Parallelization Libraries: Characterizing and Reducing Overheads

ABHISHEK BHATTACHARJEE, Rutgers University
GILBERTO CONTRERAS, Nvidia Corporation
and MARGARET MARTONOSI, Princeton University

Creating efficient, scalable dynamic parallel runtime systems for chip multiprocessors (CMPs) requires understanding the overheads that manifest at high core counts and small task sizes.

In this article, we assess these overheads on Intel's Threading Building Blocks (TBB) and OpenMP. First, we use real hardware and simulations to detail various scheduler and synchronization overheads. We find that these can amount to 47% of TBB benchmark runtime and 80% of OpenMP benchmark runtime. Second, we propose load balancing techniques such as occupancy-based and criticality-guided task stealing, to boost performance.

Overall, our study provides valuable insights for creating robust, scalable runtime libraries.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms: Experimentation, Measurement, Performance

Additional Key Words and Phrases: Parallel libraries, Intel Threading Building Blocks, OpenMP, task stealing, performance

ACM Reference Format:
Bhattacharjee, A., Contreras, G., and Martonosi, M. 2011. Parallelization libraries: Characterizing and reducing overheads. ACM Trans. Architec. Code Optim. 8, 1, Article 5 (April 2011), 29 pages.
DOI = 10.1145/1952998.1953003 http://doi.acm.org/10.1145/1952998.1953003

1. INTRODUCTION

With chip multiprocessors (CMPs) quickly becoming the new norm in computing, programmers require tools that allow them to create parallel code in a quick and efficient manner. Industry and academia have, for years, worked to develop parallel runtime systems and libraries that aim at improving application portability and programming efficiency [Kale and Krishnan 1993; Blumofe et al. 1996; OpenMP 2002; Palatin et al. 2006; Gordon et al. 2006; Halstead 1985; Gelernter 1985]. This is achieved by allowing programmers to focus their efforts on identifying parallelism rather than worrying about how parallelism is managed and/or mapped to the underlying CMP architecture.

Programmers today have a few different options when considering parallelization libraries. Older runtime systems such as OpenMP [2002] are one option. OpenMP, developed by the OpenMP Architecture Review Board (ARB), was created in 1997 for multiplatform shared-memory applications. OpenMP applications include support for parallel tasks with a data environment conducive for task-based load balancing.

Another, more recent option likely to see wide use is the Intel Threading Building Blocks (TBB) runtime library [Reindeers 2007]. Based on the C++ language, TBB pro-

Author's addresses: A. Bhattacharjee, email: abhib@cs.rutgers.edu; M. Martonosi, email: mrm@princeton.edu.

DOI 10.1145/1952998.1953003 http://doi.acm.org/10.1145/1952998.1953003

vides programmers with an API used to exploit parallelism through the use of tasks rather than parallel threads. Moreover, TBB is able to significantly reduce load imbalance and improve performance scalability through task stealing, allowing applications to exploit concurrency with little regard to the underlying CMP characteristics (number of cores).

Available as a commercial product and under an open-source license, TBB has become an increasingly popular parallelization library. Adoption of its open-source distribution into existing Linux distributions is likely to increase its usage among programmers looking to take advantage of present and future CMP systems. Given its growing importance, it is natural to perform a detailed characterization of TBB's performance.

While parallel runtime libraries such as TBB make it easier for programmers to develop parallel code, software-based dynamic management of parallelism often comes at a cost. The parallel runtime library is expected to take annotated parallelism and distribute it across available resources. This dynamic management entails instructions and memory latency—cost that can be seen as parallelization overhead. With CMPs demanding ample amounts of parallelism in order to take advantage of available execution resources, applications will be required to harness all available parallelism, including fine-grain parallelism. Fine-grain parallelism, however, may incur high overhead on many existing parallelization libraries. Identifying and understanding parallelization overheads is the first step in the development of robust, scalable, and widely used dynamic parallel runtime libraries. We also offer proposals to reduce these overheads.

This article makes the following contributions.

—We use real-system measurements and cycle-accurate simulation of CMP systems to characterize and measure basic parallelism management costs of the TBB runtime library, studying their behavior under increasing core counts.
—We port a subset of the PARSEC benchmark suite to the TBB environment. Benchmarks are originally parallelized using a static arrangement of parallelism. Porting them to TBB increases their performance portability due to TBB's dynamic management of parallelism.
—Using these and other benchmarks, we dissect TBB activities into four basic categories and show that the runtime library can contribute up to 47% of the total per-core execution time on a 32-core system. While this overhead is much lower at low core counts, it hinders performance scalability by placing a core count dependency on performance.
—We study the performance of TBB's random task stealing, showing that while effective at low core counts, it provides suboptimal performance at high core counts. This leaves applications in need of alternative stealing policies.
—We show how an occupancy-based stealing policy can improve benchmark performance by up to 17% on a 32-core system, demonstrating how runtime knowledge of parallelism availability can be used by TBB to make more informed decisions.
—We also propose and evaluate criticality-guided task stealing, showing its performance benefits over both the default random TBB task stealer and occupancy-based stealer. On average, we see a performance improvement of 22% on a 32-core system, showing that runtime knowledge of relative thread speeds can greatly aid in intelligent load balancing.
—To better showcase the behavior of TBB, we also need to compare it to existing parallelization libraries. To this end, we also characterize and measure basic management costs of the OpenMP. Our set of workloads for this section consists of the NAS benchmarks. We break down the costs into a number of categories, showing that lock contention can lead to high overheads.

—We then propose mechanisms to mitigate lock overheads in OpenMP by using an iteration-stealing strategy similar to the occupancy-based stealer proposed for TBB. Our approaches achieve speedups as high as 3–4X over the default dynamic schedules at 32 cores.

Overall, our article's insights can help parallel programmers better exploit available concurrency, while aiding runtime developers to create more efficient and robust parallelization libraries.

Our article is organized as follows Section 2 gives a general description of Intel Threading Building Blocks and its dynamic management capabilities. Section 3 illustrates how TBB is used in C++ applications to annotate parallelism. Our methodology is described in Section 4 along with our set of benchmarks. In Section 5 we evaluate the cost of some of the fundamental operations carried by the TBB runtime library during dynamic management of parallelism. Section 6 studies the performance impact of TBB on our set of parallel applications, identifying overhead bottlenecks that degrade parallelism performance. Section 7 performs an in-depth study of TBB's random task stealing, the cornerstone of TBB's dynamic load-balancing mechanism. Section 8 then introduces the OpenMP API and scheduler. Section 9 quantifies the various overheads of the scheduler followed by Section 10, which offers application programmers and runtime library developers a set of recommendations for maximizing performance in these environments. Section 11 discusses related work, and Section 12 offers our conclusions and future work.

2. THE TBB RUNTIME LIBRARY

The Intel Threading Building Blocks (TBB) library has been designed to create portable, parallel C++ code. Inspired by previous parallel runtime systems such as OpenMP [2002] and Cilk [Blumofe et al. 1996], TBB provides C++ templates and concurrent structures that programmers use in their code to annotate parallelism and extract concurrency from their code. In this section we provide a brief description of TBB's capabilities and functionality, highlighting three of its major features: task programming model, dynamic task scheduling, and task stealing.

2.1. Task Programming Model

The TBB programming environment encourages programmers to express concurrency in terms of parallel tasks rather than parallel threads. Tasks are special regions of code that perform a specific action or function when executed by the TBB runtime library. They allow programmers to create portable, scalable parallel code by offering two important attributes. (1) Tasks typically have much shorter execution bodies than threads since tasks can be created and destroyed in a more efficient manner, and classes, offering programmers object-oriented capabilities. (2) Tasks are dynamically assigned to available execution resources by the runtime library to reduce load imbalance.

In TBB applications, tasks are described using C++ classes that contain the class `tbb:task` as the base class, which provides the virtual method *execute()*, among others. The method *execute()*, which the programmer is expected to specify, completely describes the execution body of the task. Once a task class has been specified and instantiated, it is ready to be launched into the runtime library for execution. In TBB, the most basic way for launching a new parallel task is through the use of the `spawn(task *t)` method, which takes a pointer to a task class as its argument. Once a task is scheduled for execution by the runtime library, the *execute()* method of the task is called in a nonpreemptive manner, completing the execution of the task.

Tasks are allowed to instantiate and spawn additional parallel tasks through hierarchical dependencies. In this way, derived tasks become children of the tasks that

Table I. TBB Templates for Annotating Common Types of Parallelism

Template	Description
parallel_for<range, body>	Template for annotating DOALL loops. `range` indicates the limits of the loop while `body` describes the task body to execute loop iterations
parallel_reduce<range, body>	Used to create parallel reductions. The class `body` specifies a `join()` method used to perform parallel reductions.
parallel_scan<range, body>	Used to compute a parallel prefix.
parallel_while<body>	Template for creating parallel tasks when the iteration range of a loop is not known
parallel_sort<iterator, compare>	Template for creating parallel sorting algorithms.

created them, making the creator the parent task. This hierarchical formation allows programmers to create complex task execution dependencies, making TBB a versatile dynamic parallelization library capable of supporting a wide variety of parallelism types. For example, TBB includes algorithms, highly concurrent containers, locks and atomic operations, a task scheduler, and a scalable memory allocator. These enable TBB to support not only simple task parallelism but also more complex data and pipeline parallelism.

Since manually creating and managing hierarchical dependencies for commonly found types of parallelism can quickly become a tedious chore, TBB provides a set of C++ templates that allow programmers to annotate common parallelism patterns such as DOALL and reductions. Table I provides a description of the class templates offered by TBB.

Regardless of how parallelism is annotated in applications (explicitly through `spawn()` or implicitly through the use of templates), all parallelism is exploited through parallel tasks. Conversely, even though the programmer might design tasks to execute in parallel, TBB does not guarantee that they will do so. If only one processor is available at the time, or if additional processors are busy completing some other task, newly-spawned tasks may execute sequentially. When processors are available, creating more tasks than available processors allows the TBB dynamic runtime library to better mitigate potential sources of load imbalance.

2.2. Dynamic Scheduling of Tasks

The TBB runtime library consists of a dynamic scheduler that stores and distributes available parallelism as needed in order to improve performance. While this dynamic management of parallelism is largely hidden from the programmer, its overhead can sometimes be detrimental to parallelism performance. To better understand the principal sources of overhead that we measure in Sections 5 and 6, this section describes the main scheduler loop of the TBB runtime library.

When the TBB runtime library is first initialized, a set of slave worker threads is created and the caller of the initialization function becomes the master worker thread. Worker thread creation is an expensive operation, but since it is performed only once during application startup, its cost is amortized over application execution.

When a worker thread is created, it is immediately associated with a software task queue. Tasks are explicitly enqueued into a task queue when their corresponding worker thread calls the `spawn()` method. Dequeueing tasks, however, is implicit and carried out by the runtime system.

This process is better explained by Figure 1, which shows the procedure `wait_for_all()`, the main scheduling loop of the TBB runtime library. This procedure consists of three nested loops that attempt to obtain work through three different means: explicit task passing, local task dequeue, and random task stealing.

The inner loop of the scheduler is responsible for executing the current task by calling the method `execute()`. After the method is executed, the reference count of the task's

```
1     wait_for_all(task *child) {
2        task = child;
3        Loop until root is alive
4           do
5              while task available
6                 next_task = task->execute();
7                 Decrease ref_count for parent of task
8                 if ref_count==0
9                     next_task = parent of task
10                task = next_task
11             task = get_task();
12          while (task);
13       task = steal_task(random());
14       if steal unsuccessful
15        Wait for a fixed amount of time
16        If waited for too long, wait for master thread
17        to produce new work
18    }
```

(Outer loop, Middle loop, Inner loop)

Fig. 1. Simplified TBB task scheduler loop. The scheduling loop is executed by all worker threads until the master thread signals their termination. The inner, middle, and outer loops of the scheduler attempt to obtain work through explicit task passing, local task dequeue, and random task stealing, respectively.

parent is atomically decreased. This reference count allows the parent task to unblock once its children tasks have completed. If this reference count reaches one, the parent task is set as the current task and the loop iterates. The method `execute()` has the option of returning a pointer to the task that should execute next (allowing explicit task passing).

If a new task is not returned, the inner loop exits and the middle loop attempts to extract a task pointer from the local task queue in FILO order by calling `get_task()`. If successful, the middle loop iterates, calling the most inner loop once more. If `get_task()` is unsuccessful, the middle loop ends and the outer loop attempts to steal a task from other possibly existing worker threads. If the steal is unsuccessful, the worker thread waits for a predetermined amount of time. If the outer loop iterates multiple times and stealing continues to be unsuccessful, the worker thread gives up and waits until the main thread wakes it by generating more tasks.

2.3. Task Stealing in TBB

Task stealing is the fundamental way by which TBB attempts to keep worker threads busy, maximizing concurrency and improving performance through reduction of load imbalance. If there are enough tasks to work with, worker threads that become idle can quickly grab work from other worker threads.

When a worker thread runs out of local work, it attempts to steal a task by first determining a victim thread. TBB utilizes random selection as its victim policy. Once the victim is selected, the victim's task queue is examined. If a task can be stolen, the task queue is locked and a pointer describing the task object is extracted, the queue is unlocked, and the stolen task is executed in accordance with Figure 1. If the victim queue is empty, stealing fails and the stealer thread backs off for a predetermined amount of time.

Random task stealing, while fast and easy to implement, may not always select the best victim to steal from. As core counts increase, the number of potential victims also increases, and the probability of selecting the best victim decreases. This is particularly true under severe cases of work imbalance, where a small number of worker threads may have more work than others. Moreover, with process variations threatening to transform homogeneous CMP designs into an heterogeneous array of cores [Humenay et al. 2007], effective task stealing becomes even more important. We will further study the performance of task stealing in Section 7.

```
int main( ) {
  tbb::scheduler_init init(numThreads);
  rootTask &root =
        *new ( allocate_root() ) rootTask;
  spawn_root_and_wait(root);
}


class childTask:: public tbb:task
{
   void execute( ) {
      /* Task body here */
   }
};
```

```
class rootTask: public tbb::task
{
  task * execute( ) {
  childTask &A = *new( allocate_child() )
                  childTask;
  childTask &B = *new( allocate_child() )
                  childTask;
   set_ref_count(3);
   spawn(A);
   spawn_root_and_wait(B);
   return NULL;
  }
};
```

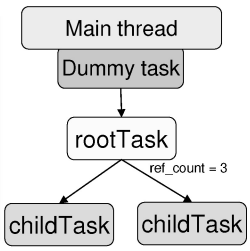

Fig. 2. TBB code example that creates a root task and two children tasks. In this example, the parent task (`rootTask`) blocks until its children terminate. **Bold** statements signify special methods provided by TBB that drive parallelism creation and behavior.

3. PROGRAMMING EXAMPLE

Figure 2 shows an example of how parallel tasks can be created and spawned in the TBB environment. The purpose of this example is to highlight typical steps in executing parallelized code. Sections 5 and 6 then characterize these overheads and show their impact on program performance.

For the given example, two parallel tasks are created by the root task (the parent task), which blocks until the two child tasks terminate. The `main()` function begins by initializing the TBB runtime library through the use of the `init()` method. This method takes the number of worker threads to create as an input argument. Alternatively, if the parameter `AUTOMATIC` is specified, the runtime library creates as many worker threads as available processors.

After initialization, a new instance of `rootTask` is created using an overloaded `new()` constructor. Since it is the main thread and not a task that is creating this task, `allocate_root()` is given as a parameter to `new()`, which attaches the newly-created task to a dummy task. Once the root task is created, the task is spawned using `spawn_root_and_wait()`, which spawns the task and calls the TBB scheduler (`wait_for_all()`) in a single call. Once the root task is scheduled for execution, `rootTask` creates two children tasks and sets its reference count to three (two children tasks plus itself). When the children tasks execute and then terminate, the reference count of the parent is decreased by one. When this count reaches one, the parent is scheduled for execution. The corresponding task hierarchy is shown to the right of Figure 2.

It is possible for `childTask()` to create additional parallel tasks in a recursive manner. As worker threads use task stealing to avoid becoming idle, child tasks start creating local tasks until the number of available tasks exceeds the number of available processors. At this point, worker threads dequeue tasks from their local queue until their contents are exhausted.

This simple example shows how parallel code can be created with little regard to the underlying machine characteristics (number of cores). While easy to use, the abstraction layer provided by the runtime library makes it difficult for programmers to assess the performance cost of exploiting available parallelism. In Section 5 we use real and simulated measurements of CMP systems to characterize the cost of basic TBB operations in order to better understand their contribution to overall parallelization overhead.

4. CHARACTERIZATION METHODOLOGY

4.1. Software Characteristics

We study the impact of the TBB runtime library on parallel applications by porting a subset of the PARSEC benchmark suite: `fluidanimate`, `swaptions`, `blackscholes`, and

Table II. Our Benchmark Suite Consists of a Subset of the PARSEC Benchmarks Parallelized Using TBB as Well as TBB Microbenchmarks. The Value N Represents the Number of Processors Being used

Benchmark	Description	Num. of tasks	Avg. cycles per task
`fluidanimate`	Fluid sim. for interactive animation	$420 \times N$	19M
`swaptions`	Heath-Jarrow-Morton framework to price portfolio of options	120,000	25K
`blackscholes`	Calculation of prices of a portfolio of European options	$1,200 \times N$ tasks	256K (@ 32 cores)
`streamcluster`	Online Clustering Problem	$11,000 + 6,000 \times N$	23M
Micro-benchmarks			
`Bitcounter`	Vector bit-counting with a highly unbalanced working set	5,740	5K
`Matmult`	Block matrix multiplication	12,224	6K
`LU`	LU dense-matrix reduction	31,200	4K
`Treeadd`	Tree-based recursive algorithm	12,290	Highly variant

`streamcluster`. These benchmarks are chosen because they provide task sizes in the hundreds of kiloinstructions while having a total task count that is sufficiently large to thoroughly exercise the TBB scheduler's load-balancing techniques. Out-of-the-box versions of these benchmarks are parallelized using a coarse-grain, static parallelization approach, where work is statically divided among N threads and synchronization directives (barriers) are placed where appropriate. We refer to this approach as *static*; it will serve as the base case when considering TBB performance.

In porting these benchmarks to the TBB environment, we use version 2.0 of the Intel Threading Building Blocks library [Intel Threading Building Blocks 2.0 Open Source] for the Linux OS. We use release tbb20_010oss, which at the start of our study was the most up-to-date commercial aligned release available (October 244, 2007). More recent releases address internal casting issues, the memory allocator, add new and improved parallel algorithm templates and data containers and makes modifications to internal task allocation and deallocation; these issues do not modify the outcome of our results.

We compile TBB using `gcc` 4.0, use the optimized `release` library, and configure it to utilize the recommended `scalable_allocator` rather than `malloc` for dynamic memory allocation. The memory allocator `scalable_allocator` offers higher performance in multithreaded environments and is included as part of TBB.

Porting benchmarks is accomplished by applying available parallelization templates whenever possible and/or by explicitly spawning parallel tasks. Since we want to take advantage of TBB's dynamic load-balancing, we aim at creating M parallel tasks in an N-core CMP system where $M \geq 4 \cdot N$. In other words, at least four parallel tasks are created for every utilized processor. In situations where this is not possible (eg. in DOALL loops with a small number of iterations), we further subpartition parallel tasks in order to create ample opportunity for load-balancing. An example of how a PARSEC benchmark is ported to the TBB environment is shown in Figure 3.

In addition to porting existing parallel applications to the TBB environment, we created a set of microbenchmarks with the purpose of stressing some of the basic TBB runtime procedures. Table II gives a description of the set of benchmarks utilized in this study.

4.2. Physical Performance Measurements

Real-system measurements are made on a system with two 1.8GHz AMD chips, each with dual cores, for a total of four processors. Cores includes 64KB of private L1 instruction cache, 64KB of private L1 data cache, and 1MB of L2 cache. Performance measurements are taken using *oprofile*, a system-wide performance profiler that uses processor performance counters to obtain detail performance information of running

```
int bs_thread(void *tid_ptr) {
 int tid = *(int *)tid_ptr;
 int start = tid * (numOptions / nThreads);
 int end = start + (numOptions / nThreads);
 BARRIER(barrier);

 for (j=0; j<NUM_RUNS; j++) {
    price[tid*LINESIZE] = 0;
    for (i=start; i<end; i++)
      price[tid*LINESIZE] +=
            BlkSchlsEqEuroNoDiv(...);
      BARRIER(barrier);
      if(tid==0) {
        acc_price = 0;
        for(i=0;i<nThreads;i++)
          acc_price += price[i*LINESIZE];
      }
  }
    return 0;
}
```

***pthread* version**

```
int bs_thread(void) {
 for (j=0; j<NUM_RUNS; j++) {
  mainWork doall;
  tbb::parallel_reduce(tbb::blocked_range<int>(0,
  numOptions, GRAIN_SIZE), doall);
  acc_price = doall.getPrice();
   }
return 0;
}

struct mainWork {
  fptype price;
public:
  void operator()(const tbb::blocked_range<int> &range)  {
   int begin = range.begin();
    int end = range.end();
    for (int i=begin; i!=end; i++)
       local_price += BlkSchlsEqEuroNoDiv(...);
    price +=local_price;

  void join(mainWork &rhs){price += rhs.getPrice();}
  fptype getPrice(){return price;}
};
```

TBB version

Fig. 3. This example shows how `blackscholes` is ported to the TBB environment. The original code consists of *pthread*ed code, where each thread executes the function `bs_thread()`. In TBB, `bs_thread()` is only executed by the main thread, and the template `parallel_reduce` is used to annotate DOALL parallelism within the function's main loop (in **bold** font). For clarity, not all variables and parallel regions are shown.

applications and libraries. We configure *oprofile* to sample the event `CPU_CLK_UNHALTED`, which counts the number of unhalted CPU cycles on each utilized processor.

4.3. Simulation Infrastructure

Since real-system measurements are limited in processor count, we augment them with simulation-based measurements. For our simulation-based studies, we use a cycle-accurate CMP simulator modeling a 1 to 32 core chip-multiprocessor similar to that used by Chen et al. [2005]. Each core models a 2-issue, in-order processor similar to the Intel XScale core [Intel Corporation 2003]. Cores have private 32KB L1 instruction and 32KB L1 data caches and share a distributed 4MB L2 cache. Since the L1 data caches are private, coherence must be maintained. For this purpose, we use an MSI directory-based protocol. Each core is connected to an interconnection network modeled as a mesh network with dimension-routing. Router throughput is one packet (32 bits) per cycle per port.

Our simulated processors are based on the ARM ISA. Our choice of the ARM ISA instead of IA32 was influenced by the slow execution speeds of existing IA32

simulation platforms. Since this study requires a detailed analysis of the various sources of overheads, we need to capture the full range of application behavior by running to completion. The slow speeds of IA32 timing simulators precludes this option; nevertheless, we do modify the ARM ISA to use atomic support (e.g. IA32's `LOCK`, `XADD`, `XCHG` instructions) equivalent to those in the IA32 architecture. This avoids penalizing TBB for its reliance on ISA support for atomicity and allows for a more direct comparison between our simulation and real-system results.

5. CHARACTERIZATION OF BASIC TBB FUNCTIONS

Dynamic management of parallelism requires the runtime library to store, schedule, and reassign parallel tasks. Since programmers must harness parallelism whose execution times are long enough to offset parallelization costs, understanding how runtime activities scale with increasing core counts allows us to identify potential overhead bottlenecks that may undermine parallelism performance in future CMPs.

In measuring some of the basic operations of the TBB runtime library, we focus on five common operations.

(1) `spawn()`. This method is invoked from user code to spawn a new task. It takes a pointer to a task object as a parameter and enqueues it in the task queue of the worker thread executing the method.
(2) `get_task()`. This method is called by the runtime library after completing the execution of a previous task. It attempts to dequeue a task descriptor from the local queue. It returns `NULL` if it is unsuccessful;
(3) `steal()`. This method is called by worker threads with empty task queues. It first selects a worker thread as the victim (at random), locks the victim's queue, and then attempts to extract a task class descriptor.
(4) `acquire_queue()`. This method is called by `get_task()` and `spawn()` in order to lock the task queue before a task pointer can be extracted. It uses atomic operations to guarantee mutual exclusion.
(5) `wait_for_all()`. This is the main scheduling loop of the TBB runtime library. It constantly executes and looks for new work to execute and is also responsible for executing parent tasks after all children are finished. We report this cost as the total time spent in this function minus the time reported by the procedures outlined in the preceding.

All of the procedures listed are directly or indirectly called by the scheduler loop shown in Figure 1, which is the heart of the TBB runtime library. They are selected based on their total execution time contribution as indicated by physical and simulated performance measurements.

5.1. Basic Operation Costs

Figure 4 shows measured and simulated execution costs of some of the basic functions performed within the TBB runtime library. We report the average cost per operation by dividing the total number of cycles spent executing a particular procedure by the total number of times the procedure is used. Physical measurements are used to show function costs at low core counts, while simulated measurements are used to study the behavior at higher core counts (up to 32 cores). Since our simulation infrastructure allows us to obtain detailed performance measurements, we divide `steal` into successful steals and unsuccessful steals. Successful steals are stealing attempts that successfully return stolen work, while unsuccessful steals are stealing attempts that fail to extract work from another worker thread due to an empty task queue.

Figure 4 shows results for two micro-benchmarks, `bitcounter`, and `treeadd`. The first one, `Bitcounter`, exploits DOALL parallelism through TBB's `parallel_for()` template.

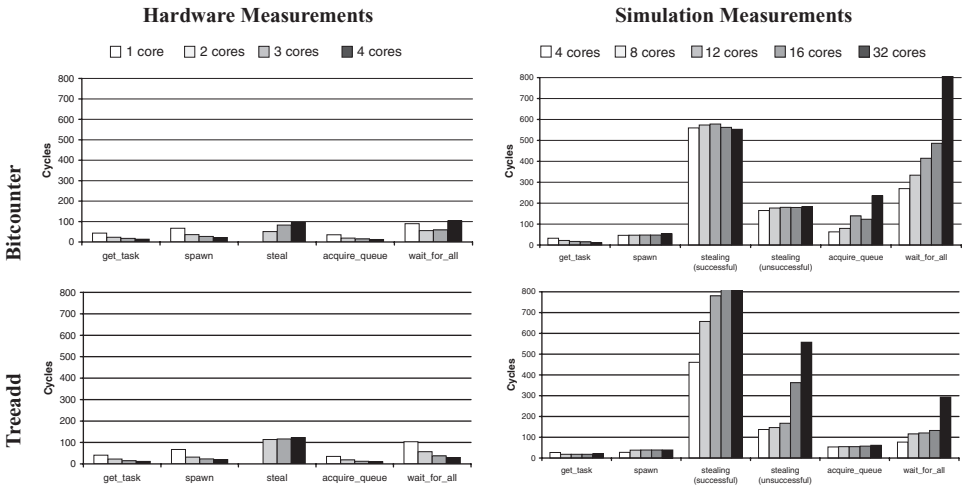

Fig. 4. Measured (hardware) and simulated costs for basic TBB runtime activities. The overhead of basic action such as `acquire_queue()` and `wait_for_all()` increases with increasing core counts above 4 cores for our simulated CMP.

Its working set is highly unbalanced, which makes the execution time of tasks highly variable. The microbenchmark `treeadd` is part of the TBB source distribution and makes use of recursive parallelism.

5.2. Hardware Measurements

On the left-hand side of Figure 4, real-system measurements show that at low core counts, the cost of some basic functions is relatively low. Functions such as `get_task()`, `spawn()`, and `acquire_queue()` remain relatively constant and even show a slight drop in average runtime with increasing number of cores. This is because as more worker threads are added, the number of function calls increases as well. However, because the cost of these functions depends on their outcomes, (`task_get()` and `steal()`, for example, have different costs depending on whether the call is successful or unsuccessful), the total cost of the function remains relatively constant, lowering its average cost per call.

Figure 4 also shows an important contrast in the stealing behavior of DOALL and recursive parallelism. In `bitcounter`, for example, worker threads rely more on stealing for obtaining work, allowing the average cost of stealing to increase slightly with increasing cores due to increasing stealing activity. For `treeadd`, where worker threads steal work once and then recursively create additional tasks, the cost of stealing remains relatively constant. `Treeadd` performs a small number of steals (less than 7,000 attempts), while `bitcounter` performs approximately 4 million attempts at 4 cores. Note that one-core results do not include stealing since all work is created and executed by the main thread.

5.3. Simulation Measurements

Similar to our physical measurements, simulated results show that functions such as `get_task()` and `spawn()` remain relatively constant, while the cost of other functions such as `acquire_queue()` and `wait_for_all()` increase with increasing cores. For `bitcounter`, the cost of `acquire_queue()` increases with increasing core counts, while for `treeadd` it remains relatively constant. Further analysis reveals that since task variables are more commonly shared among worker threads for `bitcounter`, the cost of

queue locking increases due to memory synchronization overheads. For `treeadd`, task accesses remain mostly local, avoiding cache coherence overheads.

The function `wait_for_all()` increases in cost for both studied microbenchmarks. `Treeadd` utilizes explicit task passing (see Section 2.2) to avoid calling the TBB scheduler, reducing its overall overhead. Nonetheless, for both of these benchmarks, atomically decreasing the parent's reference count creates memory coherence overheads that significantly contribute to its total cost. For `bitcounter`, memory coherence overheads account for 40% of the cost of `wait_for_all()`.

As previously noted, the two benchmarks studied in Figure 4 have different stealing behaviors, and thus different stealing costs. For `bitcounter`, the cost of a successful steal remains relatively constant at about 560 cycles per successful steal, while a failed steal attempt takes less than 200 cycles. By design, `bitcounter` sees greater increases in stealing opportunities at lower core counts; therefore from 4 to 32 cores, the stealing overheads remain roughly constant. On the other hand, the cost of a successful steal for `treeadd` increases with increasing cores, from 460 cycles at 4 cores to more than 1,100 cycles for a successful steal on a 32-core system. Despite this large overhead, the number of successful steals is small and has little impact on application performance.

While many of these overheads can be amortized by increasing task granularity, future CMP architectures will require applications to harness all available parallelism, which in many cases may present itself in the form of fine-grain parallelism. Previous work has shown that in order to efficiently parallelize sequential applications as well as future applications, support for task granularities in the range of hundreds to thousands of cycles is required [Ottoni et al. 2005; Kumar et al. 2007]. By supporting only coarse-grain parallelism, programmers may be discouraged from annotating readily available parallelism that fails to offset parallelism management costs, losing valuable performance potential.

6. TBB BENCHMARK PERFORMANCE

The previous section focused on a per-cost analysis of basic TBB operations. In this section, our goal is to study the impact of TBB overheads on overall application performance (the impact of these costs on parallelism performance). For this purpose, we first present TBB application performance (speedup) followed by a distilled overhead analysis via categorization of TBB overheads.

6.1. Benchmark Overview

Figure 5 shows simulation results for *static* versus TBB performance for 8 CMP configurations: 2, 4, 8, 9, 12, 16, 25, and 32 cores. While the use of 9, 12, or 25 cores is unconventional, it addresses possible scenarios where core scheduling decisions made by a high-level scheduler (such as the OS, for example) prevent the application from utilizing some round number of cores.

One of the most noticeable benefits of TBB is its ability to support greater performance portability across a wide range of core counts. In `swaptions`, for example, a static arrangement of parallelism fails to equally distribute available coarse-grain parallelism among available cores, causing severe load imbalance when executing on 9, 12, and 25 cores. This improved performance scalability is made possible thanks to the application's task programming approach, which allows for better load-balancing through the creation of more parallel tasks than available cores. This has prompted other parallel programming environments such as OpenMP 3.0 to include task programming model support.

While TBB is able to match or improve performance of *static* at low core counts, the performance gap between TBB and *static* increases with increasing core counts, as in the case with `swaptions`, `matmult`, and `LU`. This widening gap is caused by synchroniza-

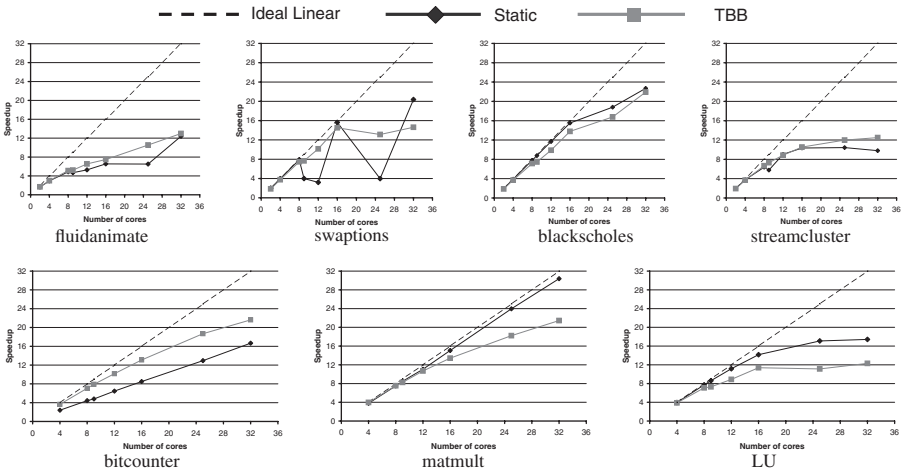

Fig. 5. Speedup results for four PARSEC benchmarks (top) and three microbenchmarks (bottom) using *static* versus TBB. TBB improves performance scalability by creating more tasks than available processors, however, it is prone to increasing synchronization overheads at high core counts.

Table III. TBB Overheads as Measured on a 4-Processor AMD System Using Medium and Large Datasets

Benchmark	Medium	Large
`fluidanimate`	2.6%	5%
`swpations`	2.4%	2.6%
`blackscholes`	14%	14.8%
`streamcluster`	11%	11%

tion overheads within `wait_for_all()` and a decrease in the effectiveness of random task stealing. To better identify sources of significant runtime library overheads, we categorize TBB overheads and study their impact on parallelism performance. The performance of random task stealing is studied in Section 7.

The performance impact of the TBB runtime library on our set of applications is confirmed by our hardware performance measurements. Table III shows the average percent time spent by each processor executing the TBB library as reported by `oprofile` for medium and large datasets. From the table it can be observed that the TBB library consumes a small, but significant amount of execution time. For example, `streamcluster` spends up to 11% executing TBB procedures. About 5% of this time is spent by worker threads waiting for work to be generated by the main thread, 4% is dedicated to task stealing, and about 3% to the task scheduler.

TBB's contribution at 4 cores is relatively low. However, it is more significant than at 2 cores. Such overhead dependency on core counts can cause applications to perform well at low core counts, but experience diminishing returns at higher core counts.

6.2. Categorization of TBB Overheads

Section 5 studied the average cost of basic TBB operations: `spawn()`, `get_task()`, `steal()`, `acquire_queue()`, and `wait_for_all()`. To better understand how the TBB runtime library influences overall parallelism performance, we categorize the time spent by these operations as well as the waiting activity of TBB (described in the following) during program execution into different overhead activities. However, since the net total execution time of task allocation, task spawn, and task dequeuing is less than

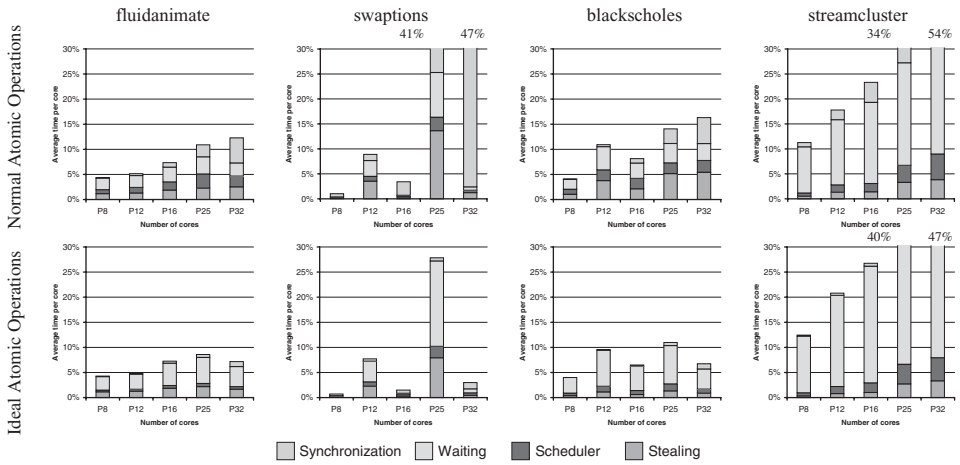

Fig. 6. Average contribution per core of the TBB runtime library on four PARSEC benchmarks. TBB contribution is broken down into four categories. The top row shows TBB contribution when latency of atomic operations is appropriately modeled. The bottom row shows TBB contributions when atomic operations are modeled as 1-cycle latency instructions.

0.5% on a 32 core system for our tested benchmarks, only the following four categories are considered.

—`Stealing`. This category captures the number of cycles spent determining the victim worker thread and attempting to steal a task (pointer extraction).
—`Scheduler`. This category included the time spent inside the `wait_for_all()` loop.
—`Synchronization`. This category captures the time spent in locking and atomic operations.
—`Waiting`. This category is not explicitly performed by parallel applications. Rather it is performed implicitly by TBB when waiting for a resource to be released or during the back-off period of unsuccessful stealing attempts.

Figure 6 plots the average contribution of the aforementioned categories. Two scenarios are shown: the top row considers the case where the latencies of all atomic operations are modeled, while the bottom row considers the case where the cost of performing atomic operations within the TBB runtime library is idealized (1-cycle execution latency). We consider the latter case since TBB employs atomic operations to guarantee exclusive access to variables accessed by all worker threads. Some of these variables include the reference count of parent tasks. As the number of worker threads is increased, atomic operations can become a significant source of performance degradation when a relatively large number of tasks are created. For example, in `swaptions`, synchronization overheads account for an average of 3% per core at 16 cores (achieving a 14.8X speedup) and grow to an average of 52% per core at 32 cores, limiting its performance to 14.5X. When atomic operations are made to happen with ideal single-cycle latency, this same benchmark achieves a 15X speedup at 16 cores and 28X at 32 cores. `Swaptions` is particularly prone to this overhead due to the relatively short duration of tasks being generated. This is typical, however, of the aggressive fine-grained applications we expect in the future. For our set of microbenchmarks, synchronization overheads degrade performance beyond 16 cores, as shown in Figure 5.

Excessive creation of parallelism can also degrade performance. For example, the benchmark `blackscholes` contains the procedure `CNDF`, which can be executed in parallel with other code. When we attempt to exploit this potential for concurrency, the

performance of `blackscholes` decreases from 19X to 10X. This slowdown is caused by the large quantities of tasks that are created (from 6K tasks on a simulated 8-core system to more than 6M tasks from parallelizing the `CNDF` procedure), quickly overwhelming the TBB runtime library as `scheduler` and `synchronization` costs overshadow performance gains.

Discouraging annotation of parallelism due to increasing runtime library overheads reduces programming efficiency as it forces extensive application profiling in order to find cost-effective parallelization strategies. Runtime libraries should be capable of monitoring parallelism efficiency and of suppressing cost-ineffective parallelism by executing it sequentially or under a limited number of cores. While the design of such runtime support is outside the scope of this article, the next section demonstrates how runtime knowledge of parallelism can be used to improve task stealing performance.

7. PERFORMANCE OF TASK STEALING

In this section, we take a closer look at the performance of task stealing. Task stealing is used by worker threads to avoid becoming idle by attempting to steal tasks from other worker threads. A number of past studies have shown that adequate and prompt stealing is necessary to reduce potential sources of imbalance. For example, Blumofe and Leiserson [1999] investigated the theoretical runtimes of various parallelism classes with work stealing, while other work has been conducted on mechanisms to improve the performance and scalability of task stealing [Acar et al. 2000; Dinan et al. 2009]. The overriding observation is that work stealing is required to mitigate load imbalance, particularly at barrier boundaries, since failure to promptly reschedule the critical path (the thread with the most amount of work) can lead to suboptimal performance.

To study the behavior of random task stealing in TBB, we monitor the following two metrics.

—*Success rate* The ratio of successful steals to the number of attempted steals.
—*False negatives* The ratio of unsuccessful steals and steal attempts given that a worker in the system had at least one stealable task.

With these metrics, we proceed to quantify the performance of random task stealing on both the microbenchmarks and TBB-ported PARSEC programs. After showing the inherent limitations of random task stealing, we then focus on two alternatives—a purely software approach called *occupancy-based* task stealing and a hardware-aided approach called *criticality-guided* task stealing.

7.1. Initial Results

Figure 7 plots the success and false negatives rates for the microbenchmarks across a number of core counts. As noted, random stealing suffers from performance degradation (decreasing success rate) as the number of cores is increased. This variability in performance is more noticeable in microbenchmarks that exhibit inherent imbalance (`bitcounter` and `LU`), where the drop in the success rate is followed by an increase in the number of false negatives as the number of cores increases.

Similarly, Figure 8 shows the performance of random task stealing on the TBB-ported PARSEC benchmarks. As with the microbenchmarks, random task stealing becomes notably less effective at higher core counts. This occurs despite the fact that load imbalance typically increases with more cores, leading to more potential steal possibilities. However, the random nature of the task stealer is unable to exploit these additional tasks. Even at lower core counts, random task stealing performs poorly for `Streamcluster`. This is because a few threads operate on longer tasks here and random stealing does not successfully steal these as victims.

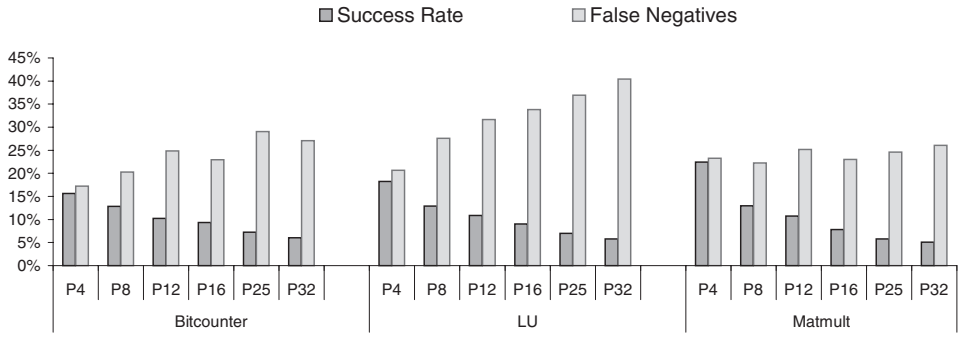

Fig. 7. Stealing behavior for three microbenchmarks. For benchmarks with significant load imbalance such as `bitcounter` and LU, random task stealing loses accuracy as the number of worker threads is increased, increasing the amount of false negatives and decreasing stealing success rate.

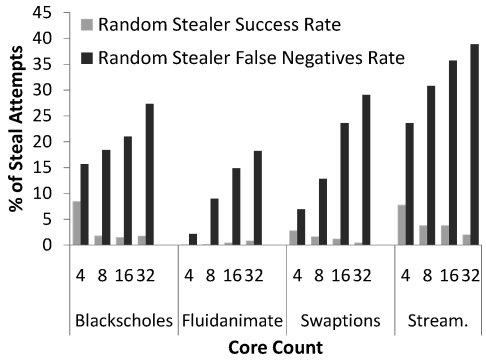

Fig. 8. Random stealing prompts a large false negatives contribution, leading to poor performance.

Overall, our results show that random victim selection, while effective at low core counts, provides suboptimal performance at high core counts by becoming less accurate, particularly in scenarios where there exists significant load imbalance.

7.2. Improving Task Stealing with Occupancy-Based Approach

We now consider mechanisms to improve stealing performance by focusing on alternative occupany-centric victim selection policies. The first is an occupancy-based selection policy, in which a victim thread is selected based on the current occupancy level of the queue. For this purpose, we have extended the TBB task queues to store their current task occupancy, increasing its value on a `spawn()` and decreasing it on a successful `get_task()` or steal.

Our occupancy-based stealer requires all queues to be probed in order to determine the victim thread with the most work (highest occupancy). This is a time consuming process for a large number of worker threads. Therefore, we also develop a variant on the pure occupany-based approach, the group-based approach. In this, stealer mitigates the temporal overheads of pure occupancy-based stealing by forming groups of cores of at most 5 worker threads. When a worker thread attempts to steal, it searches for the worker thread with the highest occupancy within its own group. If all queues in the group are empty, the stealer selects a group at random and performs a second scan. If it is still unsuccessful, the stealer gives up, waits for a predetermined amount of time, and then tries again. Note that while we choose at most 5 worker threads for our group,

Table IV. Microbenchmark Performance Improvements Over Default Random Task Stealing When Using Occupancy-Based Victim Selector, and Group-Occupancy-Based Victim Selector. Our Stealing Policies Improve Performance by Nearly 20% Over Random Stealing and Come Close to Ideal Bounds. We Expect Larger Improvements for Larger Core Counts

Benchmark	Our Approach					
	Occupancy			Group-Occupancy		
	P16	P25	P32	P16	P25	P32
`bitcntr`	2.5%	2.5%	3.7%	2.3%	3.5%	4.2%
`LU`	10%	4.1%	9.7%	9.9%	4.3%	8.3%
`matmult`	9.5%	6%	19%	8.2%	5.3%	17.8%

Table V. Microbenchmark Performance Improvements Over Default Random Task Stealing When Using an Ideal Occupancy-Based Victim Selector, and an Ideal Occupancy-Based Victim Selector with Ideal Task Extraction

Benchmark	Ideal					
	Ideal Occupancy			Ideal Stealer		
	P16	P25	P32	P16	P25	P32
`bitcntr`	2.41%	2.8%	3.7%	4.7%	6.9%	7.8%
`LU`	10.2%	4.6%	8.0%	16.0%	10.4%	20.6%
`matmult`	9.8%	7.0%	21.1%	10.8%	9.8%	28.7%

our scheme does not restrict us to this quantity. The choice of the number is based on a trade-off between the time taken to do a scan through all the queues of a group and the success rate of steals. By maintaining a relatively low number of threads in a group, as in our experiments, we test the performance benefits that this scheme could provide while keeping intercore communication for queue-checks on the lower side. While our results provide a first analysis of the benefits of this approach, future avenues may involve varying the maximum number of threads per group.

Table IV shows the performance gain of our occupancy-based and group-based selection policies for 16, 25, and 32 core systems. Table V additionally shows two scenarios: *ideal occupancy*, and *ideal stealer*. *Ideal occupancy*, similarly to our occupancy-based stealer, selects the worker thread with the highest queue occupancy as the victim, however, the execution latency of this selection policy is less than 10 cycles.[1] Our *ideal stealer* is the same as ideal occupancy stealer, but also performs actual task extraction in less than 10 cycles (as opposed to the hundreds of cycles reported in Section 5).

Overall,Table IV shows that occupancy-based and group-based victim selection policies achieve better performance on the microbenchmarks than a random selection policy. As Table V shows, when the latency of victim selection is idealized (ideal occupancy), the performance marginally improves. However, when both selection and extraction of work is idealized, speedup improvements of up to 28% can be seen (`matmult`), suggesting that most of the overhead in stealing comes from instruction and locking overheads associated with task extraction.

We have also studied the performance improvements of occupancy-based stealing (with nonidealities included) for the TBB-ported PARSEC workloads. Figure 9 shows the performance improvements across a number of core counts. As with the microbenchmarks, occupancy-based stealing provides performance improvements over random stealing, particularly for load-imbalanced `Streamcluster`. Moreover, this improvement is magnified at greater core counts, where random stealing becomes less effective. On average, we see that occupancy-based stealing provides a 13% improvement against random stealing with 32 cores.

[1]This latency is imposed by our CMP simulator.

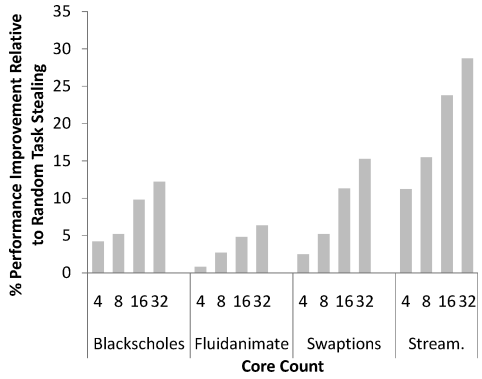

Fig. 9. Occupancy-based stealing greatly improves the performance of TBB-ported PARSEC benchmarks with an average of 13% against random task stealing at 32 cores.

7.3. Improving Task Stealing with Criticality-Guided Approach

While occupancy-based task stealing successfully selects steal victims with tasks present, it has no understanding of the relative complexities and lengths of tasks. Since different tasks could take varying amounts of time to complete, this means that occupancy-based stealing may not necessarily steal a task from the slowest or most critical thread. Since stealing from the critical thread however, holds greater performance improvement potential, we now focus on criticality-guided task stealing. We begin by introducing how to create thread criticality predictors and then investigate their application to task stealing.

7.3.1. Thread Criticality Prediction. Predicting thread criticality is a fundamental research problem for parallel programs. If a system can accurately gauge the critical or slowest threads of a parallel program, this information can be used for load rebalancing, or stealing work from the critical thread for performance improvements.

While criticality in the context of instructions has been explored in the past in detail [Fields et al. 2001; Tune et al. 2001], research on thread criticality has been more recent. Bhattacharjee and Martonosi [2009] were the first to propose metrics for thread criticality and use these to implement simple criticality predictors. This research indicates that differences in thread speeds in a parallel program can be primarily attributed to memory hierarchy statistics. Specifically, poorly cached threads tend to be slower and hence critical in computation. We use this insight to develop a criticality-guided TBB task stealer.

7.3.2. Integrating Thread Criticality with TBB. Figure 10 details the hardware changes required to accommodate thread criticality prediction, similar to hardware presented in Bhattacharjee and Martonosi [2009]. The thread criticality predictor is located at the shared, unified L2 cache where all cache miss information is centrally available. Our proposed hardware includes *Criticality Counters*, which count L1 and L2 cache misses resulting from each core's references. As cache misses define thread progress, these counters track thread criticalities, with larger ones indicative of slower, poorly cached threads. Since individual L1 cache misses contribute less to thread stall times and criticality than individual L2 misses and beyond, we propose a weighted combination of L1 instructions, L1 data, and L2 cache misses, and others when needed. Currently, our weighted criticality counter values may be expressed by:

$$N(Crit.Count.) = N(L1_{miss}) + \frac{(L1L2_{penalty}) \times N(L1L2_{miss})}{L1_{penalty}}. \quad (1)$$

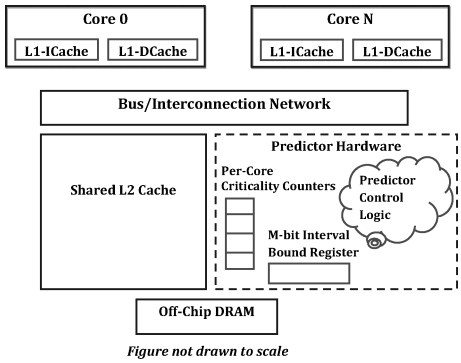

Fig. 10. High-level predictor design with per-core *Criticality Counters* placed with shared L2 cache, *Interval Bound Register*, and *Prediction Control Logic*. Hardware units are not drawn to scale.

```
if  (cache miss from Core P)  {
    Update criticality counter for Core P based on cache miss type
}

if  (Steal request from a Core)    {
    Scan all criticality counters to find the maximum value
    Report core with highest criticality counter value as steal victim
}

if  (Message indicating steal from victim Core P unsuccessful)  {
    Reset criticality counter for Core P
}

if  (   (Number of Cycles % Interval Bound)  ==  0)
    Reset all criticality counters
```

Fig. 11. Criticality-guided task stealing algorithm improves victim selection by choosing the core with the largest *Criticality Counter* value as the steal victim.

In this equation, $N(Crit.Count.)$ represents the value of the criticality counter, while $N(L1_{miss})$ and $N(L1L2_{miss})$ are equal to the number of L1 misses that hit in the L2 cache and the L1 misses that also miss in the L2 cache. Thus, since L2 misses incur a larger penalty, their weight is proportionately higher.

The counters are controlled by light-weight *Predictor Control Logic*, which is in charge of handling task stealing requests from TBB. An *Interval Bound Register*, which is incremented on every clock cycle, ensures that criticality predictions are based on relatively recent application behavior. This is accomplished by resetting all *Criticality Counters* whenever the *Interval Bound Register* reaches a predefined threshold. A threshold of 100K provides accurate readings of thread criticality [Bhattacharjee and Martonosi 2009].

With this hardware, Figure 11 details the predictor algorithm applied to task stealing. Cache misses are recorded by the *Criticality Counters*. When the TBB scheduler informs the predictor of a steal attempt, the predictor's control logic scans its criticality counters for the maximum value and replies to the stealer core that this maximum counter's corresponding core number should be the steal victim. If the steal is unsuccessful, the stealer sends a message to the predictor to reset the victim counter, minimizing further incorrect victim prediction. As before, the Criticality Counters are reset every Interval so that stealing decisions are based on recent application behavior.

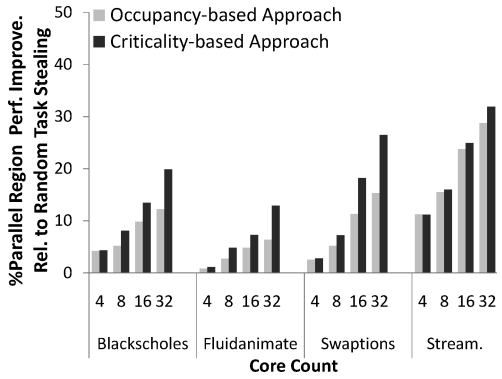

Fig. 12. Criticality-guided task stealing algorithm yields up to 32% performance improvements against random task stealing and regularly outperforms occupancy-based stealing.

Clearly, with this approach, stealing now occurs from the thread predicted to be the slowest or the most critical.

Note that unlike the random or occupancy stealing approaches, criticality-guided stealing relies on the addition of hardware. Therefore, we must account for the latency overhead of accessing this hardware. Since the predictor is placed next to the L2 cache, we assume that a predictor access imposes an additional delay equivalent to an L2 latency. This is in contrast to the random stealer, or occupancy-based stealer which are not charged this delay.

7.3.3. Results. Figure 12 plots the performance improvements of criticality-guided task stealing (parallel region of the benchmarks only) against TBB's random task stealing in the dark bars. Furthermore, we also plot the performance improvements of the occupancy-based approach, again versus random stealing in the light bars. As shown, criticality-guided stealing offers significant performance improvements over random stealing, particularly at higher core counts (average of 21.6% at 32 cores). Moreover, we also improve upon the occupancy-based stealing results. This is because occupancy-based techniques only count the number of tasks in each queue, but do not gauge their relative complexity or expected execution time. In contrast, criticality-based approaches can better account for the relative work in each task, by tracking cache miss behavior. This generally improves performance, especially at higher core counts (13.8% at 32 cores). The only exception is `Streamcluster` where a few threads hold a large number of stealable tasks of similar duration. In such a scenario, thread criticality yields little benefit over simple occupancy statistics. Generally however, the performance benefits of our approach are evident.

7.4. Summary of Observations

With future CMP systems running multiple parallel applications and sharing CPU and memory resources, future runtime libraries will require dynamic approaches that are able to scale with increasing core counts while maximizing performance. We have shown how current random stealing approaches provide suboptimal performance as the probability of selecting the best victim decreases with increasing core counts. Occupancy-based and criticality-guided policies are able to better identify the critical path and reassign parallelism to idle worker threads.

Table VI. The OpenMP API Provides Programmers with a Set of Language Pragmas Used to Annotate Parallelism in Applications. This Table Describes a Few of the Most Commonlyused Pragmas

Pragma	Description
`omp parallel`	Annotates code that should be executed by all available worker threads.
`omp single`	The body of this pragma is executed by only one thread in the team.
`omp for`	Used inside `omp parallel`, this pragma distributes the iterations of a DOALL loop across available worker threads.
`omp parallel for`	Combination of `omp parallel` and `omp for`.
`omp critical`	Forces the body of this pragma to be executed sequentially by worker threads.
`omp barrier`	Establishes a synchronization barrier across threads executing the same parallel region.

8. THE OPENMP RUNTIME LIBRARY

The OpenMP API was proposed by the OpenMP Architecture Review Board (ARB) for creating multiplatform shared-memory parallel applications. Version 1.0, released in 1997, offered OpenMP support for the FORTRAN language. Version 2.0 for the FORTRAN language was released in 2000 and in 2002 it was extended to the C/C++ programming language. The most recent revision of the OpenMP Specification, Version 3.0, adds support for parallel tasks, or encapsulated bodies of executable code with their own data environment that are dynamically assigned to worker threads, as well as better support for nested parallelism [OpenMP 2007]. When we undertook this research, no widely-available compiler support for the OpenMP 3.0 specification existed. Thus, for this chapter, and for the rest of this work, we will focus on an OpenMP 2.0 compliant runtime library. However, the methodology of our study, as well as our proposed solution in upcoming chapters, are not dependent on a specific environment and can be readily applied to upcoming software-based runtime libraries.

In OpenMP applications, programmers extract parallelism by using a series of *pragmas* that annotate specific types of parallelism. Pragmas provide programmers with a clean way of creating N-way parallelism in their applications, as it hides much of the underlying complexity of directly using threading packages such as POSIX *pthreads*. For example, one of the most important pragmas is the `omp parallel` pragma. This pragma annotates regions of code that are executed by all worker threads in parallel. With each worker thread having a unique ID, the programmer can then assign a unique subset of the problem to each worker thread. While the same mechanism can be achieved by directly using the *pthread* package, OpenMP offers a way of creating parallel applications without tying the application to a specific threading substrate.

OpenMP offers pragmas for the most basic types of parallelism operations. Critical sections within parallel regions are annotated using `omp critical`, which forces a code region to be executed sequentially by worker threads. Similarly, the pragma `omp single` can be used to force a block of code to be executed by only one worker, and synchronization among worker threads can be imposed by using the pragma `omp barrier`. Table VI describes some of the most common OpenMP pragmas.

8.1. Scheduling of Parallelism

In addition to parallel sections, many applications contain loops in which every iteration is independent of every other. For this type of parallelism, commonly referred as DOALL parallelism, OpenMP offers the pragma `omp for` for easy annotation. Used inside `omp parallel` code blocks, `omp for` allows participating worker threads to divide available loop iterations using one of three different loop scheduling policies.

```
1  #pragma omp parallel private( it,i,j,k )
2  {
3    for (it = 1; it <= NITER; it++) {
4      conj_grad(colidx, rowstr, x, z, a, p, q, r, w, &rnorm);   (1)
5      #pragma omp single
6      {
7        norm_temp11 = 0.0;   (2)
8        norm_temp12 = 0.0;
9      }
10     #pragma omp for schedule( dynamic ) reduction( +:norm_temp11,norm_temp12 )
11     for (j = 1; j <= lastcol-firstcol+1; j++) {
12       norm_temp11 = norm_temp11 + x[j]*z[j];   (3)
13       norm_temp12 = norm_temp12 + z[j]*z[j];
14     }
15   /* Additional code here */
16   }
17 }
```

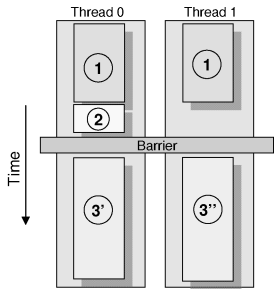

Fig. 13. (Left) OpenMP code example. Statements in **bold** are OpenMP pragma statements used to annotate parallelism. **(Right)** Diagram illustrating two OpenMP threads executing the code shown to the left. The pragma `omp parallel` configures both worker threads to execute the same code body. Code region (2) is executed only as single thread (thread 0 in our example) while all other thread, wait. The pragma `omp for` divides the iterations of a DOALL loop across available worker threads.

—*Static.* For this schedule, each worker thread is assigned an equal number of iterations (if possible). Iteration assignment occurs only once and it is not allowed to change at runtime. While this schedule offers the lowest management cost, it is particularly susceptible to load imbalance.
—*Dynamic.* This schedule attempts to dynamically assign parallel iterations to worker threads. Rather than assigning iteration counts statically, threads grab chunks of iterations as soon as they become idle. This approach reduces load imbalance, but suffers from locking contention as the number of worker threads is increased.
—*Guided.* This schedule works by progressively partitioning the iteration space across worker threads in an attempt to better load-balance computation. It attempts to reduce locking contention by initially distributing a large number of iterations followed by a smaller number in order to eliminate any remaining imbalance.

To better illustrate how programmers create parallel applications using the OpenMP API, the next section provides a short programming example.

8.2. OpenMP Programming Example

Figure 13 shows an example of how applications use OpenMP pragmas to annotate parallelism. The left side of the figure shows a code fragment from `CG`, while the right side of the figure illustrates how parallelism is executed by two worker threads.

The example starts by using an `omp parallel` pragma in line 1, which implies that all available worker threads will concurrently execute the body of the pragma (lines 2–17). Inside the parallel region we find a loop (line 3) as well as the procedure, `conj_grad()` in line 4. Since this code is being executed by all threads, the right side of the figure shows `conj_grad()` as being executed by both threads. In line 5, the pragma `omp simple` is used to annotate code that should only be executed by a single thread. The right side of the figure depicts the execution behavior of this pragma by showing only the master thread executing the code region labeled (2) while the other thread waits for its completion through an implied barrier at the end. In line 10, the pragma `omp for` annotates a loop containing DOALL parallelism and uses a `dynamic` schedule, which allows worker threads to dynamically grab loop iterations as needed. The figure on the right illustrates this behavior by showing the master thread executing (3') and the slave thread executing (3").

As with many other runtime libraries, programmers have little information regarding the actual management cost of parallelism. There seems to be, however, the general idea that dynamic management of parallelism is more expensive than static arrangements, and that coarse-grain parallelism is preferred over fine-grain parallelism in

Table VII. OpenMP NAS benchmarks used to characterize the Omni OpenMP runtime library.

		OpenMP Pragmas			
Benchmark	Description	`for`	`critical`	`single`	`barrier`
`LU`	LU dense matrix decomposition.	29	2	2	3
`MG`	Multigrid method to compute the solution of a 3D scalar Poisson equation.	11	3	10	11
`CG`	Conjugate gradient method to compute an approximation of an unstructured sparse matrix.	22	2	12	9
`SP`	Simulated computational fluid dynamics.	63	1	0	2

order to hide runtime library overheads. This may discourage programmers from using dynamic management of parallelism, or from annotating parallelism that does not seem sufficiently large to be cost-effective. Both of these approaches are contradictory to the needs of future CMP systems. It is of prime importance, then, to understand the major sources of overheads of existing runtime libraries and improve their performance through alternative, more cost-effective dynamic approaches.

The following section characterizes the Omni runtime library, an OpenMP-compliant runtime library, identifying some of its most significant sources of overheads. It also proposes an alternative loop-scheduling mechanism, which offers static-like costs while providing dynamic loop management capabilities.

8.3. Methodology

For the study of OpenMP, we use the Omni 1.6 infrastructure [The Omni OpenMP Compiler Project 2007], which consists of a C-language front-end compiler and a runtime library. The front-end compiler is responsible for reading C files annotated with OpenMP pragmas and creating an intermediate C file containing calls to the Omni runtime library. The runtime library is a high-performance [Kusano et al. 2000] open-source runtime library for the Linux OS that implements the OpenMP 2.0 specifications.

Our benchmarks consist of a subset of the OpenMP NAS benchmarks [Jin et al. 1999] described in Table VII. We use dataset `W` and compile the benchmarks using the Omni compiler front-end and `gcc` 3.0 as the back-end compiler with optimization flags `-O3`.

9. OPENMP RESULTS

9.1. Characterization of the Omni Runtime Library

Using a similar approach to that used in the characterization of the TBB runtime library in Section 5, we classify the time spent by each core during the execution of an OpenMP benchmark into four different categories.

(1) *Scheduler.* This category captures the time spent determining the next loop iteration to execute.
(2) *Barrier.* Captures the time spent executing implicit or explicit barriers.
(3) *Lock.* This category includes the time spent trying to obtain a lock to a shared resources within the runtime library.
(4) *Support.* Accounts for the time spent executing other runtime functionality, such as determining the number of active worker threads and/or obtaining local data structures.

Figure 14 shows the average time spent by cores executing these categories when executing `CG` and `MG` OpenMP benchmarks under static and dynamic schedules. The figure shows that for both static and dynamic schedules, the overall contribution of the runtime library increases with increasing core counts. For static, barrier costs from

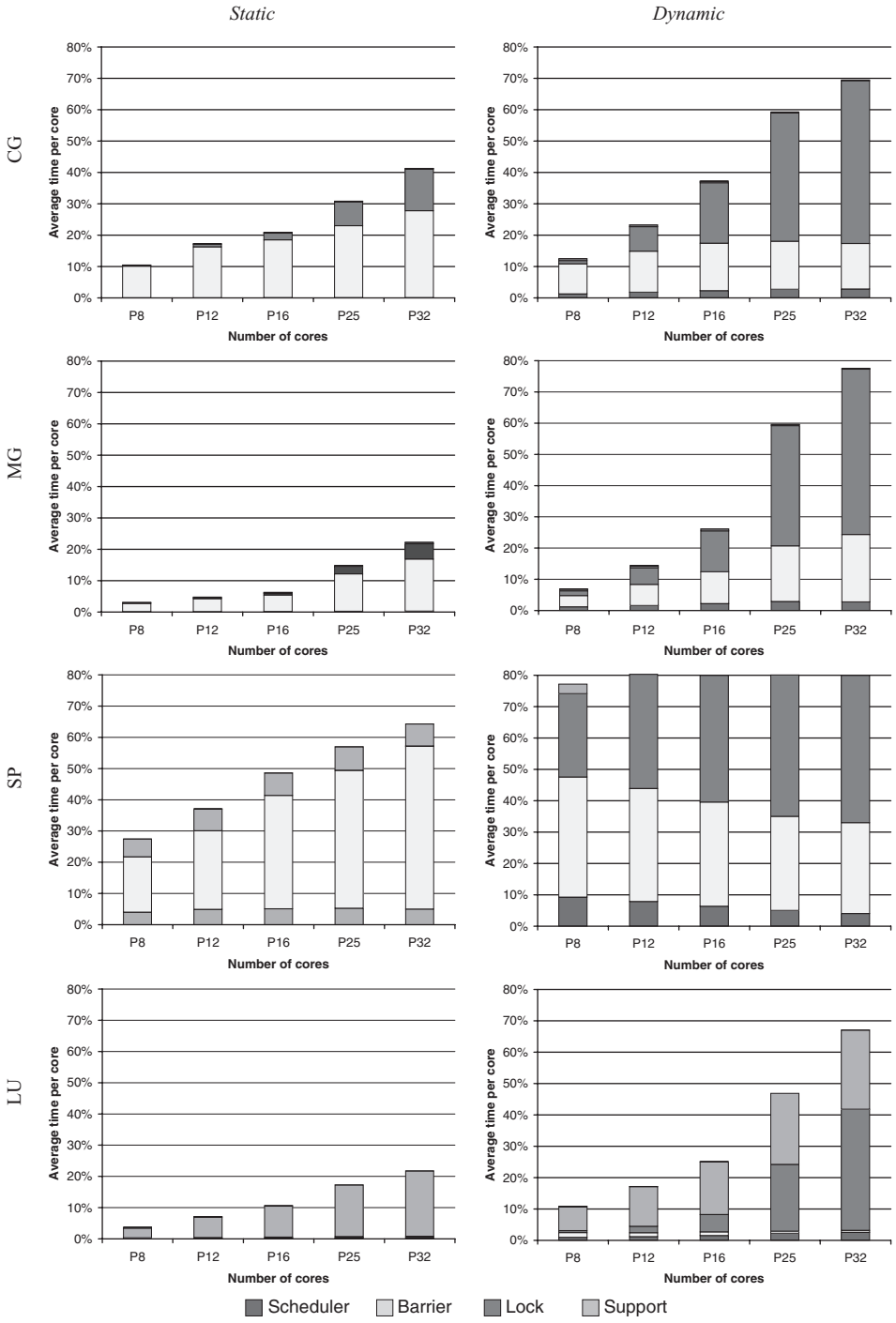

Fig. 14. Average time spent per core executing OpenMP activities. For dynamic loop scheduling, high lock contention decreases performance as the number of cores increases.

implicit synchronization at the end of parallel loops account for a significant portion of the runtime execution. For *dynamic*, lock contention becomes a major bottleneck with increasing core counts. This is because in the Omni runtime library, threads are required to lock the centralized location where the remaining iterations are being stored. Thus, as more worker threads attempt to grab work, synchronization overheads become significant, eventually overshadowing any parallelism performance gains.

Increasing synchronization overheads decreases the performance potential of applications, even when the annotated parallelism in the applications is able to scale well. For example, for the benchmark `CG`, dynamic loop scheduling achieves a speedup of only 1.6X on a 32-core system. When artificially treating atomic operations as 1-cycle latency instructions, the performance of `CG` increases to 15X. Similarly, for `MG`, which achieves a speedup of 1.9X on a 32 core CMP, 1-cycle latency instructions causes it to achieve a speedup of 20X.

It is possible to reduce lock contention by increasing the number of iterations that worker threads are allowed to grab (increasing chunk size). However, as worker threads grab a larger number of iterations, there remains little opportunity for scalability as there might not be sufficient number of iterations to keep all cores busy. Given the abstraction layer provided by runtime libraries, it is difficult for programmers to assess these trade-offs without extensive profiling.

9.2. Improving the Performance of OpenMP

The previous section demonstrated that locking contention within the Omni runtime library can significantly degrade the performance of dynamic loop scheduling. As a way of improving parallelism management, we propose a scheduling mechanism that offers management cost very close to that of static scheduling while offering the benefits of a dynamic schedule. For this, we have extended the *Omni* runtime system to support iteration stealing. Iteration stealing is implemented on the *Omni* runtime system by first dividing the iteration space equally across active worker threads—much like static scheduling—but rather than working on the entire assigned iteration space at once, threads extract iteration chunks from their local assignment. Extracting iteration chunks from their local assignment requires local locking, avoiding global lock contention and improving application scalability. When a worker runs out of local iterations, it attempts to steal from other threads using a highest-occupancy selection policy.

Our OpenMP occupancy-based stealing policy is similar to that used in the TBB runtime library, except that instead of using the number of task pointers stored in the container, it uses the number of pending iterations as an indication of occupancy. Hence, when worker threads consume their assigned iterations, they select a victim thread by scanning the occupancy of other worker threads and selecting the one with the highest occupancy (number of unfinished iterations). To further improve the performance of this policy, assembly code is used in critical sections to reduce instruction overheads.

Figure 15 shows the performance of `CG` and `MG` when using static, dynamic, and stealing schedules. As shown, iteration stealing offers significant performance improvements over a dynamic loop schedule. Despite significant performance improvements, however, the runtime library continues to exhibit significant overhead for large core counts. Barrier completion overhead continues to be a significant source of performance degradation. Furthermore, iteration stealing introduces its own cost as well. Scanning available worker threads for potential work consumes an average of 3% of the execution time of worker threads. Locking overhead, while less significant, continues to be expensive at large core counts as stealing threads are required to obtain exclusive access to the victim's data structures during stealing.

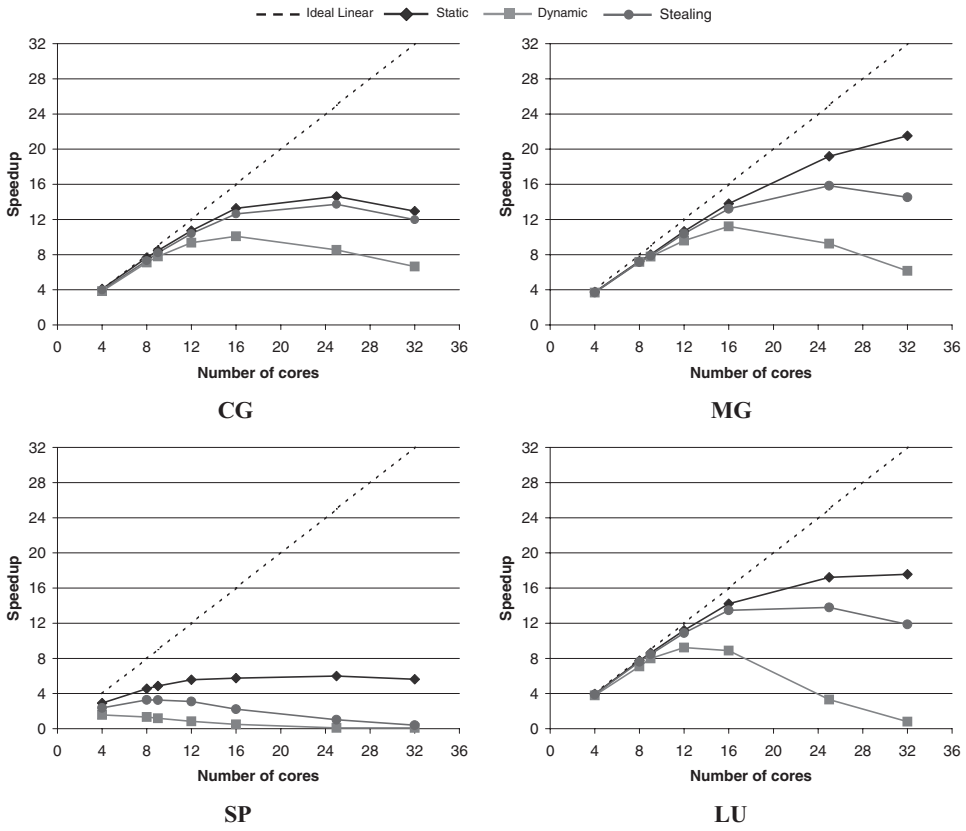

Fig. 15. Iteration stealing is able to offer higher performance by replacing global locking of variables by local locking. However, for large core counts, synchronization costs from application barriers continue to degrade performance.

The implementation of policies such as occupancy-based stealing shows that it is possible to create parallelism redistribution approaches that more adequately adapt to system behavior. Unfortunately, the implementation of these policies through software-based approaches causes them to be too heavyweight as overheads begin to dominate at high core counts, quickly overshadowing any benefits they provide.

Support such as container access, container locking, and assignment of parallelism to available processors are basic operations carried by runtime libraries that inherently contribute to overall parallelization overhead. Any solution that aims at reducing the overheads of these basic operations is likely to provide greater support for fine-grain parallelism and faster parallelism redistribution among execution resources.

10. GENERAL RECOMMENDATIONS

Based on our characterization results and experience with the TBB and OpenMP runtime libraries, we offer the following recommendations for programmers and runtime library developers:

—*For programmers.* At low core counts (2 to 8 cores), most of the overhead comes from the use of various library runtime procedures. For example, for TBB, creating a relatively small number of tasks might be sufficient to keep all processors busy with sufficient opportunity for load balancing. At higher core counts, synchronization

overheads start becoming significant. Excessive task creation can induce significant overhead if task granularity is not sufficiently big (approximately 100K cycles). In either case, using explicit task passing (see Section 2.2) is recommended to decrease some of these overheads.

—*For TBB developers.* While it might be difficult to reduce synchronization overheads caused by atomic operations within the TBB runtime library (unless specialized synchronization hardware becomes available [Sampson et al. 2006]), offering alternative task stealing policies that consider the current state of the runtime library (queue occupancy, for example) can offer higher parallelism performance at high core counts. Moreover, knowledge of existing parallelism can help drive future creation of concurrency. For example, when too many tasks are being created, the runtime library might be able to throttle the creation of additional tasks. In addition, while not highly applicable to our tested benchmarks, we noted an increase in simulated memory traffic caused by the random assignment of tasks to available processors. An initial deterministic assignment of tasks followed by stealing for load-balancing might help maintain data locality of tasks.

—*For OpenMP developers.* The primary overheads associated with the OpenMP scheduler revolve around synchronization primitives. For example, barrier overheads become particularly severe at higher core counts, while locking overheads, though less severe, continue to be expensive at large core counts. While the implementation of policies like iteration stealing can possible redistribute parallelism to better adapt to system behavior, overheads are likely to remain dominant at high core counts. Therefore it is key to devise new mechanisms to address overheads associated with container access, container locking, and parallelism assignment. Any progress on these fronts would provide greater support for fine-grain parallelism and faster redistribution among execution resources.

11. RELATED WORK

CMPs demand parallelism from existing and future software applications in order to make effective use of available execution resources. The extraction of concurrency from applications is not new however. Multiprocessor systems previously influenced the creation of software runtime libraries and parallel languages in order to efficiently make use of available processors. Classical UNIX semantics (`fork()`, posix threads) are widely used though they are error-prone and may be unsuitable for the management of fine-grained parallelism. Alternate approaches like Hood [Blumofe and Dionisios 1999] or Continuations [Hieb and Dybvig 1990] aid in hiding low-level primitives while allowing the programmer to handle concurrency access.

At the same time, parallel languages such as Linda [Gelernter 1985], Orca, Emerald and Cilk [Blumofe et al. 1996], among many others, were designed with the purpose of extracting course-grain parallelism from applications. Some of these languages have been further refined to aid programmers; for example, Cilk++ builds upon Cilk by additionally providing new constructs to solve data race problems [Leiserson 2009]. Nevertheless, the coarse-grained parallel nature of these languages remains. Furthermore, many of these paradigms, including Orca and MGS [Giavitto and Michel 2001], are domain-specific. In response, intentional programming, generative programming, and language-oriented programming [Ward 1994] allow for domain-specific programming but also permit the generation of standard code automatically. Additional approaches exist in the form of X10 [Charles et al. 2005] and Fortress [Allen et al. 2006], where implicit transactions and weak atomicity have been used to express concurrency constraints. Moreover, speculative synchronization [Martinez and Torrellas 2002] has been proposed to overcome expensive lock checking through rollback mechanisms.

Despite these benefits however, libraries that extended sequential languages for parallelism extraction, such as Charm++ [Kale and Krishnan 1993], STAPL, and OpenMP [OpenMP 2002] have become valuable tools, as they allow programmers to create parallel applications in an efficient and portable way. Many of these tools and techniques can be directly applied to existing CMP systems, but in doing so, runtime libraries also bring their preferred support for coarse-grain parallelism. In this regard, our work is an important step towards the development of efficient runtime libraries targeted at CMPs with high core counts. By highlighting the critical overheads of TBB and Open MP, we provide programmers and developers with guidance on how best to tailor parallel libraries to CMPs.

There has also been important characterization work in the context of suites of parallel programs. For example, the SPLASH suite was one of the first workload suites for the purpose of studying shared-address cache-coherent systems. The associated SPLASH characterization study looked at a number of parallelization issues such as scalability, working set sizes, and communication to computation ratios [Woo et al. 1995]. More recently, in recognition of the prevalence of CMPs, the PARSEC benchmark suite [Bienia et al. 2008b] has been developed to represent workloads with the complexity and computational demands expected for future parallel applications. Ensuing characterization studies have been undertaken for PARSEC [Bienia et al. 2008b], again addressing scalability issues, synchronization overheads, and memory demands. Moreover, comparison studies of both the PARSEC and Splash-2 suites have been conducted [Bienia et al. 2008a]. We view our work as a logical extension of these characterization studies in the context of parallelization libraries.

Aside from the development and characterization of parallel languages, libraries, and benchmarks, much work has also addressed hardware and software mechanisms to reduce parallelization overheads. For example, Hoffman et al. [2004a, b] explore a variety of task-queue implementations to reduce overheads with a number of software and hardware synchronization primitives. While some of the surveyed applications do see benefits with the proposed hardware approaches, they are modest. Hankins et al. [2006] propose the alternate approach of Multiple Instruction Stream Processing (MISP), which allows for fast spawn and manipulation of user-level threads on CMP hardware contexts. These threads are not visible to the OS and can be invoked and terminated quickly by application threads, allowing for efficient fine-grained task management. However, task scheduling is retained in software. Unlike this proposal, Kumar et al. [2007] propose Carbon, which not only supports high performance for fine-grained parallelism through hardware task queues, but also minimizes scheduling overheads by implementing scheduling and task prefetching techniques in hardware. Our work is partly inspired by this body of previous work to propose hardware/software techniques to improve load-balancing issues on both the TBB and OpenMP libraries.

12. CONCLUSIONS AND FUTURE WORK

The advent of CMPs and expected increase in core counts necessitates the creation of parallelization libraries that can create and manage parallelism in an efficient manner across a range of granularities. To this end, Intel's Threading Building Blocks (TBB) runtime library is an increasingly popular parallelization library, which encourages the creation of portable, scalable parallel applications through the creation of parallel tasks. This allows TBB to dynamically store and distribute parallelism among available execution resources, utilizing task stealing for improving resilience to sources of load imbalance.

This article has presented a detailed characterization and identification of some of the most significant sources of overhead within the TBB runtime library. Through the use of a subset of PARSEC benchmarks ported to TBB, we show that the TBB runtime library

can contribute up to 47% of the total execution time on a 32-core system, attributing most of this overhead to synchronization within the TBB scheduler. We have studied the performance of random task stealing, which fails to scale with increasing core counts, and shown how a queue occupancy-based stealing policy can improve performance of task stealing by up to 17%.

In conjunction with our studies on TBB, we have also explored the sources of critical overheads in the OpenMP runtime system. Our results show that synchronization primitives are a primary bottleneck in performance and become more critical at higher core counts. While the implementation of more intelligent load-balancing techniques like iteration stealing can better redistribute parallelism among cores, overheads are likely to remain high at higher core counts.

The community's future work should focus on approaches that aim at reducing many of the overheads identified in our work. For example, one could accomplish this through an underlying support layer capable of offering low-latency, low-overhead parallelism management operations [Kumar et al. 2007]. One way to achieve such support is through a synergistic cooperation between software and hardware layers, giving parallel applications the flexibility of software-based implementations and the low-overhead, low-latency response of hardware implementations.

REFERENCES

ACAR, U., BLELLOCH, G., AND BLUMOFE, R. 2000. The data locality of work stealing. *Proceedings of the Symposium on Parallel Algorithms and Architectures*.

ALLEN, A., CHASE, D., LUCHANGCO, V., MAESEN, J. W., RYU, S., STEELE, G., AND TOBIN-HOCHSTADT, S. 2006. The Fortress Language specification. Sun Microsystems.

BHATTACHARJEE, A., AND MARTONOSI, M. 2009. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. *Proceedings of the International Symposium on Computer Architecture*.

BIENIA, C., KUMAR, S., AND LI, K. 2008. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip multiprocessors. *Proceedings of the International Symposium on Workload Characterization*.

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. PARSEC-2.0: Characterization and architectural implications. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

BLUMOFE, R. AND DIONISIOS, D. 1999. Hood: A user-level threads library for multiprogramming multiprocessors. *Tech. rep. University of Texas-Austin*.

BLUMOFE, R. AND LEISERSON, C. 1999. Scheduling multithreaded computations by work stealing. *J. ACM*.

BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. 1996. Cilk: An efficient multithreaded runtime system. *J. Parall. Distrib. Comput. 37,* 1, 55–69.

CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., PRAUN, C., AND SARKAR, V. 2005. X10:An object-oriented approach to non-uniform cluster computing. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

CHEN, J., JUANG, P., KO, K., CONTRERAS, G., PENRY, D., RANGAN, R., STOLER, A., PEH, L.-S., AND MARTONOSI, M. 2005. Hardware-modulated parallelism in chip multiprocessors. *Proceedings of the Workshop on Design, Architecture and Simulation of Chip Multiprocessors (dasCMP).*

DINAN, J., LARKINS, D., SADAYAPPAN, P., KRISHNAMOORTHY, S., AND NIEPLOCHA, J. 2009. Scalable work stealing. *Proceedings of the Conference on High Performance Computing Networking, Storage, and Analysis*.

FIELDS, B., RUBIN, S., AND BODIK, R. 2001. Focusing processor policies via critical-path reduction. *Proceedings of the International Symposium on Computer Architecture*.

GELERNTER, D. 1985. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst. 7,* 1, 80–112.

GIAVITTO, J. AND MICHEL, O. 2001. MGS: A rule-based programming language for complex objects and collections. *Proceedings of the International Workshop on Rule-Based Programming*.

GORDON, M. I., THIES, W., AND AMARASINGHE, S. 2006. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

HALSTEAD, R. H. 1985. MULTILISP: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Sys. 7,* 4, 501–538.

HANKINS, R., CHINYA, G., COLLINS, J., WANG, P., RAKVIC, R., WANG, H., AND SHEN, J. 2006. Multiple instruction stream processor. *Proceedings of the International Conference on parallel Processing*.

HIEB, R. AND DYBVIG, R. 1990. Continuations and concurrency. *Proceedings of the Symposium on Principles and Practices of Parallel Programming*.

HOFFMAN, R., KORCH, M., AND RAUBER, T. 2004a. Performance evaluation of task pools based on hardware synchronization. *Proceedings of the International Symposium on Supercomputing*.

HOFFMAN, R., KORCH, M., AND RAUBER, T. 2004b. Using hardware operations to reduce the synchronization overhead of task pools. *Proceedings of the International Conference on Parallel Processing*.

HUMENAY, E., TARJAN, D., AND SKADRON, K. 2007. Impact of process variations on multicore performance symmetry. In *Proceedings of the Conference on Design, Automation and Test in Europe*. ACM Press, 1653–1658.

Intel Corporation 2003. *Intel PXA255 Processor: Developer's Manual*. Intel Corporation. Order Number 278693001.

INTEL THREADING BUILDING BLOCKS 2.0 OPEN SOURCE. http://threadingbuildingblocks.org/.

JIN, H., FRUMKIN, M., AND YAN, J. 1999. The OpenMP implementation of NAS parallel benchmarks and its performance. *Tech. Rep. NAS-99-011*.

KALE, L. V. AND KRISHNAN, S. 1993. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications*.

KUMAR, S., HUGHES, C., AND NGUYEN, A. 2007. Carbon: Architectural support for fine-grained parallelism in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*.

KUSANO, K., SATOH, S., AND SATO, M. 2000. Performance evaluation of the omni OpenMP compiler. Lecture Notes in Computer Science vol. 1940, 403+.

LEISERSON, C. 2009. The cilk++ concurrency platform. *Proceedings of the Design Automation Conference*.

MARTINEZ, J. AND TORRELLAS, J. 2002. Speculative Synchronization: Applying thread-level speculation to explicitly parallel applications. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

OPENMP. 2002. OpenMP C/C++ Application Programming Interface.

OpenMP. 2007. OpenMP Application Program Interface. Draft 3.0.

OTTONI, G., RANGAN, R., STOLER, A., AND AUGUST, D. I. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the International Symposium on Microarchitecture*.

PALATIN, P., LHUILLIER, Y., AND TEMAM, O. 2006. CAPSULE: Hardware-assisted parallel execution of component-based programs. In *Proceedings of the International Symposium on Microarchitecture*.

REINDEERS, J. 2007. Intel threading building blocks: Outfitting C++ for multicore parallelism. O'Reilly Publishers.

SAMPSON, J., GONZALEZ, R., COLLARD, J.-F., JOUPPI, N. P., SCHLANSKER, M., AND CALDER, B. 2006. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *Proceedings of the International Symposium on Microarchitecture*.

THE OMNI OPENMP COMPILER PROJECT. 2007. http://phase.hpcc.jp/omni.

TUNE, E. LIANG, D., TUKSEN, D. M., AND CALDER, B. 2001. Dynamic prediction of critical path instructions. *Proceedings of the International Symposium on High Performance Computer Architecture*.

WARD, M. 1994. Language oriented programming. *Soft. Concepts Tools 15,* 4, 146–161.

WOO, S., OHARA, M., TORRIE, E., AND SINGH, J. 1995. The SPLASH-2 Programs: Characterization and methodological Considerations. *Proceedings of the International Symposium on Computer Architecture*.

Received November 2009; revised September 2010; accepted October 2010