Beyond Nested Parallelism: Tight Bounds on Work-Stealing Overheads for Parallel Futures

Daniel Spoonhower† Guy E. Blelloch† Phillip B. Gibbons* Robert Harper†

†Carnegie Mellon University
{spoons,blelloch,rwh}@cs.cmu.edu

*Intel Research Pittsburgh
phillip.b.gibbons@intel.com

ABSTRACT

Work stealing is a popular method of scheduling fine-grained parallel tasks. The performance of work stealing has been extensively studied, both theoretically and empirically, but primarily for the restricted class of nested-parallel (or fully strict) computations. We extend this prior work by considering a broader class of programs that also supports pipelined parallelism through the use of parallel futures.

Though the overhead of work-stealing schedulers is often quantified in terms of the number of steals, we show that a broader metric, the number of *deviations*, is a better way to quantify work-stealing overhead for less restrictive forms of parallelism, including parallel futures. For such parallelism, we prove bounds on work-stealing overheads—scheduler time and cache misses—as a function of the number of deviations. Deviations can occur, for example, when work is stolen or when a future is touched. We also show instances where deviations can occur independently of steals and touches.

Next, we prove that, under work stealing, the expected number of deviations is $O(Pd+td)$ in a P-processor execution of a computation with span d and t touches of futures. Moreover, this bound is existentially tight for any work-stealing scheduler that is *parsimonious* (those where processors steal only when their queues are empty); this class includes all prior work-stealing schedulers. We also present empirical measurements of the number of deviations incurred by a classic application of futures, Halstead's quicksort, using our parallel implementation of ML. Finally, we identify a family of applications that use futures and, in contrast to quicksort, incur significantly smaller overheads.

Categories and Subject Descriptors

C.4 [**Performance of systems**]: Performance attributes; D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages

General Terms

Performance

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.

Keywords

scheduling, work stealing, futures, performance bounds

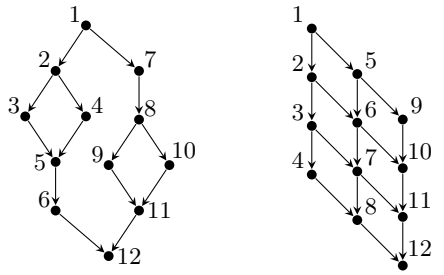
1. INTRODUCTION

Work stealing [12, 18] is a popular class of schedulers for balancing load among processors with relatively low overhead. The performance of these schedulers has been extensively studied, both theoretically [9] and empirically [10] for nested-parallel (or fully strict) programs. Nested parallelism, however, is a fairly restrictive form of parallelism and cannot be used to implement parallel pipelining [5]. In contrast, parallel pipelines can be constructed using parallel futures [18] as well as other more general forms of parallelism. Though previous work has described and bounded the number of steals for a work-stealing scheduler that supports futures [3], the scheduler overhead, including time and additional cache misses, cannot be bounded in terms of the number of steals.

As an alternative to the number of steals, we show that the number of *deviations* is a better metric for measuring the performance of work-stealing schedulers for unrestricted parallelism. Informally, a deviation occurs in a parallel execution whenever a processor executes instructions in a different order than the sequential implementation (but without violating sequential dependencies). Deviations are necessary to exploit parallelism in an application, but in general, the fewer the deviations the better. We prove that for general forms of parallelism, the number of deviations can be used to bound both forms of overhead mentioned above.

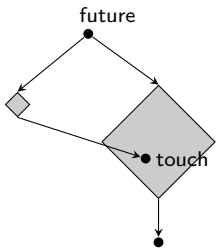
Unfortunately, in programs that use futures, deviations can occur much more frequently than steals. For example, parallel executions under work stealing with only a single steal and a single "touch" of a future can still result in up to d deviations where d is the span of the computation. We provide the first non-trivial upper bound on the number of deviations, as follows. We say that a work-stealing scheduler is *parsimonious* if there is one task queue per processor and a processor only steals when its local queue is empty. (Both the implementation in Cilk [15] and the scheduler described by Arora et al. [3] are parsimonious.) We prove that the expected number of deviations using a parsimonious work-stealing scheduler in a P-processor execution is $O(Pd + td)$, where t is the number of touches of futures. Moreover, we describe a family of computation graphs derived from programs that use futures for which this bound is tight.

These ideas arose in the context of our parallel extension of Standard ML [23]. Lacking a good model for performance, it was not clear to us how to implement futures using a work-stealing scheduler and, in particular, how to apply opti-



(a) nested parallelism

(b) pipelining



(c) parallel future

Figure 1: Computation Graphs. Parallelism is well-nested in part (a) but not parts (b) or (c). Part (b) shows an example of a parallel pipeline, and part (c), a single future interleaved with nested parallelism.

mizations described in previous work to our implementation. The current work serves to guide how and when these optimizations can be applied and suggests some alternatives that we plan to explore in our ongoing work.

Finally, we consider applications that use futures and have appeared in previous literature. We show one case where parsimonious work stealing will incur high overheads. We also give a set of strong but not onerous restrictions that guarantee significantly less overhead.

In summary, the contributions of this work include an analysis of an implementation of parallel futures using work stealing and specifically: (i) examples of new and subtle cases where deviations can occur and bounds on overheads as a function of deviations (Section 3); (ii) upper and lower bounds on the number of deviations for parsimonious work-stealing schedulers in the presence of futures (Section 4); and (iii) an empirical study of a classic application of parallel futures and a characterization of a class of programs that use futures in an efficient way (Section 5).

2. PRELIMINARIES

Futures and Parallel Pipelining. Parallel futures enable an expression to be computed in parallel with the context in which it appears. We consider parallel futures where the body of the future begins evaluation before its enclosing context [19]. Any expression can be computed in parallel by adding the future keyword in front of it. We refer to the result of evaluating that expression as the *value of the future*. Programmers demand the value of a future using an explicit touch operation [4].[1] If that value has not yet been computed, then the touch will suspend the current thread and that thread will be restarted after the value is ready.

Futures can be used to write strictly more programs than can be written with nested parallelism. The intuition behind this is twofold: first, futures syntactically separate the beginning of parallel execution from its end; second, futures may be embedded within data structures to create streams of values that are computed in parallel. Parallel pipelining is an example of a style of program that uses futures and cannot be expressed with nested parallelism. Parallel pipelines can be as simple as a single producer-consumer pair or far more sophisticated [5].

This paper focuses on futures as they seem to be a modest extension of nested parallelism that already reveals fundamental differences. Many of our results carry over to even more expressive forms of parallelism, including synchronization variables [6], that can be used to implement futures.

Parallel Computation Graphs. As in previous work [8, 9, 7], we use directed acyclic graphs to represent parallel computations. Each graph represents the dynamic execution of a program for a particular input, but as we limit ourselves to deterministic parallelism, this graph does not depend on the order in which nodes are executed. Nodes in a computation graph represent instructions or tasks, while edges represent sequential dependencies. An edge from u to v means that u must be executed before v. Edges are added to the graph to show where new threads are created, at join points, at touches of a future, or because of ordinary control dependencies. Each node is given a (non-zero, integral) weight that indicates the time required to execute it. A node v is *ready* if all of v's predecessors have been executed but v has not. The *span* (equivalently, *depth* or *critical path length*) of a graph is the weight of the heaviest weighted path from the root node to the sink.

Figure 1(a) shows an example of a program that uses nested parallelism. This form of parallelism is also known as *fork-join* or *fully strict* parallelism, and the resulting graphs as *series-parallel* graphs. Figure 1(b) shows an example of a use of futures to create a parallel pipeline. Here the pipeline has three stages, represented by the three columns of nodes. Figure 1(c) shows the use of a single future and touch within the context of a larger computation. Note that parallelism is not well-nested in this case, because there is a fork on the right-hand side that occurs after the future is spawned but is not joined until after the touch. The left edge leading out from the root of this example is a *future edge* and the edge leading back from the left-hand sub-graph is a *touch edge*.

As in previous work [3, 1], we assume that nodes in the computation graph have both out-degree and in-degree of at most two. This assumption fits with our model of nodes as the instructions of the program. A future that is touched several times can be implemented by a sequence of nodes with out-degree two. Below, we define a unique sequential execution for each computation graph. Nodes in Figure 1(a) and (b) are labeled according to the order of this execution, and we will render graphs so that the sequential execution is the left-to-right, depth-first traversal of the graph.

Work-Stealing Schedulers. In this work, we are interested in a style of work stealing implemented and studied as

[1]Explicit touches enable the language implementation to omit runtime checks to determine if a value is a future or not and instead to insert these checks only when the user has explicitly used a touch operation.

```
/* Each processor i maintains a deque Q_i of tasks. */
Processor i repeatedly executes the following:
  If Q_i is empty, select another deque Q_j at random and
    steal the top task of Q_j (if any), pushing it onto Q_i
  Else
    Pop the bottom task t from Q_i and execute it
    If t forks a new task
†     Push the continuation task onto the bottom of Q_i
      Push the spawned task onto the bottom of Q_i
    Else if t joins two tasks (as part of a fork-join)
      If both such tasks are finished, push the
        continuation task onto the bottom of Q_i
      Else do nothing
    Else if t spawns a future
      Push the continuation task onto the bottom of Q_i
      Push the future task onto the bottom of Q_i
    Else if t touches a future but that future has not yet
        been computed
      Mark t as a task suspended on that future
    Else if t finishes computing the value of a future
      If some task is suspended on that future, push the
        suspended task onto the bottom of Q_i
      Push the continuation task (if any) onto the bottom
        of Q_i
    Else /* t is an ordinary task */
      Push the continuation task onto the bottom of Q_i
```

Figure 2: WS Scheduler. This abstract description of the WS scheduler for programs using parallel futures shows the order in which parallel tasks will be executed but does not include any optimizations. The line marked † is considered in Section 3.2.

part of the Cilk programming language [9, 15]. While Cilk supports only nested parallelism, Arora et al. [3] extend the work to more general forms of deterministic parallelism, and prove that the expected number of steals in a P-processor execution of span d is $O(Pd)$ (the same as in the nested-parallel case). Though they did not explicitly study futures, the class of parallelism considered subsumes the use of futures, and hence we can consider how their scheduler would handle them. Figure 2 depicts the scheduler, which we call WS.

For our lower bounds, we shall consider any work-stealing scheduler that is *parsimonious*, that is, any scheduler that maintains one deque per processor and only steals when a processor's local deque is empty. Parsimonious work-stealing schedulers attempt to avoid paying for steals by executing tasks that are available locally. This class includes WS and, to our knowledge, all prior work-stealing schedulers.

3. STEALS AND DEVIATIONS

As in previous work, our goal is to quantify the performance of a parallel execution in terms of the performance of the sequential execution and properties of the parallel computation graph. We define a metric, called a deviation, that precisely captures when and where a parallel execution differs from the sequential execution. Though similar metrics have been used in previous work [11, 1] as vehicles for relating performance to the number of steals, we show that for programs that use futures, deviations are a better metric than steals. In this section, we define deviations, discuss the sources of deviations, and use deviations to extend previous bounds on the performance of work stealing to more general forms of parallelism.

Deviations. We define the sequential execution to be the strict linear order on nodes of the computation graph determined by the 1-processor execution of the WS scheduler and write $u <_1 v$ if u appears before v in the sequential execution. Any order gives rise to a notion of *adjacency* (or a *covering relation*). Given an order $<$ we say that two nodes u and v are *adjacent* and write $u \prec v$ if $u < v$ and there is no node w such that $u < w$ and $w < v$. If the order is linear, then there is at most one u for each v such that $u \prec v$.

For any fixed schedule, each processor p gives rise to a strict linear order $<_p$ on the nodes executed by that processor. We now define a *deviation from the sequential execution*, or more briefly a *deviation*, as follows.

Definition 1 (Deviation)**.** *A* deviation *occurs when a processor p executes node v if there exists a node u such that $u \prec_1 v$ but $u \not\prec_p v$.*

In other words, if v is executed immediately after u in the sequential execution, a deviation occurs whenever the processor that executes v does not execute u, when that processor executes one or more additional nodes between u and v, or when that processor executes u after v. We define the total deviations of a parallel execution as the sum of the deviations of each processor.

Deviations in Nested Parallelism. Acar et al. [1] show that for nested-parallel programs, each deviation incurred by WS is caused either directly or indirectly by a steal and that at most two deviations can be attributed to each steal.[2] We briefly review these sources of deviations before considering those in programs that use parallel futures.

Figure 3(a) shows a small parallel computation graph where nodes are labeled by the order in which they are executed in the sequential execution. Nodes at which deviations occur are circled. Part (b) shows one possible schedule using two processors. Maximal sequences of non-deviating execution are boxed. Note that while deviations take up space in our illustrations, they do not correspond to a period of time during which a processor is idle.

In the schedule in Figure 3(b), processor p begins by executing tasks 1, 2 and 3. Processor q steals work from p and executes tasks 4 and 5. Since processor q does not execute task 3, this steal results in a deviation at task 4. A steal will always result in at least one deviation. A second deviation at task 6 will occur because q finishes task 5 before p finishes task 3. Though p itself does not steal any work, we can attribute this deviation to the earlier steal. As noted above, for nested-parallel programs, WS will incur at most two deviations per steal.

3.1 Deviations Resulting From Futures

We now describe additional sources of deviations in programs that use parallel futures when using a work-stealing scheduler. Figure 3(c) shows a computation graph that will be used as a running example in this section.

Suspend Deviations. A deviation occurs when a task attempts to touch a future that has not yet been computed. In Figure 3(d), task 17 is the touch of the future that is

[2]Acar et al. [1] use "drifted node" instead of deviation.

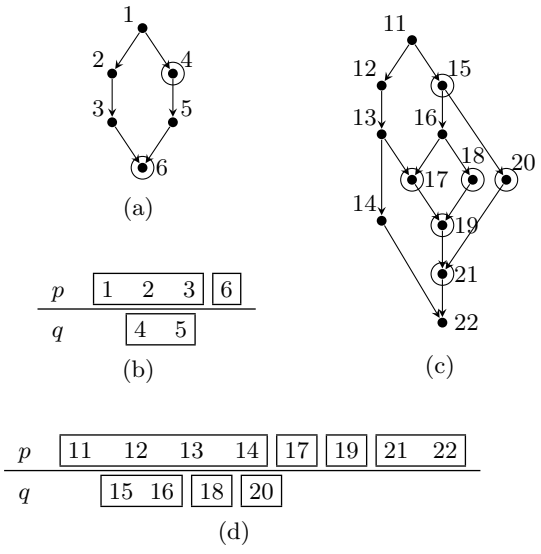


Figure 3: Example Deviations. A parallel computation graph (a) and a schedule (b) for nested parallelism. A parallel computation graph (c) and a schedule (d) for parallel futures. The edges between nodes (13, 17) and (14, 22) are touch edges. Nodes at which deviations occur are circled.

computed by task 13. In the schedule shown in part (d), processor q executes task 16 but deviates by executing task 18 (since task 17 is not yet ready). Note that this does not require an additional steal: in WS, task 18 sits at the bottom of q's deque.

Resume Deviations. A deviation also occurs when a suspended task is resumed. In the figure, task 17 is ready only after task 13 is completed. Thus after task 13 is executed, both tasks 17 and 14 will be pushed on to p's deque (in that order). When task 17 is popped and executed, a deviation will occur. In this example, there are no other tasks in p's deque when task 17 is executed, but it is possible that there could be more tasks in the deque above task 17. In that case, a second deviation would occur when the first of these tasks is popped and executed.

Indirect Deviations. The most subtle deviations occur through the use of multiple futures or a combination of nested parallelism and futures. Deviations can occur when a task is suspended waiting for another task which is itself suspended. In Figure 3, this occurs when task 20 is executed. This deviation cannot be attributed directly to a steal. Instead, it is caused indirectly by the suspension of task 17. As shown in Section 4.2, a single steal followed by a single touch of a future can result in up to $d/2$ deviations (where d is the span of the graph).

Indirect deviations can also occur on the processor that resumes a suspended touch. In this example, deviations occur at tasks 19 and 21 on a processor that has neither stolen work nor touched a future. These deviations occur when a series of suspended tasks must be resumed once the value of a future is finished. In general, finishing the evaluation of the value of a future and resuming suspended tasks resulting from a single touch can also lead to up to $d/2$ deviations (see Section 4.2).

In the remainder of the text, we shall refer to any deviation incurred by a processor after suspending on a touch as a suspend deviation (whether it is directly or indirectly caused by that touch), and similarly for resume deviations. Just as at most one deviation occurs at a join for each steal in nested parallelism [1], deviations that occur as a result of futures can also be loosely paired as shown by the following lemma.

Lemma 1. *There is at least one steal or suspend deviation for every two resume deviations.*

Proof. Assume that there is some node v that corresponds to a touch. We will show that if a resume deviation occurs when executing v, there is at least one steal or suspend deviation and that the number of steal and suspend deviations related to v is no less than half the number of resume deviations related to v. Assume that processor p deviates when executing (and therefore, resuming) v. This implies that some processor q suspended rather than executing v and incurred either a steal or suspend deviation at that time. Processor p may also deviate after executing v and one or more of its descendants, so we have one or two resume deviations and one steal or suspend deviation. Now consider each descendant u of v at which p incurs an indirect resume deviation. This implies that some other processor (possibly q) already executed u's other parent but not u itself, at which time it incurred a suspend deviation. □

3.2 Performance Bounds

For nested-parallel computations, the number of steals can be used to bound the overhead of the scheduler, including scheduler time and cache misses. Using deviations, we can extend these bounds to more general parallel computations. In the remainder of this section, we consider a scheduler called *generalized* WS that is a version of the scheduler shown in Figure 2 extended for unrestricted parallelism (and also defined in Arora et al. [3]). These results, together with results in the next section, show that for programs that use parallel futures, WS will incur overhead proportional to both the number of touches and the span of the graph. Below, Δ indicates the number of deviations from the sequential execution.

Scheduler Time. Cilk-style work stealing derives much of its performance from the application of the "work-first" principle. This states that overhead should be moved away from the "work" and onto the critical path [15]. For example, Cilk compiles two versions of each function: one to be run in the common case (the fast clone) and one to be run only after a steal (the slow clone). Because fast clones are expected to be run much more frequently, most of the synchronization overhead is moved into the slow clones. When running a fast clone, no checks are required to determine which sub-tasks have already finished: in a fast clone, sub-tasks are always executed by the same processor as the parent task. In contrast, a slow clone must stop at each synchronization point to check for a concurrently executing task and see if it has finished. In the implementation shown in Figure 2, if the continuation task pushed in the line marked † is *not* executed by the same processor that pushed it, then the last task before the corresponding join must be run using the

slow clone; if it is executed by the same processor, then the fast clone can be used. It can be shown for nested-parallel computations that the number of invocations of slow clones is bounded by the number of steals. By bounding the number of invocations of slow clones, we also bound a significant portion of the time used by the scheduler. This optimization for nested parallelism is less effective in programs that also use other forms of parallelism. In this case, the number of executions of slow clones is bounded only by the number of deviations.

Theorem 1. *The number of invocations of slow clones using generalized* WS *is no more than the number of deviations.*

Proof. Assume a slow clone is executed for a node v by processor p. This is only the case when (i) v has a successor w, and w has an in-degree greater than one, (ii) $v \prec_1 w$, and (iii) the other predecessor of w (call it u) has not (yet) been executed by p. (If v is the sole predecessor of w then no synchronization is necessary. Likewise, if $v \not\prec_1 w$ then v need not be executed using a slow clone since any necessary synchronization will occur as part of popping an element from the deque. Finally, if $u <_p v$ then p need not synchronize before executing w since it already knows that u has been executed.) From assumption (ii) it follows that $u <_1 v$ since otherwise w would not be ready after the execution of v. Let x be the last node executed by p no later than v at which a deviation occurred. Such a node x must exist either because another processor executed u or because p executes u after v. For example, the first node x executed by p with $u <_1 x$ would incur a deviation. If no additional deviations occur between x and v (which is the case since we take x to be the last such node) then no node between x and v is executed with a slow clone other than v. This follows from the fact that every task executed by p in this interval is pushed by p onto its own deque. Thus the number of invocations of slow clone is bounded by the number of deviations. □

The number of deviations also provides a lower bound for the number of slow clone invocations for parallel programs that use futures. A slow clone will be executed after every steal and every suspend deviation (direct or indirect). In the latter case, this follows since each such deviation corresponds to the execution of a task pushed at the line marked † of Figure 2 where one or more predecessors of the task pushed on the following line have not yet been executed. By Lemma 1 this is a constant fraction of all deviations.

Cache Misses. If we consider a shared memory architecture with a hardware-controlled cache, then we can bound the additional cache misses incurred during a parallel execution. We assume that each processor has a private cache with capacity C. Acar et al. [1] bound the number of cache misses in nested-parallel computations in terms of the number of steals. To extend this to general parallelism, we must understand how data communicated between processors through uses of futures can affect cache behavior. In these results, we assume a fully associative cache with an LRU replacement policy.

Theorem 2 (extension of Theorem 3 in Acar et al. [1])**.** *The number of cache misses incurred using generalized* WS *is less than $M_1(C) + C\Delta$ where C is the size of the cache, $M_1(C)$ is the number of cache misses in the sequential execution, and Δ is the number of deviations.*

The original theorem is proved using the fact that the computation is race-free, *i.e.*, for two unrelated nodes in the computation graph, there is no memory location that is written by one node and either read or written by the other. We assume deterministic computations are race-free. For nested-parallel computations, race-free implies that any locations read by the thief must have been written before the steal or by the thief. Therefore, after C misses, the thief's cache will be in the same state as that of the sole processor in the sequential schedule. From that point forward, such a processor will only incur misses that also occur during the sequential schedule. In this proof, we consider an implementation that flushes the cache at each deviation. An alternative implementation that did not flush the cache after each deviation will incur no more misses.

Proof. Given a graph g and a schedule $\mathcal{S}$ determined by generalized WS, assume that the implementation flushes the cache after each deviation. Consider a node v that reads a memory location m and a second node u that writes to that memory location. Assuming the program is deterministic, then there is exactly one such u whose value should be read by v and there is at least one path from u to v in g.

If no deviation occurs between u and v then a cache miss occurs at v in $\mathcal{S}$ if and only if a cache miss occurs at v in the sequential schedule: either m must have remained in the cache since u was visited, or C other locations were accessed in the interval between u and v.

If a deviation does occur between u and v, then let w be the most recent deviation incurred by the processor q that executes v. Assume that no miss occurs at v in the sequential schedule, and further assume, for a contradiction, that q has already incurred C cache misses since w. This implies that there have been C distinct memory accesses since w. However, q visits all nodes between w and v in the same order as the sequential schedule, so the state of q's cache at v must be identical to that of the sequential schedule. Therefore, if no miss occurs in the sequential schedule at v then no miss occurs when v is visited by q, a contradiction. Therefore, q incurs at most C more misses than the sequential schedule. □

Acar et al. [1] showed a family of graphs (which can be constructed using futures) where $\Omega(Ct)$ cache misses occur with two processors (where t is the number of touches), while only $O(C)$ cache misses occur when using a single processor. Because each of these touches incurs a deviation under a parsimonious work-stealing scheduler, the number of misses is $M_1(C) + \Omega(C\Delta)$.

4. TIGHT BOUNDS ON DEVIATIONS

In this section, we give the main results of this paper: upper and lower bounds on the number of deviations that occur when using WS to schedule parallel programs that use futures. Given the results of the previous section, these bounds imply that using work stealing to schedule tasks in programs that use futures may incur high overheads.

4.1 Upper Bound

This bound is given in terms of the number of touches that appear in the computation graph. Note that this is not the number of touches that appear in the program source

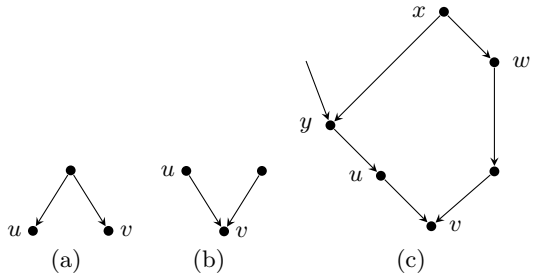


Figure 4: Graphs Used in Proof of Upper Bound.

code, but the number of touches that occur dynamically in the program execution.

Theorem 3 (Upper Bound)**.** *For computations with span d and t touches, the expected number of deviations by* WS *on P processors is $O(Pd + td)$.*

A proof of this theorem appears below, but we first give the following lemmata that are used in the proof. These rely critically on the structure graphs arising from uses of futures and the behavior of parsimonious work-stealing schedulers.

Lemma 2. *Given a node v in a computation graph with out-degree two and right child u, if y is a descendant of v and $y <_1 u$, but y has a parent that is not a descendant of v, then y is a touch.*

Proof. If y is a descendant of v but it also has a parent that is not a descendant of v, then y must have in-degree two. Therefore it is either a join or a touch. Suppose the other parent of y is x. As we have $y <_1 u$ and $x <_1 y$, it follows that $x <_1 u$. Furthermore, x is not a descendant of v and so $x <_1 v$. If x was executed before v in the sequential execution then x and v share a common predecessor w (possibly x). Since the two parallel paths between w and y are not well-nested with respect to v's two children, y must be a touch. □

Lemma 3. *Given a node of out-degree two with children u and v, let V be the set of nodes executed in the sequential schedule between u and v (inclusive). If for some processor p in a parallel execution and for each touch $y \in V$, $x \prec_1 y \Rightarrow x \prec_p y$, then for all $y, z \in V$, $y <_1 z \Rightarrow y <_p z$.*

Proof. Assume otherwise, for the sake of contradiction. Take the first pair of nodes $y, z \in V$ (ordered lexicographically according to the sequential order of execution) such that $y <_1 z$, but $y \not<_p z$. There are three possibilities: either y or z is not executed by p (and thus p does not relate y and z) or $z <_p y$. If y is not executed by p (and similarly for z) then it must have a parent that is not a descendant of u. By Lemma 2, y is a touch, a contradiction, since we assumed that p executed all touches in V. Alternatively, we have $z <_p y$. If this is the case, then up to the point that p executes z, p has executed every node x such that $x <_1 y$ (since otherwise y and z would not be the first offending pair). Let x be the last such node: $x \prec_1 y$. If p does not execute y immediately after x, then y must have another parent that is not in V. This again (by Lemma 2) implies that y is a touch, a contradiction, since we assumed that $x \prec_1 y$ implies $x \prec_p y$ for all touches in y in V. □

Lemma 4. *When executing a graph of span d, the number of elements in any processor's deque is at most d.*

Proof. The number of elements in a deque only increases when executing a node with out-degree two. Each processor only steals when its deque is empty, thus if there are d elements in a processor's queue, there must exist a path of length d composed of nodes of out-degree two. If such a path exists, then the span of the graph is at least d. □

Lemma 5. *For a graph g with root u and sink v, for any other node w in g, if there is a path from w to v that does not include any touch edges, then there is a path from u to w that does not include any future edges.*

Proof. By induction on the length of the path from w to v. First take the case where there is an edge connecting w and v (a path of length one) that is not a touch edge. Edges in the computation graph are either (i) future or touch edges, (ii) are added to between two sub-graphs that must be computed serially, or (iii) are added to compose two sub-graphs as a nested-parallel computation. The edge from w to v cannot be a future edge (i) as a future edge cannot lead to the sink of a graph. If that edge was added as part of the serial composition of two sub-graphs (ii), then every path from u to v that does not include a future or touch edge must include w (and there is always at least one path from u to v that does not include either future or touch edges). If an edge was added as part of a nested-parallel composition (iii) (*i.e.* leading to a join), then there must be a corresponding fork with an edge that leads to w as well as a path without future edges that leads from u to that fork.

Now take the case where the length of the path from w to v is greater than one. Assume that there is a node y with an edge (w, y), a path from y to v, and that none of those edges is a touch edge. By induction, there is a path from u to y that does not include any future edges. As above, consider how the edge (w, y) was added to the graph. This edge cannot be a future edge (i) since if it was then every path from y to v must include a touch edge and we assumed at least one path that does not. If that edge was added during a serial composition (ii) then again every path from u to v that does not include a future or touch edge must include w. (iii) If the edge (w, y) is a join edge, then there must be a corresponding fork and that fork must lie on the path between u and y; it follows that there is path without future edges from that fork to w. Finally, if the edge (w, y) is a fork edge, then every path without future edges from u to y also goes through w. □

Proof of Theorem 3. We consider each node in the graph and determine whether or not a deviation could occur at that node. We divide nodes into three classes based on the in-degree of each node and (possibly) the out-degree of its parent. Only the root node has in-degree zero, but a deviation never occurs at the root. If a node v has in-degree one and either (i) its parent has out-degree one, or (ii) its parent has out-degree two and v is the left child of its parent, then v will always be executed immediately after its parent and no deviation will occur.

Consider the case where v has in-degree one, its parent has out-degree two, and v is the right child of its parent as shown in Figure 4(a). In this case, there is some descendant x of u such that $x \prec_1 v$. Assume that v is executed by processor p. A deviation occurs when executing v only if at least one of the following three conditions holds: (i) v is stolen, (ii) there is a touch at some node w that is executed between u and

v in the sequential schedule along with a node y such that $y \prec_1 w$ and p executes y but $y \not\prec_p w$, or (iii) there is a node x such that $x \prec_1 v$ and x is executed by p, but p also executes at least one other node between x and v.

First, we show that these conditions are exhaustive. Assume that a deviation occurs when executing v but suppose, for the sake of contradiction, that none of these conditions is true. If v was not stolen then u was also executed by p and v was pushed onto p's deque. By assumption, for every touch w that appears in the sequential execution between u and v with $y \prec_1 w$, we have $y \prec_p w$. It follows from Lemma 3 that for every pair of nodes y and z that appears in the sequential execution between u and v (inclusive), we have $y <_1 z \Rightarrow y <_p z$. This implies $x <_p v$ for the node x such that $x \prec_1 v$. Finally, by assumption, there is no other node that p executes between x and v and so $x \prec_p v$. Therefore, no deviation occurs at v, a contradiction.

Now we count the number of nodes that can be described by one of the three conditions above. First, from the proof of Theorem 9 in Arora et al. [3], the expected number of steals is $O(Pd)$, bounding the number of nodes that fall into the first category. Second, if there is a touch w such that $y \not\prec_p w$ and v was not stolen, then v was on p's deque when p executed y. Moreover, since $y \prec_1 w$, there is exactly one such y for each touch. By Lemma 4, there are at most d elements on p's queue at any time, and thus $O(td)$ nodes in the second category. Finally, there is exactly one node v for each node x such that $x \prec_1 v$. The number of nodes that finish the value of a future is bounded by the number of touches. If p executes another node between x and v, then x is a future and the number of nodes such as x is $O(t)$. It follows that the number of nodes that immediately follow x is also $O(t)$. Thus the expected number of deviations occurring at nodes of in-degree one is $O(Pd + td)$.

Now take the case where v has in-degree two. Deviations only occur here if v is enabled by its left parent (*i.e.*, if its left parent is visited after its right parent), so consider that case (Figure 4(b)). Define $\hat{g}_x$ to be the sub-graph of g rooted at some node x formed by removing any future or touch edges as well as any nodes that are not reachable from x through the remaining edges. In this case, a deviation occurs when executing v only if one of the two following conditions holds: (i) there is a touch of a node y and a node x (possibly the same as y) that is a predecessor of both v and y such that v lies on all paths from y to the sink of $\hat{g}_x$, or (ii) there exists a node w such that $u \prec_1 w$ and w was stolen.

Assume that a deviation occurs when executing v but suppose again, for the sake of contradiction, that neither of the above conditions is true. First, v cannot be a touch. (If it were, then it would lie on all paths between itself and the sink of $\hat{g}_v$.) Therefore, v must be a join. Let x be the corresponding fork and w be the right child of x. It follows that $u \prec_1 w$. By assumption, w was not stolen and was therefore executed by the same processor p that executed x. Since no deviation occurred, w is also executed before u. However, if p executes w before u, then there must be some node y that is a predecessor of u and a descendant of x but is not dominated by x. The graph might look something like Figure 4(c). If y is not dominated by x then it must be a touch. If there is more than one such touch, let y denote the one that is closest to u. Then any edges between y and u are not touch edges. By Lemma 5 there is a path from x to y that does not include any future edges. Thus y is in $\hat{g}_x$. Since $\hat{g}_x$ is series-parallel and y appears after x but before v, any path in $\hat{g}_x$ from y to the sink must go though v, a contradiction. Thus no deviation can occur unless one of the above conditions holds.

We now count the number of nodes that can fall into one of the two categories above: in the first case, the number of nodes such as v is bounded by the number of touches times the span of $\hat{g}_x$ (which is in turn bounded by the span of g); in the second case, there is exactly one node v for each steal of w such that v follows some node u with $u \prec_1 w$. Thus the number of deviations occurring at nodes of in-degree two is $O(Pd + td)$, and the total number of expected deviations, including nodes with both in-degree one and two, is also $O(Pd + td)$. □

4.2 Lower Bound

While this upper bound is high, we show that there are programs that use futures for which this upper bound is tight. We assume that the scheduler is parsimonious (*i.e.*, it maintains one task queue per processor and each processor only steals when its local queue is empty) and also greedy. This class of schedulers includes WS. In the computations we construct, each time we increase the span by one, we can either double the number of touches (and the number of deviations incurred as a result of those touches) or we can cause each existing touch to incur an additional deviation.

Theorem 4 (Lower Bound)**.** *There exists a family of parallel computations derived from programs that use parallel futures with span d and t touches where $\Omega(Pd + td)$ deviations occur when executed on P processors with any parsimonious work-stealing scheduler and when $d > 8 \log t$ and $d > 2 \log P$.*

Proof. The proof is given by constructing such a family of computations. Graphs that incur $\Omega(Pd)$ deviations can be constructed using only nested parallelism, for example, by chaining together sequences of fork-join pairs. In such examples, d must be greater than $2 \log P$ to ensure there is adequate work for P processors. Below, we construct graphs with $\Omega(td)$ deviations for two processors and assume that they can be combined (in parallel) with such chains yielding a family of graphs with the required number of deviations.

For ease of presentation, we will carry out this construction in stages. In the first step, we show that one steal and one touch is sufficient to generate $\Omega(d)$ deviations on a graph of span d with two processors. Figure 5(a) shows the one such graph. The program starts by using a future to split the computation into two parts. The value of the future is finished at node w and the edge from w to v is a touch edge.

Assume that processor p starts execution of the graph and the processor q steals work corresponding to the right-hand sub-graph. We assume that steals require unit time, but our examples can be extended to account for larger costs. (A heavy line in the figure divides those nodes executed by p from those executed by q.) This steal means that q immediately incurs a deviation. As above, nodes at which deviations occur are circled. The left-hand sub-graph is a sequence of nodes that ensures that q finishes node u before p finishes w.

After q finishes u, it cannot continue sequentially to v. However, there are several nodes in q's deque. Since we assume the work-stealing scheduler is parsimonious, q will continue by removing elements from its queue, incurring

a deviation each time. These deviations occur as indirect results of a suspension.

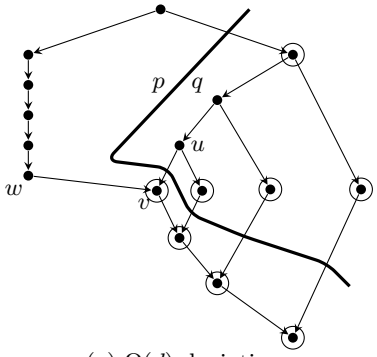
Meanwhile, p will finish the value of the future and continue by executing v (again, since this node was on its own deque and it has no other nodes to execute). Then p will execute all of the nodes along the lower-left side of the right-hand sub-graph, incurring a deviation at each node. These are examples of indirect deviations that occur after resuming a suspended touch.

To generalize this graph to an arbitrary span, we add an additional fork on the right-hand sub-graph and one more node on the left, ensuring there is sufficient work to delay p. This will increase both the span and the number of deviations by two but will not change the number of steals or touches. (In fact, similar graphs can be constructed that will incur $\Omega(d)$ deviations for any greedy scheduler.)

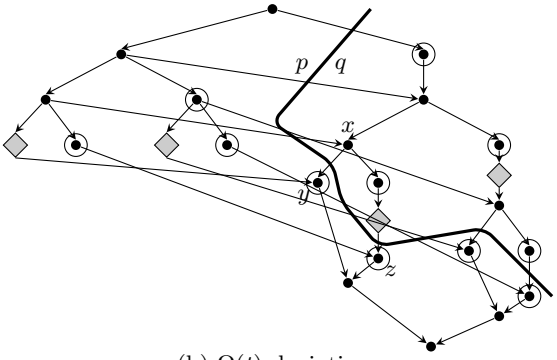
In the second step, we show that one steal and t touches is sufficient to generate $\Omega(t)$ deviations on a graph of span $\Omega(\log t)$ as in Figure 5(b). Graphs of this form are generated from a pair of functions which produce and consume a binary tree in parallel. Each node in this tree, both internal and external, is defined by a future.

These graphs are constructed so that p and q alternate between suspending on the touch of a future and resuming it. The gray sub-graphs (drawn as diamonds in the figure) are small sequences of nodes introduced to ensure this alternation. In the figure, each such sub-graph requires two or three nodes. Processor q incurs a deviation after executing node x. There are several nodes in q's deque, but the graph is constructed so the next touch will also result in suspend, no matter which node is chosen. This graph may be generalized to an arbitrary number of touches so long as the span of the graph is at least $4 \log t$.

Finally, we define a family of computations with span d that incur $\Omega(td)$ deviations on two processors. We do so by first taking a graph as described in the second step (Figure 5(b)) of span $d/2$. We then replace each touch in that graph (*e.g.*, nodes y and z in the figure) with an instance of the right-hand portion of the graph described in the first step (Figure 5(a)) of span $d/2$. In addition, each diamond sub-graph on the producer-side of the graph must be extended with an additional delay of $d/2$ nodes. This results in a graph of span d with t touches, each of which will result in $\Omega(td)$ deviations. Note that this construction requires that $d > 8 \log t$ so that there is adequate parallelism to produce and consume the tree in parallel. □



(a) $\Omega(d)$ deviations



(b) $\Omega(t)$ deviations

Figure 5: Graphs Demonstrating Lower Bounds. In part (a), one future, one steal, and one touch lead to $\Omega(d)$ deviations on a graph of span d. In part (b), one steal and t touches leads to $\Omega(t)$ deviations.

5. IMPLEMENTATION & APPLICATIONS

In this section, we give empirical results showing that deviations occur more frequently than steals not just in constructed examples, but also in a program proposed in previous literature. We also examine a class of programs that use futures in a limited but more efficient way. We first give some background on our parallel implementation of ML.

Parallel ML. We performed our experiments with Halstead's quicksort (described below) using our parallel implementation of ML. This is a parallel extension of Standard ML [23] based on the MLton compiler and runtime system [26]. In addition to adding multi-processor support to this runtime, we also implemented several different schedulers, including work-stealing and parallel depth-first schedulers [7]. One of our goals was to factor the implementation so that the scheduler can be easily replaced with an alternative while at the same time allowing for (and implementing) optimizations that have been suggested in previous work (*e.g.*, Cilk). For example, we have implemented the fast clone/slow clone compilation strategy used by Cilk at the source level in ML using higher-order functions. This implementation relies heavily on MLton's aggressive inlining and optimizations to achieve good performance.

We have also implemented several parallel primitives, including primitives for nested parallelism, futures, and write-once synchronization variables [6]. Though all of these constructs could be implemented using expressive primitives like synchronization variables, some optimizations can be performed only if the more restrictive constructs are also built as primitives. While implementing futures, we faced certain issues as to how and when these optimizations (*e.g.*, fast/slow clones) could be applied and these issues have led us to the current work. These and other implementation details are described in our previous work [25].

Halstead's Quicksort. Halstead [19] uses an implementation of quicksort as an application of parallel futures. (The full source code for this example is given in Figure 1 of that work.) This implementation uses futures in two ways. First, it streams the partition of the input so that while one processor continues to partition the remainder of the input, other processors can select new pivots and recursively partition the two halves of the input. Second, it uses a future to compute both recursive calls in parallel. While in many cases this

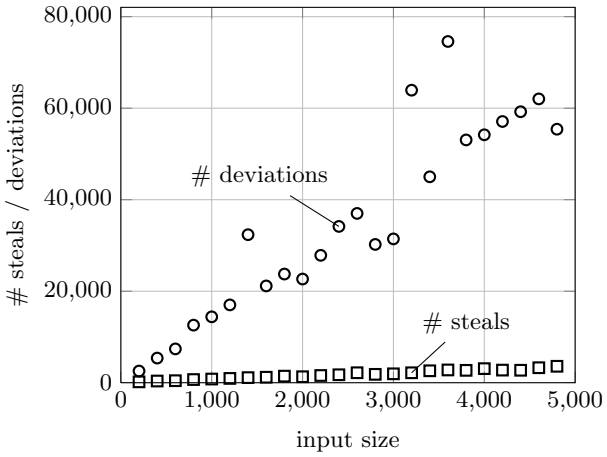


Figure 6: Empirical Measurements for Quicksort.

can be accomplished using nested parallelism, this implementation uses a well-known technique to avoid concatenating the results of the two recursive calls. Instead it passes the result of one recursive call as an argument to the other. To enable these to execute in parallel, this argument is passed as a future. It appears that it is not possible to implement this with nested parallelism.

This implementation of quicksort has span $O(n)$ and work $O(n \log n)$. It also creates $O(n \log n)$ futures and touches each future at least once. Figure 6 shows the number of steals (squares) and deviations (circles) measured empirically in our implementation of the WS scheduler. (The variation in some points is due to effects of the operating system scheduler.) Note that only a small fraction of the deviations can be attributed to steals.

Pipelining. Recall that Figure 1(b) shows an example of a computation graph derived from a program that uses futures to implement a parallel pipeline. Though the computation shown in the figure does not use any nested parallelism, this example can be extended by replacing each node in such a graph with a nested-parallel computation (*e.g.*, if each stage in the pipeline is a matrix multiplication). We call such a computation a *pure linear pipeline* if it meets the following restrictions: (i) there is at most one read for each future, (ii) touches are nested within a constant number of forks and futures, and (iii) the value of each future has at most one additional future embedded within it. The last condition requires more explanation. It allows programmers to use linear streams of futures (*e.g.*, lists whose tails are futures) but not trees of futures. This restriction can be easily enforced in a typed language by restricting those types α that appear as α future. These three conditions imply that at most one thread can suspend waiting for the value of each future and that at most one thread must be resumed. As touches only appear within a constant parallel nesting, there is at most a constant number of tasks on a processor's deque at the time of a suspension. Finally, there is at most one unfinished future in the environment of each thread. This implies that after a suspension, only a constant number of deviations can occur before another steal. As the expected number of steals is $O(Pd)$, this is also an upper bound on the expected number of deviations for pure linear pipelines.

6. RELATED WORK

The principle of work stealing goes back at least to work on parallel implementations of functional programming languages in the early 1980's [12, 18]. It is a central part the implementation of the Cilk programming language [15]. Cilk-style work stealing has also been implemented in Java [22], X10 [2, 13], and C++ [20].

Work stealing for nested-parallel programs (such as those in Cilk) was studied by Blumofe and Leiserson [9] who bounded the execution time in terms of the number of steals as well as the additional space required. Agarwal et al. [2] extend Blumofe and Leiserson's work on space bounds to *terminally strict* parallelism, a class of parallel programs broader than nested-parallel (fully strict) programs but still not sufficiently expressive to build parallel pipelines. Arora et al. [3] give a work-stealing scheduler for unrestricted parallelism and bound the number of steals and execution time but not space. They also consider performance in a multi-programmed environment. Acar et al. [1] bound the cache overhead of this scheduler using *drifted nodes* (which we have re-dubbed deviations) but only for nested-parallel programs.

Parallel depth-first scheduling [7] is another form of dynamic scheduling with proven bounds on overhead. It has been extended to support synchronization variables [6], which in turn can be used to implement futures, but it is unclear if the overhead of this scheduler can be reduced to support parallelism at the same granularity as work stealing.

Static scheduling determines the mapping between parallel tasks and processors at compile time and has been used in the implementation of stream processing languages (*e.g.*, [17]) where the hardware platforms are determined in advance. Stream processing includes limited forms of parallel pipelining, but static scheduling is not applicable when the structure of the pipeline depends on the program input.

Parallel futures of the form we use were first proposed by Halstead [18] as part of Multilisp and also appeared in work by Kranz et al. [21]. Light-weight implementations of futures were described by Mohr et al. [24] and Goldstein et al. [16]. Parallel pipelining can also be implemented using a number of other language features such as I-structures in ID [4] and streams in SISAL [14]. Blelloch and Reid-Miller [5] give several examples of applications where the use of futures can asymptotically reduce the span of the computation graphs. They also give an upper bound on the time required to execute these computations by giving an implementation of a greedy scheduler that supports futures. Blelloch and Reid-Miller limit each future to a single touch to enable a more efficient implementation.

7. CONCLUSION AND FUTURE WORK

Using deviations, we have shown that the overheads of work-stealing schedulers for programs using parallel futures can (only) be bounded in terms of the number of touches and the span of the computation graph: steals do not suffice once we go beyond nested parallelism.

While previously described work-stealing schedulers have focused on limiting the number of steals and moving overhead into the critical path, it would be interesting to explore alternatives that attempted to limit the number of deviations, perhaps at the cost of additional steals. Moreover, deviations may also be of use in bounding the overhead of schedulers that do not rely on work stealing. We also leave as future

work a formal characterization of the family of graphs arising from uses of futures as well as bounds on the number of deviations for other forms of deterministic parallelism.

Acknowledgments. We would like to thank the MLton developers for their support and the anonymous reviewers for their helpful comments. This work was funded in part by an IBM OCR gift, the Center for Computational Thinking sponsored by Microsoft, and a gift from Intel.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 1–12. ACM, 2000.

[2] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 229–240. ACM, 2007.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 119–129. ACM, 1998.

[4] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, 1989.

[5] Guy E. Blelloch and Margaret Reid-Miller. Pipelining with futures. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 249–259. ACM, 1997.

[6] Guy E. Blelloch, Phillip B. Gibbons, Girija J. Narlikar, and Yossi Matias. Space-efficient scheduling of parallelism with synchronization variables. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 12–23. ACM, 1997.

[7] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, 1999.

[8] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proc. of the Symp. on Theory of Comput.*, pages 362–371, 1993.

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of the Symp. on Principles and Practice of Parallel Program.*, pages 207–216. ACM, 1995.

[11] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. of the Symp. on Parallel Algorithms and Architectures*, pages 297–308. ACM, 1996.

[12] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proc. of the Conference on Funct. Program. Lang. and Comp. Architecture*, pages 187–194. ACM, 1981.

[13] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Proc. of the Intl. Conf. on Parallel Processing*, pages 536–545. IEEE Computer Society, 2008.

[14] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *J. Parallel Distrib. Comput.*, 10(4):349–366, 1990.

[15] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the Conf. on Program. Lang. Design and Implementation*, pages 212–223. ACM, 1998.

[16] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: implementing a fast parallel call. *J. Parallel Distrib. Comput.*, 37(1):5–20, 1996.

[17] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. of the Int. Conf. on Architectural Support for Program. Lang. and Operating Sys.*, pages 151–162. ACM, 2006.

[18] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proc. of the Symp. on LISP and Funct. Program.*, pages 9–17. ACM, 1984.

[19] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[20] Intel Corporation. Intel Threading Building Blocks. `www.threadingbuildingblocks.org`, 2009.

[21] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: a high-performance parallel Lisp. In *Proc. of the Conf. on Program. Language Design and Implementation*, pages 81–90. ACM, 1989.

[22] Doug Lea. A Java fork/join framework. In *Proc. of the Conference on Java Grande*, pages 36–43. ACM, 2000.

[23] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[24] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proc. of the Conf. on LISP and Funct. Program.*, pages 185–197. ACM, 1990.

[25] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space profiling for parallel functional programs. In *Proc. of the Int. Conf. on Funct. Program.*, pages 253–264. ACM, 2008.

[26] Stephen Weeks. Whole-program compilation in MLton. In *Proc. of the Workshop on ML*, page 1. ACM, 2006.