First Lazy Work Stealing

In this section, we describe *Depth-First Lazy Scheduling* (DF-LS) as applied to work stealing, which we originally implemented on XMT [Tzannes et al. 2010]. We call it *depth-first* because it does not follow the *breadth-first thefts* order of work stealing, and although we were aware of the issue at the time, we did not see performance degradation on XMT, so we chose it for its simpler implementation. Depth-first lazy work stealing was presented in Tzannes et al. [2010] under the name of *Lazy Binary Splitting* (LBS). In Section 6.1, we present *Breadth-First Lazy Scheduling* (BF-LS), which follows the breadth-first thefts order and has scalability advantages over DF-LS on commercial multicores.

DF-LS checks if the local deque is empty and only then splits the current task descriptor. Figure 3 shows how a DF-LS worker processes a task descriptor upon creating it, popping it off the local deque, or stealing it. The gray boxes highlight the differences compared to Figure 1. Unlike deque transactions that require expensive memory fences, deque-is-empty checks do not, thus making them very cheap operations.

Figure 3 shows an additional improvement in DF-LS—t hat it also stops splitting when the number of tasks in the task descriptor is equal to or below a statically

determined *profitable parallelism-threshold* (*ppt*). This is useful because creating excessively small amounts of parallel work is never profitable, regardless of the number of cores, the input, or the context, because the overheads of task creation and synchronization will negate any gain from parallelism. The static coarsening pass that picks the *ppt* is described in Tzannes [2012a]. Because the *ppt* is independent of the number of cores, the input, or the context, and because it only depends on the work per task and the implementation-specific costs of creating parallelism, it can usually be easily determined by the compiler for each parallel loop without sacrificing performance portability. The performance portability of DF-LS comes from the deque-is-empty check, which ensures that enough but not too much parallelism is created for good load balancing by adapting to runtime conditions. Note that although the *ppt* is syntactically similar to the *sst* used by EBS (SP & AP), its role is merely to prevent parallelism that is too fine-grained, as opposed to also controlling the splitting of TDs. Essentially, the role of the *ppt* is to *amortize* the scheduling overheads per task while letting lazy scheduling *prune* parallelism at runtime, whereas *sst* must achieve both goals for EBS.

We now revisit the example of Section 2.3 to show how DF-LS overcomes the shortcomings of eager work stealers. When DF-LS is run, assuming processor *A* encounters a parallel loop with 16 tasks and a threshold (*ppt*) of 1, it creates a TD with those 16 tasks and starts processing it (Figure 3): Since the TD has multiple tasks, it proceeds to check if the deque is empty; assuming it is, it splits the TD and places half (tasks 9–16) on its deque. Then, seeing that its deque contains a TD, *A* starts working on task 1. Note that, at this point, SP and AP would have continued splitting the TD and pushing TDs with 4, 2, and 1 tasks before doing some actual work, likely incurring unnecessary runtime and memory overheads. In the meantime, processor *B* steals the TD in *A*'s deque and processes it: *B*'s deque is empty, or it would not have stolen a TD, so *B* splits the TD and places half on its deque (tasks 13–16), and it starts working on task 9. Then *A* finishes executing task 1 and, since its deque is empty because of *B*'s theft, it splits its remaining TD (2–8), places half (5–8) on its deque, and starts working on task 2. *B* finishes task 9, its deque is not empty, so it continues with the remaining tasks in its TD (10–12), checking before each task to see if the deque is empty. Similarly, *A* continues with its TD (3–4). When their TDs run out of tasks, *A* and *B* pop the TDs off their deques, split them, push half back on their deque, and work on their half.

The example shows how DF-LS overcomes the serialized access or creation of TDs by splitting them (like EBS, whether it be SP or AP) and how it also keeps the number of splits to a minimum by checking the deque frequently, making DF-LS more performance portable than EBS. The next section presents a detailed comparison of the number of deque transactions and synchronizations for DF-LS, EBS with SP and AP, and work-first and help-first work stealing. The comparison illustrates the benefits of DF-LS's runtime adaptivity to load conditions.

4. SYNCHRONIZATION AND DEQUE TRANSACTION SAVINGS OF LAZY SCHEDULING

Unlike existing work stealing variants (e.g. SP, AP, work-first, and help-first work stealing), lazy scheduling is able to effectively combine tasks *at runtime* by postponing pushing work while the local deque is not empty. This saves expensive deque transactions that require memory fences. It is easy to appreciate the difference between DF-LS and the other work stealers by analyzing the number of deque transactions and parent-child synchronizations (the main sources of overhead for work stealing) needed to schedule an *N* task parallel loop in the three scenarios described here. We call these three scenarios *worst*, *intermediate,* and *best* because they require a decreasing number of deque transactions and synchronizations from all compared schedulers and especially DF-LS. Loosely speaking, an execution can be approximated as a

Table II. Transaction and Synchronization Costs

	# Deque Transactions		
	Worst	Intermediate	Best
DF-LS(gr)	$2\left\lceil\frac{N}{gr}\right\rceil - 1$	$\log\frac{N}{gr} + 1$	0
SP(gr)	$2\left\lceil\frac{N}{gr}\right\rceil - 1$	$\frac{3N}{2gr} - 1$	$\frac{3N}{2gr} - 1$
AP(K, V)	$2(N-1)$	$\frac{3K \cdot P}{2} - 1$	$\frac{3K \cdot P}{2} - 1$
Work-First(gr)	$2\left\lceil\frac{N}{gr}\right\rceil - 1$	$\frac{N}{gr}$	$\frac{N}{gr}$
Help-First(gr)	$2\left\lceil\frac{N}{gr}\right\rceil - 1$	$2\left\lceil\frac{N}{gr}\right\rceil - 1$	$2\left\lceil\frac{N}{gr}\right\rceil - 1$
	# Synchronization Points		
	Worst	Intermediate	Best
DF-LS(gr)	$\frac{N}{gr}$	$\log\frac{N}{gr} + 1$	1
SP(gr)	$\frac{N}{gr}$	$\frac{N}{gr}$	$\frac{N}{gr}$
AP(K, V)	N	$K \cdot P$	$K \cdot P$
Work-First(gr)	$\frac{N}{gr}$	$\frac{N}{gr}$	$\frac{N}{gr}$
Help-First(gr)	$\frac{N}{gr}$	$\frac{N}{gr}$	$\frac{N}{gr}$

combination of these three scenarios, which is why it is important to understand how the compared schedulers operate in these cases.

The results are summarized in Table II. In the analysis here, we treat the *sst* and *ppt* thresholds (in SP and DF-LS, respectively) as parameter *gr* (for *grain*), and, without loss of generality, we assume that N is divisible by *gr* and both are powers of 2 to avoid cluttering the notation with floor and ceiling functions. We also assume that the *gr* parameter of the parallel loop is honored by work-first and help-first work stealing. As we see, both transactions and synchronizations are linear in N for SP, work-first, and help-first, but the situation for DF-LS is much different: The metrics go from linear in the worst case, to logarithmic in the intermediate case, to constant in the best case. AP's metrics go from linear in N in the worst case, to linear in P in the other two cases.

In our XMTC implementation [Tzannes et al. 2010], we implemented the sequence of pop-split-push operations as a single *pop-half* operation in order to further reduce deque transactions and, thus, runtime overheads. The pop-half operation occurs when a worker runs out of work and its deque contains a TD with more than *ppt* tasks,[10] in which case half of the TD is popped atomically.[11] For work-first work stealing, the pop-take-and-task-push sequence is also implemented as a single deque transaction in XMTC. These optimizations are included in the analytical comparison of deque transactions for the different work stealing approaches presented in the following section.

Worst Case. *The worst case happens when a worker encounters a parallel loop creating N tasks, and there are enough idle workers ($N \leq P$) to immediately steal all TDs, effectively keeping all deques empty*. This happens, for example, when parallelism is first created by the original sequential task, and it is barely enough to make all

[10]Each task descriptor may store a different *ppt* value.

[11]We based our deque implementation for XMTC on the algorithm described in Almsasi and Gottlieb [1994] "*10.3.11 Case Study: Highly Parallel Queue Management Using Fetch-and-Add.*" In our TBB implementation of lazy scheduling (described later), we made use of TBB's existing deque implementation, which only provides the regular pop operation.

workers active. In this case, SP and DF-LS behave identically: DF-LS always finds an empty deque because of the thefts and keeps splitting and pushing TDs. Similarly, the stolen TDs are split and stolen, so eventually N/gr TDs are created. That means that N/gr parent-child synchronizations occur, one for each TD. Also $2(N/gr-1)$ deque transactions happen: The factor of 2 accounts for the push and steal transaction for every task descriptor, and the -1 accounts for the fact that one of the N/gr task descriptors is never pushed on a deque but is locally executed by the worker that created it. Similarly, for work-first and help-first, we have N/gr synchronizations and $2(N/gr-1)$ deque transactions. For AP, $N \leq P$ implies $N \leq K \cdot P$, for $K \geq 1$. This means that AP will split the TD into N chunks, resulting in $2(N-1)$ transactions and N synchronizations. If a *grain size* is provided for AP, it will also appear in the denominator, but we did not include it in Table II because AP's main advantage over SP is relieving the programmer from picking grain sizes.

Intermediate Case. *The intermediate case happens when a worker encounters a parallel loop creating N tasks, the local deque is empty, but no thefts occur during its execution*. This can happen when a worker encounters a nested parallel loop while the outer parallelism was enough to feed all workers but not enough to fill the deques. This is very common in the XMT implementation because the outer parallelism is scheduled in hardware [Tzannes 2012a], and nested parallelism, which is scheduled using software, always finds the local deque to be empty. In the intermediate case, all N tasks will be executed on the worker creating them. For SP and work-first, the difference of this intermediate case compared to the worst case is that some deque transactions can be combined using the pop-half and pop-one transactions, bringing their total number down. For SP $N/grain$, TDs will be created over the course of this execution, as in the worst case. One will never be pushed on the deque, but the rest will, resulting in $(N/grain-1)$ pushes and $(N/grain-1)$ pops. This number can be reduced if we use the pop-half transaction, which combines a pop and a subsequent push of half of the popped TD. It is straightforward to show that the number of such pop-half transactions is equal to the number of nodes in a perfect binary tree[12] with N/gr leaves, excluding the leaves, which represent the execution of an indivisible amount of work (gr tasks), and their parent nodes, which represent an indivisible TD at the top of the deque that cannot benefit from the pop-half transaction. The number of the remaining nodes is $\frac{N}{2gr}-1$, so the number of transactions becomes $\frac{3N}{2gr}-1$. The number of synchronizations remains N/gr, as before. For AP, $K \cdot P$ TDs will be created, and, by following the same reasoning, the number of transactions will be $\frac{3K \cdot P}{2}-1$, and the number of synchronizations will be $K \cdot P$. For work-first, the number of transactions is N/gr: One push of $N-gr$ tasks initially, followed by $N-2$ *pop-grain* operations removing gr tasks each, and finally a pop of the remaining tasks. The number of synchronizations is also N/gr; one after every gr tasks. Help-first cannot benefit from the pop-grain transaction since it begins by creating N/gr TDs that cannot be split further. For that reason, the number of transactions and synchronizations is the same as in the worst case.

For DF-LS, the situation here is much different. Initially, half the tasks $(N/2)$ are pushed on the deque, and the other half are executed, checking the size of the deque after every gr tasks but finding it full. Then, a pop-half operation reclaims half of the pushed tasks (i.e., $N/4$) that will be executed. Then, a pop-half will reclaim $N/8$ tasks, and so on, until the last $\frac{N}{2^k} = gr$ tasks are popped and executed. This amounts to $\log \frac{N}{gr}+1$ transactions. The number of synchronizations is also $\log \frac{N}{gr}+1$ because they happen before every pop-half, before the last pop, and at the very end.

[12]A binary tree that has all leaf nodes at the same depth, and all internal nodes have exactly two children.

Best Case. *The best case happens when a worker encounters a parallel loop creating N tasks, no thefts occur, and the deque is* **not** *empty*. This happens when nested parallelism is encountered and the outer parallelism was sufficient to fuel all workers and deques, and it is particularly common for recursively nested parallelism.

For SP, AP, work-first, and help-first, nothing changes from the previous case because these schedulers do not change their behavior based on the status of the deque. For DF-LS, things are very simple: No transactions occur, and synchronization occurs only once, after all tasks have executed. We call this the *best case* because DF-LS incurs almost zero overhead in terms of deque transactions and synchronizations. In fact, even that single synchronization can be optimized away by detecting that none of the tasks was ever placed on the deque, but we have not implemented this optimization.

Synchronization Coalescing. In retrospect, another interesting optimization to reduce the number of synchronizations for DF-LS from logarithmic to constant in the intermediate case would be the following: Instead of directly returning and synchronizing with the parent task after processing a TD (Figure 3), the scheduler will try to consume the rest of the TD from the local deque through a series of pop-half and pop transactions, if the TD was split. In the absence of thefts, this attempt will be successful, and no synchronization will be needed. We did not implement this optimization because we conceived it too late, and we are unconvinced that it would result in significant cumulative improvements compared to those already achieved by lazy scheduling. We mention it here, however, for completeness and for the benefit of anyone planning to implement lazy scheduling.

4.1. Deque Checks

So far, we have focused on the overhead of deque transactions and synchronizations, but there is one more source of overheads in DF-LS: the checks to the local deque to decide whether to postpone pushing work or not. These checks are very lightweight and fast, but they are linear in the number of tasks ($N/gr - 1$) in all three cases just presented. Thus, for very fine-grained tasks, they can become a significant source of overhead. Note that SP, AP, work-first, and help-first also perform deque checks to determine if pushing a TD will overflow the deque. In all three cases just described (best, intermediate, and worst), DF-LS, SP, work-first, and help-first perform $O(N/gr)$ deque checks, whereas AP performs $O(K \cdot P)$ checks. When tasks are very fine-grained, the linear overhead of these checks can become more important than the logarithmic or constant overhead of deque transactions and synchronizations of DF-LS. This motivates the need for a profitable parallelism-threshold for DF-LS, as described in the next section.

4.2. Role of the Profitable Parallelism Threshold (*ppt*)

As outlined earlier, the function of the profitable *ppt* of DF-LS is to amortize scheduling costs by reducing the frequency of deque checks. Conversely, the *sst* of SP focuses mainly on pruning parallelism to control the number of deque transactions and synchronizations by stopping the splitting. DF-LS achieves this goal without using the *sst* (or the *ppt*), by postponing pushing work onto the work-pool based on the deque size, which is our heuristic for inferring the system load. There is also a second source of overheads associated with the deque checks: the scheduler executes a task by calling its closure, and, to check the size of the deque, the execution must return to the scheduler code. So, for each deque check, DF-LS also pays the overhead of a function call. Since these overheads are linear in the number of tasks, it is important to combine fine-grained tasks by means of the profitable parallelism threshold (*ppt*). Recall that the *ppt*, also referred to as *grain size*, is picked by the compiler for each parallel loop, as described in Tzannes et al. [2010, 2012a], so it does not burden the programmer.

Another thing to note from the analysis in Section 4 is that the *ppt* threshold (*grain*) in the intermediate and best cases plays a minimal role in controlling the number of transactions in DF-LS. The worst case, which is triggered by thefts, is rare enough, as backed up by our results showing better performance for lazy scheduling (Section 7), so that it is fair to say that *ppt* is not the primary factor controlling the number of transactions and synchronizations in DF-LS. Conversely, *sst* is the **only** way these overheads are controlled in SP, work-first, and help-first. In AP, the only way to control the number of transactions and synchronizations is to also provide a *grain*, which supersedes AP's automatic coarsening. However, the *grain* parameter was not included in the analysis because AP is typically used without a grain size—after all, this is AP's advantage over SP.

As an aside, a *grain* (ppt or sst) larger than 1 for a parallel loop is semantically very similar to loop strip-mining with a parallel outer loop and a sequential inner loop of *grain* iterations. A compiler can perform all the classic textbook loop transformations on this sequential inner loop.

5. SCALABILITY ISSUES OF DEPTH-FIRST LAZY SCHEDULING (DF-LS)

In the interest of reducing duplication, we will not reproduce the experimental result of Tzannes et al. [2010]. In summary, we showed that DF-LS (a.k.a. LBS) outperformed the default configuration of AP by 38.9% over a set of benchmarks on our experimental XMT architecture [Wen and Vishkin 2008] (not Cray XMT). DF-LS also outperformed SP without tuning (i.e., $sst = 1$) by 56.7%. These two comparisons represent the performance benefits of DF-LS without programmer tuning of the grain size. Moreover, the self-relative speedup results (Table 4 in Tzannes et al. [2010]) do not indicate scalability issues for DF-LS on XMT. Similarly, the results in Bergstrom et al. [2010] seem to suggest the same on commercial multicores for Lazy-Tree Splitting, a derivation of DF-LS for a functional language with arrays represented as trees. Nevertheless, in this section, we show that DF-LS has serious scalability issues on multicores for codes with extremely fine-grained tasks.

The deviation of DF-LS from breadth-first thefts is reason for concern because it can greatly increase the number of thefts by pushing work from the TD being processed (i.e., describing the innermost nested parallel construct at the point of execution) when the deque is found empty, instead of the oldest postponed TD (i.e., from the outermost parallel construct). The number of thefts can increase because deeply nested tasks often contain less work (in their computation subtree) than shallower tasks, and DF-LS makes smaller chunks of work available to hungry workers by pushing the innermost postponed tasks instead of the oldest postponed task.

Thefts are more expensive on multicores than on XMT, in part because their memory hierarchy includes private caches. First, stealing a TD involves acquiring exclusive write permission to a cache-line that is typically owned by the victim worker. Furthermore, the activation frame for that task typically also resides in the private cache of the victim worker. Think of the theft as a lightweight context switch in which parts of the private cache of the victim worker are transferred to the thief. On XMT, because it does not have private caches, a theft only involves stealing a task descriptor, an operation that is almost as cheap (or as expensive) as popping a TD from the local deque.

To show that DF-LS has scaling issues on traditional multicores, we implemented it in Intel's TBB library (v3.0). TBB implements work stealing and provides the programmer with an API that implements parallel loops, reducers, and other operations. We chose TBB because parallel TBB code achieved good speedups versus serial implementations, indicating that TBB is implemented efficiently, and because TBB supports various target platforms, which allowed us to run experiments on a variety of machines. Furthermore, we are very thankful that the TBB team and Intel chose to release their

code under the open source GPL license, allowing us and others to experiment and contribute to the state of the art. We made our modified TBB implementation publicly available [Tzannes 2013b], as well as the implementation of all our benchmarks in this article [Tzannes 2013a].

Our original intention was to implement our lazy schedulers in one of the Cilk languages (Cilk, Cilk++, or the new Cilk Plus). We excluded Cilk due to its lack of parallel loop support and Cilk++ because it was not open source for us to modify. Cilk Plus [Cilk-Plus 2011] was not publicly available when this work was done, and, even as these lines are written, the open-source version of Cilk Plus is not mature enough to support the simple benchmarks presented later [Tzannes 2012b]. Even though the scheduling overheads of the Cilk languages are lower than those of TBB, thanks in part to the parallel constructs being integrated in the language rather than being library functionality, our lazy scheduling would still benefit them by significantly reducing the scheduling overhead per task, thus enabling more fine-grained parallelism to be profitable. As we have argued, support of the finest possible grain of parallelism improves ease of programming and performance portability, both important issues for general-purpose parallel programming.

ALGORITHM 2: NQueens(N, *partialSolution*, *depth*)

Input: The size of the board N, a partialSolution $c_1c_2 \ldots c_k$, where $1 \leq c_i \leq N$ is the column position of the queen on the i^{th} row, and depth the depth of the recursion (and size of the partial solution), where $1 \leq k \leq N$.

Result: The number of solutions to placing N queens on a N × N board is added to the global variable solutionCount.

```
NQueens (N, partialSolution, depth) begin
    forall the i ∈ {1, 2, ..., N} do
        if OkToAdd (i, partialSolution, depth) then      // ok to add queen on column i at row
        depth
            partialSolution′ ← partialSolution ∪ (i, depth); // Append (i, depth) to partialSolution
            if depth < N then                                                    // Recursion
                NQueens (N, partialSolution′, depth +1)
            else                                                          // Found a Solution
                atomic{solutionCount += 1 }
            end
        end
    end
end
```

To demonstrate the lack of scalability of DF-LS, we use *NQUEENS* (with N = 14) for its recursively nested parallelism but without parallelism cutoff, as shown in Algorithm 2, to intensify the repercussions of choosing the innermost task. Because one of our goals is to set the foundations of efficient support for declarative parallel programming, it is important to ensure good scalability in the absence of manual coarsening.

Algorithm 2 shows that *NQUEENS* recursively calls itself up to $depth = N$. At the i^{th} recursive invocation, *NQUEENS* tries to place a queen on each column of the i^{th} row of the chessboard and checks if adding a queen does not attack the queens already placed on rows 1 to $i - 1$. Parallelism is introduced by trying to place each of the N queens at each row in parallel. Each time the N^{th} queen has been placed successfully, the counter of solutions is incremented atomically. Note that the `OkToAdd` check prunes the search space at every level of the recursion, resulting in an unbalanced and irregular execution tree.

Table III. Platform Descriptions

Name	i7	Xeon	T2
Threads	8	24	64
Cores	4	24	8
CPU	i7 CPU 920	4 Intel Xeon E7450	UltraSPARC-T2
Clock	2.67GHz	2.4GHz	1.2GHz
L3 cache	8MB	4×12MB	4MB
RAM	24GB DDR3	48GB DDR2	32GB DDR2
kernel	linux 3.2.0	linux 2.6.18	Solaris 5.10
g++	4.6.3	4.1.2	3.4.3
libc	2.15	2.5	N/A

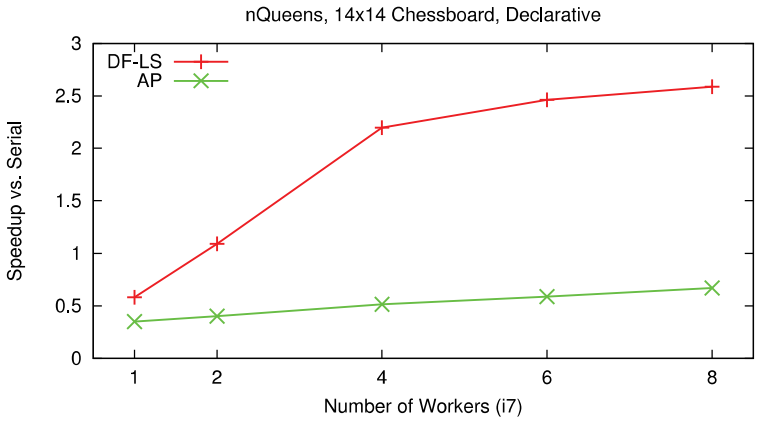

Fig. 4. Performance scaling of schedulers on i7 (nQueens).

Fig. 5. Performance scaling of schedulers on Xeon (nQueens).

We used three commercial multicores for our evaluation, summarized in Table III. The three machines are very different and include a multicore desktop (i7) with hyperthreading, an SMP multicore (Xeon), and a Niagara2 multithreaded multicore (T2). Figures 4, 5, and 6 show the performance scaling of DF-LS on *NQUEENS* on these three machines. The performance of AP, TBB's default scheduler, is also shown for reference. We also collected numbers for TBB's SP, but AP always performed at least as well as SP for the benchmarks presented in this article, so we only present

Fig. 6. Performance scaling of schedulers on T2 (nQueens).

the results for AP. Each data point is computed as the average of 10 executions. The standard deviations can be found in Table XX in the appendix. The standard deviations for AP are below 2%, and most of those for DF-LS are below 4%. The few data points with higher standard deviation for DF-LS reveal the greater degree of randomness in DF-LS, which pushes tasks from the TD currently being processed at the time when a deque is found empty, as opposed to always pushing the oldest postponed tasks. Nevertheless, this higher variability does not negate the conclusion that DF-LS has scalability issues on multicores.

On the small i7 machine (Figure 4), DF-LS scales well and greatly outperforms AP. The sublinear performance scaling of DF-LS is attributed to the fact that, for worker counts larger than the number of cores (4), hyperthreading kicks in and the workers compete for shared core resources.

On the two larger machines, Xeon (Figure 5) and T2 (Figure 6), DF-LS fails to scale. It scales well up to 8 workers, which shows the promise of lazy scheduling, but then flattens out and even decreases in performance as more workers are used. DF-LS's performance degrades because workers push their innermost postponed tasks, which can contain an exponentially smaller amount of work than their outermost postponed tasks. This greatly increases the number of thefts (the most expensive scheduling operation) and prevents DF-LS from scaling to larger numbers of workers. In Section 7.1, we show that we can fix the scaling problem of DF-LS and allow it to scale to larger machines. We also count the thefts for the compared approaches and show that DF-LS incurs by far the most thefts. In the next section, we present two ways to make lazy scheduling scale on multicores by reducing the frequency of thefts it incurs.

6. LAZY SCHEDULING FOR DECLARATIVE CODE

We describe two approaches for solving the scalability issues that DF-LS has with declarative code. The first (BF-LS) is more robust but a bit harder to implement and involves pushing the outermost postponed tasks instead of the innermost ones. The second (DF2-LS) is less robust but trivial to implement and actually achieves comparable performance to the first one on our set of benchmarks and may be a reasonable alternative for mid-size machines (between 10 and 60 hardware threads). It involves increasing the deque size threshold of DF-LS from 1 to 2. This simple change has deeper effects on the scheduling algorithm than is immediately apparent; these are described in Section 6.2.

6.1. Breadth-First Lazy Scheduling (BF-LS)

In eager scheduling, a worker always pushes tasks on its deque as soon as it encounters them. Conversely, in lazy scheduling, if the system has enough parallelism, a worker postpones pushing tasks on its deque. Later, during the execution, the worker may infer that other workers are hungry and decide to push work onto the shared work-pool (e.g., its deque). At that point, the worker may have many postponed TDs to choose from if the code has nested parallelism.

From an implementation standpoint, the simplest solution is to push the innermost postponed task—the TD being processed at the time of the decision to push work onto the work-pool. We called this approach DF-LS, and it is the approach taken by our earlier work [Tzannes et al. 2010] and by Lazy Tree Splitting [Bergstrom et al. 2010]. Although, in our experience, DF-LS works well on XMT for the benchmarks we tried, using it on commercial multicores may not scale because it pushes deeply nested tasks that are likely to contain less work than shallower tasks, leading to more thefts and deque transactions (pushes and pops), which we are trying to reduce by lazy scheduling in the first place. In fact, in Section 5, we showed experimentally that DF-LS fails to scale to large numbers of threads on multicores for a recursively nested declarative parallel code. On the other hand, the depth-first approach may reduce the memory footprint in practice, although it is unlikely that a better theoretical bound can be proven.

The dual approach to DF-LS is to push onto the deque the oldest postponed task, which also has the shallowest nesting depth. This approach honors the principle of *breadth-first thefts*, so we call it BF-LS. This approach is a bit trickier to implement because we must keep track of the postponed tasks and be able to push them on the deque, for which we use an additional deque data structure per worker. Because of this added bookkeeping, BF-LS has a slightly higher scheduling overhead per task than DF-LS.

A third approach could push postponed tasks that are somewhere between the outermost and the innermost ones. From an implementation standpoint, that would incur the same (or more) bookkeeping overhead as BF-LS. The only apparent advantage of this approach is that it might reduce the memory footprint without dramatically increasing the number of thefts. However, it would still violate the principle of breadth-first thefts. Such an approach did not seem promising, so we did not investigate it.

Algorithm 3 presents a high-level description of one possible implementation of BF-LS. For simplicity, we only show TDs that represent 1D iteration ranges, but our TBB implementation also supports 2D and 3D ranges. A TD has an integer index for the first iteration to execute (*id*), and one for the number of iterations in the TD (*nrt*), as well as a minimum grain size (*grain*). It also maintains a function pointer to the code to be executed and a pointer to the data accessible on the task's parent frame (these pointers are identical for all tasks of a dynamic instance of a loop). Each worker now maintains a private deque for postponed TDs in addition to its shared deque for shared TDs. The private deque can be implemented as a doubly linked list. Readers interested in possible implementations of shared deques are referred to existing work [Frigo et al. 1998; Arora et al. 1998]. TDs are pushed onto the private deque before processing (line 1) and are popped upon completion (line 5). If the shared deque size is below the threshold and the oldest TD in the private deque is not splittable, it may be dequeued by a nested invocation of Algorithm 3 (line 2). For that reason, it is necessary to check if the private deque is empty before popping the processed TD on line 5.

Having replaced the original shared deque of work stealing with a shared deque with a size threshold, a private deque for postponed tasks may seem counterproductive, but

ALGORITHM 3: BF-LS Processing of a TD representing a parallel loop

```
   struct TD { int id, nrt, grain; void (*func) (); void *args,};
   Input: TD: a Task Descriptor Representing a Range of Tasks
   Input: worker: a data-structure maintaining references to the local shared and private
          deques
   Result: totalExec: The number of tasks executed (for synchronizing with the parent task)
   totalExec ← 0 ;                                              // Number of Tasks executed
1  push TD on worker.privateDeque.top ;
   while TD.nrt > TD.grain do                                   // number of tasks > grain (=ppt)
       if worker.sharedDeque.size < THRESHOLD then              // push work on shared deque
           oldestTD = worker.privateDeque.bottom;
           if oldestTD.nrt > oldestTD.grain then                // can split oldestTD
               split oldestTD and push half on top of worker.sharedDeque
           else                                                 // cannot split oldestTD
2              remove oldestTD from bottom of worker.privateDeque ;
               push oldestTD on top of worker.sharedDeque ;
           end
       else                                  // postpone pushing work while executing TD.grain tasks
           id ← TD.id;
3          TD.id ← TD.id + TD.grain;
4          TD.nrt ← TD.nrt − TD.grain;
           TD.func(id, TD.grain, TD.args) ;                     // Execute grain tasks
           totalExec ← totalExec + TD.grain;
       end
   end
   if worker.privateDeque is not empty then
5      pop worker.privateDeque.top
   end
   if TD.nrt > 0 then                                           // execute any remaining tasks
       TD.func(TD.id, TD.nrt, TD.args) ;
       totalExec ← totalExec + TD.nrt
   end
   return totalExec ;                                 // Used to decr. continuation's pending tasks
```

there are several advantages in doing that. *(1)* The shared deques can be statically allocated because of their constant size, this avoiding expensive dynamic memory allocation. *(2)* Operations on the private deque, which greatly outnumber those on the shared deque, do not need any synchronization since the private deque is not shared. *(3)* Postponed TDs are not recursively split ($\log N$ times) but are placed whole in the private deque. *(4)* The construction and consumption of TDs in the private deque (mostly) follows the depth-first execution order; in conjunction with the previous point of placing postponed TDs unsplit in the private deque, this allows us to allocate the private deque on the worker's stack and avoid expensive dynamic memory allocation for the private deque as well. In fact, the allocation and deallocation of the private deque comes free by being embedded in the existing stack allocation and deallocation. The only management overhead comes by maintaining the forward and backward pointers.

One subtle point of this algorithm is that, on lines 3 and 4, the worker must remove the tasks from the *TD before* executing them. This is because executing them may create additional tasks TD', at which point the worker may push the rest of *TD* on its deque, potentially resulting in executing those tasks twice, which is generally not correct if the tasks have side effects. Relaxing the requirement of unique execution

can lead to less strict synchronization requirements and improved performance, even though some work is duplicated [Michael et al. 2009; Leijen et al. 2009]. In our case, however, duplicate execution violates correctness.

6.2. DF-LS with a Deque-Check Delay (DF2-LS)

Simply increasing the deque size threshold of DF-LS from 1 to 2 can dramatically improve its performance scaling by introducing a delay between pushing work onto an empty deque and deciding to postpone exposing more parallelism. We call this variant DF2-LS to distinguish it from DF-LS, which has a threshold of 1.

The fundamental difference between the two is most noticeable when the machine is starving for parallel work (i.e., when many workers are trying to steal, and most deques are empty), as occurs when switching from sequential to parallel execution for example. During that initial period, a few workers have tasks to push onto their empty deques. If the deque threshold is 1, a busy worker will push a TD with half its tasks onto its deque, immediately check its size, and find it equal to the threshold because thieves have not had time to steal the work. Consequently, the worker will falsely conclude that other workers are not hungry and will start executing a task. If nested parallelism is encountered, the worker will discover that its deque is empty and push some of the inner tasks onto its deque, instead of the outer ones. Conversely, with a threshold of 2, the worker pushes a TD with half its tasks onto its empty deque, then pushes another TD with half of its remaining tasks. This second push of a TD gives thieves some time[13] to steal the first TD the worker pushed. The worker will notice that theft and subsequently keep pushing outer tasks until thefts become less frequent and the system is no longer starving for work. Therefore, having a threshold of 2 effectively creates a *delay* between pushing work onto an empty deque and concluding that other processors are not hungry, thus making the heuristic of checking the size of the deque a more accurate indicator of the system load.

We also experimented with adding an artificial delay between pushing work on a deque and the subsequent size check of that deque, instead of increasing the threshold to 2. We invoked `usleep` with arguments ranging from 1 to 25, but always noticed a performance degradation compared to DF-LS. We did not try to implement the artificial delay as a shorter busy wait (e.g., a loop of 100 iterations that do nothing) because we do not believe that wasting power to simply wait is a good strategy, especially since power is already one of the factors limiting performance.

Despite the good scalability results that we present in Section 7.1, DF2-LS remains a depth-first approach, with the same problem of pushing the innermost tasks and incurring an increased number of thefts. Moreover, in the absence of nested parallelism, the higher deque threshold of DF2-LS causes more deque transactions (pushes and pops) without any additional benefit since all tasks are outer tasks. In those scenarios, DF2-LS is a bit slower than DF-LS, but its superior scalability generally justifies its use over it.

Table IV presents a high-level view of the four schedulers we compare in the next section, including TBB's SP, which is also used by Cilk++ and CilkPlus. We used a bold font to highlight the good qualities of each scheduler. The number of thefts and the number of pops (reclaiming work from one's own deque) are a measure of wasted overheads and should be minimized.

The table shows that BF-LS is the best approach, but it incurs slightly more overhead per push by accessing the oldest postponed task in the private deque, instead of pushing the current task like all the other compared schedulers do. Nevertheless, BF-LS is the only scheduler that minimizes both the number of thefts and the number

[13]Depending on the relative speed of splits versus steals, a threshold higher than 2 might be needed.

Table IV. Comparison of Schedulers

	BF-LS	DF2-LS	DF-LS	AP	SP
BF-Thefts	**Yes**	No	No	**Yes**	**Yes**
Lazy	**Yes**	**Yes**	**Yes**	No	No
Cost/Push	Low+□	**Low**	**Low**	**Low**	**Low**
#Thefts	**Low**	Medium	Very High	**Low**	**Low**
#Pops	**Low**	Medium	**Low**	High	Very High

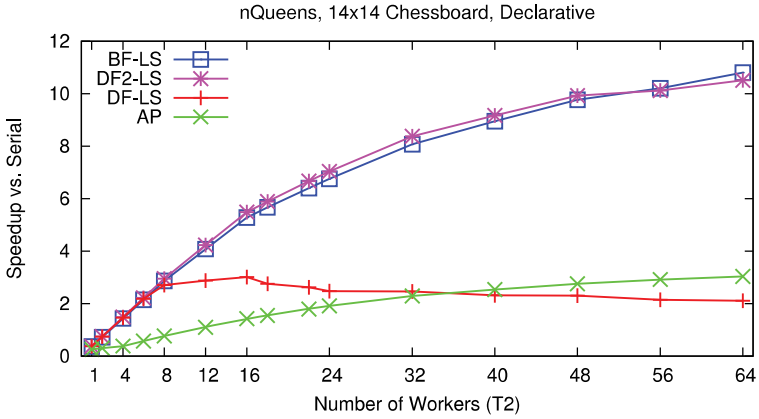

Fig. 7. Performance scaling of schedulers on T2 (nQueens).

of pop operations, so it is likely to be the best choice for performance and performance portability, in particular for codes with fine-grained tasks. The experimental results in the next section support this hypothesis.

7. EXPERIMENTAL EVALUATION OF LAZY WORK STEALING ON MULTICORES

In this section, we start by showing that our proposed solutions, BF-LS and DF2-LS, amend the scalability issues of DF-LS on the *NQUEENS* benchmark presented in Section 5, and we count the number of thefts incurred by the different schedulers to show that our hypothesis was correct and that DF-LSs do indeed cause significantly more thefts. Then, we evaluate BF-LS and DF2-LS on a set of benchmarks on three significantly different multicore platforms and show their performance improvement over DF-LS and TBB's default scheduler, the AP. We also collected performance numbers for TBB's SP, but we are not presenting those because they were equal to or worse than those of AP. Furthermore, we compare the *software performance optimality ratio* [Tzannes 2012a] (software optimality for short) of the compared schedulers on declarative code and on code that has been statically coarsened (manually or otherwise) to amortize scheduling overheads. This comparison demonstrates the performance and programmability advantages of lazy scheduling for fine-grained tasks.

7.1. Scaling of Lazy Scheduling on Multicores

Figures 7, 8, and 9 augment Figures 6, 5, and 4 with the results for BF-LS and DF2-LS to show how they overcome the scaling issues of DF-LS. To do that, we implemented those two alternative schedulers within TBB. Given DF-LS, the additional effort to implement DF2-LS was trivial, and the effort for BF-LS was relatively modest. As before, we timed 10 executions of each data point and took the average. Speedups are computed versus the execution time of a *sequential version of the program*, as opposed

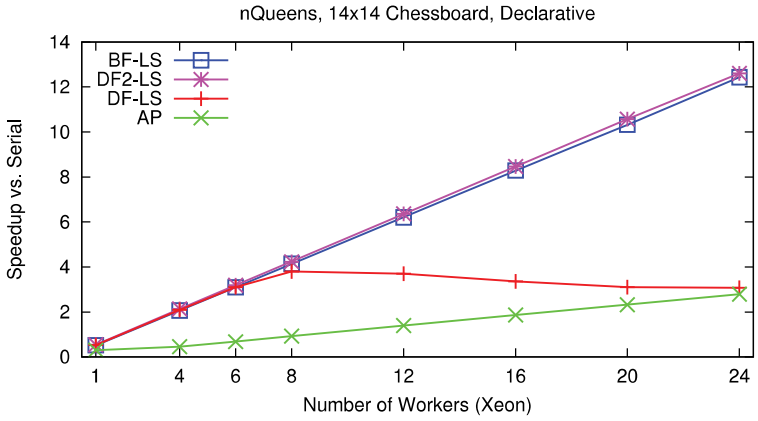

Fig. 8. Performance scaling of schedulers on Xeon (nQueens).

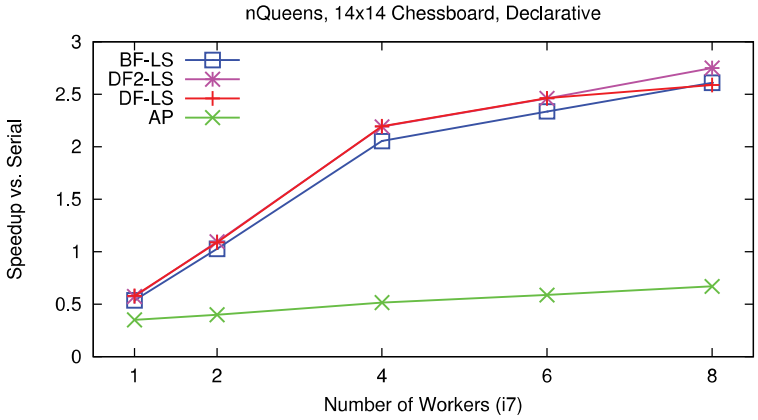

Fig. 9. Performance scaling of schedulers on i7 (nQueens).

to the execution of the parallel code on one worker. The standard deviations are shown in Table XX in the appendix.

On T2, our largest platform, the improvement is very substantial (Figure 7): BF-LS and DF2-LS achieve speedups of 10.8× and 10.5× compared to the speedup of 2.1× for DF-LS and 3.0 for AP. Another interesting trend is that DF2-LS performs marginally better than BF-LS up to 56 workers, and BF-LS comes ahead for larger worker counts. This illustrates two things: *(1)* that BF-LS has a higher overhead per task because it keeps track of postponed TDs in private deques, which causes it to fall slightly behind for smaller worker counts, and *(2)* that for a large number of workers, DF2-LS starts suffering from the same scaling issues as DF-LS because it pushes work from the innermost postponed TD.[14] On Xeon (Figure 8), we observe the same trends. BF-LS and DF2-LS achieve speedups of 12.4× and 12.6×, whereas DF-LS and AP only reach 3.0×.

On the small i7 (Figure 9), BF-LS and DF2-LS achieve similar performance to DF-LS. This supports our hypothesis that, for small machines, the DF-LS strategy of pushing

[14]Increasing the deque threshold from 2 (e.g., to 3 or 4) would improve scaling at the cost of more deque transactions. However, DF*k*-LS would still suffer scalability issues on larger machines due to its deviation from the breadth-first thefts principle. For that reason, we favor using BF-LS over the DF variants.

Table V. Number of Thefts (Average over 10 Runs)

Platform	DF-LS	DF2-LS	BF-LS	AP
T2(64)	55,593,973.2	1,045,072.0	3,130.5	3,303.2
Xeon(24)	4,161,559.1	10,562.9	791.9	906.6
i7(8)	316,337.6	973.8	228.1	274.1

the innermost tasks does not cause thefts to become excessively frequent to the point of hurting performance. The same observation can be made by looking at the scaling of DF-LS on the larger machines for small numbers of workers (Figures 7 and 8 for 1–8 workers).

On all three platforms, all three of TBB's available EBS schedulers (i.e., SP, AP, and affinity-partitioner) fall *significantly* behind BF-LS on the declarative version of *NQUEENS*. Of the three TBB schedulers, only AP is shown because it consistently achieved better performance.

7.2. Counting Thefts

As argued earlier, the limited scalability of DF-LS is caused by the choice of pushing the innermost postponed tasks, resulting in a greater number of thefts. We measure the number of thefts incurred by the four competing approaches on our three platforms (Table V); displayed is the cumulative number of thefts performed by all workers, averaged over 10 runs. The number of thefts with DF-LS is orders of magnitude larger than with any other approach, whereas with DF2-LS, it is orders of magnitude smaller than with DF-LS but much larger than with BF-LS or AP. Finally, the number with BF-LS roughly matches that of AP. We believe that AP incurs slightly more thefts than BF-LS in this example because it runs much longer, wasting time pushing and popping tasks from the local deque. In an effort to load balance over this longer execution time, some additional thefts occur. The added instrumentation to count thefts did not significantly affect the running time of the benchmark because it only requires some bookkeeping local to each worker and no communication, so we believe the results accurately reflect the number of thefts of uninstrumented executions.

7.3. Evaluation on a Set of Benchmarks

NQUEENS was a useful toy example to experimentally demonstrate how DF-LS fails to scale up to a large number of workers but grants us little confidence that using BF-LS as the default scheduler instead of AP (or SP) is a good idea. To address this, we compare the different approaches over the set of benchmarks summarized in Table VI. These benchmarks were selected because they exhibit a variety of computation and communication patterns [Asanovic et al. 2006], which is important because we want to support general-purpose parallel code. Moreover, we needed benchmarks with nested parallelism to ensure scaling under composition of parallelism and to expose the limitations of AP.

The benchmarks in Table VI are described as follows: *TSP* is the traveling salesman problem on a dense graph, *NQUEENS* finds all possible solutions to placing N queens on an N by N chessboard, *BFS* is a breadth-first search over a sparse graph, and we used the `DIMACS10/rgg_n_2_24_s0` dataset from the University of Florida sparse matrix collection [Davis and Hu 2011], *SpMV* is a sparse matrix by (dense) vector multiplication, *FW* is the Floyd-Warshall all-pairs shortest path on a dense graph, *MM* is the naive (N^3) dense matrix multiplication, and *CONV* is an image-by-filter convolution. Moreover, *TSP(cut)* and *NQUEENS(cut)* are coarsened versions with manual parallelism cut off after depth $N/2$, whereas *TSP(decl)* and *NQUEENS(decl)* are the declarative versions with parallelism exposed all the way down to the leaves of the recursion. Finally, *SpMV(coarse)* computes each row of the sparse array sequentially,

Table VI. Benchmark Summary

Declarative	**Dataset**	**Grain**
NQUEENS	N = 14 Nodes	1
TSP	N = 12 Nodes	1
SpMV	80Kx5K ($N \times M$), 40M nonzero (V)	77
BFS	16.8M Nodes, 132.5M Edges	53
FW	N = 2048 Nodes	91
Coarsened	**Dataset**	**Grain**
NQUEENS	N = 14 Nodes	1
TSP	N = 12 Nodes	1
SpMV	80Kx5K ($N \times M$), 40M nonzero (V)	1
MM	1024x1024 (N^2)	1
CONV	4Kx4K (N^2) image, 16x16 (M^2) filter	1

Declarative	**Nesting**	**Parallelism**	**Work/Task**	**Description**
NQUEENS(decl)	N	O($N!$)	O(1)	fine/irregular
TSP(decl)	N	O($N!$)	O(1)	fine/irregular
SpMV(decl)	2	$40M$	O(1)	fine/irregular
BFS	2	$O(\frac{\lvert E\rvert}{Diameter})$	O(1)	fine/rregular
FW	1 (2D)	N^2	O(1)	fine/regular
Coarsened	**Nesting**	**Parallelism**	**Work/Task**	**Description**
NQUEENS(cut)	$N/2 = 7$	$O\frac{N!}{(N/2)!}$	$O\frac{N}{2}!$	coarse/irregular
TSP(cut)	$N/2 = 6$	$O\frac{N!}{(N/2)!}$	$O\frac{N}{2}!$	coarse/irregular
SpMV(coarse)	1	$80K$	$O(V/N)$	medium/irregular
MM	1 (2D)	N^2	O(N)	coarse/regular
CONV	1 (2D)	N^2	O(M^2)	coarse/regular

whereas *SpMV(decl)* uses TBB's parallel-reduce construct to expose all the parallelism (one task per nonzero element of the sparse array) and to efficiently aggregate the results. To that end, we modified TBB's reduction operation to support the DF-LS, DF2-LS, and BF-LS schedulers.

In our PPoPP paper [Tzannes et al. 2010], we had also benchmarked *Quicksort* in order to demonstrate that the `partition` procedure could also be easily parallelized using XMT's novel hardware prefix-sum unit. On the commercial multicores we used in this article, we did not find an easy (declarative) way to profitably parallelize *Quicksort's* `partition` procedure, and implementing it with a sequential `partition` would classify it as a coarse-grained benchmark, of which we already had enough and better candidates. For those reasons, we decided to exclude it from this evaluation.

Table VI divides our benchmarks into *declarative*, in which all the parallelism has been exposed, and *coarsened*, in which either some of the parallelism was manually hidden (*TSP(cut)* and *NQUEENS(cut)*), or not all parallelism was exposed (*SPMV(coarse)*, *MM*, and *CONV*).

All the coarsened benchmarks can be rewritten as declarative ones with $O(1)$ work per task. For example, *MM* and *CONV* can be further parallelized using a parallel reduce operation, but, due to their more regular nature, it is unlikely that the additional parallelism would improve performance in most realistic scenarios. Because this parallelization could be viewed as unnatural and unnecessary, we opted against it. However, in extreme cases where, for example, the multiplied arrays are vectors and the result is a single value, using the parallel reduce operation may be the only way to achieve a speedup.

The *grain* column shows the profitable parallelism threshold (*ppt*) automatically picked by the XMTC compiler [Tzannes et al. 2010; Tzannes 2012a], the *nesting* column

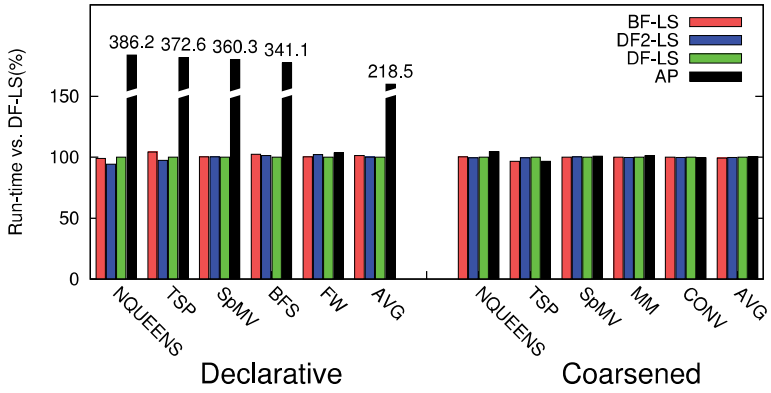

Fig. 10. Benchmarks on the i7 using all eight workers.

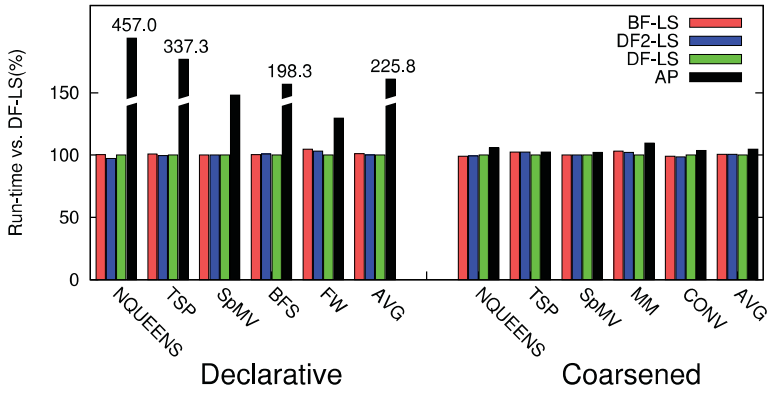

Fig. 11. Benchmarks on the Xeon using only six workers.

is the nesting depth of parallelism, and *parallelism* is the degree of parallelism (i.e., the maximum number of tasks that can be executed in parallel); or, in other words, the maximum width of the computation Directed Acyclic Graph (DAG). For *BFS*, it is on average in the order of the number of edges divided by the graph diameter. For our dataset, the diameter is 4. The next column represents the work per task, which happens to be constant for our declarative benchmarks and nonconstant for our coarsened benchmarks.

In addition to1D iteration ranges, TBB also provides *range* objects that describe 2D and 3D iteration spaces, which can be used to effectively flatten nested parallelism for dense, affine matrix computations. This allows us to expose a multi-dimensional range of parallelism while avoiding the explicit use of nested parallelism, something that AP and SP are not very good for. We used multidimensional ranges wherever possible (*FW*, *MM*, and *CONV*).

Comparisons. Figures 10, 11, 12, 13, and 14 show the results on our three machines, grouped into declarative and coarsened benchmarks. We used the average of 10 runs for the plots, and the standard deviations are shown in Tables XXI, XXII, and XXIII in the Appendix. We used the geometric mean to compute the averages since we are averaging percentages (scaled values for the execution time). Our main goal is to demonstrate the superior performance of BF-LS and DF2-LS compared to DF-LS, so we

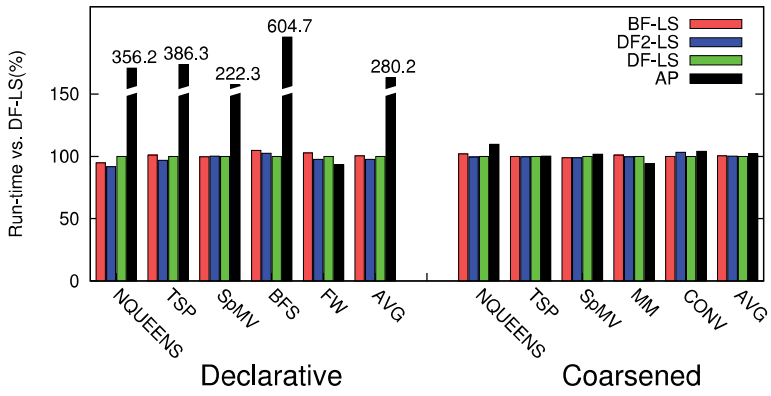

Fig. 12. Benchmarks on the T2 using only eight workers.

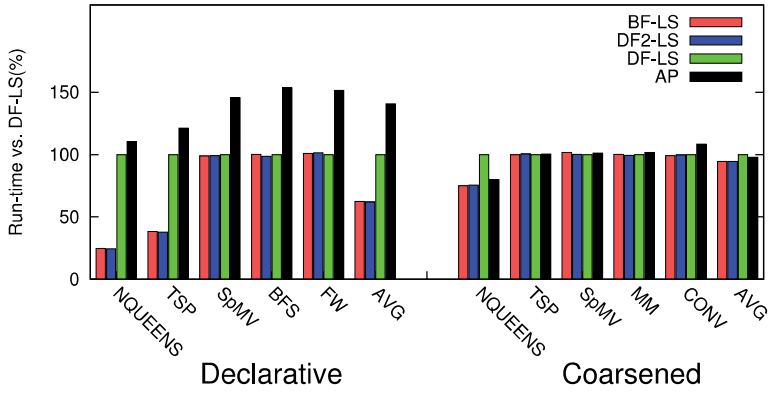

Fig. 13. Benchmarks on the Xeon using all 24 workers.

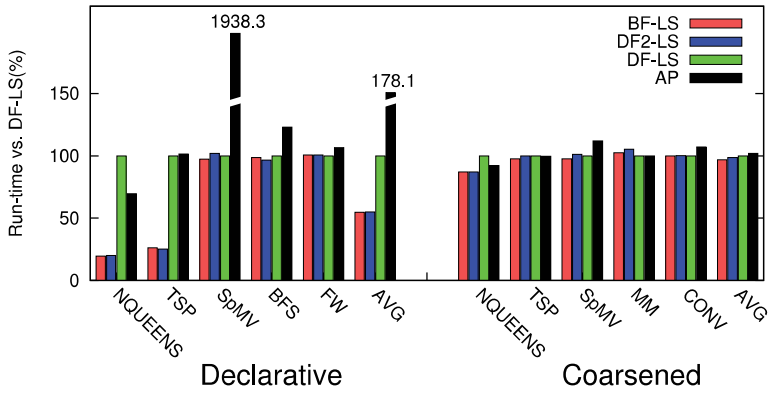

Fig. 14. Benchmarks on the T2 using all 64 workers.

normalized the execution time versus DF-LS, but since DF-LS has not been compared to AP, neither on declarative code nor on multicores, we also included AP in our performance evaluation. Compared to AP (TBB's default scheduler), all three lazy approaches are faster on declarative code and competitive on coarsened code. We also measured the performance of TBB's SP and affinity-partitioner, but because AP was consistently the best choice, we only present those results.

On small-size machines, such as the 4-core/8-thread i7 (Figure 10), the additional overheads of BF-LS and DF2-LS match the benefits of incurring fewer thefts, and DF-LS is marginally faster. We also run the same comparison on the Xeon machine using six workers (Figure 11) and on the 8-core T2 using eight workers (Figure 12). These represent small multicore platforms. The conclusion is the same for all three small platforms: the three lazy approaches perform similarly; therefore, BF-LS is preferable because it scales better.

Notice that the six-worker Xeon configuration has higher standard deviations (Table XXII in the Appendix). We believe the reason for this high variability is that workers are not pinned to cores, and the operating system naively migrates them across chips, causing them to lose their cached values. Whatever the reason, the results in Figure 11 are unreliable, but the ones in Figures 10 and 12 have low variability and are reliable.

On the larger machines, the situation is much different. On the 24-core Xeon (Figure 13), DF-LS fails to scale on the recursively nested declarative benchmarks *TSP(decl)* and *NQUEENS(decl)* and falls behind on *NQUEENS(cut)*. BF-LS and DF2-LS perform equally well, outperforming the other two approaches on average both on declarative (by 48% vs. DF-LS) and on coarsened benchmarks (by 5.5% vs. DF-LS). On the 64-thread T2 (Figure 14), DF2-LS and BF-LS are also comparable, with BF-LS being marginally better. BF-LS is 45.2% faster than DF-LS on declarative code and 3.2% on coarsened. Compared to AP, BF-LS and DF2-LS achieve significant speedups on declarative codes, while being competitive on coarsened codes.

Scaling up. Our hypothesis is that, on larger platforms, DF2-LS will suffer from a similar lack of scalability as DF-LS because they both violate the breadth-first theft principle of work stealing. Because we do not have access to larger machines, in order to test our assumption, we use a synthetic code with extremely fine-grained recursively nested tasks, which stresses the scheduler as much as possible. The synthetic code is computing the 36th Fibonnaci by exposing all the available parallelism, and Figure 15 shows the performance scaling of the three lazy schedulers on T2. We are not showing results on the smaller machines because they have too few processors to expose the scalability problems of DF2-LS. As usual, we plotted the average of 10 runs, and the standard deviations are shown in Table XXIV in the Appendix.

As expected, DF-LS does not scale well, and, similarly, DF2-LS gradually stops scaling for larger numbers of workers. With more than 32 workers, BF-LS becomes the best approach because it scales better, but for fewer workers, its higher overhead per task makes it fall behind DF2-LS. Although this single experiment is insufficient to conclusively prove or challenge our hypothesis that DF2-LS will not scale, it serves as a means to highlight the issue and offers some insight.

The low speedup numbers in Figure 15 are not surprising because we did not implement any manual cutoff and because TBB's overheads for creating a TD are relatively high. This is in part due to TBB being implemented as a library and having to create several objects to call the scheduler for each new task. However, the goal here is to show how the performance of the different schedulers scales under extreme stress to try and emulate the stress of running on a larger machine. To that effect, we think that these results provide some insight.

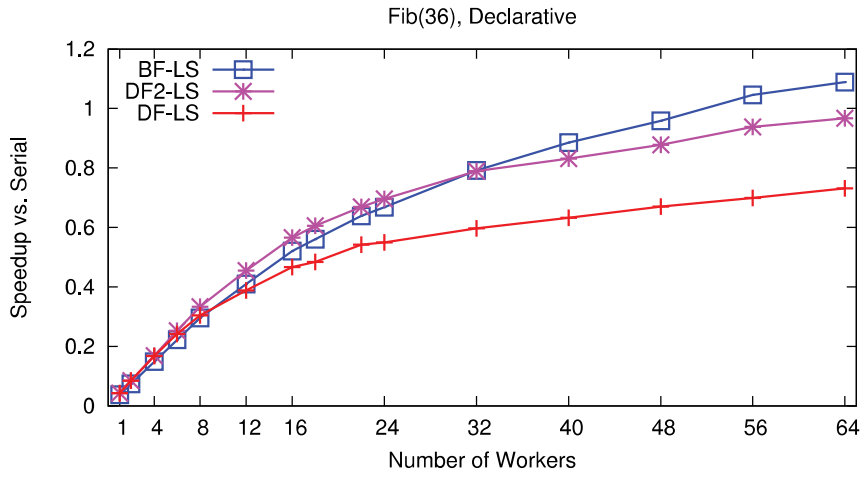

Fig. 15. Performance scaling of schedulers on T2 (Fib(36)).

Given the above numbers, our recommendation would be to always use BF-LS because its performance is more robust than the depth-first lazy approaches (DF2-LS and DF-LS) and because it does not fall significantly behind the best alternative when it is not itself the best scheduler. This gives us greater confidence that, if used on other benchmarks and applications than those presented here, BF-LS will not surprise with lower than expected performance. Moreover, if one wants to create a binary of his parallel application to be executed on platforms of varying sizes, BF-LS is the best choice. Alternatively, it may be attractive to recompile a parallel application to use a different scheduler depending on the target platform, to improve performance. This work has outlined trends to help select the best scheduler.

7.4. Software Optimality of Declarative Code

One of the claims we have made is that our work on lazy scheduling brings us one step closer to efficient execution of declarative code. We substantiate this claim by comparing the performance of declarative code to that of its coarsened counterpart. We use the definition of *software optimality* from Tzannes [2012a, Section 3.4]. Intuitively, given an input environment (the input data, the execution platform, and the subset of available workers), the software optimality of a code is the ratio of the performance[15] it achieves over the best achievable performance by changing programmer-tuned variables, such as the coarsening, the algorithm used, or the choice of runtime. In other words, software optimality shows how close or far from optimal a choice of user variables is for a given input environment.

To properly define software optimality, we should take the minimum over all possible coarsenings, but, given that the coarsenings we used were selected to maximize performance, the effort of trying all possible coarsenings to get a slightly more accurate lower bound for execution time was not justified.

In Tables VII, VIII, and IX, we present, for each of our three platforms, the software optimalities of the four compared schedulers for the three benchmarks for which we had both a declarative and a coarsened version. The results show that DF-LS achieves significantly better software optimality that AP on small platforms (i7), but fails to

[15] $Performance = \frac{1}{Execution\ Time}$.

Table VII. Software Optimality (%) of Declarative Code on i7

i7	BF-LS	DF2-LS	DF-LS	AP
NQUEENS	56.2	59.2	55.7	14.4
TSP	43.1	46.2	45.0	12.0
SpMV	97.6	97.5	97.8	27.1

Table VIII. Software Optimality (%) of Declarative Code on Xeon

Xeon	BF-LS	DF2-LS	DF-LS	AP
NQUEENS	55.3	56.0	13.7	12.4
TSP	45.5	46.2	17.4	14.4
SpMV	99.7	99.5	98.7	67.8

Table IX. Software Optimality (%) of Declarative Code on T2

T2	BF-LS	DF2-LS	DF-LS	AP
NQUEENS	38.5	37.5	7.5	10.8
TSP	25.2	26.0	6.6	6.5
SpMV	86.2	82.2	84.0	4.3

deliver on larger platforms. On the other hand, BF-LS and DF2-LS achieve several times better software optimality than AP on all platforms.

When the compiler is able to perform coarsening to amortize the overheads per task, such as for *SpMV*, the software optimality of declarative code scheduled with BF-LS (and DF2-LS) becomes competitive with that of manually coarsened code, but without compromising the performance portability. This is an indication that, using BF-LS, programmers will no longer need to prune the exposed parallelism, which is tedious and hurts performance portability. Instead, they will only be responsible for amortizing the overheads per task, which is easier and more performance portable, as the next section shows.

On the other hand, when the compiler is unable to perform coarsening to amortize the overheads per task, such as for *TSP* and *NQUEENS*, the software optimality is not close to optimal. Even so, novice programmers will not be discouraged because their first parallel implementations will achieve significant speedups with BF-LS compared to AP.

7.5. Software Optimality of Code with Amortizing Coarsening

In this section, we ask how much software optimality can be achieved by using the lazy schedulers, assuming that the compiler or the programmer have performed coarsening to amortize scheduling overheads. To that effect, we repeat the previous experiment, but this time, we add a cutoff depth for *NQUEENS* and *TSP* and use the *ppt,* computed for the lazy schedulers, with AP as well. The *amortizing cutoff depth* is such that, if the subproblem would not benefit from parallelization, it is solved sequentially. In other words, we find the *ppt* as a cutoff depth.

To find the cutoff depth for *NQUEENS*, we run increasing input sizes $n \in \{1, 2, 3, \ldots\}$ and measure the parallel execution on two workers with the cutoff depth equal to 1 (only the outermost parallel loop stays exposed), comparing it to the time taken by the sequential execution. Let k be the the minimum n for which $ParTime(n) < SerTime(n)$. Then, the cutoff function will be $N - (k - 1)$; in other words, $k - 1$ recursive levels from the bottom of the recursion. We repeat this process on each target platform to get the platform-specific value of k for each of them. We repeat the same procedure for *TSP* and find the amortizing cutoff depths for each of our multicore platforms.

Table X shows the values of $k - 1$ for each of the platforms and benchmarks. There is little variation of the profitable parallelism cutoff depth between platforms because

Table X. Amortizing Cut-Off Depths for NQUEENS and TSP

	T2	Xeon	i7
NQUEENS	5	5	5
TSP	5	5	4

Table XI. Software Optimality (%) of Amortized Code on i7

i7	BF-LS	DF2-LS	DF-LS	AP
NQUEENS	92.7	93.8	92.0	58.8
TSP	99.2	96.3	96.3	96.1
SpMV	97.6	97.5	97.8	67.5

Table XII. Software Optimality (%) of Amortized Code on Xeon

Xeon	BF-LS	DF2-LS	DF-LS	AP
NQUEENS	85.6	81.4	33.1	51.0
TSP	100.0	98.8	98.0	98.9
SpMV	99.7	99.5	98.7	81.3

only the cutoff for *TSP* on the i7 is lower. This is because, due to its smaller scale, the i7 can profit from smaller granularity of parallelism.

Tables XI, XII, and XIII show the software optimality results of the different schedulers compared to amortized code. BF-LS is the clear winner in this comparison, achieving above 85% on all three machines for all benchmarks, with DF2-LS following closely. DF-LS performs well on the small i7 but fails on the bigger ones, with worst- case software optimality below 50%, and AP has software optimality below 60% on at least one of the three benchmarks on all three machines.

These results show two things. First, lazy scheduling constitutes a significant step toward supporting declarative code because it achieves very high software performance optimality ratios on irregular codes that have been coarsened just enough to amortize scheduling overheads. Second, pruning parallelism is often necessary to achieve a good software optimality ratio with AP and other eager schedulers, even when coarsening to amortize scheduling overheads has been performed. Pruning parallelism, however, is tedious for programmers and likely to compromise performance portability by overconstraining parallelism.

Finally, Table XIV shows the number of tasks that *NQUEENS* on a 14-by-14 board exposes to the runtime with the different cutoff depths used. It shows that, despite the fact that amortized *NQUEENS* exposes an order of magnitude more tasks than its coarsened counterpart, lazy scheduling achieves 85% or more of the performance achievable with the coarsened code (Tables XI, XII, and XIII).

8. LAZY SCHEDULING FOR NONLOOP PARALLELISM – AN EXAMPLE

In this section, we apply the concepts of lazy scheduling to a custom scheduler implemented for the Unbalanced Tree Search (UTS) benchmark [Olivier et al. 2007]. Our goal is to reinforce our claim that lazy scheduling can be applied to algorithms other than work stealing and also that it can improve the performance of task parallelism introduced one at a time, not through parallel loops. Later in this section, we also experiment with our own TBB implementation of UTS, which obviates the need for custom scheduling. We used parallel for-loops in our TBB implementation because they fit best the type of parallelism present in UTS and because we have not added support to TBB for lazy parallel-invoke (TBB's version of task parallelism). Adding support for lazy scheduling of parallel-invoke task parallelism in TBB would be straightforward but tedious, and it would not grant us further insights, so we opted against it. Overall,

Table XIII. Software Optimality (%) of Amortized Code on T2

T2	BF-LS	DF2-LS	DF-LS	AP
NQUEENS	85.6	84.6	47.6	53.3
TSP	100.0	96.0	97.1	93.3
SpMV	86.2	82.2	84.0	54.9

Table XIV. Parallelism of *NQUEENS* (14x14) for Different Cutoffs

Kind	**cut-off**	**# Tasks**
Declarative	none	377,901,398
Amortized	$N-5$	46,951,002
Coarsened	$N/2$	4,294,066

we aim to illustrate that the principles and benefits of lazy scheduling extend to task parallelism and to custom schedulers, and they do not only apply to parallel loops.

UTS fully visits an unbalanced (irregular) tree, and the challenge is that the size of each subtree (and therefore the granularity of each task) is unknown until the entire subtree is visited. Two types of random trees are considered, geometric and binomial: A node in a binomial tree has m children with probability q and no children with probability $1-q$, and a node in a geometric tree has a branching factor that follows a geometric distribution with an expected value specified by a parameter $b_0 > 1$. Geometric trees have larger subtrees closer to the root (and require a cutoff depth in order to be finite); whereas binomial trees are finite when $qm < 1$, and each subtree has the same expected number of nodes regardless of its depth in the tree. Olivier et al. implemented a UTS in UPC, in OpenMP, and in Pthreads. Because our work targets UMA platforms, we only describe and modify their shared memory Pthreads implementation (their UPC and OpenMP implementations are primarily targeting NUMA clusters).

UTS implements a custom scheduler, which is eager. Each time a node is visited, its children are discovered and added, one at a time, to a shared data structure that is very similar to a deque. The difference is that the top of the deque is thread-private, and nodes pushed in that private segment can subsequently become shared through a separate operation. To avoid excessive overheads, a parameter `chunk-size` can be set on the command line (`chunk-size = 1` by default) that controls the number of nodes to move from the private to the shared segments and vice versa; whenever a deque has at least $2 \cdot$ `chunk-size` nodes in its private segment, it makes `chunk-size` of them shared by updating a pointer that delimits the private from the shared segment. Similarly, when a worker has exhausted the private segment of its deque, it reclaims `chunk-size` nodes from its shared segment or attempts to steal `chunk-size` nodes if its deque is empty.

Having a deque with a shared and a private segment deviates from the traditional work-stealing algorithm, which only has the former. It also differs from our proposed lazy approach because the decision to share work does not depend on (inferred) runtime load conditions. Olivier et al. showed that performance is significantly sensitive to the choice of `chunk-size`, even for shared memory machines, which is a serious drawback for performance portability.

To make their UTS *chunky work stealing* approach lazy, we modify the condition when nodes (tasks) are moved from the private to the shared segment: in addition to requiring the private segment to have at least $2 \cdot$ `chunk-size` nodes, we also require the shared segment to be empty. Furthermore, when both those conditions are met, we share *half* of the private segment (instead of only `chunk-size` nodes), and thefts steal the *entire* shared segment (instead of only `chunk-size` nodes) because the private

Table XV. Datasets for UTS

Tree	Type	Depth	MNodes	MLeaves
T1L	Geometric (fixed)	13	102.181	81.746 (80.00%)
T2L	Geometric (cyclic)	67	96.794	53.791 (55.57%)
T3L	Binomial	17844	111.346	89.077 (80.00%)

segment is expected to hold approximately as many nodes. These modifications follow in spirit the recursive splitting of task descriptors created by loops. Finally, when a worker runs out of work in its private deque segment, it reacquires half of its shared segment if it contains at least $2 \cdot$ `chunk-size` nodes or all of it otherwise. We call this scheduler *Lazy Splitter* and the original UTS scheduler *Eager Chunker*.

The decision to share half of the private segment whenever the shared segment is empty and to steal all of the shared segment is similar to the steal-half approach [Hendler and Shavit 2002], although it is just a *best-effort* heuristic; the scheduler does not constantly keep rebalancing the private and shared segments to keep them approximately equal as tasks (nodes) are locally produced or consumed.

In an effort to separately evaluate the performance gains of lazily moving nodes to the shared segment and those of our best-effort steal-half optimization, we also implemented a *Naive Lazy* scheduler and an *Eager Steal Half* one. Naive lazy simply shares `chunk-size` nodes whenever the shared segment is empty. Steal-half is eager, so it shares nodes as they are produced, `chunk-size` at a time, but it steals half of the available nodes from any given victim and immediately shares all but `chunk-size` of them on the thief's deque.

Table XV briefly describes the most important features of the datasets used in our evaluation. The datasets are taken as-is from the UTS project [UTSproject]. We refer the interested reader to their website for additional details.

Figure 16 shows that, for all three trees, lazy scheduling is superior (as before, we plot the average of 10 runs). On T2, the gap between lazy and eager schedulers is small ($< 25\%$), whereas on the Xeon it is at least $4\times$. Overall, Eager Chunker has high overheads because of constantly moving nodes between the shared and the private segments and because of frequent thefts of small numbers of nodes. Eager Split Half significantly reduces thefts, but apparently they account for a small fraction of the overhead and thus result in negligible performance gains.[16] Naive Lazy minimizes moving nodes to and from the shared segment, which results in significant performance benefits, but it does not minimize thefts, which causes it to fall behind Lazy Splitter on Xeon, where thefts are more expensive due to the potential of cross-chip communication. Finally, Lazy Splitter minimizes both thefts and moving nodes locally to and from the shared segment thereby achieving the best performance. It is also worth noticing that the biggest performance gap between Lazy Splitter and Naive Lazy (on Xeon) occurs with the binomial tree dataset (T3L). This is expected because, unlike geometric trees where shallower nodes are more likely to have larger subtrees, binomial trees do not, thus leading to a smaller probability that a theft will result in significant work being stolen without the use of a higher `chunk-size`. Luckily, the best-effort steal-half heuristic of Lazy Splitter is apparently sufficient to approximate the ideal `chunk-size`, as we will see below. We omit the results on i7 for brevity and because they do not offer any additional insights.

Next, we show the difference in sensitivity to the `chunk-size` parameter between the eager and lazy schedulers. Figure 17 shows that the performance with Lazy Splitter only degrades when the chunk size is too large, therefore limiting parallelism

[16]This negative result for the eager steal half concept on UTS is not sufficient to make any general claims for its benefits to other computations, schedulers, or platforms.

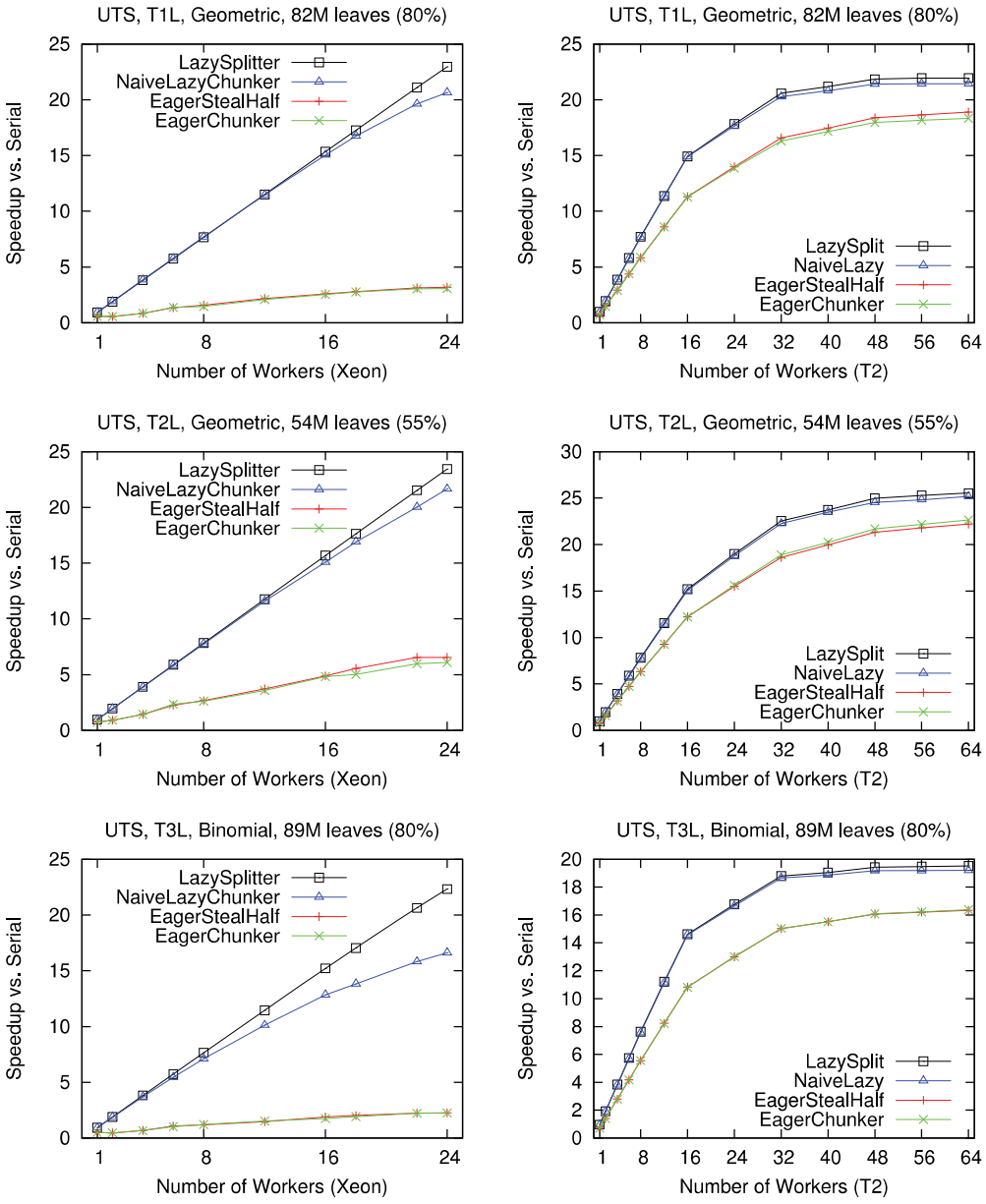

Fig. 16. UTS speedup results on Xeon and T2 (original PThreads implementation).

excessively; but, even when the chunk size is equal to 1 (i.e., as small as possible), the performance does not degrade. This is not the case for the eager approach originally used for UTS and our custom eager steal half scheduler: The choice of chunk size is crucial to achieve good performance, especially when the memory hierarchy is deeper, as in the case of the Xeon. If the chunk size is too small, scheduling overheads stifle performance, whereas if the chunk size is too large, the lack of parallelism prevents performance scalability. Moreover, the optimal chunk size

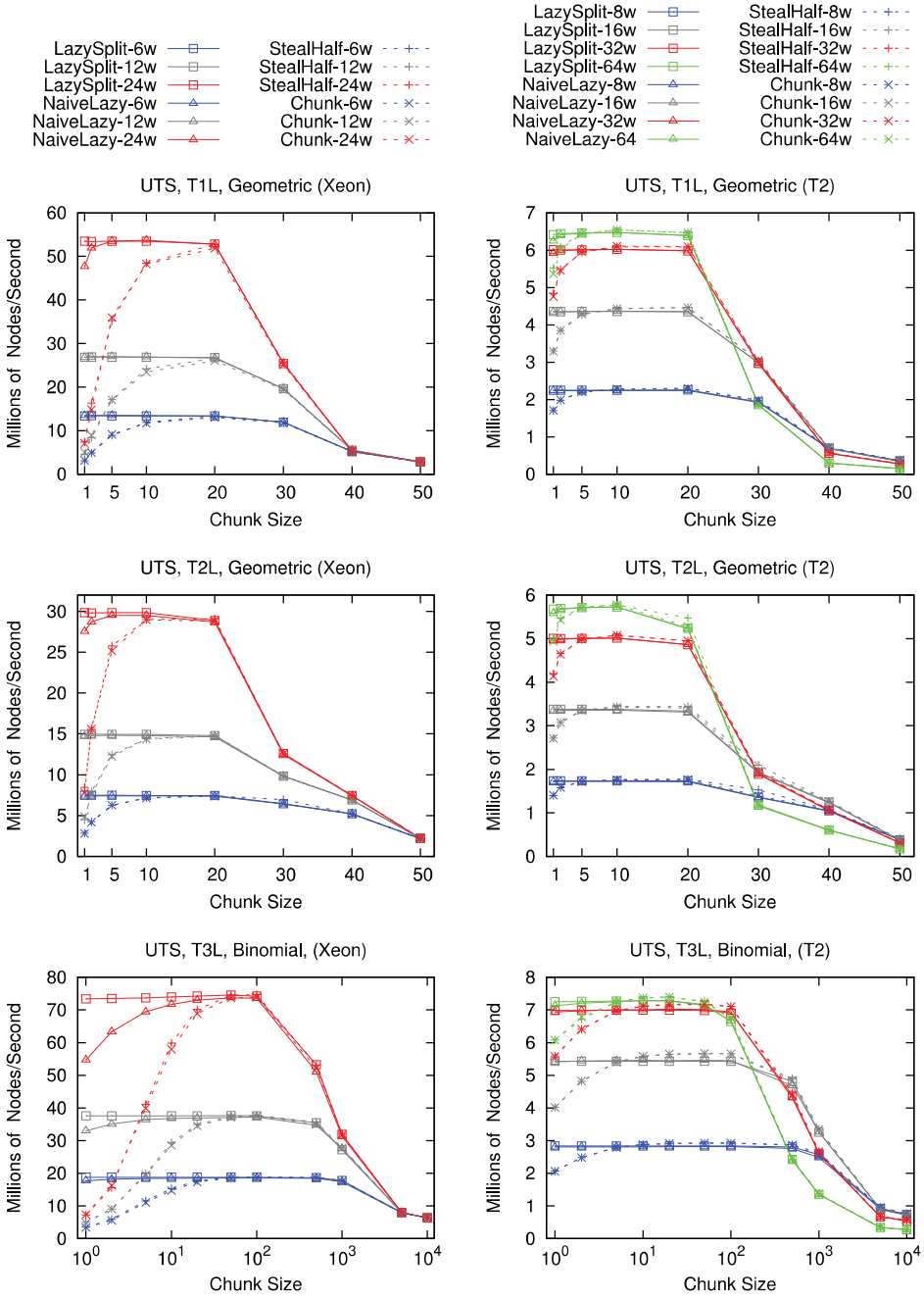

Fig. 17. UTS sensitivity results on Xeon and T2 (original PThreads implementation).

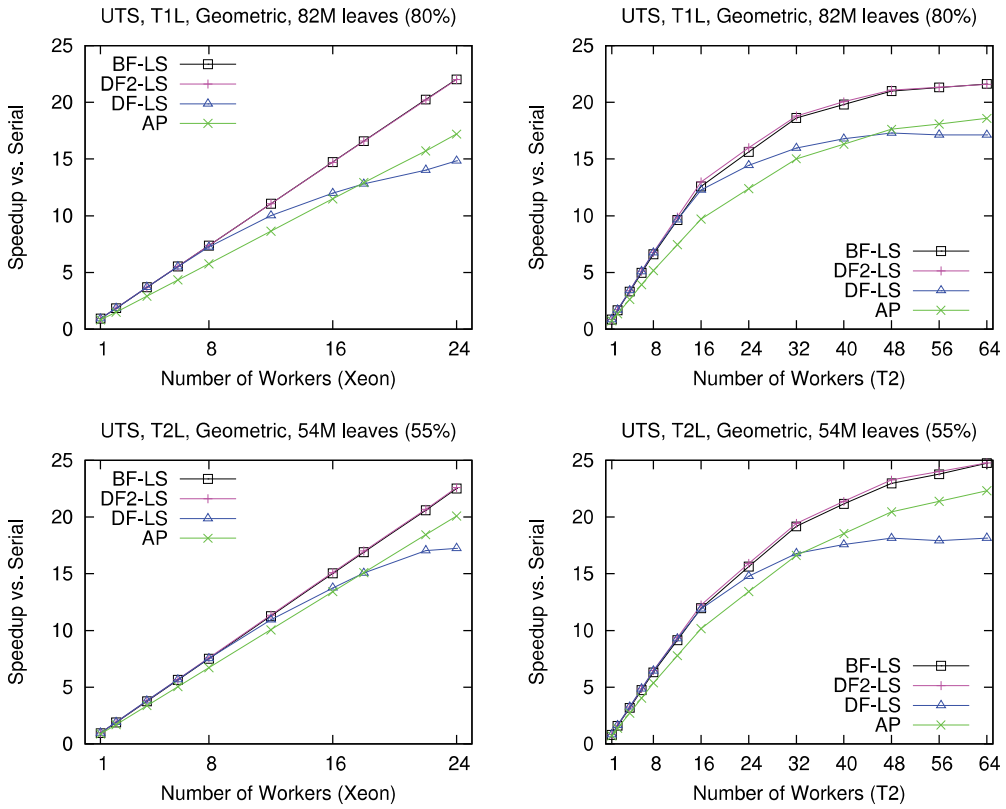

Fig. 18. UTS Speedup results on Xeon and T2 (TBB Implementation).

depends on the type of the tree (geometric vs. binomial) and possibly on the size of the tree as well [Olivier et al. 2007]. Finally, the Naive Lazy scheduler is somewhat sensitive to the choice of `chunk-size` on Xeon, but to a much lesser degree than the eager alternatives.

8.1. TBB Implementation of UTS

In this section, we present and evaluate the performance of the lazy schedulers (BF-LS, DF2-LS, DF-LS) on our implementation of UTS in TBB. We made minimal changes to the UTS code to replace PThreads parallelism and the custom scheduling code with TBB parallelism, and we used parallel for-loops because they were the best fit for UTS. We ran UTS on the two large geometric trees (T1L, and T2L), but the binomial tree (T3L) was too deep, causing the TBB execution to run out of memory, both with the existing eager schedulers and with our lazy ones. Unsurprisingly, TBB has a larger memory footprint than the custom schedulers presented in the previous section.

Figure 18 plots the speedup results for the TBB implementation with the schedulers examined throughout this article. BF-LS and DF2-LS give the best performance, AP is gives somewhat slower performance, and DF-LS only scales well up to a small number of workers, trailing AP for larger worker counts. We also ran UTS with SP, whose performance matched that of AP, so we excluded it from the plot. The gap between BF-LS and AP is smaller than for TSP or nQueens because the work per task for UTS is significantly larger. AP is performing significantly better than the eager custom

schedulers (without manual tuning of `chunk-size`) because it recursively pushes half of the children of each node instead of pushing them one at a time, effectively resulting in larger chunk sizes. Finally, notice that, on the Xeon, BF-LS is achieving 22× speedup, marginally lower than the speedup of the custom lazy splitter scheduler and very close to perfect linear speedup (24×), thus indicating that our TBB implementation is performing well. (The speedups are not self-relative but relative to an optimized sequential implementation.)

9. TIME AND SPACE BOUNDS FOR LAZY AND EAGER WORK STEALERS

In this section, we revisit the question of time and space bounds for the different work stealing schedulers that we discussed. As we mentioned, the theoretical bounds that were shown in Blumofe and Leiserson [1999] apply to computations with parallel function calls but not to parallel loops or other constructs that introduce multiple tasks at once. The bounds rely on the fact that tasks are created one at a time and need to be amended for parallel loops. We start with the space bounds for different variants of work stealing, then discuss the time bounds. We show that AP and lazy work stealing have time bounds without additional linear or logarithmic overheads in N, but this improved common-case performance comes at the cost of poorer worst-case behavior, as demonstrated by synthetic examples in Sections 9.3 and 9.4. For lazy scheduling, it is important to poll the work-pool frequently in order to maintain enough tasks available for hungry workers, which is practically always the case with declarative codes.[17] For AP, there is no good way to mitigate its worst-case behavior.

9.1. Space Bounds

Remember that the bound for stack space for vanilla work stealing (work stealing without parallel loops) is $P \cdot S_1$, where P is the number of workers and S_1 the stack space needed by the sequential (depth-first) execution. It is important to note that the bound is a function of the sequential space as we proceed to generalize the result in the presence of parallel loops.

ALGORITHM 4: Generic Parallel Loop

```
forall the i ∈ {1, ..., N} do
    CODE(i);                          // Performs O(k) work sequentially
end
```

Let us assume a generic parallel loop with N iterations (tasks), such as the one shown in Algorithm 4. Work-first work stealing creates a single TD per loop and therefore the $P \cdot S_1$ bound still holds, ignoring O(1) terms. Help-first work stealing starts by creating N TDs, one for each iteration. Thus, the space needed is $P \cdot S_1 + O(N)$. EBS with SP will recursively split the iteration range, creating $\log N$ TDs, some of which will be stolen and further split, so the space will be $P \cdot (S_1 + O(\log N/P))$. AP will only create $K \cdot P$ chunks initially instead of N for SP; thefts can induce further splits into V chunks, so the space requirement is $P \cdot (S_1 + O(\log \max(K, V)))$. Finally, all the lazy scheduling approaches have a constant upper bound B on the number of TDs per deque (e.g., one TD per deque: $B = 1$); therefore, the original bound of $P \cdot S_1$ holds (with an additional $O(B \cdot P)$ term). Table XVI summarizes these results, as well as the time

[17] In other words, long-running tasks may harm load balancing by preventing postponed tasks from becoming available unless the work-pool is polled mid-task.

Table XVI. Space and Time Bounds for Generic Parallel Loop

Scheduler	Space	Time
Work-First	$P \cdot S_1$	$T_1/P + T_\infty + \mathrm{O}(N)$
Help-First	$P \cdot S_1 + \mathrm{O}(N)$	$T_1/P + T_\infty + \mathrm{O}(N)$
Simple-Partitioner	$P \cdot S_1 + P \cdot \mathrm{O}(\log N/P)$	$T_1/P + T_\infty + \mathrm{O}(\log N)$
Auto-Partitioner	$P \cdot S_1$	$T_1/P + T_\infty + \mathrm{O}(\log P)$
Lazy Scheduling (starved)	$P \cdot S_1$	$T_1/P + T_\infty + \mathrm{O}(\log P)$
Lazy Scheduling (busy)	$P \cdot S_1$	$T_1/P + T_\infty$

bounds discussed hereafter. For AP and lazy scheduling, we have dropped the K, V, and B terms since they are typically small constants.

9.2. Time Bounds

The time bound for vanilla work stealing is $T_1/P + \mathrm{O}(T_\infty)$, where T_1 is the work (i.e., the time taken by sequential execution of the parallel code) and T_∞ is the depth (i.e., length of the critical path). For the generic loop of Algorithm 4, for example, T_∞ is the length of the longest of its N tasks.

Table XVI shows the time bounds as well. Work-first work stealing has to sequentially remove the N tasks from the single TD (which is never split), so the time bound has an additional $\mathrm{O}(N)$ term. Help-first work stealing starts by creating N TDs before even executing the first task, so it also incurs an added $\mathrm{O}(N)$ on the critical path. SP has a logarithmic overhead for the same reason. AP also has a logarithmic overhead but in $K \cdot P$ instead of N. The behavior of lazy scheduling depends on whether the system is starved or full. In the first case, lazy scheduling stops creating TDs as soon as thefts stop and the deque is above a threshold size; in other words, after at most $\mathrm{O}(B + \log P)$ time, to account for distributing in-parallel TDs to all P workers in $\log P$ rounds and for filling the B slots of the deque thereafter. In the second case, where thefts do not occur (e.g., for a nested parallel loop when parallelism from outer scopes provided workers with enough work), lazy scheduling does not incur any overhead on the critical path. These two cases appear as distinct rows of Table XVI.

Overall, lazy scheduling has the best bounds, both for space and for time, whereas the EBS approaches (SP and AP) are the next best.

Note that the time bounds for AP and lazy scheduling hold only if the work of all tasks in the parallel loop is of the same order of magnitude. In the worst case, if an adversary picked the computational cost of each task in the loop, the time bounds for AP and lazy scheduler degrade, as discussed next.

9.3. Worst-Case Scenario for Auto-Partitioner

For AP, imagine that the first chunk of tasks ($\frac{N}{KP}$ tasks) of a parallel loop contains $\mathrm{O}(W)$ work per task, whereas all of the other tasks have practically no work (i.e., $\mathrm{O}(1)$). The depth will be $T_\infty = W$, and, because the first chunk will contain N/KP tasks, the total work will be:

$$T_1 = W \cdot \frac{N}{KP} + \left(N - \frac{N}{KP}\right) = (W + KP - 1)\frac{N}{KP}.$$

However, since AP will execute all N/KP tasks as a single chunk, its critical path will be $W \cdot N/KP$, which is N/KP times longer than the critical path of the given code.

9.4. Worst-Case Scenarios for Lazy Scheduling

In this section, we show how the running time of very long tasks may, if not properly handled, reduce the amount of parallelism accessible by idle workers, resulting in

longer critical paths (and execution times). For lazy scheduling, we present two scenarios: a flat one with a single parallel loop and a nested one with recursive parallelism and a parallel loop at the leaf of the recursion. For the flat one, the critical path increases by a logarithmic factor, and, for the nested one, it increases by an arbitrary factor $g(N)$ under specific conditions. In both cases, long-running tasks prevent postponed tasks from becoming available to idle workers in a timely manner. There are several possible solutions:

—The scheduler could share all postponed tasks before executing a *long* task. Long tasks can be identified either naively (e.g., tasks with loops, I/O, etc.), by using an interprocedural cost estimation pass, or by manual user annotations. In TBB, for example, it is possible to select a different scheduler per loop, allowing us to schedule coarse-grained loops using eager scheduling and fine-grained ones using lazy scheduling.
—A compiler could inject work-pool polling in long tasks.
—Lightweight interrupts or hardware support for core-to-core signaling (e.g., Sanchez et al. [2010]) could trigger a remote function call to request that more postponed tasks be shared.

Flat Scenario. Assume worker A starts executing a parallel loop with N tasks whose first $\log N$ tasks have $W(N)$ work, and the rest have O(1). Also, assume that we stop pushing work onto a deque if it is not empty (the size threshold is one) and that thefts happen more slowly than checking the deque size after pushing a TD. Worker A will push a TD with half the tasks onto its deque and start executing the first of the "thick" $W(N)$ tasks. In the meantime, all of the $N/2$ "thin" O(1) tasks are split among the remaining workers and executed before worker A completes its first thick task. This means that $W(N) \in \square(N)$.[18] After executing its thick task, worker A will find its deque empty, push a TD with half of the remaining tasks, and start executing the next thick task. Once again, the $N/4$ thin tasks are split and executed by the remaining workers, and so on for $\log N$ rounds. Therefore, worker A will execute $O(W(N)) \log N$ work, which is $\log N$ times longer than the critical path of the original parallel loop.

Lazy scheduling is better than AP in the worst-case flat scenarios because its critical path increases by a logarithmic factor, instead of a linear one.

Nested Scenario. Algorithm 5 shows a synthetic code that forces all N tasks of a parallel loop to execute on a single worker with BF-LS, while all other workers are practically idle. Assume we call function `bar` with `depth=0` on worker A. While `depth` $< N$, it spawns off in a new task function `foo`, which does nothing (i.e., `foo(x)` $\in$ O(1)), and it calls itself recursively with a depth incremented by one.[19] When `depth` $= N$, `bar` finally executes the parallel loop. Assume no thefts have occurred until worker A starts executing the first of the N iterations.[20] At that point, worker A has task `foo(1)` on its deque, and tasks `foo(2)`, ..., `foo(N)` are postponed. While A executes that first iteration, worker B steals `foo(1)` and executes it. Worker A notices the theft and pushes `foo(2)` before executing the second iteration, and so on. Eventually, worker A has executed all N iterations of the parallel loop while worker B has executed N dummy tasks, and all other workers have remained idle.

[18]More precisely, $W(N) \in \square(\log P + \frac{N}{2P})$, which for $N \gg P$ gives $W(N) \in \square(N)$.
[19]Here, we assume a help-first scheduling order for parallel function calls, which means that the worker that calls `bar` will fork off the computation of `foo` and execute the recursive `bar` invocation. With work-first, we would have to first `spawn bar` then call `foo` to get the same effect.
[20]Alternatively, assume that a recursive call of `bar` is faster than a theft; then, for some constant $c > 1$ at recursive depth $c \cdot N$ worker A will have N postponed tasks.

ALGORITHM 5: Recursive Synthetic Scenario

```
foo (depth) begin
  | does nothing
end

bar (depth)
begin
  | if depth < N then
  |   | spawn foo (depth +1);
  |   | bar (depth +1);
  |   | sync ;
  | else
  |   | forall the i ∈ {1, ..., N} do
  |   |   | CODE(i) ;                    // Performs O(f(N)) work sequentially
  |   | end
  | end
end
```

Assuming each parallel iteration is sequential and has $f(N)$ work, the critical path of `bar(0)` is $T_\infty = N + f(N)$, and its work is $T_1 = 2N + N \cdot f(N)$. The BF-LS schedule takes $N + N \cdot f(N)$ time. For any function $f(N) \in \square(N)$, the critical path of the computation is in $O(f(N))$ but that of BF-LS is in $O(N \cdot f(N))$.

We can further generalize this example by having $g(N)$ iterations (and recursive depth) instead of N. The critical path becomes $T_\infty = g(N) + f(N)$, and the work $T_1 = 2g(N) + g(N) \cdot f(N)$. The BF-LS schedule takes $g(N) + g(N) \cdot f(N)$ time, and, for any function $f(N) \in \square(g(N))$, the critical path of the computation is $O(f(N))$, but that of BF-LS is $O(g(N) \cdot f(N))$.

This says that the critical path of a lazy schedule may asymptotically increase if the complexity of tasks ($f(N)$) is at least of the same order of magnitude as the number of tasks ($g(N)$). In other words, to avoid this situation, the time between two deque-checks (the inverse of the polling frequency) should be small relative to the available parallelism. In our implementation, the time between two deque checks is the same as the task granularity, so the task granularity should be small relative to the number of tasks. This is a good guideline, and it has been the case in all codes we have encountered. For matrix multiplication, for example, where each cell of the result array is evaluated in parallel, we have N^2 parallelism and $\square(N)$ task granularity. If we only had one task per row of the result array, we would have N parallelism and $\square(N^2)$ granularity, which would cross into the danger zone for lazy scheduling. In this example, however, because all tasks have the same granularity, lazy scheduling is unlikely to suffer imbalance penalties.

One optimization that lazy schedulers could implement is to push half of their postponed tasks, which may originate from different (dynamic) spawn sites and therefore be stored in multiple TDs. This follows in spirit our lazy scheduling implementation for the UTS benchmark and also the steal-half approach proposed by Hendler and Shavit [2002].

10. RELATED WORK

In this section, we present previous work on two types of schedulers: *(1)* schedulers that support parallel function calls or futures but not parallel loops and *(2)* schedulers that explicitly support parallel loops. Then, we present work on *throttling parallelism*,

coarsening parallelism at runtime to minimize overheads, and more related work on scheduling.

10.1. Schedulers without Parallel Loop Support

These approaches do not explicitly support parallel loops; instead, they introduce parallelism through function calls or futures, one task at a time. Handling of parallel loops explicitly opens optimization opportunities not available to parallel function calls because loops create many tasks simultaneously, instead of one at a time. Multiple tasks can be packaged into a single TD, thus greatly reducing the number of deque transactions and leading to much better performance. Work stealers that do not explicitly support parallel loops miss these optimization opportunities and deliver inferior performance. This explains why EBS (AP and SP) is our primary competitor because it explicitly supports parallel loops. Nevertheless, methods for parallel function calls are outlined here because they were the first results on work stealing and made it popular.

Work stealing has become popular in part because of its efficient implementation in the Cilk programming language [Frigo et al. 1998]. The Cilk compiler creates two clones of functions, a fast and a slow one. The fast one simply skips the synchronization of a task with its continuation if they both execute on the same worker (i.e., the continuation is not stolen). The slow clone is executed if the task is stolen and may have, therefore, executed concurrently with one of its siblings or children. This optimization is orthogonal to our proposed lazy scheduling, and the two should be combined for optimal performance. Cilk [Frigo et al. 1998] was designed for parallel function calls (i.e., relatively coarse-grained parallelism), however, and it is not optimized for parallel loops. Other approaches that focus on coarser parallelism, such as parallel function calls and futures [Kranz et al. 1989; Mohr et al. 1990; Goldstein et al. 1996; Taura et al. 1999], have the same limitations.

Arora et al. [1998] propose a nonblocking implementation of work stealing that is well suited for multiprogrammed systems. Their approach suffers from deque overflows, which can cause the program to crash. Two other approaches [Chase and Lev 2005; Hendler et al. 2006] propose complicated solutions to the overflow problem. Lazy scheduling sidesteps the problem of overflowing the deques since it will stop pushing task descriptors on a deque that exceeds a threshold size. Therefore, deques are implemented as constant-size circular arrays, and overflow is not an issue.

Acar et al. [2000] describe a method to improve the locality of work stealing. This approach is implemented in TBB [Robison et al. 2008] and called *Affinity-Partitioner* (AfP). We also compared our solutions to AfP and found it to be slower than AP on average, which is why we excluded it from the presentation. In fact, we also implemented a lazy version of AP and AfP, but they were also slower than our proposed solution, BF-LS. We believe the reason is that lazy scheduling relies on frequent deque checks to push work for hungry workers, as we argued in Section 9.4, but AP and AfP coarsen tasks into large chunks, which prevents frequent checks.

Hendler and Shavit [2002] propose stealing half the TDs of a deque instead of just one in order to better spread the work across the system, and they prove good theoretical bounds for load balance. Their approach is not applicable to lazy scheduling because, unless a higher size threshold is selected, each deque will have at most one (or two) TDs at all times. In our experience, picking a higher threshold is detrimental to performance. However, in the case of parallel loops, in which binary splitting (lazy or eager) starts by pushing a TD with half the tasks on the deque, one could say that Hendler's advice to steal half of the remaining tasks is heeded. The lazy aspect of scheduling is an added benefit in addition to the binary splitting.

10.2. Schedulers with Parallel Loop Support

When we began this work, the only work stealing schedulers that explicitly supported parallel loops were TBB's SP) and AP [Robison et al. 2008], which is why they were the focus of our comparisons. To use SP, the programmer is expected to determine a good value for the *sst* of each parallel loop by trying out various values. Moreover, this fixed threshold limits the performance portability of the code to a different number of cores, datasets, and contexts. Lazy scheduling frees the programmer from choosing a threshold manually and adapts to runtime conditions to avoid excessive splitting, without falling behind on performance. AP does not require programmer tuning, but it still falls behind lazy scheduling because it lacks context portability because it does not perform runtime adaptive coarsening; this can result in exposing excessive parallelism to the runtime in the presence of nested parallelism.

Cilk++ [Leiserson 2009] implements EBS using the SP approach with a default *sst* of one. This approach falls significantly behind lazy scheduling on code with fine-grained parallelism due to scheduling overheads. CilkPlus is the latest reincarnation of the Cilk language and follows the same approach as Cilk++ for parallel loops. Other implementations of EBS SP for loops include Microsoft's TPL [Leijen et al. 2009] and Java ForkJoin [Lea 2000].

Guo et al. [2010] present a scheduler that adaptively chooses between two work stealing approaches: work-first and help-first. In work-first, the worker picks the child task and places its continuation on the deque, whereas in help-first, it places the child task on the deque and executes the continuation. In the absence of parallel loops, choosing between the two approaches is orthogonal to lazy scheduling. If parallel loops are introduced, both approaches serialize parallelism creation and fork off work *grain* tasks at a time. On the contrary, binary splitting approaches (eager or lazy) overcome this serialization and create TDs with more tasks, thus improving the load-balancing effect of thefts.

Bergstrom et al. [2010] combined lazy scheduling with *zippers*, an approach for splitting trees, which is how arrays are represented in their functional programming language. Their *lazy tree splitting* approach shows improved performance robustness across their benchmark suite, compared to EBS with SP.

The rest of the schedulers in this subsection support parallel loops but not work stealing. OpenMP starting with version 3.0 [OpenMP Architecture Review Board 2008] recognizes the need for nested parallelism by providing primitives, but whether nesting is truly supported is often implementation specific. Frequently, OpenMP implementations serialize inner parallelism, which has serious performance limitations [Ayguade et al. 1999; Bücker et al. 2004; Tzannes et al. 2010; Tzannes 2012a].

The nano-threads library supports nested parallelism [Martorell et al. 1999] and can be used for OpenMP, but uses a ready queue or a hierarchical ready queue [Nikolopoulos et al. 1998] for scheduling, both of which can have an arbitrarily higher memory footprint than work stealing. Additionally, access to the head or tail of a queue must be synchronized among all threads (i.e., workers), and a hierarchical ready queue (a tree of queues) has a single enqueue point—the root—and it requires multiple operations to get work to the leaves, where it is dequeued. This makes them unsuitable for our goal of supporting declarative fine-grained parallelism.

Duran et al. [2005] propose a system that assigns processors to tasks by instrumenting the code and getting runtime statistics to refine the distribution. They assume, however, that the programmer has coarsened the outer parallelism into *ngroups* (similar to setting the *sst*) and has also defined the grain size (*sst*) of the inner parallelism. Lazy scheduling does not need to collect runtime statistics and does not place the burden of coarsening on the programmer.

NESL [Blelloch et al. 1993] employs complex compiler transformations to support nested parallelism by flattening [Blelloch and Sabot 1990]. NESL is an interpreted functional language without side effects, which limits its scope. Moreover, it is unclear if good performance can be achieved since only three benchmarks are evaluated (only one with nested parallelism) on three architectures, and, in most cases, their approach falls behind native code for these machines. The claim is that much better performance will be achieved if the language is compiled instead of interpreted, but we are unaware of a study quantifying this claim. The approach of flattening nested parallelism seems less fit for multithreaded platforms, such as the ones that work stealing targets, because it effectively tries to make some of the runtime scheduling choices at compile-time, with the limited information it has available, so as to partition the computation as evenly as possible among the processing units. Flattened code is, however, particularly important for the vector machines that were the basis of most supercomputers throughout the 1980s and into the '90s, when this work was published.

10.3. Parallelism Throttling

Kranz et al. [1989] and Certner et al. [2008] have also used runtime conditions to decide between creating more parallelism or executing work serially, but they rely on maintaining extra state (e.g., a global counter), which creates a memory hot-spot and does not scale well. Moreover, these approaches make irrevocable serialization decisions that may hurt load balancing. Lazy scheduling only postpones exposing parallelism to other workers and runs one or *ppt* tasks before checking the system load again.

Duran et al. [2008] propose an interesting way to limit the creation of excessive parallelism, which is not related to scheduling. In fact, they experiment with several schedulers to show that their method works well with all of them. They inject code that collects statistics about the amount of work of different procedures as a function of the depth (of the call-stack) at which they are called. When enough statistics have been collected, they turn off this profiling and use the information to decide which procedures to serialize and at what depth. Given a recursive parallel procedure such as quicksort, their approach will decide at which depth of the recursion to start calling a serial version of quicksort. This approach is orthogonal to lazy scheduling because it does not solve the need to schedule the work, and it can be applied on top of it. In fact, our coarsened recursively nested benchmarks (*TSP* and *NQUEENS*) have coarse versions with manual parallelism cutoff that achieves the same performance benefits as Duran's scheme. As our results show, even for these benchmarks, lazy scheduling (BF-LS) was able to match or exceed the performance achieved with eager schedulers. It is important to note, however, that parallelism cutoff is only applicable to certain programs.

Acar et al. [2011] propose *oracle scheduling*, a combination of static and dynamic techniques for managing the granularity of parallelism to minimize the scheduling overheads of work stealing without adversely reducing parallelism. Oracle scheduling relies on annotating every function with its asymptotic complexity, and it estimates its constant factors by means of runtime profiling. Using this information, it achieves significant performance improvements by deciding at runtime whether to serialize a task (and all of its subtasks). Combining their approach with ours to create *lazy oracle scheduling* would achieve the best of both approaches: lazy scheduling would reduce the annotation burden on the programmer[21] by efficiently executing more fine-grained tasks, and it would allow programmers to enable the dynamic coarsening of oracle scheduling on fine-grained recursive functions simply by annotating them.

[21]We assume that functions that are not annotated do not benefit from oracle scheduling but do benefit from lazy scheduling.

10.4. Other Schedulers

Vandierendonck et al. [2011] present a unified scheduler for both fork-join and dataflow parallelism, which simplifies programming of pipeline parallelism, which is not always natural or efficient with most platforms using work stealing. Sanchez et at. [2011] present an adaptation of work stealing for fine-grained and irregular pipeline parallelism.

Wen et al. [Wen and Vishkin 2008; Wen 2008] propose the eXplicit Multi-Threading (XMT) architecture as a programmable parallel platform aimed at single-task completion time. XMT offers hardware primitives that allow fast, constant-time scheduling of outer parallelism. The hardware prefix-sum unit combines and serves simultaneous work requests from workers in unit time, and a special instruction helps detect the global termination of a parallel section, which triggers a return to sequential execution. XMT does not provide a one-size-fits-all hardware scheduler but rather primitives that the compiler can harness creatively. For example, we implemented a hybrid hardware-software scheduler using the XMT hardware for outer parallelism and detecting global termination and software for scheduling inner parallelism lazily [Tzannes et al. 2010; Tzannes 2012a].

Kumar et al. [2007] propose a hardware implementation of work stealing with a fallback on software when the hardware deques overflow. We agree that some architectural support for scheduling or synchronization will be needed, but the proposed approach relies on a central unit, which does not scale. Sanchez et al. [2010] seem to share our opinion and propose Asynchronous Direct Messages (ADM), a lightweight core-to-core messaging hardware mechanism that is fully virtualizable. Their scheduling algorithm is super-eager because it tries to preemptively balance work among queues that are not empty, thus requiring some cores to act solely as scheduling managers. ADMs can cause a User-Level interrupt (ULI) to invoke a handler to receive or send a message. Such a mechanism could be used in conjunction with lazy scheduling to remove the need to poll the size of the work-pool: Theft attempts would use an ADM, which would trigger an immediate response from the victim worker via ULI.

Acar et al. [2013] investigate two implementations of work stealing using private deques and prove time bounds. Deques are not shared but private, leading to similar savings in expensive shared queue transactions as with BF-LS. Instead, the victim is responsible to answer theft requests (or to push work to hungry workers in the second version of their algorithm), so their approach has the same sensitivity to long-running tasks as lazy scheduling. Both could benefit from the ADMs described earlier, and both have significant flexibility in how to share work (e.g., by sharing the oldest or newest created tasks or by sharing half of the tasks in the private deque). Their approach is still eager, incurring a $\log N$ overhead on the critical path, and hungry processors always have to wait for a busy processor to send them work, but it avoids the $\log N$ deque transactions incurred by lazy scheduling in the intermediate case discussed in Section 4. Ideally, we think the scheduler should have private deques, be lazy (not eagerly split ranges in the private deque into $\log N$ TDs), and use ADMs or compiler-inserted deque polling to prevent long-running tasks from starving other workers.

Li et al. [2010] also propose a hardware scheduler, but not a work stealing one. It assumes a mesh interconnect with a bounded-size task queue at each port. Tasks are pushed away from the worker creating them "like a gas expands in space," and tasks are parallelized conditionally based on load conditions that are approximated by the occupancy (size) of the local task queue. This is very similar in spirit to our approach but requires hardware support and a point-to-point interconnect. Furthermore, their approach may cause depth (priority) inversion, where a task of shallower nesting level takes precedence over a task of deeper nesting created locally (i.e., depth-first execution

is not always followed). This inversion happens because tasks being migrated are taken from the head (oldest) of the local queue and pushed onto the tail of the remote queue. In addition to potentially affecting temporal locality, this may also lead to an unbounded memory footprint by having a "mostly depth-first but sometimes breadth-first" execution.

Shirako et al. [2009] present parallel loop transformations for statically chunking parallel loops that contain synchronizations (e.g., barriers) that are illegal in the execution models of the languages we are focusing on, such as Cilk++, TBB, XMTC, and others. This style of coding (i.e., parallel for-loop with barriers) is common in OpenMP code, which traditionally targets clusters. On those parallel machines, locality of data is much more important, and barriers allow the same workers to work on the same data after synchronizing on a barrier. To achieve the same synchronization, we would use consecutive parallel loops, but then tasks could dynamically map to different processors, thus losing the locality benefits of barriers. TBB actually has an *affinity partitioner* that is a best-effort approach for mapping the same iteration subspace of consecutive parallel loops to the same workers [Robison et al. 2008]. Shirako et al. do not make clear the motivation of using unintuitive barriers in parallel loops and do not compare to TBB's AfP, but their use of well-known static transformations (strip mining, loop interchange, loop unswitching, and loop distribution) to achieve static coarsening in their complex execution model is nonetheless interesting.

Kumar et al. [2012] propose a series of dynamic (just-in-time) optimizations that attempt to move as much of the work stealing overhead from the critical path of busy workers to that of thieves. They rely on the stack-walk capabilities of the managed runtime of Java (and X10) and achieve impressive results. Unfortunately, their techniques are unlikely to transfer in some form to languages closer to the metal, like C/C++.

Finally, work stealing is starting to cross into distributed memory platforms with several tweaks, such as hierarchies of deques to improve locality and reduce randomized cross-node thefts [Quintin and Wagner 2010; Min et al. 2011] or asynchronous thefts [Li et al. 2013].

11. CONCLUSION

In this article, we presented *lazy scheduling*, a scheduling technique that adapts to runtime load conditions in order to minimize scheduling overheads. We combined lazy scheduling with work stealing, the most popular dynamic scheduling algorithm currently used for general-purpose shared-memory task-parallel programs, and we presented three variants of this lazy work stealing, BF-LS, DF-LS, and DF2-LS. We experimented with these three variants, as well as with TBB's AP on a set of benchmarks on three different commercial multicores and showed that BF-LS has distinct performance advantages on codes with fine-grained nested parallelism over AP, TBB's default scheduler. We also showed that DF-LS fails to scale to larger numbers of workers on multicores and demonstrated that BF-LS is the most scalable of the three variants. In terms of software performance optimality ratio, we found that BF-LS is significantly better than AP, both on declarative and amortized codes. This allows programmers to expose parallelism more liberally since finer task granularity can be efficiently executed, thus enhancing ease of programming and performance portability. We also implemented a lazy variant of a custom scheduler for the UTS benchmark to demonstrate how the principles of lazy scheduling can be applied to nonloop parallelism and that they are successful in reducing scheduling overheads in that case as well. Finally, we presented the time and space bounds of work-stealing variants (lazy and eager) for parallel loops and constructed artificial scenarios to expose the worst-case behaviors

of lazy scheduling and AP. The insight gained is that the frequency of deque checks should be large relative to the program parallelism in order to avoid increasing the critical path of lazy schedules.

REFERENCES

2008. Intel Threading Building Blocks Reference Manual, Rev. 1.9. (2008).

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The data locality of work stealing. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*. ACM, New York, NY, 1–12. DOI:http://dx.doi.org/10.1145/341800.341801

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2011. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'11)*. ACM, New York, NY, 499–518. DOI:http://dx.doi.org/10.1145/2048066.2048106

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, 219–228.

George S. Almsasi and Allan Gottlieb. 1994. *Highly Parallel Computing* (2nd ed.). Benjamin/Cummings.

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'98)*. ACM, New York, NY, 119–129. DOI:http://dx.doi.org/10.1145/277651.277678

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, Berkeley. Available at http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

Eduard Ayguade, Xavier Martorell, Jesus Labarta, Marc Gonzalez, and Nacho Navarro. 1999. Exploiting multiple levels of parallelism in OpenMP: A case study. In *Proceedings of the 1999 International Conference on Parallel Processing (ICPP'99)*. IEEE Computer Society, Washington, DC.

Lars Bergstrom, Mike Rainey, John Reppy, Adam Shaw, and Matthew Fluet. 2010. Lazy tree splitting. In *Proceedings of the 15th International Conference on Functional Programming (ICFP'10)*.

Guy Blelloch and Gary W. Sabot. 1990. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing* 8 (1990), 119–134.

Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. 1993. Implementation of a portable nested data-parallel language. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93)*. ACM, New York, NY, 102–111. DOI:http://dx.doi.org/10.1145/155332.155343

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM* 46, 5 (Sept. 1999), 720–748. DOI:http://dx.doi.org/10.1145/324133.324234

H. Martin Bücker, Arno Rasch, and Andreas Wolf. 2004. A class of OpenMP applications involving nested parallelism. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*. ACM, New York, NY, USA, 220–224. DOI:http://dx.doi.org/10.1145/967900.967948

F. Warren Burton and M. Ronan Sleep. 1981. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture (FPCA'81)*. ACM, New York, NY, 187–194. DOI:http://dx.doi.org/10.1145/800223.806778

Olivier Certner, Zheng Li, Pierre Palatin, Olivier Temam, Frederic Arzel, and Nathalie Drach. 2008. A practical approach for reconciling high and predictable performance in non-regular parallel programs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*. ACM, New York, NY, 740–745. DOI:http://dx.doi.org/10.1145/1403375.1403555

David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'05)*. ACM, New York, NY, 21–28.

CilkPlus. 2011. Homepage. Available at http://software.intel.com/en-us/articles/intel-cilk-plus/.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software* 38, 1, Article 1 (Dec. 2011), 25 pages. DOI:http://dx.doi.org/10.1145/2049662.2049663

Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. 2008. An adaptive cut-off for task parallelism. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press, 36:1–36:11.

Alejandro Duran, Marc Gonzàlez, and Julita Corbalán. 2005. Automatic thread distribution for nested parallelism in OpenMP. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*, 121–130.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, 212–223.

Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. 1996. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing* 37, 1 (1996), 5–20. http://www.cs.cmu.edu/~seth/papers/goldstein96-jpdc.pdf

Yi Guo, Jisheng Zhao, V. Cave, and V. Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of the 2010 IEEE International Symposium Parallel Distributed Processing*. 1–12. DOI:http://dx.doi.org/10.1109/IPDPS.2010.5470425

Robert H. Halstead, Jr. 1984. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP'84)*. ACM, New York, NY, 9–17. DOI:http://dx.doi.org/10.1145/800055.802017

Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. 2006. A dynamic-sized nonblocking work stealing deque. *Distributed Computing* 18, 3 (February 2006), 189–207. DOI:http://dx.doi.org/10.1007/s00446-005-0144-5

Danny Hendler and Nir Shavit. 2002. Non-blocking steal-half work queues. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing (PODC'02)*. ACM, New York, NY, 280–289. DOI:http://dx.doi.org/10.1145/571825.571876

D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. 1989. Mul-T: A high-performance parallel Lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI'89)*. ACM, New York, NY, 81–90. DOI:http://dx.doi.org/10.1145/73141.74825

Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural support for ed parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, NY, 162–173. DOI:http://dx.doi.org/10.1145/1250662.1250683

Vivek Kumar, Daniel Frampton, Stephen M. Blackburn, David Grove, and Olivier Tardieu. 2012. Work-stealing without the baggage. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*. ACM, New York, NY, USA, 297–314. DOI:http://dx.doi.org/10.1145/2384616.2384639

Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande (JAVA'00)*. ACM, New York, NY, 36–43. DOI:http://dx.doi.org/10.1145/337449.337465

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*. ACM, New York, NY, 227–242. DOI:http://dx.doi.org/10.1145/1640089.1640106

Charles E. Leiserson. 2009. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*. ACM, New York, NY, 522–527. DOI:http://dx.doi.org/10.1145/1629911.1630048

Shigang Li, Jingyuan Hu, Xin Cheng, and Chongchong Zhao. 2013. Asynchronous work stealing on distributed memory systems. In *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'13)*. 198–202. DOI:http://dx.doi.org/10.1109/PDP.2013.35

Zheng Li, Olivier Certner, Jose Duato, and Olivier Temam. 2010. Scalable hardware support for conditional parallelization. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, NY, 157–168. DOI:http://dx.doi.org/10.1145/1854273.1854297

Xavier Martorell, Eduard Ayguadé, Nacho Navarro, Julita Corbalán, Marc González, and Jesús Labarta. 1999. Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors. In *Proceedings of the 13th International Conference on Supercomputing (ICS'99)*. 294–301.

Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. 2009. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM, New York, NY, USA, 45–54. DOI:http://dx.doi.org/10.1145/1504176.1504186

S. Min, C. Iancu, and K. Yelick. 2011. Hierarchical work stealing on manycore clusters. In *5th Conference on Partitioned Global Address Space Programming Models*.

Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. 1990. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP'90)*. ACM, New York, NY, 185–197. DOI:http://dx.doi.org/10.1145/91556.91631

Dorit Naishlos, Joseph Nuzman, Chau-Wen Tseng, and Uzi Vishkin. 2001. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*. ACM, New York, NY, USA, 93–102. DOI:http://dx.doi.org/10.1145/378580.378597

Dorit Naishlos, Joseph Nuzman, Chau-Wen Tseng, and Uzi Vishkin. 2003. Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. *Theory of Computing Systems* 36, 5 (2003), 521–552. DOI:http://dx.doi.org/10.1007/s00224-003-1086-6

Dimitrios S. Nikolopoulos, Eleftherios D. Polychronopoulos, and Theodore S. Papatheodorou. 1998. Efficient runtime thread management for the nano-threads programming model. In *Proceedings of the 2nd IEEE IPPS/SPDP Workshop on Runtime Systems for Parallel Programming, LNCS*. 183–194.

Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2007. UTS: An unbalanced tree search benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing (LCPC'06)*. Springer-Verlag, Berlin, 235–250.

OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface, Ver. 3.0 May 2008. Available at http://www.openmp.org.

Jean-Noël Quintin and Frédéric Wagner. 2010. Hierarchical work-stealing. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I (EuroPar'10)*. Springer-Verlag, Berlin, 217–229.

A. Robison, M. Voss, and A. Kukanov. 2008. Optimization via reflection on work stealing in TBB. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. 1–8. DOI:http://dx.doi.org/10.1109/IPDPS.2008.4536188

Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic fine-Grain scheduling of pipeline parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE Computer Society, Washington, DC, 22–32. DOI:http://dx.doi.org/10.1109/PACT.2011.9

Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible architectural support for fine-grain scheduling. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, NY, 311–322. DOI:http://dx.doi.org/10.1145/1736020.1736055

Jun Shirako, Jisheng M. Zhao, V. Krishna Nandivada, and Vivek N. Sarkar. 2009. Chunking parallel loops in the presence of synchronization. In *Proceedings of the 23rd International Conference on Supercomputing (ICS'09)*. ACM, New York, NY, 181–192. DOI:http://dx.doi.org/10.1145/1542275.1542304

Kenjiro Taura, Kunio Tabata, and Akinori Yonezawa. 1999. StackThreads/MP: Integrating futures into calling standards. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*. ACM, New York, NY, 60–71. DOI:http://dx.doi.org/10.1145/301104.301110

Alexandros Tzannes. 2012a. *Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming*. Ph.D. Dissertation. University of Maryland, College Park.

Alexandros Tzannes. 2012b. Segmentation fault with recursively nested parallelism (gcc snapshot). Intel Cilk Plus User Forum. Available at http://software.intel.com/en-us/forums/intel-cilk-plus/.

Alexandros Tzannes. 2013a. Code and datasets for all benchmarks presented in this article (TBB & UTS). Available at https://github.com/atzannes/TBBBenchmarks.

Alexandros Tzannes. 2013b. Implementation of Lazy TBB. https://github.com/atzannes/LazyTBB-v3.0. (2013).

Alexandros Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. 2010. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. ACM, 179–190.

UTSproject. 2007. The Unbalanced Tree Search Benchmark. Available at http://sourceforge.net/p/uts-benchmark/wiki/Home/.

Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. 2011. A unified scheduler for recursive and task dataflow parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE Computer Society, Washington, DC, 1–11. DOI:http://dx.doi.org/10.1109/PACT.2011.7

Uzi Vishkin. 2011. Using simple abstraction to reinvent computing for parallelism. *Communications of the ACM* 54 (Jan. 2011), 75–85. Issue 1. DOI:http://dx.doi.org/10.1145/1866739.1866757

Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. 1998. Explicit multi-threading (XMT) bridging models for instruction parallelism (extended abstract). In *Proceedings of the 10th Annual ACM*

Symposium on Parallel algorithms and architectures (SPAA'98). ACM, New York, NY, USA, 140–151. DOI:http://dx.doi.org/10.1145/277651.277680

Xingzhi Wen. 2008. *Hardware Design, Prototyping and Studies of the Explicit Multi-Threading (XMT) Paradigm*. Ph.D. Dissertation. University of Maryland, College Park.

Xingzhi Wen and Uzi Vishkin. 2008. FPGA-based prototype of a PRAM-on-chip processor. In *Proceedings of the 2008 Conference on Computing Frontiers (CF'08)*. 55. DOI:http://dx.doi.org/10.1145/1366230.1366240

Received October 2012; revised December 2013; accepted January 2014