Characterizing and Improving the Performance of Intel Threading Building Blocks

Gilberto Contreras, Margaret Martonosi
Department of Electrical Engineering
Princeton University
{gcontrer,mrm}@princeton.edu

Abstract— **The Intel Threading Building Blocks (TBB) runtime library [1] is a popular C++ parallelization environment [2][3] that offers a set of methods and templates for creating parallel applications. Through support of parallel tasks rather than parallel threads, the TBB runtime library offers improved performance scalability by dynamically redistributing parallel tasks across available processors. This not only creates more scalable, portable parallel applications, but also increases programming productivity by allowing programmers to focus their efforts on identifying concurrency rather than worrying about its management.**

While many applications benefit from dynamic management of parallelism, dynamic management carries parallelization overhead that increases with increasing core counts and decreasing task sizes. Understanding the sources of these overheads and their implications on application performance can help programmers make more efficient use of available parallelism. Clearly understanding the behavior of these overheads is the first step in creating efficient, scalable parallelization environments targeted at future CMP systems.

In this paper we study and characterize some of the overheads of the Intel Threading Building Blocks through the use of real-hardware and simulation performance measurements. Our results show that synchronization overheads within TBB can have a significant and detrimental effect on parallelism performance. Random stealing, while simple and effective at low core counts, becomes less effective as application heterogeneity and core counts increase. Overall, our study provides valuable insights that can be used to create more robust, scalable runtime libraries.

I. 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 has responded to this need by developing parallel runtime systems and libraries that aim at improving application portability and programming efficiency [4]-[11]. 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.

A recently introduced parallelization library that is likely to see wide use is the Intel Threading Building Blocks (TBB) runtime library [1]. Based on the C++ language, TBB provides 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 (i.e. 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 [12] 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 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, which in many cases may exist in the form of 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.

Our paper makes the following important 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.
- 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 sub-optimal 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.

Overall, our paper provides valuable insights that can help parallel programmers better exploit available concurrency, while aiding runtime developers create more efficient and

978-1-4244-2778-9/08/$25.00 ©2008 IEEE

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.

TABLE I

TBB TEMPLATES FOR ANNOTATING COMMON TYPES OF PARALLELISM.

```
wait_for_all(task *child) {
   task = child;
   Loop until root is alive
        do
           while task available
              next_task = task->execute();
              Decrease ref_count for parent of task
              if ref_count==0
                  next_task = parent of task
              task = next_task
            task = get_task();
        while (task);
    task = steal_task(random());
    if steal unsuccessful
     Wait for a fixed amount of time
     If waited for too long, wait for master thread
     to produce new work
}
```

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 loop of the scheduler attempt to obtain work through explicit task passing, local task dequeue, and random task stealing, respectively.

robust parallelization libraries.

Our paper is organized as follows: Section II gives a general description of Intel Threading Building Blocks and its dynamic management capabilities. Section III illustrates how TBB is used in C++ applications to annotate parallelism. Our methodology is described in Section IV along with our set of benchmarks. In Section V we evaluate the cost of some of the fundamental operations carried by the TBB runtime library during dynamic management of parallelism. Section VI studies the performance impact of TBB on our set of parallel applications, identifying overhead bottlenecks that degrade parallelism performance. Section VII performs an in-depth study of TBB's random task stealing, the cornerstone of TBB's dynamic load-balancing mechanism. In Section VIII we provide programmers and runtime library developers a set of recommendations for maximizing concurrency performance and usage. Section IX discusses related work, and Section X offers our conclusions and future work.

II. 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 [7] and Cilk [6], 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*.

A. 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 (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 non-preemptive manner, completing the execution of the task.

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

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.

B. 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 completely hidden from the programmer, it imposes a management "tax" on performance, which at significant levels can be detrimental to parallelism performance.

```
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;
   }
};
```

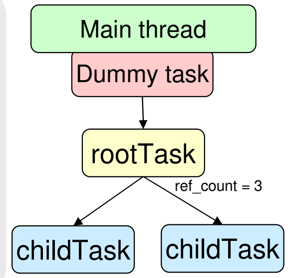

Fig. 2. TBB code example that creates a root tasks and two children tasks. In this example, the parent task (`rootTask`) blocks until its children terminate. **Bold** statements signify special methods provided by TBB that drive parallelism creation and behavior.

To better understand the principal sources of overhead that we measure in Sections V and VI, 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 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).

C. 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 2.0 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 pre-determined amount of time.

Random task stealing, while fast and easy to implement, may not always select the best victim to steal from. As the number of potential victims increase, 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 [13], effective task stealing becomes even more important. We will further study the performance of task stealing in Section VII.

III. 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 V and VI 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 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

Benchmark	Description	Number of tasks	Avg. task duration (cycles)
fluidanimate	Fluid simulation for interactive animation purposes	$420 \times N$	19M
swaptions	Heath-Jarrow-Morton framework to price portfolio of swaptions	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	

TABLE II

OUR BENCHMARK SUITE CONSISTS OF A SUBSET OF THE PARSEC BENCHMARKS PARALLELIZED USING TBB AS WELL AS TBB MICRO-BENCHMARKS. THE VALUE N REPRESENTS THE NUMBER OF PROCESSORS BEING USED.

```
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 consist 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 above). For clarity, not all variables and parallel regions are shown.

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 (i.e. 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 V 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.

IV. CHARACTERIZATION METHODOLOGY

A. Software Characteristics

We study the impact of the TBB runtime library on parallel applications by porting a subset of the PARSEC [14] benchmark suite: fluidanimate, swaptions, blackscholes, and streamcluster. 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 [15] 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 24, 2007). The most recent release (tbb20_020oss, dated April 25, 2008), addresses internal casting issues and makes modifications to internal task allocation and de-allocation, issues which 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 (in DOALL loops with a small number of iterations, for example), we further sub-partition 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.

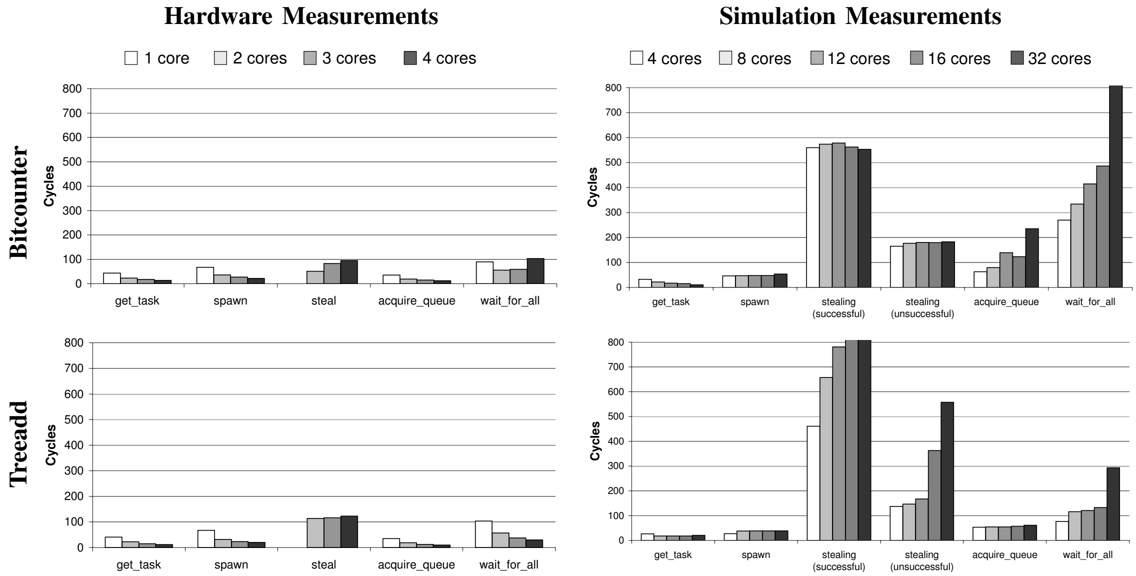

Fig. 4. Measured (hardware) and simulated costs for basic TBB runtime activities. The overhead of basic action such as `acquire_queue()` and `wait_for_all()` increase with increasing core counts above 4 cores for our simulated CMP.

In addition to porting existing parallel applications to the TBB environment, we created a set of micro-benchmarks 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.

B. 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 [16]. Performance measurements are taken using *oprofile* [17], a system-wide performance profiler that uses processor performance counters to obtain detail performance information of running 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.

C. 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 presented in [18]. Each core models a 2-issue, in-order processor similar to the Intel XScale core [19]. Cores have private 32KB L1 instruction and 32KB L1 data caches and share a distributed 4MB L2 cache. L1 data caches are kept coherent using a 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. Since the TBB system is designed to use atomic ISA support (e.g. IA32's `LOCK`, `XADD`, `XCHG` instructions), we extended our ARM ISA to support atomic operations equivalent to those in the IA32 architecture. This avoids penalizing TBB for its reliance on ISA support for atomicity.

V. Characterization of Basic TBB Functions

Dynamic management of parallelism requires the runtime library to store, schedule, and re-assign parallel tasks. Since programmers must harness parallelism whose execution time is 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 above.

All of the procedures listed above are directly or indirectly called by the scheduler loop shown in Figure 1, which constitute 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.

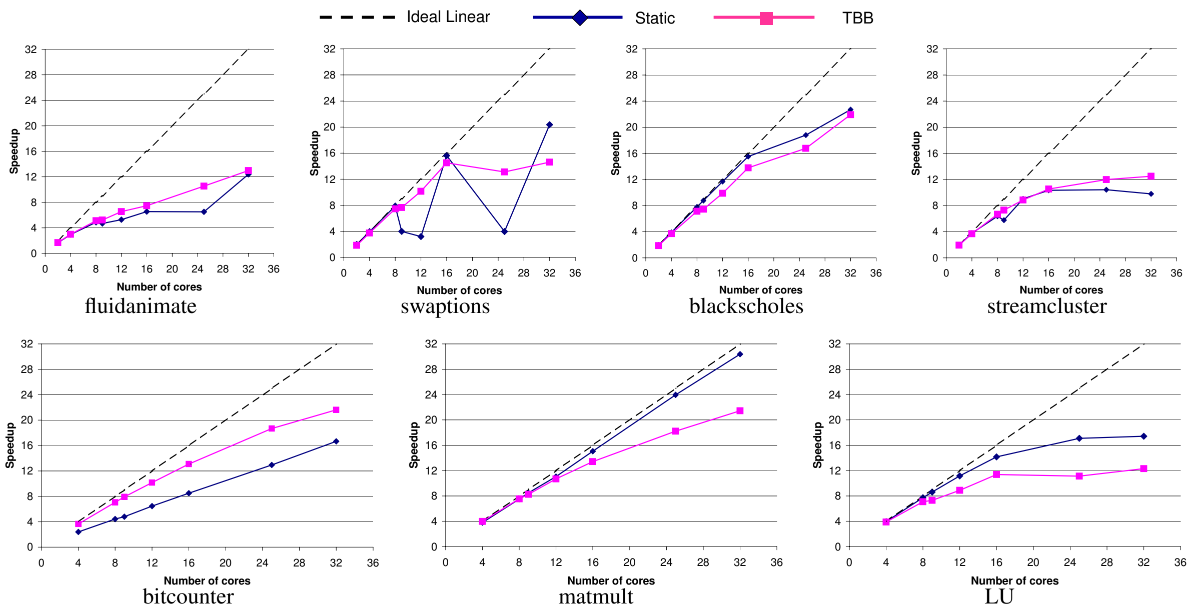

Fig. 5. Speedup results for three PARSEC benchmarks (top) and three micro-benchmarks (bottom) using *static* versus TBB. TBB improves performance scalability by creating more tasks than available processors, however, it is prone to increasing synchronization overheads at high core counts.

A. 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 high 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`. `Bitcounter` exploits DOALL parallelism through TBB's `parallel_for()` template. Its working set is highly unbalanced, which makes the execution time of tasks highly variable. The micro-benchmark `treeadd` is part of the TBB source distribution and makes use of recursive parallelism.

A.1. 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.

A.2. Simulation Measurements

In a similar way 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 tasks structure 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 micro-benchmarks. `Treeadd` utilizes *explicit task passing* (see Section II-B) 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 accounts for 40% of the cost of `wait_for_all()`.

As previously noted, the two benchmarks studied in Figure 4 have different stealing behavior, 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. 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 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 [20][21]. 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.

VI. 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 (i.e. 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.

A. 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 the full set of physical 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 [22].

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 synchronization 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 VII.

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.

B. Categorization of TBB Overheads

Section V 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 below) 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 0.5% on a 32 core system for our tested benchmarks, only four categories are considered:

- `Stealing` 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` 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 when 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 accounts for an average of 3% per core at 16

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

TABLE III
TBB overheads as measured on a 4-processor AMD system using medium and large datasets.

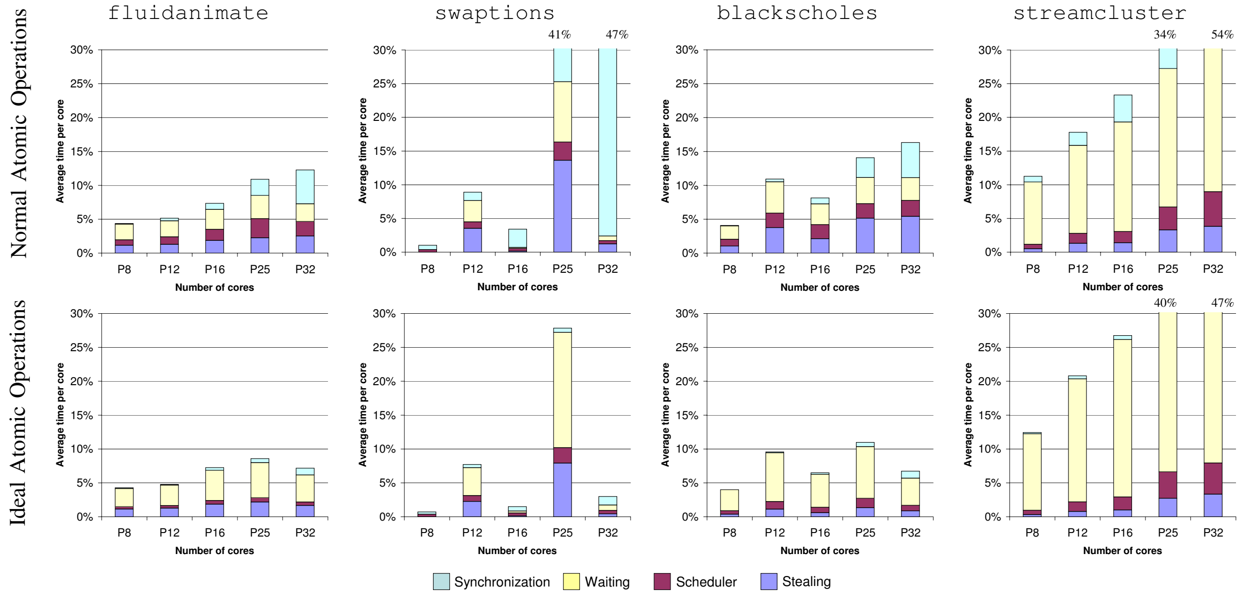

Fig. 6. Average contribution per core of the TBB runtime library on three PARSEC benchmarks. TBB contribution is broken down into four categories. The top row shows TBB contribution when latency of atomic operations is appropriately modeled. The bottom row shows TBB contributions when atomic operations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

cores (achieving a 14.8X speedup) and grows 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 micro-benchmarks, 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 paper, the next section demonstrates how runtime knowledge of parallelism can be used to improve task stealing performance.

VII. 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. Adequate and prompt stealing is necessary to reduce potential sources of imbalance. This is particularly true at barrier boundaries since failure to promptly reschedule the critical path (the thread with the most amount of work) can lead to sub-optimal performance.

To study the behavior of random task stealing in TBB, we make use of three micro-benchmarks and 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.

A. Initial Results

Figure 7 shows our results. As noted by this figure, random stealing suffers from performance degradation (decreasing success rate) as the number of cores is increased. This variability in performance is more noticeable in micro-benchmarks 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. Our results show that random victim selection, while effective at low core counts, provides sub-optimal performance at high core counts by becoming less "accurate," particularly in scenarios where there exists significant load imbalance.

B. Improving Task Stealing

We attempt to improve stealing performance by implementing two alternative victim selection policies: *occupancy-based*, and *group-based* stealing. In our *occupancy-based* selection policy, 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.

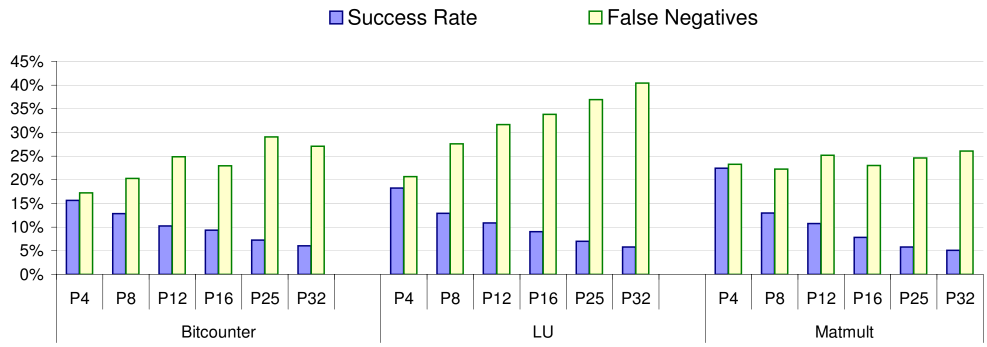

Fig. 7. Stealing behavior for three micro-benchmarks. For benchmarks with significant load imbalance such as `bitcounter` and `LU`, random task stealing losses accuracy as the number of worker threads is increased, increasing the amount of *false negatives* and decreasing stealing *success rate*.

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. Our *group-based* stealer mitigates this limitation 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.

Table IV shows the performance gain of our occupancy-based and group-based selection policies for 16, 25, and 32 core systems. Two additional scenarios are also shown: *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 hundreds of cycles reported in Section V).

Overall, occupancy-based and group-based victim selection policies achieve better performance than a random selection policy. 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.

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 sub-optimal performance as the probability of selecting the "best" victim decreases with increasing core counts. Occupancy-based policies are able to better identify the critical path and re-assign parallelism to idle worker threads.

VIII. General Recommendations

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

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

For programmers: At low core counts (2 to 8 cores), most of the overhead comes from the usage of TBB runtime procedures. 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 II-B) 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 [23]), 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.

IX. 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. Multi-processor systems previously influenced the creation of software runtime libraries and parallel languages in order to efficiently make use of available processors. Parallel languages such as Linda [11], Orca [24], Emerald [25] and Cilk [6], among many others, were designed with the purpose of extracting course-grain parallelism from applications. Runtime libraries that extended sequential languages for parallelism extraction such as Charm++ [4], STAPL [5] and OpenMP [7] 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. This work has taken an important step towards the development of efficient runtime libraries targeted at CMPs with high core counts by

Benchmark	Our Approach						Ideal					
	Occupancy			Group-Occupancy			Ideal Occupancy			Ideal Stealer		
	P16	**P25**	**P32**	**P16**	**P25**	**P32**	**P16**	**P25**	**P32**	**P16**	**P25**	**P32**
`bitcounter`	2.5%	2.5%	3.7%	2.3%	3.5%	4.2%	2.41%	2.8%	3.7%	4.7%	6.9%	7.8%
`LU`	10%	4.1%	9.7%	9.9%	4.3%	8.3%	10.2%	4.6%	8.0%	16.0%	10.4%	20.6%
`matmult`	9.5%	6%	19%	8.2%	5.3%	17.8%	9.8%	7.0%	21.1%	10.8%	9.8%	28.7%

TABLE IV

MICRO-BENCHMARK 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. 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.

highlighting some of the most critical overheads found within the TBB runtime library.

X. CONCLUSIONS AND FUTURE WORK

The Intel Threading Building Blocks (TBB) runtime library is an increasingly popular parallelization library that 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 improve resilience to sources of load imbalance.

This paper 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%.

Our future work will focus on approaches that aim at reducing many of the overheads identified in our work. We hope to accomplish this through an underlying support layer capable of offering low-latency, low-overhead parallelism management operations. 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

[1] "Intel Threading Building Blocks 2.0," March 2008. [Online]. Available: http://www.intel.com/software/products/tbb/

[2] "Product Review: Intel Threading Building Blocks," December 2006. [Online]. Available: http://www.devx.com/go-parallel/Article/33270

[3] D. Bolton, "A Free C++ Multi Threading Library from Intel!" December 2007. [Online]. Available: http://cplus.about.com/b/2007/12/04/a-free-c-multi-threading-library-from-intel.htm

[4] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *OOPSLA '93: Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM Press, 1993, pp. 91–108.

[5] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "STAPL: A Standard Template Adaptive Parallel C++ Library," in *International Workshop on Advance Compiler Technology for High Performance and Embedded Processors*, July 2001, p. 10.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[7] OpenMP C/C++ Application Programming Interface, "Version 2.0," March 2002.

[8] P. Palatin, Y. Lhuillier, and O. Temam, "CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 247–258.

[9] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs," in *ASPLOS-XII: Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM Press, 2006, pp. 151–162.

[10] R. H. Halstead, "MULTILISP: a Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 4, pp. 501–538, 1985.

[11] D. Gelernter, "Generative Communication in Linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.

[12] K. Farnham, "Threading Building Blocks Packaged into Ubuntu Hardy Heron," January 2008. [Online]. Available: http://cplus.about.com/b/2007/12/04/a-free-c-multi-threading-library-from-intel.htm

[13] E. Humenay, D. Tarjan, and K. Skadron, "Impact of Process Variations on Multicore Performance Symmetry," in *DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe*. ACM Press, 2007, pp. 1653–1658.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Princeton University Technical Report TR-811-08*, January 2008.

[15] Intel Threading Building Blocks 2.0 Open Source, "http://threadingbuildingblocks.org/."

[16] *AMD Opteron Processor Product Data Sheet*, Advanced Micro Devices, March 2007, Publication Number 23932.

[17] OProfile, "http://oprofile.sourceforge.net/."

[18] J. Chen, P. Juang, K. Ko, G. Contreras, D. Penry, R. Rangan, A. Stoler, L.-S. Peh, and M. Martonosi, "Hardware-Modulated Parallelism in Chip Multiprocessors," *In Proceedings of the Workshop on Design, Architecture and Simulation of Chip Multiprocessors (dasCMP)*, 2005.

[19] *Intel PXA255 Processor: Developer's Manual*, Intel Corporation, March 2003, order Number 278693001.

[20] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic Thread Extraction with Decoupled Software Pipelining," in *Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, November 2005.

[21] S. Kumar, C. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism in Chip Multiprocessors," in *Proceedings of the 34th International Symp. on Computer Architecture*, 2007, pp. 60–71.

[22] "OpenMP Application Program Interface. Draft 3.0," October 2007. [Online]. Available: http://www.openmp.org/drupal/mp-documents/spec30_draft.pdf

[23] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 235–246.

[24] H. Bal, M. Kaashoek, and A. Tanenbaum, "Experience with Distributed Programming in Orca," *International Conference on Computer Languages*, pp. 79–89, March 1990.

[25] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distrbution and Abstract Types in Emerald," *IEEE Trans. Softw. Eng.*, vol. 13, no. 1, pp. 65–76, 1987.