Experimental Analysis of Space-Bounded Schedulers

Harsha Vardhan Simhadri
Carnegie Mellon University,
Lawrence Berkeley Lab
harshas@lbl.gov

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Jeremy T. Fineman
Georgetown University
jfineman@cs.georgetown.edu

Phillip B. Gibbons
Intel Labs Pittsburgh
phillip.b.gibbons@intel.com

Aapo Kyrola
Carnegie Mellon University
akyrola@cs.cmu.edu

ABSTRACT

The running time of nested parallel programs on shared memory machines depends in significant part on how well the scheduler mapping the program to the machine is optimized for the organization of caches and processors on the machine. Recent work proposed "space-bounded schedulers" for scheduling such programs on the multi-level cache hierarchies of current machines. The main benefit of this class of schedulers is that they provably preserve locality of the program at every level in the hierarchy, resulting (in theory) in fewer cache misses and better use of bandwidth than the popular work-stealing scheduler. On the other hand, compared to work-stealing, space-bounded schedulers are inferior at load balancing and may have greater scheduling overheads, raising the question as to the relative effectiveness of the two schedulers in practice.

In this paper, we provide the first experimental study aimed at addressing this question. To facilitate this study, we built a flexible experimental framework with separate interfaces for programs and schedulers. This enables a head-to-head comparison of the relative strengths of schedulers in terms of running times and cache miss counts across a range of benchmarks. (The framework is validated by comparisons with the Intel® Cilk™ Plus work-stealing scheduler.) We present experimental results on a 32-core Xeon® 7560 comparing work-stealing, hierarchy-minded work-stealing, and two variants of space-bounded schedulers on both divide-and-conquer micro-benchmarks and some popular algorithmic kernels. Our results indicate that space-bounded schedulers reduce the number of L3 cache misses compared to work-stealing schedulers by 25–65% for most of the benchmarks, but incur up to 7% additional scheduler and load-imbalance overhead. Only for memory-intensive benchmarks can the reduction in cache misses overcome the added overhead, resulting in up to a 25% improvement in running time for synthetic benchmarks and about 20% improvement for algorithmic kernels. We also quantify runtime improvements varying the available bandwidth per core (the "bandwidth gap"), and show up to 50% improvements in the running times of kernels as this gap increases 4-fold. As part of our study, we generalize prior definitions of space-bounded schedulers to allow for more practical variants (while still preserving their guarantees), and explore implementation tradeoffs.

SPAA'14, June 23–25, 2014, Prague, Czech Republic.

ACM 978-1-4503-2821-0/14/06 ...$15.00.
http://dx.doi.org/10.1145/2612669.2612678.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Runtime environments

Keywords

Thread schedulers, space-bounded schedulers, work stealing, cache misses, multicores, memory bandwidth

1. INTRODUCTION

Writing nested parallel programs using fork-join primitives on top of a unified memory space is an elegant and productive way to program parallel machines. Nested parallel programs are portable, sufficiently expressive for many algorithmic problems [28, 5], relatively easy to analyze [22, 3], and supported by many programming languages including OpenMP [25], Cilk++ [17], Intel® TBB [18], Java Fork-Join [21], and Microsoft TPL [23]. The unified memory address space hides from programmers the complexity of managing a diverse set of physical memory components like RAM and caches. Processor cores can access memory locations without explicitly specifying their physical location. Beneath this interface, however, the real cost of accessing a memory address from a core can vary widely, depending on where in the machine's cache/memory hierarchy the data resides at time of access. Runtime thread schedulers can play a large role in determining this cost, by optimizing the timing and placement of program tasks for effective use of the machine's caches.

Machine Models and Schedulers. Robust schedulers for mapping nested parallel programs to machines with certain kinds of simple cache organizations such as single-level shared and private caches have been proposed. They work well both in theory [10, 11, 8] and in practice [22, 24]. Among these, the *work-stealing scheduler* is particularly appealing for private caches because of its simplicity and low overheads, and is widely deployed in various run-time systems such as Cilk++. The *PDF scheduler* [8] is suited for shared caches and practical versions of this schedule have been studied. The cost of these schedulers in terms of cache

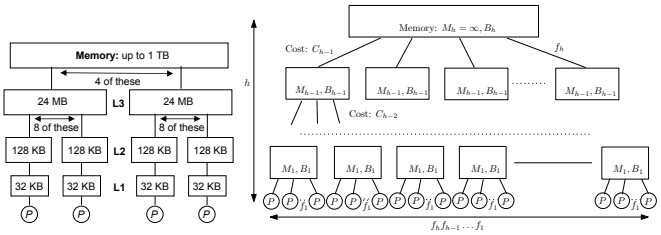
misses or running times can be bounded by the locality cost of the programs as measured in certain abstract program-centric cost models [10, 1, 6, 7, 29].

However, modern parallel machines have multiple levels of cache, with each cache shared amongst a subset of cores (*e.g.*, see Fig. 1(a)). A parallel memory hierarchy (PMH) as represented by a tree of caches [2] (Fig. 1(b)) is a reasonably accurate and tractable model for such machines [16, 15, 14, 6]. Because previously studied schedulers for simple machine models may not be optimal for these complex machines, recent work has proposed a variety of hierarchy-aware schedulers [16, 15, 14, 27, 6] for use on such machines. For example, hierarchy-aware work-stealing schedulers such as PWS and HWS schedulers [27] have been proposed, but no theoretical bounds are known.

To address his gap, *space-bounded schedulers* [15, 14] have been proposed and analyzed. To use space-bounded schedulers, the computation needs to annotate each function call with the size of its memory footprint. The scheduler then tries to match the memory footprint of a subcomputation to a cache of appropriate size in the hierarchy and then run the subcomputation fully on the cores associated with that cache. Note that although space annotations are required, the computation can be oblivious to the size of the caches and hence is portable across machines. Under certain conditions these schedulers can guarantee good bounds on cache misses at every level of the hierarchy and running time in terms of some intuitive program-centric metrics. Chowdhury et al. [15] (updated as a journal article in [14]) presented such schedulers with strong asymptotic bounds on cache misses and runtime for highly balanced computations. Our follow-on work [6] presented slightly generalized schedulers that obtain similarly strong bounds for unbalanced computations.

Our Results: The First Experimental Study of Space-Bounded Schedulers. While space-bounded schedulers have good theoretical guarantees on the PMH model, there has been no experimental study to suggest that these (asymptotic) guarantees translate into good performance on real machines with multi-level caches. Existing analyses of these schedulers ignore the overhead costs of the scheduler itself and account only for the program run time. Intuitively, given the low overheads and highly-adaptive load balancing of work-stealing in practice, space-bounded schedulers would seem to be inferior on both accounts, but superior in terms of cache misses. This raises the question as to the relative effectiveness of the two types of schedulers in practice.

This paper presents the first experimental study aimed at addressing this question through a head-to-head comparison of work-stealing and space-bounded schedulers. To facilitate a fair comparison of the schedulers on various benchmarks, it is necessary to have a framework that provides separate modular interfaces for writing portable nested parallel programs and specifying schedulers. The framework should be light-weight, flexible, provide fine-grained timers, and enable access to various hardware counters for cache misses, clock cycles, etc. Prior scheduler frameworks, such as the Sequoia framework [16] which implements a scheduler that closely resembles a space-bounded scheduler, fall short of these goals by (i) forcing a program to specify the specific sizes of the levels of the hierarchy it is intended for, making it non-portable, and (ii) lacking the flexibility to readily support work-stealing or its variants.



(a) 32-core Xeon® 7560 (b) PMH model of [2]

Figure 1: Memory hierarchy of a current generation architecture from Intel®, plus an example abstract parallel hierarchy model. Each cache (rectangle) is shared by all cores (circles) in its subtree.

This paper describes a scheduler framework that we designed and implemented, which achieves these goals. To specify a (nested-parallel) program in the framework, the programmer uses a Fork-Join primitive (and a Parallel-For built on top of Fork-Join). To specify the scheduler, one needs to implement just three primitives describing the management of tasks at Fork and Join points: `add`, `get`, and `done`. Any scheduler can be described in this framework as long as the schedule does not require the preemption of sequential segments of the program. A simple work-stealing scheduler, for example, can be described with only 10s of lines of code in this framework. Furthermore, in this framework, program tasks are completely managed by the schedulers, allowing them full control of the execution.

The framework enables a head-to-head comparison of the relative strengths of schedulers in terms of running times and cache miss counts across a range of benchmarks. (The framework is validated by comparisons with the commercial Cilk$^{\text{TM}}$ Plus work-stealing scheduler.) We present experimental results on a 32-core Intel® Nehalem series Xeon® 7560 multicore with 3 levels of cache. As depicted in Fig. 1(a), each L3 cache is shared (among the 8 cores on a socket) while the L1 and L2 caches are exclusive to cores. We compare four schedulers—work-stealing, priority work-stealing (PWS) [27], and two variants of space-bounded schedulers—on both divide-and-conquer micro-benchmarks (scan-based and gather-based) and popular algorithmic kernels such as quicksort, sample sort, matrix multiplication, and quad trees.

Our results indicate that space-bounded schedulers reduce the number of L3 cache misses compared to work-stealing schedulers by 25–65% for most of the benchmarks, while incurring up to 7% additional overhead. For memory-intensive benchmarks, the reduction in cache misses overcomes the added overhead, resulting in up to a 25% improvement in running time for synthetic benchmarks and about 20% improvement for algorithmic kernels. To better understand how the widening gap between processing power (cores) and memory bandwidth impacts scheduler performance, we quantify runtime improvements over a 4-fold range in the available bandwidth per core and show further improvements in the running times of kernels (up to 50%) as the bandwidth gap increases.

Finally, as part of our study, we generalize prior definitions of space-bounded schedulers to allow for more practical variants, and explore implementation tradeoffs, e.g., in a key parameter of such schedulers. This is useful for engineering space-bounded schedulers, which were previously described

only at a high level suitable for theoretical analyses, into a form suitable for real machines.

Contributions. The contributions of this paper are:

- A modular framework for describing schedulers, machines as tree of caches, and nested parallel programs (Section 3). The framework is equipped with timers and counters. Schedulers that are expected to work well on tree of cache models such space-bounded schedulers and certain work-stealing schedulers are implemented.
- A precise definition for the class of space-bounded schedulers that retains the competitive cache miss bounds expected for this class, but also allows more schedulers than previous definitions (which were motivated mainly by theoretical guarantees [15, 14, 6]) (Section 4). We describe two variants, highlighting the engineering details that allow for low overhead.
- The first experimental study of space-bounded schedulers, and the first head-to-head comparison with work-stealing schedulers (Section 5). On a common multicore machine configuration (4 sockets, 32 cores, 3 levels of caches), we quantify the reduction in L3 cache misses incurred by space-bounded schedulers relative to both work-stealing variants on synthetic and non-synthetic benchmarks. On bandwidth-bound benchmarks, an improvement in cache misses translates to improvement in running times, although some of the improvement is eroded by the greater overhead of the space-bounded scheduler.

2. DEFINITIONS

We start with a recursive definition of nested parallel computation, and use it to define what constitutes a schedule. We will then define the parallel memory hierarchy (PMH) model—a machine model that reasonably accurately represents shared memory parallel machines with deep memory hierarchies. This terminology will be used later to define schedulers for the PMH model.

Computation Model, Tasks and Strands. We consider computations with nested parallelism, allowing arbitrary dynamic nesting of fork-join constructs including parallel loops, but no other synchronizations. This corresponds to the class of algorithms with series-parallel dependence graphs (see Fig. 2(left)).

Nested parallel computations can be decomposed into "tasks", "parallel blocks" and "strands" recursively as follows. As a base case, a ***strand*** is a serial sequence of instructions not containing any parallel constructs or subtasks. A ***task*** is formed by serially composing $k \geq 1$ strands interleaved with $(k-1)$ "parallel blocks", denoted by $\mathtt{t} = \ell_1; \mathtt{b}_1; \ldots; \ell_k$. A ***parallel block*** is formed by composing in parallel one or more tasks with a fork point before all of them and a join point after (denoted by $\mathtt{b} = \mathtt{t}_1 \| \mathtt{t}_2 \| \ldots \| \mathtt{t}_k$). A parallel block can be, for example, a parallel loop or some constant number of recursive calls. The top-level computation is a task. We use the notation $\mathsf{L}(\mathtt{t})$ to indicate all strands that are recursively included in a task.

Our computation model assumes all strands share a single memory address space. We say two strands are ***concurrent*** if they are not ordered in the dependence graph. Concurrent reads (*i.e.*, concurrent strands reading the same memory location) are permitted, but not data races (*i.e.*, concurrent strands that read or write the same location with at least one write). For every strand ℓ, there exists a task $\mathtt{t}(\ell)$ such that ℓ is nested immediately inside $\mathtt{t}(\ell)$. We call this the ***task of strand*** ℓ.

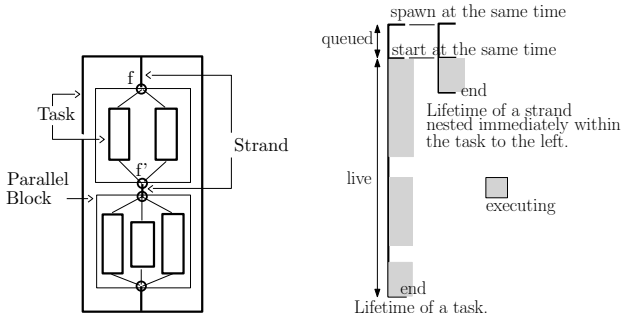


Figure 2: (left) Decomposing the computation: tasks, strands and parallel blocks. f, f' are corresponding fork and join points. (right) Timeline of a task and its first strand, showing the difference between being *live* and execution.

Schedule. We now define what constitutes a valid schedule for a nested parallel computation on a machine. These definitions will enable us to precisely define space-bounded schedulers later on. We restrict ourselves to non-preemptive schedulers—schedulers that cannot migrate strands across cores once they begin executing. Both work-stealing and space-bounded schedulers are non-premptive. We use P to denote the set of cores on the machine, and L to denote the set of strands in the computation. A non-preemptive schedule defines three functions for each strand ℓ.

- **Start time:** $start : \mathsf{L} \to \mathbb{Z}$, where $start(\ell)$ denotes the time the first instruction of ℓ begins executing;
- **End time:** $end : \mathsf{L} \to \mathbb{Z}$, where $end(\ell)$ denotes the (post-facto) time the last instruction of ℓ finishes; and
- **Location:** $proc : \mathsf{L} \to P$, where $proc(\ell)$ denotes the core on which the strand is executed. Note that $proc$ is well defined because of the non-preemptive policy for strands.

We say that a strand ℓ is ***live*** at any time τ with $start(\ell) \leq \tau < end(\ell)$.

A non-preemptive schedule must also obey the following constraints on the ordering of strands and timing:

- (*ordering*): For any strand ℓ_1 ordered by the fork-join dependence graph before ℓ_2: $end(\ell_1) \leq start(\ell_2)$.
- (*processing time*): For any strand ℓ, $end(\ell) = start(\ell) + \gamma_{\langle \text{schedule,machine} \rangle}(\ell)$. Here γ denotes the processing time of the strand, which may vary depending on the specifics of the machine and the history of the schedule. The schedule alone does not control this value.
- (*non-preemptive execution*): No two strands may be live on the same core at the same time, i.e., $\ell_1 \neq \ell_2, proc(\ell_1) = proc(\ell_2) \implies [start(\ell_1), end(\ell_1)) \cap [start(\ell_2), end(\ell_2)) = \emptyset$.

We extend the same notation and terminology to tasks. The start time $start(\mathtt{t})$ of a task $\mathtt{t}$ is a shorthand for $start(\mathtt{t}) = start(\ell_s)$, where ℓ_s is the first strand in $\mathtt{t}$. Similarly $end(\mathtt{t})$ denotes the end time of the last strand in $\mathtt{t}$. The function $proc$, however, is undefined for tasks as a task's contained strands may execute on different cores.

When discussing specific schedulers, it is convenient to consider the time a task or strand first becomes available to execute. We use the term ***spawn time*** to refer to this time, which is the instant at which the preceding fork or join finishes. Naturally, the spawn time is no later than the start time, but a schedule may choose not to execute the task or strand immediately. We say that the task or strand is ***queued*** during the time between its spawn time and start time and ***live*** during the time between its start time and end time. Fig. 2(right) illustrates the spawn, start and end times of a task and its initial strand. The task and initial strand are spawned and start at the same time by definition. The strand is continuously executed until it ends, while a task goes through several phases of execution and idling before it ends.

Machine Model: Parallel Memory Hierarchy (PMH). Following prior work addressing multi-level parallel hierarchies [2, 12, 4, 13, 31, 9, 15, 14, 6], we model parallel machines using a tree-of-caches abstraction. For concreteness, we use a symmetric variant of the parallel memory hierarchy (PMH) model [2] (see Fig. 1(b)), which is consistent with many other models [4, 9, 12, 13, 15, 14]. A PMH consists of a height-h tree of memory units, called ***caches***. The leaves of the tree are at level-0 and any internal node has level one greater than its children. The leaves (level-0 nodes) are cores, and the level-h root corresponds to an infinitely large main memory. As described in [6] each level in the tree is parameterized by four parameters: the size of the cache M_i at level i, the block size B_i used to transfer to the next higher level, the cost of a cache miss C_i which represents the combined costs of latency and bandwidth, and the fanout f_i (number of level $i-1$ caches below it).

3. EXPERIMENTAL FRAMEWORK

We implemented a C++ based framework with the following design objectives in which nested parallel programs and schedulers can be built for shared memory multicore machines. The implementation, along with a few schedulers and algorithms, is available on the web page `http://www.cs.cmu.edu/~hsimhadr/sched-exp`. Some of the code for the threadpool module has been adapted from an earlier implementation of threadpool [20].

Modularity: The framework separates the specification of three components—programs, schedulers, and description of machine parameters—for portability and fairness. The user can choose any of the candidates from these three categories. Note, however, some schedulers may not be able to execute programs without scheduler-specific hints (such as space annotations).

Clean Interface: The interface for specifying the components should be clean, composable, and the specification built on the interface should be easy to reason about.

Hint Passing: While it is important to separate program and schedulers, it is useful to allow the program to pass hints (extra annotations on tasks) to the scheduler to guide its decisions.

Minimal Overhead: The framework itself should be lightweight with minimal system calls, locking and code complexity. The control flow should pass between the functional modules (program, scheduler) with negligible time spent outside. The framework should avoid generating background memory traffic and interrupts.

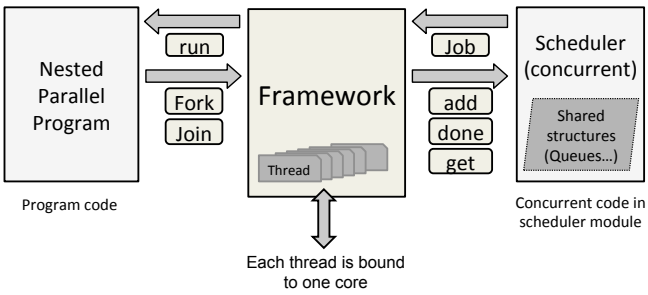


Figure 3: Interface for the program and scheduling modules.

Timing and Measurement: It should enable fine-grained measurements of the various modules. Measurements include not only clock time, but also insightful hardware counters such as cache and memory traffic statistics. In light of the earlier objective, the framework should avoid OS system calls for these, and should use direct assembly instructions.

3.1 Interface

The framework has separate interfaces for the program and the scheduler.

Programs: Nested parallel programs, with no other synchronization primitives, are composed from tasks using `fork` and `join` constructs. A `parallel_for` primitive built with `fork` and `join` is also provided. Tasks are implemented as instances of classes that inherit from the `Job` class. Different kinds of tasks are specified as classes with a method that specifies the code to be executed. An instance of a class derived from `Job` is a task containing a pointer to a strand nested immediately within the task. The control flow of this function is sequential with a terminal `fork` or `join` call. (This interface could be readily extended to handle non-nested parallel constructs such as futures [30] by adding other primitives to the interface beyond `fork` and `join`.)

The interface allows extra annotations on a task such as its size, which is required by space-bounded schedulers. Such tasks inherit a derived class of `Job` class, the extensions in the derived class specifying the annotations. For example, the class `SBJob` suited for space-bounded schedulers is derived from `Job` by adding two functions—`size(uint block_size)` and `strand_size(uint block_size)`—that allow the annotations of the job size.

Scheduler: The scheduler is a concurrent module that handles queued and live tasks (as defined in Section 2) and is responsible for maintaining its own queues and other internal shared data structures. The module interacts with the framework that consists of a thread attached to each processing core on the machine, through an interface with three call-back functions.

- `Job* get (ThreadIdType)`: This is called by the framework on behalf of a thread attached to a core when the core is ready to execute a new strand, after completing a previously live strand. The function may change the internal state of the scheduler module and return a (possibly null) `Job` so that the core may immediately begin executing the strand. This function specifies *proc* for the strand.
- `void done(Job*,ThreadIdType)` This is called when a core finishes the execution of a strand. The scheduler

is allowed to update its internal state to reflect this completion.

- `void add(Job*,ThreadIdType)`: This is called when a `fork` or `join` is encountered. In case of a `fork`, this call-back is invoked once for each of the newly spawned tasks. For a `join`, it is invoked for the continuation task of the `join`. This function decides where to enqueue the job.

Other auxiliary parameters to these call-backs have been dropped from the above description for clarity and brevity. The `Job*` argument passed to these functions may be instances of one of the derived classes of `Job*` that carry additional information helpful to the scheduler. Appendix A presents an example of a work-stealing scheduler implemented in this scheduler interface.

Machine configuration: The interface for specifying machine descriptions accepts a description of the cache hierarchy: number of levels, fanout at each level, and cache and cache-line size at each level. In addition, a mapping between the logical numbering of cores on the system to their left-to-right position as a leaf in the tree of caches must be specified. For example, Fig. 4 is a description of one Nehalem-EX series 4-socket × 8-core machine (32 physical cores) with 3 levels of caches as depicted in Fig. 1(a).

```
int num_procs=32;
int num_levels = 4;
int fan_outs[4] = {4,8,1,1};
long long int sizes[4] = {0, 3*(1<<22), 1<<18, 1<<15};
int block_sizes[4] = {64,64,64,64};
int map[32] = {0,4,8,12,16,20,24,28,
               2,6,10,14,18,22,26,30,
               1,5,9,13,17,21,25,29,
               3,7,11,15,19,23,27,31};
```

Figure 4: Specification entry for a 32-core Xeon® machine depicted in Fig. 1(a).

3.2 Implementation

The runtime system initially fixes a POSIX thread to each core. Each thread then repeatedly performs a call (`get`) to the scheduler module to ask for work. Once assigned a task and a specific strand inside it, the thread completes the strand and asks for more work. Each strand either ends in a `fork` or a `join`. In either scenario, the framework invokes the `done` call back. For a `fork`, the `add` call-back is invoked to let the scheduler add new tasks to its data structures.

All specifics of how the scheduler operates (*e.g.*, how the scheduler handles work requests, whether it is distributed or centralized, internal data structures, where mutual exclusion occurs, etc.) are relegated to scheduler implementations. Outside the scheduling modules, the runtime system includes no locks, synchronization, or system calls (except during the initialization and cleanup of the thread pool), meeting our design objective.

3.3 Measurements

Active time and overheads: Control flow on each thread moves between the program and the scheduler modules. Fine-grained timers in the framework break down the execution time into five components: (i) active time—the time spent executing the program, (ii) `add` overhead, (iii) `done` overhead, (iv) `get` overhead, and (v) empty queue overhead. While active time depends on the number of instructions and the communication costs of the program, `add`, `done` and `get` overheads depend on the complexity of the scheduler, and the number of times the scheduler code is invoked by forks and joins. The empty queue overhead is the amount of time the scheduler fails to assign work to a thread (`get` returns null), and reflects on the load balancing capability of the scheduler. In most of the results in Section 5, we usually report two numbers: active time averaged over all threads and the average overhead, which includes measures (ii)–(v). Note that while we might expect this partition of time to be independent, it is not so in practice—the background coherence traffic generated by the scheduler's bookkeeping may adversely affect active time. The timers have very little overhead in practice—less than 1% in most of our experiments.

Measuring hardware counters: Modern multicores are equipped with hardware counters that can provide various performance statistics such as the number of cache misses at various levels. Such counters, however, are somewhat challenging to use. Appendix B details the specific methodology we used for the Intel® Nehalem architecture.

4. SCHEDULERS

In this section, we will define the class of space-bounded schedulers and describe the schedulers we compare using our framework.

4.1 Space-bounded Schedulers

Space-bounded schedulers are designed to achieve good cache performance on PMHs and use the sizes of tasks and strands to choose a schedule $(start, end, proc)$ mapping the hierarchy of tasks in a nested parallel program to the hierarchy of caches. They have been described previously in [15] and [6]. The definitions in [15] are not complete in that they do not specify how to handle "skip level" tasks (described below). Our earlier definition [6] handled skip level tasks, but was more restrictive than necessary, ruling out practical variants, as discussed below. Here, we provide a broader definition for the class of space-bounded schedulers that allows for practical variants, while retaining the strong analytical bounds on cache misses that are the hallmark of the class of space-bounded schedulers. Specifically, our new definition provides more flexibility in the scheduling of strands by (i) allowing each strand to have its own size, and (ii) accounting for strand sizes differently from task sizes.

Informally, a space-bounded schedule satisfies two properties: (i) *Anchored:* Each task `t` gets "anchored" to a smallest possible cache that is larger than its size—strands within `t` can only be scheduled on cores in the tree rooted at the cache; and (ii) *Bounded:* At any point in time, the sum of the sizes of all tasks and strands occupying space in a cache is at most the size of the cache. These two conditions (when formally defined) are sufficient to imply strong bounds on the number of cache misses at every level in the tree of caches. A good space-bounded scheduler would also handle load balancing subject to anchoring constraints to quickly complete execution.

More formally, a space-bounded scheduler is parameterized by a global ***dilation*** parameter $0 < \sigma \leq 1$ and machine parameters $\{M_i, B_i, C_i, f_i\}$. We will need the following terminologies for the definition (which are both simplified and generalized from [6]).

Chowdhury et. al. [15] also suggest two other approaches to scheduling: CGC (coarse-grained contiguous) and CGC-on-SB (which combines CGC with space bounded schedulers). CGC is designed to keep nearby blocks of iterations close together so they are run on the same cache. This can be simulated in our framework by grouping iterations recursively (which is what we do). This means that our approach will not have constant critical path length for certain algorithms, but experimentally we are using a modest number of cores. CGC-on-SB is primarily a mechanism for skipping levels in the cache. Our variant of space-bounded scheduler already allows for level skipping.

Task Size and Strand Size: The size of a task (strand) is defined as a function of cache-line size B, independent of the scheduler. Let $loc(\mathsf{t}; B)$ denote the set of distinct cache lines touched by instructions within a task t. Then $S(\mathsf{t}; B) = |loc(\mathsf{t}; B)| \cdot B$ denotes the ***size*** of t. The size of a strand is defined in the same way. While results in [6] show that it is not necessary for the analytical bounds that strands be allowed to have their own size, we found that the flexibility it enables is an important running time optimization.[1] Note that even strands immediately nested within the same task may have different sizes.

Cluster: For any cache X_i, its *cluster* is the set of caches and cores nested below X_i. Let $P(X_i)$ denote the set of cores in X_i's cluster.

Befitting Cache: Given a particular cache hierarchy and dilation parameter $\sigma \in (0, 1]$, we say that a level-i cache ***befits*** a task t if $\sigma M_{i-1} < S(\mathsf{t}, B_i) \leq \sigma M_i$.

Maximal Task: We say that a task t with parent task t' is ***level-i maximal*** if and only if a level-i cache befits t but not t', i.e., $\sigma M_{i-1} < S(\mathsf{t}, B_i) \leq \sigma M_i < S(\mathsf{t}', B_i)$.

Anchored: A task t with strand set $\mathsf{L}(t)$ is said to be ***anchored*** to level-i cache X_i (or equivalently to X_i's cluster) if and only if (i) it is executed entirely in the cluster, i.e., $\{proc(\ell) | \ell \in \mathsf{L}(t)\} \subseteq P(X_i)$, and (ii) the cache befits the task. Anchoring prevents the migration of tasks to a different cluster or cache. The advantage of anchoring a task to a befitting cache is that once it loads its working set, it can reuse the working set without the risk of losing it from the cache. If a task is not anchored anywhere, for notational convenience we assume it is anchored at the root of the tree.

Cache-occupying tasks: The definition depends on whether the cache is inclusive or non-inclusive. For a level-i inclusive cache X_i and time τ, the set of ***cache-occupying tasks*** for a cache X_i at time τ, denoted by $\mathsf{Ot}(X_i, \tau)$, is the union of (a) the maximal tasks live at time τ that are anchored to X_i, and (b) the maximal tasks live at time τ that are anchored to any cache below X_i in the hierarchy whose immediate parents are anchored to a cache above X_i in the hierarchy. The tasks in (b) are called "skip level" tasks. For a non-inclusive cache, only type (a) tasks are included in $\mathsf{Ot}(X_i, \tau)$. Tasks in $\mathsf{Ot}(X_i, \tau)$ are the tasks that consume space in the cache at time τ. Note that we need account only for *maximal* tasks because any non-maximal task t' is anchored to the same cache as its closest enclosing maximal task t and $loc(\mathsf{t}'; B) \subseteq loc(\mathsf{t}; B)$.

[1]On the other hand, it does require additional size information on programs—thus we view it as optional: Any strand whose size is not specified is assumed by default to be the size of its enclosing task.

Cache-occupying strands: The set of ***cache-occupying strands*** for a cache X_i at time τ, denoted by $\mathsf{Ol}(X_i, \tau)$, is the set of strands $\{\ell\}$ such that (a) ℓ is live at time τ (b) ℓ is executed below X_i, i.e., $proc(\ell) \in P(X_i)$, and (c) ℓ's task is anchored strictly above X_i.

A ***space-bounded scheduler*** for a particular cache hierarchy is a scheduler parameterized by $\sigma \in (0, 1]$ that satisfies the following two properties:

- *Anchored:* Every subtask (recursively) of the root task is anchored to a **befitting** cache.
- *Bounded:* At every time τ, for every level-i cache X_i, the sum of the sizes of cache-occupying tasks and strands is at most M_i:

$$\Sigma_{\mathsf{t} \in \mathsf{Ot}(X_i, \tau)} S(\mathsf{t}, B_i) + \Sigma_{\ell \in \mathsf{Ol}(X_i, \tau)} S(\ell, B_i) \leq M_i.$$

A key property of the definition of space-bounded schedulers in [6] is that for any PMH and any level i, one can upper bound the number of level-i cache misses incurred by executing a task on the PMH by $Q^*(\mathsf{t}; \sigma M_i, B_i)$, where Q^* is the cache complexity as defined in the Parallel Cache-Oblivious (PCO) model in [6]. Roughly speaking, the cache complexity Q^* of a task t in terms of a cache of size M and line size B is defined as follows. Decompose the task into a collection of maximal subtasks that fit in M space, and "glue nodes" – instructions outside these subtasks. For a maximal size M task t', the PCO cache complexity $Q^*(\mathsf{t}'; M; B)$ is defined to be the number of distinct cache lines it accesses, counting accesses to a cache line from unordered instructions multiple times. The model then pessimistically counts all memory instructions that fall outside of a maximal subtask (i.e., glue nodes) as cache misses. The total cache complexity of an algorithm is the sum of the complexities of the maximal subtasks, and the memory accesses outside of maximal subtasks. Note that Q^* is a *program-centric* or *machine-independent* metric, capturing the inherent locality in a parallel algorithm [7, 29].

THEOREM 1. *Consider a PMH and any dilation parameter $0 < \sigma \leq 1$. Let t be a task anchored to the root of the tree. Then the number of level-i cache misses incurred by executing t with any space-bounded scheduler is at most $Q^*(\mathsf{t}; \sigma M_i, B_i)$, where Q^* is the cache complexity as defined in the Parallel Cache-Oblivious (PCO) model in [6].*

The proof is a simple adaptation of the proof in [6] to account for our more general definition. At a high level, the argument is that for any cache X_i, the cache-occupying tasks and strands for X_i bring in their working sets into the cache X_i exactly once because the boundedness property prevents cache overflows. Thus a replacement policy that keeps these working sets in the cache until they are no longer needed will incur no additional misses; because the PMH model assumes an ideal replacement policy, it will perform at least as well.

In the course of our experimental study, we found that the following minor modification to the boundedness property of space-bounded schedulers improves their performance. Namely, we introduce a new parameter $\mu \in (0, 1]$ ($\mu = 0.2$ in our experiments) and modify the boundedness property to be such that at every time τ, for every level-i cache X_i:

$$\Sigma_{\mathsf{t} \in \mathsf{Ot}(X_i, \tau)} S(\mathsf{t}, B_i) + \Sigma_{\ell \in \mathsf{Ol}(X_i, \tau)} \min\{\mu M_i, S(\ell, B_i)\} \leq M_i,$$

The minimum term with μM_i is to allow several large strands to be explored simultaneously without their space measure

taking too much of the space bound. This helps the scheduler to quickly traverse the higher levels of recursion in the DAG and reveal parallelism so that the scheduler can achieve better load balance.

Given this modified boundedness condition, it is easy to show that the bound in Theorem 1 becomes $Q^*(\mathtt{t}; \mu\sigma M_i, B_i)$.

Note that while setting σ to 1 yields the best bounds on cache misses, it also makes load balancing harder. As we will see later, a lower value for σ like 0.5 allows greater scheduling flexibility.

4.2 Schedulers implemented

Space-bounded schedulers: SB and SB-D. We implemented a space-bounded scheduler by constructing a tree of caches based on the specification of the target machine. Each cache is assigned one logical queue, a counter to keep track of "occupied" space and a lock to protect updates to the counter and queue. Cores can be considered to be leaves of the tree; when a scheduler call-back is issued to a thread, that thread can modify an internal node of the tree after gathering all locks on the path to the node from the core it is mapped onto. This scheduler accepts `Job`s which are annotated with task and strand sizes. When a new `Job` is spawned at a fork, the `add` call-back enqueues it at the cluster where its parent was anchored. For a new `Job` spawned at a join, `add` enqueues it at the cluster where the `Job` that called the corresponding fork of this join was anchored.

A basic version of such a scheduler would implement logical queues at each cache as one queue. However, this presents two problems: (i) It is difficult to separate tasks in queues by the level of cache that befits it, and (ii) a single queue might be a contention hotspot. To solve problem (i), behind each logical queue, we use separate "buckets" for each level of cache below to hold tasks that befit those levels. Cores looking for a task at a cache go through these buckets from the top (heaviest tasks) to bottom. We refer to this variant as the **SB** scheduler. To solve problem (ii) involving queueing hotspots, we replace the top bucket with a distributed queue—one queue for each child cache—like in the work-stealing scheduler. We refer to the **SB** scheduler with this modification as the **SB-D** scheduler.

Work-Stealing scheduler: WS. A basic work-stealing scheduler based on Cilk++ [11] is implemented and is referred to as the **WS** scheduler. Since the Cilk++ runtime system is built around work-stealing and deeply integrated and optimized exclusively for it, we focus our comparison on the **WS** implementation in our framework for fairness and to allow us to implement variants. The head-to-head micro-benchmark study in Section 5 between our **WS** implementation and Cilk$^{\text{TM}}$ Plus (the commercial version of Cilk++) suggests that, for these benchmarks, **WS** well-represents the performance of Cilk++'s work-stealing. We associate a double-ended queue (dequeue) of ready tasks with each core. The function `add` enqueues new tasks (or strands) spawned on a core to the bottom of its dequeue. When in need of work, the core uses `get` to remove a task from the bottom of its dequeue. If its dequeue is empty, it chooses another dequeue uniformly at random, and *steals* the work by removing a task from the *top* of that core's dequeue. The only contention in this type of scheduler is on the distributed dequeues—there is no other centralized data structure.

To implement the dequeues, we employed a simple two-locks-per-dequeue approach, one associated with the owning core, and the second associated with all cores currently attempting to steal. Remote cores need to obtain the second lock before they attempt to lock the first. Contention is thus minimized for the common case where the core needs to obtain only the first lock before it asks for work from its own dequeue.

Priority Work-Stealing scheduler: PWS. Unlike in the basic **WS** scheduler, cores in the **PWS** scheduler [27] choose victims of their steals according to the "closeness" of the victims in the socket layout. Dequeues at cores that are closer in the cache hierarchy are chosen with a higher probability than those that are farther away to improve scheduling locality while retaining the load balancing properties of **WS** scheduler. On our 4 socket machines, we set the probability of an intra-socket steal to be 10 times that of an inter-socket steal.

5. EXPERIMENTS

The goal of our experimental study is to compare the performance of the four schedulers on a range of benchmarks, varying the available memory bandwidth. Our primary metrics are runtime and L3 (last level) cache misses. We have found that the cache misses on other levels do not vary significantly among the schedulers (within 5%). We will also validate our work-stealing implementation via a comparison with the commercial Cilk$^{\text{TM}}$ Plus scheduler. Finally, we will quantify the overheads for space-bounded schedulers and study the performance impact of the key parameter σ for space-bounded schedulers.

5.1 Benchmarks

We use seven benchmarks in our study. The first two are synthetic micro-benchmarks that mimic the behavior of memory-intensive divide-and-conquer algorithms. Because of their simplicity, we use these benchmarks to closely analyze the behavior of the schedulers under various conditions and verify that we get the expected cache behavior on a real machine. The remaining five benchmarks are a set of popular algorithmic kernels.

Recursive repeated map (RRM): This benchmark takes two n-length arrays A and B and a point-wise map function that maps elements of A to B. In our experiments each element of the arrays is a double and the function simply adds one. RRM first does a parallel point-wise map from A to B, and repeats the same operation multiple times. It then divides A and B into two by some ratio (e.g., 50/50) and recursively calls the same operation on each of the two parts. The base case of the recursion is set to some constant at which point the recursion terminates. The input parameters are the size of the arrays n, number of repeats r, the cut ratio f, and the base-case size. We set $r = 3$ (the number of repeats in quicksort), and $f = 50\%$ in the experiments unless mentioned otherwise. RRM is a memory intensive benchmark because there is very little work done per memory operation. However, once a recursive call fits in a cache (i.e., the cache size is at least $16n$ bytes for subproblem size n), all remaining accesses are cache hits.

Recursive repeated gather (RRG): This benchmark is similar to RRM but instead of doing a simple map it does a gather. In particular at any given level of the recursion it takes three n-length arrays A, B and I and for each location i sets $B[i] = A[I[i] \mod n]$. The values in I are random in-

tegers. As with RRM after repeating r times it splits the arrays in two and repeats on each part. RRG is even more memory intensive than RRM because its accesses are random instead of linear. Again, however, once a recursive call fits in a cache all remaining accesses are cache hits.

Quicksort: This is a parallel quicksort algorithm that both parallelizes the partitioning and the recursive calls, using a median-of-3 pivot selection strategy. It switches to a version which parallelizes only the recursive calls for $n < 128K$ and a serial version for $n < 16K$. These parameters worked well for all the schedulers. We note that our quicksort is about 2x faster than the Cilk code found in the Cilk+ guide [22]. This is because it does the partitioning in parallel. It is also the case that the divide does not exactly partition the data evenly, because it depends on how well the pivot divides the data. For an input of size n the program has cache complexity $Q^*(n; M, B) = O(\lceil n/B \rceil \log_2(n/M))$ and therefore is reasonably memory intensive.

Samplesort: This is cache-optimal parallel Sample Sort algorithm described in [9]. The algorithm splits the input of size n into $\sqrt{n}$ subarrays, recursively sorts each subarray, "block transposes" them into $\sqrt{n}$ buckets and recursively sorts these buckets. For this algorithm, $Q^*(n; M, B) = O(\lceil n/B \rceil \log_{2+(M/B)} n/B)$ making it relatively cache friendly, and optimally cache oblivious even.

Aware samplesort: This is a variant of sample-sort that is aware of the cache sizes. In particular it moves elements into buckets that fit into the L3 cache and then runs quicksort on the buckets. This is the fastest sort we implemented and is in fact faster than any other sort we found for this machine. In particular it is about 10% faster than the PBBS sample sort [28].

Quad-tree: This generates a quad tree for a set of n points in two dimensions. This is implemented by recursively partitioning the points into four sets along the mid line of each of two dimensions. When the number of points is less than 16K we revert to a sequential algorithm.

Matrix multiplication: This benchmark is an 8-way recursive matrix multiplication. To allow for an in-place implementation, four of the recursive calls are invoked in parallel followed by the other four. Matrix multiplication has cache complexity $Q^*(n; M, B) = (\lceil n^2/B \rceil \times \lceil n/\sqrt{M} \rceil)$. The ratio of instructions to cache misses is therefore very high, about $B\sqrt{M}$, making this is a very compute-intensive benchmark. We switch to serial Intel® Math Kernel Library's `cblas_dgemm` matrix multiplication for sizes of $\leq 128 \times 128$ to make use of the floating point SIMD operations.

5.2 Experimental Setup

The benchmarks were run on a 4-socket 32-core Xeon® 7560 machine (Nehalem-EX architecture), as described in Fig. 1(a) and Fig. 4, using each of the four schedulers. Each core uses 2-way hyperthreading. The last level cache on each socket is a 24MB L3 cache that is shared by eight cores. Each of the four sockets on the machine has memory links to distinct DRAM modules. The sockets are connected with the Intel® QPI interconnect. Memory requests from a socket to a DRAM module connected to another socket pass through the QPI, the remote socket, and the remote memory link. To prevent excessive TLB cache misses, we use Linux hugepages of size 2MB to pre-allocate the space required by the algorithms. We configured the system to have a pool of $10,000$ huge pages by setting `vm.nr_hugepages` to that value using `sysctl`. We used the `hugectl` tool to execute memory allocations with hugepages.

Monitoring L3 cache. We focus on L3 cache misses, the most expensive cache level before DRAM on our machine. While we will report runtime subdivided into application code and scheduler code, such partitioning was not possible for L3 misses because of software limitations. Even if it were possible to count them separately, it would be difficult to interpret the results because of the non-trivial interference between the data cached by the program and by the scheduler. Further details can be found in Appendix B.

Controlling bandwidth. As part of our study, we will quantify runtime improvements varying the available memory bandwidth per core (the "bandwidth gap"). We control the memory bandwidth available to the program as follows. Because the QPI has high bandwidth, if we treat the RAM as a single functional module, the bandwidth between the L3 and the RAM depends on the number of memory links used, which in turn depends on the mapping of pages to the DRAM modules. If all the pages used by a program are mapped to DRAM modules connected to one socket, the program effectively utilizes one-fourth of the memory bandwidth. On the other hand, an even distribution of pages to DRAM modules across the sockets provides the full bandwidth to the program. The `numactl` tool can be used to control this mapping.

Code. We use the same code for the applications/algorithms for all benchmarks, except for the CilkPlus code, which we kept as close as possible. The code always includes the space annotations, but those annotations are ignored by the schedulers that do not need them. The code was compiled with a CilkPlus fork of `gcc 4.8.0` compiler.

5.3 Results

We use $\sigma = 0.5$ and $\mu = 0.2$ in the **SB** and **SB-D** schedulers, after some experimentation with the parameters, unless otherwise noted. All numbers reported in this paper are the average of at least 10 runs with the smallest and largest readings across runs removed. The results are not sensitive to this particular choice of reporting.

Synthetic benchmarks. Fig. 5 and Fig. 6 show the number of L3 cache misses of RRM and RRG, respectively, along with their active times and scheduling overheads on 64 hyperthreads at different bandwidth values. In addition to the four schedulers discussed in Section 4.2 (**WS**, **PWS**, **SB** and **SB-D**), we include the CilkPlus work-stealing scheduler in these plots to validate our **WS** implementation. We could not separate overhead from time in CilkPlus because it does not supply such information.

These plots show that the space-bounded schedulers incur roughly 42–44% fewer L3 cache misses than the work-stealing schedulers. As expected, the number of L3 misses does not significantly depend on the available bandwidth. On the other hand, the active time is most influenced by the number of instructions in the benchmark (constant across schedulers) and the costs incurred by L3 misses. The extent to which improvements in L3 misses translates to an improvement in active time depends on the memory bandwidth given to the program. When the bandwidth is low (25%), the active times are almost directly proportional to

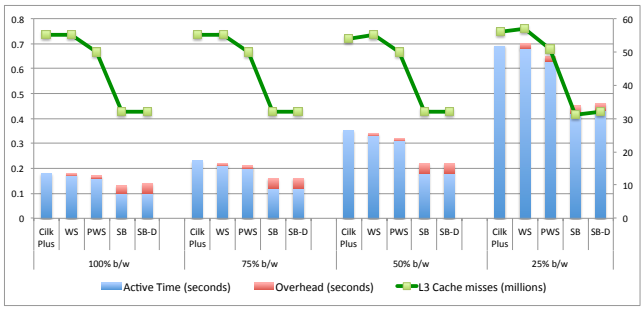


Figure 5: RRM on 10 million `double` elements, varying the memory bandwidth. Left axis is running time in seconds. Right axis is L3 misses in millions.

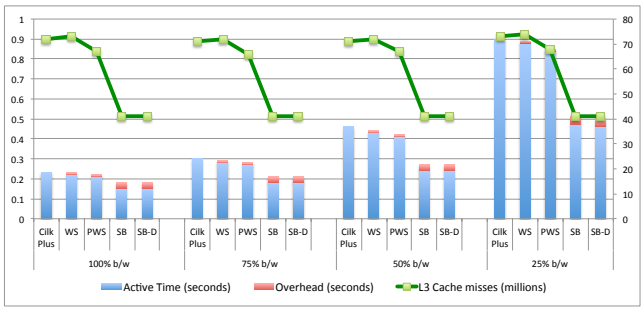


Figure 6: RRG on 10 million `double` elements, varying the memory bandwidth. Left axis is running time in seconds. Right axis is L3 misses in millions.

the number of L3 cache misses, while at full bandwidth, the active times are far less sensitive to misses.

The differences in L3 cache costs of space-bounded and work-stealing schedulers roughly corresponds to the difference between the cache complexity of the program with a cache of size σM_3 (M_3 being the size of L3) and a cache of size $M_3/16$ (because eight cores with sixteen hyperthreads share an L3 cache). In other words, space-bounded schedulers share the cache constructively while work-stealing schedulers effectively split the cache between the cores. To see this, consider our RRM benchmark. Each Map operation that does not fit in L3 touches $2 \times 10^7 \times 8 = 160$ million bytes of data, and RRM has to unfold four levels of recursion before it fits in $\sigma M_3 = 0.5 \times 24MB= 12$MB space with space-bounded schedulers. Therefore, since the cache line size $B_3 = 64$ bytes, space-bounded schedulers incur about $(160 \times 10^6 \times 3 \times 4)/64 = 30 \times 10^6$ cache misses, which matches closely with the results in Fig. 5. On the other hand, the number of cache misses of the **WS** scheduler (55 million) corresponds to unfolding about 7 levels of recursions, three more than with space-bounded schedulers. Loosely speaking, this means that the recursion has to unravel to one-sixteenth the size of L3 before work-stealing schedulers start preserving locality.

To support this observation, we ran the RRM and RRG benchmarks varying the number of cores per socket; the results are in Fig. 7. The number of L3 cache misses when using the **SB** and **SB-D** schedulers do not change with the number of cores, because cores constructively share the L3 cache independent of their number. However, when using the **WS** and **PWS** schedulers, the number of L3 misses is

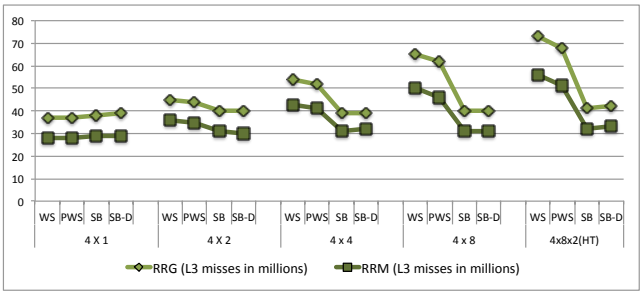


Figure 7: L3 cache misses for RRM and RRG, varying the number of cores used per socket.

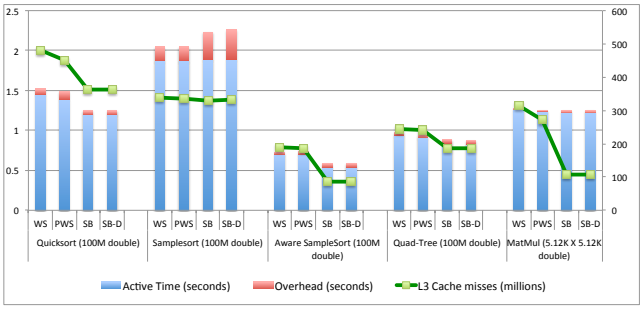


Figure 8: Active times, overheads, and L3 cache misses for the 5 benchmark algorithms at full bandwidth.

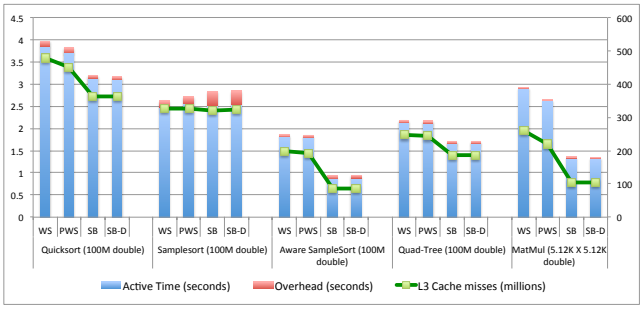


Figure 9: Active times, overheads, and L3 cache misses for the 5 benchmark algorithms at 25% bandwidth.

highly dependent on the number of cores: when fewer cores are active on each socket, there is lesser contention for space in the shared L3 cache. Thus, again the experimental results coincide with the theoretical analysis.

These experiments indicate that the advantages of space-bounded schedulers over work-stealing schedulers improve as (i) the number of cores per socket goes up, and (ii) the bandwidth per core goes down. At 8 cores there is a 30–35% reduction in L3 cache misses. At 64 cores we would expect (by the analysis) over a 60% reduction.

Algorithms. Fig. 8 and Fig. 9 show the active times, scheduling overheads, and L3 cache misses of the five algorithmic kernels at 100% and 25% bandwidth, respectively, with 64 hyperthreads. These plots show that the space-bounded schedulers incur significantly fewer L3 cache misses on 4 of the 5 benchmarks, with up to 65% on matrix multiply. The cache-oblivious sample sort is the sole benchmark

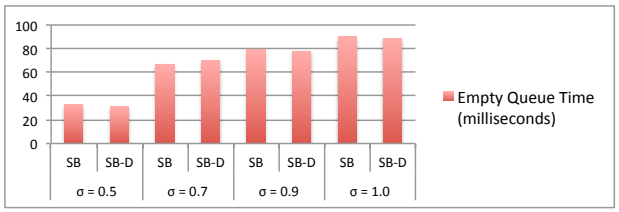


Figure 10: Empty queue times for Quad-tree.

with little difference in L3 cache misses across schedulers. Given the problem size $n = 10^8$ doubles and sample sort's $\sqrt{n}$-way recursion, all subtasks after one level of recursion are much smaller than the L3 cache. Thus, all four schedulers avoid overflowing the cache. Because of their added overhead relative to work-stealing, the space-bounded schedulers are 7% slower for this benchmark.

As with the synthetic benchmarks, the sensitivity of active time to L3 cache misses depends on whether the algorithm is memory-intensive enough to stress the bandwidth. Matrix Multiplication, although benefitting from space-bounded schedulers in terms of cache misses, shows no significant improvement in active time at full bandwidth because it is very compute intensive. However, when the machine bandwidth is reduced to 25%, Matrix Multiplication is bandwidth bound and the space-bounded schedulers are about 50% faster than work-stealing. The other three benchmarks—Quicksort, Aware Samplesort and Quad-tree—are memory intensive and see improvements of up to 25% in running time at full bandwidth (Fig. 8). At 25% bandwidth, the improvement is even more significant and up to 40% (Fig. 9).

Load balance and the dilation parameter σ. The choice of σ, determining which tasks are maximal, is an important parameter affecting the performance of space-bounded schedulers, especially their ability to load balance. If σ were set to 1, it is likely that one task that is about the size of the cache gets anchored to the cache, leaving little room for other tasks or strands. This adversely affects load balance, and we would expect to see greater empty queue times. On the other hand, if σ were set to a lower value like 0.5, then each cache can allow more than one task or strand to be simultaneously anchored, leading to better load balance. Fig. 10 gives an example algorithm (Quad-tree) demonstrating this. If σ is too low (closer to 0), then the number of levels of recursion until the space-bounded schedulers preserve locality would increase, resulting in less effective use of shared caches.

Comparison of scheduler variants. Looking at the results, we find that while **PWS** can reduce the number of cache misses by up to 10% compared to standard **WS**, it has negligible impact on running times for the benchmarks studied. Similarly, while **SB-D** is designed to remove a serial bottleneck in the **SB** scheduler, the runtime (and cache miss) performance of the two are nearly identical. This is because our benchmarks call the scheduler sufficiently infrequently so that the performance difference of each invocation is not noticeable in the overall running time.

6. CONCLUSION

We developed a framework for comparing schedulers, and deployed it on a 32-core machine with 3 levels of caches. We used it to compare four schedulers, two each of work-stealing and space-bounded types. As predicted by theory, we did notice that space-bounded schedulers demonstrate some, or even significant, improvement over work-stealing schedulers in terms of cache miss counts on shared caches for most benchmarks. In memory-intensive benchmarks with low instruction count to cache miss count ratios, an improvement in L3 miss count because of space-bounded schedulers can improve running time, despite their added overheads. On the other hand, for compute-intensive benchmarks or benchmarks with highly optimized cache complexities, work-stealing schedulers are slightly faster, because of their low scheduling overhead. Improving the overhead of space-bounded schedulers further could make the case for space-bounded schedulers stronger and is an important direction for future work.

Our experiments were run on an Intel® multicore with (only) 8 cores per socket, 32 cores total, and one level of shared cache (the L3). The experiments make it clear that as the core count per socket goes up (as is expected with each new generation), the advantages of space-bounded schedulers should increase due to the increased benefit of avoiding cache conflicts among the many unrelated threads sharing the limited on-chip cache capacity. As the core count per socket goes up, the available bandwidth per core decreases, again increasing space-bounded schedulers' advantage. We also anticipate a greater advantage for space-bounded schedulers over work-stealing schedulers when more cache levels are shared and when the caches are shared amongst a greater number of cores. Such studies are left to future work, when such multicores become available. On the other hand, compute-intensive benchmarks will likely continue to benefit from the lower scheduling overheads of work-stealing schedulers, for the next few generations of multicores, if not longer.

Acknowledgements

This work is supported in part by the National Science Foundation under grant numbers CCF-1018188, CCF-1314633, CCF-1314590, the Intel Science and Technology Center for Cloud Computing (ISTC-CC), the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics and Computer Science programs under contract No. DE-AC02-05CH11231 through the Dynamic Exascale Global Address Space (DEGAS) programming environments project, and a Facebook Graduate Fellowship.

7. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory Comp. Sys.*, 35(3), 2002.

[2] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers*, 1993.

[3] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), 1996.

[4] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA*, 2008.

[5] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*, 2012.
[6] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *SPAA*, 2011.
[7] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Program-centric cost models for locality. In *MSPC*, 2013.
[8] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *JACM*, 46(2), 1999.
[9] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low-depth cache oblivious algorithms. In *SPAA*, 2010.
[10] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *SPAA*, 1996.
[11] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5), 1999.
[12] R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *SPAA*, 2007.
[13] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *SPAA*, 2008.
[14] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 73(7):911 – 925, 2013. Best Papers of IPDPS 2010, 2011 and 2012.
[15] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *IPDPS*, 2010.
[16] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, et al. Sequoia: Programming the memory hierarchy. In *Supercomputing*, 2006.
[17] Intel. Intel Cilk++ SDK programmer's guide. `https://www.clear.rice.edu/comp422/resources/Intel_Cilk++_Programmers_Guide.pdf`, 2009.
[18] Intel. Intel Thread Building Blocks reference manual. `http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm\#reference/reference.htm`, 2013. Version 4.1.
[19] Intel. Performance counter monitor (PCM). `http://www.intel.com/software/pcm`, 2013. Version 2.4.
[20] R. Kriemann. Implementation and usage of a thread pool based on posix threads. `www.hlnum.org/english/projects/tools/threadpool/doc.html`, 2004.
[21] D. Lea. A java fork/join framework. In *ACM Java Grande*, 2000.
[22] C. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51, 2010.
[23] Microsoft. Task Parallel Library. `http://msdn.microsoft.com/en-us/library/dd460717.aspx`, 2013. .NET version 4.5.
[24] G. J. Narlikar. Scheduling threads for low space requirement and good locality. In *SPAA*, 1999.
[25] OpenMP Architecture Review Board. OpenMP API. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008. v 3.0.
[26] Perfmon2. libpfm. `http://perfmon2.sourceforge.net/`, 2012.
[27] J.-N. Quintin and F. Wagner. Hierarchical work-stealing. In *EuroPar*, 2010.
[28] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the problem based benchmark suite. In *SPAA*, 2012.
[29] H. V. Simhadri. *Program-Centric Cost Models for Locality and Parallelism*. PhD thesis, CMU, 2013.
[30] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *SPAA*, 2009.
[31] L. G. Valiant. A bridging model for multi-core computing. In *ESA*, 2008.

APPENDIX

A. WORK STEALING SCHEDULER

Fig. 11 provides an example of a work-stealing scheduler implemented using the scheduler interface presented in Section 3.1. The `Job*` argument passed to the `add` and `done` functions may be instances of one of the derived classes of `Job*` that carry additional information helpful to the scheduler.

B. MEASURING HARDWARE COUNTERS

Multicore processors based on newer architectures like Intel® Nehalem-EX and Sandybridge contain numerous functional components such as cores (which includes the CPU and lower level caches), DRAM controllers, bridges to the inter-socket interconnect (QPI) and higher level cache units (L3). Each component is provided with a performance monitoring unit (PMU)—a collection of hardware registers that can track statistics of events relevant to the component.

For instance, while the core PMU on Xeon® 7500 series (our experimental setup, see Fig. 1(a)) is capable of providing statistics such as the number of instructions, L1 and L2 cache hit/miss statistics, and traffic going in and out, it is unable to monitor L3 cache misses (which constitute a significant portion of active time). This is because L3 cache is a separate unit with its own PMU(s). In fact, each Xeon® 7560 die has eight L3 cache banks on a bus that also connects DRAM and QPI controllers (see Fig. 12). Each L3 bank is connected to a core via buffered queues. The address space is hashed onto the L3 banks so that a unique bank is responsible for each address. To collect L3 statistics such as L3 misses, we monitor PMUs (called C-Boxes on Nehalem-EX) on all L3 banks and aggregate the numbers in our results.

Software access to core PMUs on most Intel® architectures is well supported by several tools including the Linux kernel, the Linux perf tool, and higher level APIs such as libpfm [26]. We use the libpfm library to provide fine-grained access to the core PMU. However, access to *uncore* PMUs—complex architecture-specific components like the C-Box—is

```
void WS_Scheduler::add (Job *job, int thread_id) {
  _local_lock[thread_id].lock();
  _job_queues[thread_id].push_back(job);
  _local_lock[thread_id].unlock();
}
int
WS_Scheduler::steal_choice (int thread_id) {
  return (int)((((double)rand())/((double)RAND_MAX))
               *_num_threads);
}
Job* WS_Scheduler::get (int thread_id) {
  _local_lock[thread_id].lock();
  if (_job_queues[thread_id].size() > 0) {
    Job * ret = _job_queues[thread_id].back();
    _job_queues[thread_id].pop_back();
    _local_lock[thread_id].unlock();
    return ret;
  } else {
    _local_lock[thread_id].unlock();
    int choice = steal_choice(thread_id);
    _steal_lock[choice].lock();
    _local_lock[choice].lock();
    if (_job_queues[choice].size() > 0) {
      Job * ret = _job_queues[choice].front();
      _job_queues[choice].erase(_job_queues[choice].begin());
      ++_num_steals[thread_id];
      _local_lock[choice].unlock();
      _steal_lock[choice].unlock();
      return ret;
    }
    _local_lock[choice].unlock();
    _steal_lock[choice].unlock();
  }
  return NULL;
}
void WS_Scheduler::done (Job *job, int thread_id,
                    bool deactivate) {}
```

Figure 11: WS scheduler implemented in scheduler interface

not supported by most tools. Newer Linux kernels (3.7+) are incrementally adding software interfaces to these PMUs at the time of this writing, but we are only able to make program-wide measurements using this interface rather than fine-grained measurements. For accessing uncore counters, we adapt the Intel® PCM 2.4 tool [19].

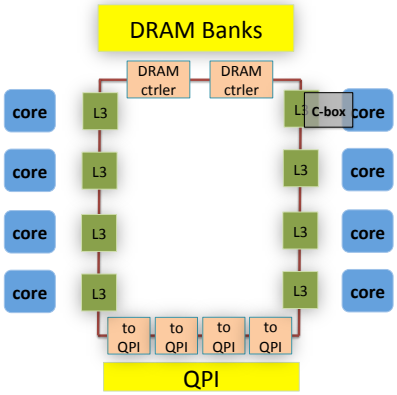


Figure 12: Layout of 8 cores and L3 cache banks on a bidirectional ring in Xeon® 7560. Each L3 bank hosts a performance monitoring unit called C-box that measures traffic into and out of the L3 bank.

To count L3 cache misses, the uncore counters in the C-boxes were programmed using the Intel® PCM tool to count misses that occur due to any reason (`LLC_MISSES - event code: 0x14, umask: 0b111`) and L3 cache fills in any coherence state (`LLC_S_FILLS - event code: 0x16, umask: 0b1111`). Both the numbers concur up to three significant digits in most cases. Therefore, only the L3 cache miss numbers are reported in this paper.