An Efficient Work-Stealing Scheduler for Task Dependency Graph

Chun-Xun Lin
Department of Electrical and Computer Engineering, University of Illinois Urbana-Champaign, Urbana, IL, USA
clin99@illinois.edu

Tsung-Wei Huang
Department of Electrical and Computer Engineering, The University of Utah, Salt Lake City, Utah
twh760812@gmail.com

Martin D. F. Wong
Department of Electrical and Computer Engineering, University of Illinois Urbana-Champaign, Urbana, IL, USA
mdfwong@illinois.edu

2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS) | 978-1-7281-9074-7/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICPADS51040.2020.00018

***Abstract*—Work-stealing is a key component of many parallel task graph libraries such as Intel Threading Building Blocks (TBB) FlowGraph, Microsoft Task Parallel Library (TPL) Batch.Net, Cpp-Taskflow, and Nabbit. However, designing a correct and effective work-stealing scheduler is a notoriously difficult job, due to subtle implementation details of concurrency controls and decentralized coordination between threads. This problem becomes even more challenging when striving for optimal thread usage in handling parallel workloads with complex task graphs. As a result, we introduce in this paper an effective work-stealing scheduler for execution of task dependency graphs. Our scheduler adopts a simple and efficient strategy to *adapt* the number of working threads to available task parallelism at any time during the graph execution. Our strategy is provably good in preventing resource underutilization and simultaneously minimizing resource waste when tasks are scarce. We have evaluated our scheduler on both micro-benchmarks and a real-world circuit timing analysis workload, and demonstrated promising results over existing methods in terms of runtime, energy efficiency, and throughput.**

***Keywords*-task dependency graph; work stealing; parallel computing; scheduling; multithreading;**

I. Introduction

Work stealing has been proved to be an efficient approach for parallel task scheduling on multi-core systems and has received wide research interest over the past few decades [1]–[12]. Several task-based parallel programming libraries and language have adopted work-stealing scheduler as the runtime for thread management and task dispatch such as Intel Threading Building Blocks (TBB) [13], Cilk [2] [14], X10 [15] [8], Nabbit [16], Microsoft Task Parallel Library (TPL) [17], and Golang [18]. The efficiency of the work-stealing scheduler can be attributed to the way it manages the threads: The scheduler spawns multiple threads (denoted as workers) on initialization. Each worker first carries out tasks in its private queue. Then, a worker without available tasks becomes a thief and randomly steals tasks from others. By having thieves actively steal tasks, the scheduler is able to balance the workload and maximize the performance. However, implementing an efficient work-stealing scheduler is not an easy job, especially when dealing with task dependency graph where the parallelism could be very irregular and unstructured. Due to the decentralized architecture, developers have to sort out many implementation details to efficiently manage workers such as deciding the number of steals attempted by a thief and how to mitigate the resource contention between workers. The problem is even more challenging when considering throughput and energy efficiency, which have emerged as critical issues in modern scheduler designs [4] [19]. The scheduler's worker management can have a huge impact on these issues if it is not designed properly. For example, a straightforward method is to keep workers busy in waiting for tasks. Apparently, this method consumes too much resource and can result in a number of problems, such as decreasing the throughput of co-running multithreaded applications and low energy efficiency [4] [19]. Several methods have been proposed to remedy this deficiency, e.g., making thieves relinquish their cores before stealing [5] or backing off a certain time [6], [13], or modifying OS kernel to directly manipulate CPU cores [4]. Nevertheless, these approaches still have drawbacks especially from the standpoints of solution generality and performance scalability.

In this paper, we propose an adaptive work-stealing scheduler with provably good worker management for executing task dependency graphs. Our scheduler employs a simple yet effective strategy, in no need of sophisticated data structures, to adapt the number of working threads to dynamically generated task parallelism. Not only can our strategy prevent resource from being underutilized which could degrade the performance, but it also avoids overly subscribing threads or thieves when tasks are scarce. As a result, we can significantly improve the overall system performance including runtime, energy usage, and throughput. We summarize our contributions as follows:

- **An adaptive scheduling strategy**: We developed an adaptive scheduling strategy for executing task dependency graph. The strategy is simple and easy to implement in a standalone fashion. It does not require sophisticated data structures or hardware-specific code. A wide range of parallel processing libraries and runtimes that leverage

2690-5965/20/$31.00 ©2020 IEEE
DOI 10.1109/ICPADS51040.2020.00018

work-stealing can benefit from our scheduler designs.

- **Provably good worker management**: We proved our scheduler can simultaneously prevent the under-subscription problem and the over-subscription problem during the graph execution. When tasks are scarce, the number of wasteful thieves will drop within a bounded interval to save resources for other threads both inside and outside the application.
- **Improved system performance**: We showed through balancing working threads on top of dynamically generated task parallelisms can improve both application performance and overall system performance including runtime, energy efficiency, and throughput. Improved system performance translates to better scalability.

We evaluated the proposed scheduler on both micro-benchmarks and real-world applications, and we show that on a very large scale integration (VLSI) timing analyzer benchmark our scheduler can achieve up to 15% less runtime and 36% less energy consumption over a baseline algorithm.

II. Adaptive Work-Stealing Scheduler

In this section we first introduce the task dependency graph model and present the details of the proposed work-stealing scheduler. Then, we provide an analysis of our worker management to show its efficiency.

A. Scheduler Overview

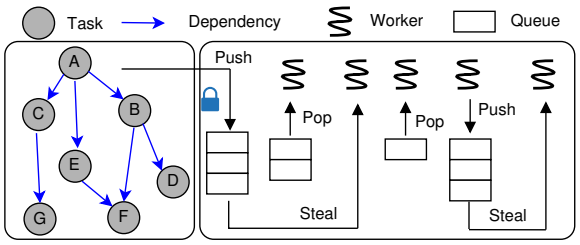

Figure 1. A task dependency graph and the architecture of our scheduler.

Algorithm 1: Task insertion from users

```
Input: tasks: source tasks
1 lock();
2 for t in tasks do
3 |   master_queue.push(t);
4 end
5 unlock();
6 notifier.notify_one();
```

Figure 1 shows a task dependency graph (left) and the architecture of the proposed scheduler (right). A task dependency graph is a directed acyclic graph, where a node contains a task which can be any computation, and an edge denotes the dependency between tasks. A node is ready for execution only after all its predecessors finish. The execution of a task dependency graph begins from source nodes (nodes without predecessors, e.g. node A) and ends when all sink nodes (nodes without successors, e.g. node G) finish.

Our scheduler consists of a set of workers, a master queue, a lock and a notifier. On initialization the scheduler spawns workers waiting for tasks. Each worker is equipped with a private queue to store tasks ready for execution. To begin executing a task graph, users insert source nodes to the master queue and notify workers via the notifier. Algorithm 1 is the pseudo code of task insertion from users. A lock (line 1) is used to protect the master queue for concurrent submissions of task graphs. When a worker completes a task, it automatically decrements the dependency of successive tasks and pushes new tasks to its queue whenever dependency is met. Each worker keeps one local cache of task for continuation.

We implement the queue based on the Chase-Lev algorithm [20] [21]. Each worker can add and pop task tasks from one end of its queue, while other workers can steal tasks from the other end. The `notifier` is a two-phase synchronization protocol that efficiently supports: (1) putting a worker into sleep and (2) waking up one or all sleeping workers. We leverage the `EventCount` construct from the Eigen library [22] as the `notifier` in our scheduler. The usage of `EventCount` is similar to a condition variable. The notifying thread sets a condition to true and then signals waiting workers via `EventCount`. On the other side, a worker first checks the condition and returns to work if the condition is true. Otherwise, the worker updates the `EventCount` to indicate it is waiting and checks the condition again. If the condition is still false, the worker is suspended via the `EventCount`.

B. Worker Management

Each worker iterates the following two phases:

1) **Task exploitation**: The worker repeatedly executes tasks from its queue and exploits new successor tasks whenever dependency is met.
2) **Task exploration**: The worker drains out its queue and turns into a thief to explore tasks by randomly stealing tasks from peer workers. If the worker successfully steals a task, it returns to the task exploitation phase, or becomes a sleep candidate to wait for future tasks.

Algorithm 2 is the pseudo code of a worker's top-level control flow. The `exploit_task` (line 5) and `wait_for_task` (line 6) functions correspond to the two phases, respectively. The worker exits the control flow when the scheduler terminates (`wait_for_task` returns `false`). Inside these two phases, our scheduler uses two variables: `num_actives` and `num_thieves`, to adaptively adjust the number of thieves. We present the details of these two functions below to illustrate our scheduler's worker management.

Algorithm 2: Top-level worker control flow

```
   Input: id: worker id
1  Function worker_loop(id):
2      w ← workers[id] ; // the worker's associated
       data
3      t ← NIL ; // a task holder
4      while true do
5          exploit_task(t, w) ; // Algorithm 3
6          if wait_for_task(t, w) = false then // Algorithm 5
7              break;
8          end
9      end
10 return
```

Algorithm 3 is the pseudo code of the task exploitation phase. A worker enters this phase when successfully stealing a task (line 1), either from other queues or the master queue. The worker first increments the `num_active` and checks the `num_thieves` (line 2). If the worker is the first one that increments the `num_active` and there is no thief (i.e. `num_thieves` is 0), the worker makes a notification using the notifier (line 3). Next the worker carries out available tasks until both the cache and private queue are empty (line 5-12). When the worker runs out all available tasks, it decrements the `num_actives` (line 13) and leaves the `exploit_task`. Next in task exploration phase, a worker becomes a thief to randomly steal tasks from others. **Algorithm 5** is the pseudo code of task exploration. The thief begins by incrementing `num_thieves` (line 2) and then invokes the `explore_task` (**Algorithm 4**) to perform random stealing. Once the thief obtains a task, it decrements the `num_thieves` (line 5) and returns to task exploitation phase (line 8). A successful thief has to make a notification (line 6) if the `num_thieves` becomes zero. Otherwise, the thief that failed to derive any task prepares waiting for tasks via the notifier (line 10) and then checks the master queue (line 11) and the scheduler's status (line 23) sequentially. If the master queue is empty and the scheduler is not terminated, the thief decrements the `num_thieves` (line 29). If there exists an active worker (i.e. `num_actives` is greater than 0), the last thief has to reiterate the `wait_for_task` procedure (line 31). Otherwise, the thief will be suspended by the notifier (line 33).

Algorithm 4 is the pseudo code of how a thief performs random stealing. The thief steals randomly from either other queues or the master queue (line 5:9) and returns once it succeeds. When the number of failed steal exceeds a threshold (line 14), the thief has to call `yield` after each failed steal. The yielding intends to let other workers with tasks to run first so that resource can be better utilized. A thief stops stealing if it cannot obtain a task after yielding several times (line 18).

Algorithm 3: exploit_task

```
   Input: t: a task holder
   Input: w: the worker's associated data
1  if t ≠ NIL then
2      if AtomInc(num_actives) == 1 and num_thieves == 0 then
3          notifier.notify_one();
4      end
5      do
6          execute(t);
7          if w.cache ≠ NIL then
8              t ← w.cache;
9          else
10             t ← pop(w.queue);
11         end
12     while t ≠ NIL;
13     AtomDec(num_actives);
14 end
```

Algorithm 4: explore_task

```
   Input: t: a task holder
   Input: w: the worker's associated data
1  num_failed_steals ← 0;
2  num_yields ← 0;
3  while scheduler not stops do
4      victim ← random();
5      if victim == w then
6          t ← steal(master_queue);
7      else
8          t ← steal_from(victim);
9      end
10     if t ≠ NIL then
11         break;
12     else
13         num_failed_steals + +;
14         if num_failed_steals ≥ STEAL_BOUND then
15             yield();
16             num_yields + + ;
17             if num_yields = YIELD_BOUND then
18                 break;
19             end
20         end
21     end
22 end
```

C. Analysis

We show the scheduler's worker management is very efficient in two fronts: (1) it does not have the under-subscription problem (2) it mitigates the over-subscription problem by putting most thieves into sleep after they failed to steal within a time bound. We first define the states of a worker and present the Lemma 1:

Definition 1. A worker is **active** if it is exploiting tasks (Alg 3: line 2:13). A worker is a **thief** if it is not exploiting tasks (Alg 3: line 2:13) nor sleeping (Alg 5: line 33).

Lemma 1. *When a worker is active and at least one worker is inactive, one thief always exists.*

Proof: Assume there exists one active worker and one inactive worker. The inactive worker is either awake (Alg 5: line 1:28) or sleeping (Alg 5: line 33). If the inactive

Algorithm 5: wait_for_task

```
   Input: t: a task holder
   Input: w: the worker's associated data
   Output: A boolean value to indicate continuation of worker-loop
1  wait_for_task:
2  AtomInc(num_thieves);
3  explore_task:
4  if explore_task(t, w) ; t ≠ NIL then
5      if AtomDec(num_thieves) == 0 then
6          notifier.notify_one();
7      end
8      return true;
9  end
10 notifier.prepare_wait(w);
11 if master_queue is not empty then
12     notifier.cancel_wait(w);
13     t ← steal(master_queue);
14     if t ≠ NIL then
15         if AtomDec(num_thieves) == 0 then
16             notifier.notify_one();
17         end
18         return true;
19     else
20         go to explore_task;
21     end
22 end
23 if scheduler stops then
24     notifier.cancel_wait(w);
25     notifier.notify_all();
26     AtomDec(num_thieves);
27     return false;
28 end
29 if AtomDec(num_thieves) == 0 and num_actives > 0 then
30     notifier.cancel_wait(w);
31     go to wait_for_task;
32 end
33 notifier.commit_wait(w);
34 return true;
```

worker is awake, then it is a thief and the lemma holds. Otherwise the inactive worker is sleeping and it must have decremented the `num_thieves` without seeing any active worker (Alg 5: line 29). This happens only when the active worker just enters the `exploit_task` function and is right before incrementing `num_actives` (Alg 3: line 2). Subsequently the active worker shall wake up a thief (Alg 3: line 3) and the lemma holds. ■

With Lemma 1, we show that our scheduler does not have the *under-subscription* problem:

Definition 2. An under-subscription problem means:

$$T = 0 \text{ and } 0 < Q < W$$

where

T : number of thieves, W : number of total workers

Q : number of non-empty private queues

According to the definition, the under-subscription problem occurs when all thieves go into sleep (i.e. $T = 0$) while at least one queue is non-empty (i.e. $0 < Q < W$). Lemma 1 guarantees at least one thief exists in our scheduler when there is an active worker. Since a worker must be active if its queue is not empty, according to the Lemma 1 a thief must exist (except when all workers are active), and thus our scheduler does not have the under-subscription problem.

Our scheduling method not only prevents the under-subscription problem but can also mitigate the over-subscription problem. The over-subscription problem means the number of thieves is greater than the number of available tasks. Over-subscription can lead to substantial resource wasted on failed steals if those thieves remain awake for a long time.

We now show the scheduler will put most thieves into sleep if they fail to steal any task within a time bound. First we give the definition of a group:

Definition 3. Assume there are multiple thieves. We call those thieves are in a *group* and a thief leaves the group if it goes into sleep (Alg 5: line 33) or successfully steals a task.

Given a group, we prove that only one thief exists in the group after a certain time. This implies excessive thieves will go into sleep when there is no sufficient tasks. In the following proof, we assume the master queue is empty since thieves always check the master queue before going to sleep.

Lemma 2. *Given a group of thieves, only one thief in the group exists after* $O((STEAL\_BOUND + YIELD\_BOUND) * S + C)$ *time, where* S *is the time to perform a steal and* C *is a constant.*

Proof: Given a group of thieves, we denote the thief that lastly decrements the `num_thieves` (Alg 5: line 5, 15 and 29) in this group as the *last thief*. Thieves in a group except the last thief must either (1) become active workers if they successfully steal tasks or (2) go into sleep (Alg 5 line 33) after they decrement the `num_thieves`. Therefore, eventually only one thief stays in the group when the last thief performs the decrement.

Next we analyze the runtime taken by the last thief to perform the decrement on `num_thieves`. There are two cases to consider: the last thief either successfully steals a task (Alg 5: line 4) or fails to steal any task (Alg 5: line 29). For the first case, the runtime is bounded by $O((STEAL\_BOUND + YIELD\_BOUND) * S)$ where S is the time of conducting one steal and $(STEAL\_BOUND + YIELD\_BOUND)$ is the maximum number of steals that can be attempted. For the second case, the last thief will attempt $(STEAL\_BOUND + YIELD\_BOUND)$ steals, prepare for sleep, and check the master queue and scheduler's status before doing the decrement.

Because the latter two steps are simple routines, we can use a constant C to denote the maximal runtime taken by these two steps. Then the runtime of the

second case is bounded by $O((STEAL\_BOUND + YIELD\_BOUND) * S + C)$. Therefore, the runtime for the last thief to perform the decrement will be bounded by $O((STEAL\_BOUND + YIELD\_BOUND) * S + C)$. ■

To sum up, we proved our scheduler can prevent the under-subscription problem (Lemma 1) and effectively mitigate the over-subscription problem (Lemma 2) during graph execution. When tasks are abundant, one worker will incrementally wake up another and so on until no starvation or maximum workers reached. When tasks are scarce, the number of thieves will drop within a bounded interval to reduce wasteful steals. Our scheduler adaptively maintains this invariant to balance the number of workers on top of dynamically generated task parallelisms.

III. Evaluation

We evaluated our scheduler on both micro-benchmarks and a real-world timing analyzer for VLSI systems. We compare our scheduler with two approaches: (1) The ABP method proposed by Aurora et al. [5] and (2) MBWS which is modified from the BWS method of Ding et al. [4]. For fair comparison, we implement all schedulers on top of the same task execution engine, Cpp-Taskflow [23], a modern C++ parallel programming library using task dependency graphs. We briefly summarize our implementation of ABP and MBWS: In ABP, thieves repeatedly steal until they succeed, and thieves will invoke `yield` system call every time before attempting a steal. BWS [4] introduces two methods to enhance ABP's resource utilization: (1) BWS modifies the OS kernel so that workers can query the running status of others and yield their cores directly to others. (2) BWS uses two counters, a wake-up counter and a steal counter, to make thieves wake up two sleeping workers for busy workers and limit the number of steals a thief can attempt. We modified BWS as follows: First, we do not modify the OS kernel as we aim for a portable solution that does not introduce system-specific hard code. To compensate for this, we associate each worker with a status flag which is set by the owner to inform its current status, and thieves do not yield their cores.

Second, we included the steal and wake-up counters into our `explore_task`. As multiple thieves can concurrently modify the wake-up counter, we use atomic compare-and-swap operation to decrement the wake-up counter. A deficiency of BWS is that all thieves could be sleeping while parallelism is increased. To resolve this problem, BWS has to keep a watchdog worker which never goes into sleep even no tasks are available. We also implemented this mechanism in MBWS by keeping a thief busy in a stealing loop if it is the last one that decrements `num_thieves`. Notice that the modified BWS may not be reflective of the true implementation but it provides a good reference to implement the wake-up-two heuristic.

We conducted all experiments on a machine with two Intel Xeon Gold 6138 processors (2 NUMA nodes) and 256 GB memory. Each processor has 20 cores with 2 threads per core. The OS is Ubuntu 19.04 and the compiler is GCC 8.3.0. We compile all source code with the optimization flag `O2` and C++ 17 standard flag (`-std=c++17`). To reduce the impact of thread migration, we use the system command `taskset` to bond the threads on a set of cores, and we split the threads equally on the two processors. The `STEAL_BOUND` is set to $2 * (\text{number of workers} + 1)$ and the `YIELD_BOUND` is 100. For MBWS we select 64 as the `SleepThreshold`, which is the same as the experiment setting in [4]. We report the results measured by Linux profiling utility `perf`.

A. Micro-benchmarks

We investigate the performance of each scheduler under a set of task dependency graphs each representing a unique parallel pattern:

- Linear chain: Each task has one successor and one predecessor except the first task and the last tasks. Each task increments a global counter by one. The graph has 8388608 tasks.
- Binary tree: Each task has one predecessor and two successors except the root and leaf tasks. The task has no computation and the graph has 8388607 tasks.
- Graph traversal: We generate a directed acyclic graph with random dependency. The degree of each node, i.e. number of successors and predecessors, is bounded by four. Each task marks a boolean variable to true indicating the node is visited. The graph has 4000000 tasks.
- Matrix multiplication: We generate a task graph to multiply two square matrices of size 2048×2048. The task graph has two parts: (1) The first part initializes the values of matrices. (2) The second part performs the matrix multiplication. Computations are performed row by row to represent a generic parallel-for pattern.

In this experiment, we vary the number of cores from 1, 4, 8, 12, 16, 20, 24, 28, 32, 36 to 40 to study the scalability and CPU utilization of each scheduler. All data is an average over ten runs reported by the command `perf stat -r`.

Figure 2 and 3 show the runtime and CPU utilization of each benchmark, respectively. For the linear chain, the runtime does not decrease when adding more cores. This is expected as the linear chain has no parallelism at all and one core is sufficient for optimal performance. The CPU utilization of ABP increases along with the number of cores while both MBWS and ours remain nearly uninfluenced. In fact, the CPU utilizations of MBWS and ours stay around 2.0 and 1.2 from 4 to 40 cores respectively. For the binary tree and graph traversal, the runtimes of all schedulers drop to a stable point after 4 cores. Adding more cores does not improve the performance as a worker can quickly carry out all tasks in its queue before thieves discover them. We can

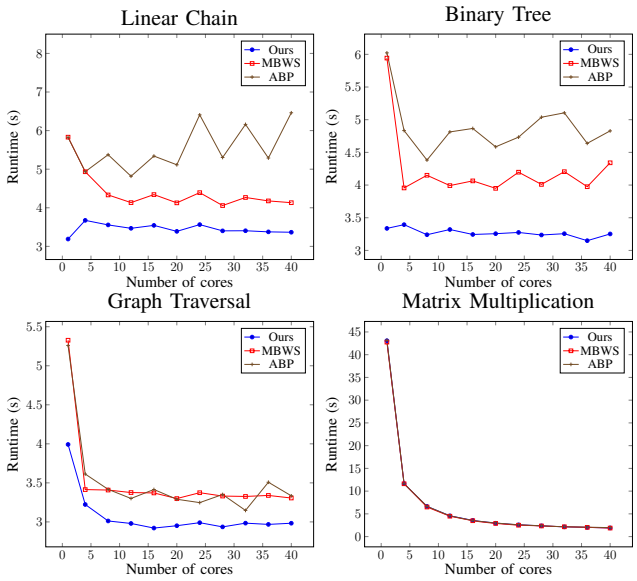

Figure 2. Runtime comparisons between ours, MBWS, and ABP on micro-benchmarks.

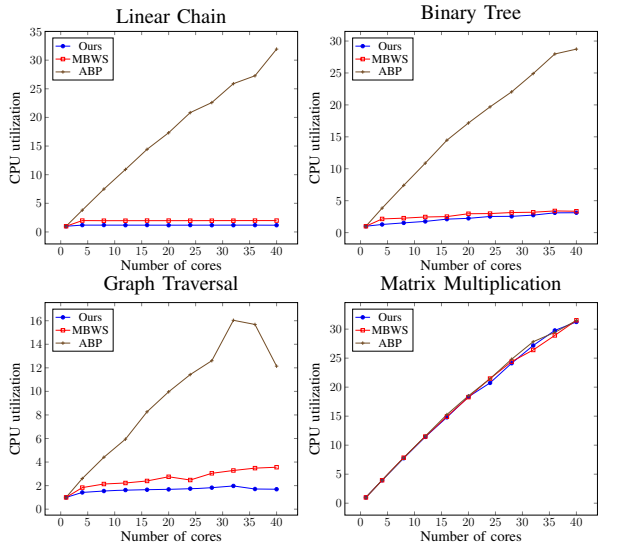

Figure 3. CPU utilization comparisons between ours, MBWS, and ABP on micro-benchmarks.

clearly inspect the overhead of stealing under this scenario. ABP has the highest CPU utilization among all schedulers and ours is the lowest in both cases. The CPU utilization of ABP also grows more rapidly than others in these two cases. Lastly, for the matrix multiplication, which has better scalability than the previous three cases, the runtimes of all schedulers are very close and their CPU utilizations exhibit similar growth. There are two reasons: (1) In the matrix multiplication, intra-level tasks are independent of each other and those tasks have nearly equal workload. (2) the multiplication is compute-intensive and thus the runtime is dominated by the computation rather than the scheduling overhead.

Table I
STATISTICS OF CIRCUITS

Circuit	# of gates (K)	# of nets (K)	# of operations
c6288	1.7	1.7	80800
tv80	5.3	5.3	51000
b19	255.3	255.3	10100

B. VLSI Timing Analysis

We study the performance of our scheduler on a real-world VLSI static timing analyzer, OpenTimer [24]. Static timing analysis (STA) plays a critical role in the circuit design flow. For a circuit to function correctly, its timing behavior must meet all requirements under different design constraints and environment settings. Thus, circuit designers have to apply STA to examine the circuit's timing behavior during different stages in the design flow. STA calculates the timing-related information by propagating through the gates in a circuit. This workload can be naturally described using task dependency graph, where a task encloses the computation of a gate and the edge denotes the propagation direction. In this experiment, we use these schedulers to execute the task graph built in OpenTimer [24], an open-source VLSI timer. We randomly generate a set of operations which incrementally modify a given circuit and then perform STA to update the timing. We use the circuits from TAU 2015 timing contest [25] and the statistics of the circuits are listed in Table I. For each circuit we ran OpenTimer five times and report the average runtime and CPU utilization recorded by `perf`. Figure 4 show the runtime (left) and CPU utilization (right) of each circuit respectively.

We categorize the circuits into different groups based on their sizes and discuss the results. On the smallest circuit c6288, the runtime does not scale with the number of cores because the size of the circuits is small. The CPU utilizations of all schedulers increase along with the number of cores, and ABP has the highest CPU utilization followed by the MBWS and ours is the smallest. Next for the medium size circuit tv80, the runtimes of all schedulers decrease after adding more cores. ABP is faster than others except at single core and the runtimes at 40 cores are 60.4 (ours), 60.8 (MBWS) and 52.5 (ABP), respectively. We attribute this to the overhead of notifying workers. Both ours and MBWS will put thieves into sleep and wake them up when tasks present, while in ABP all thieves are stay awake in waiting for tasks. In terms of the CPU utilization, ABP is still the highest, and ours and MBWS are very close. Lastly, for the large circuit with over 100,000 gates: b19, the performance scales with the number of cores in all schedulers. When using multiple cores, ABP is slower than others even though ABP's CPU utilization remains the highest. For instance,

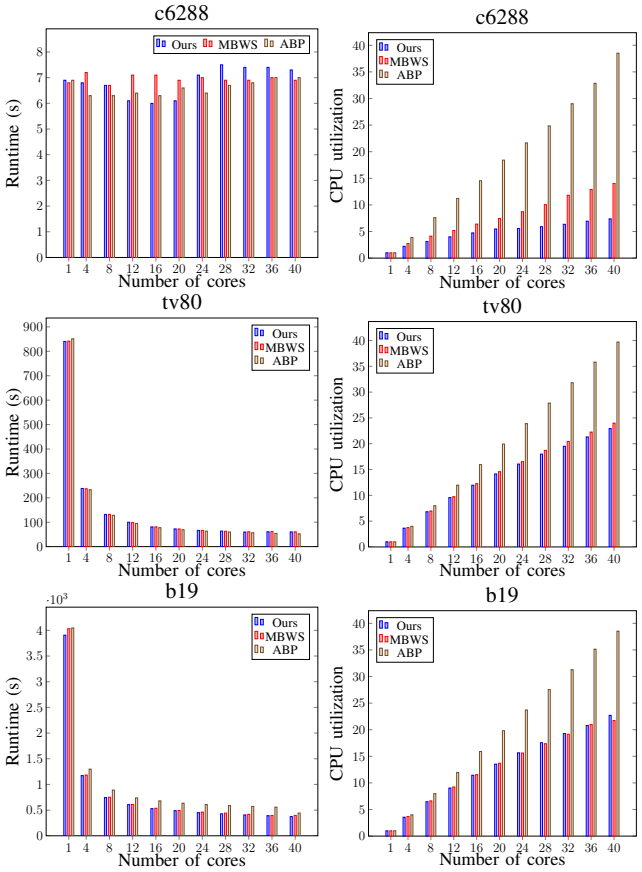

Figure 4. Runtime and CPU utilization comparisons between ours, MBWS, and ABP on OpenTimer.

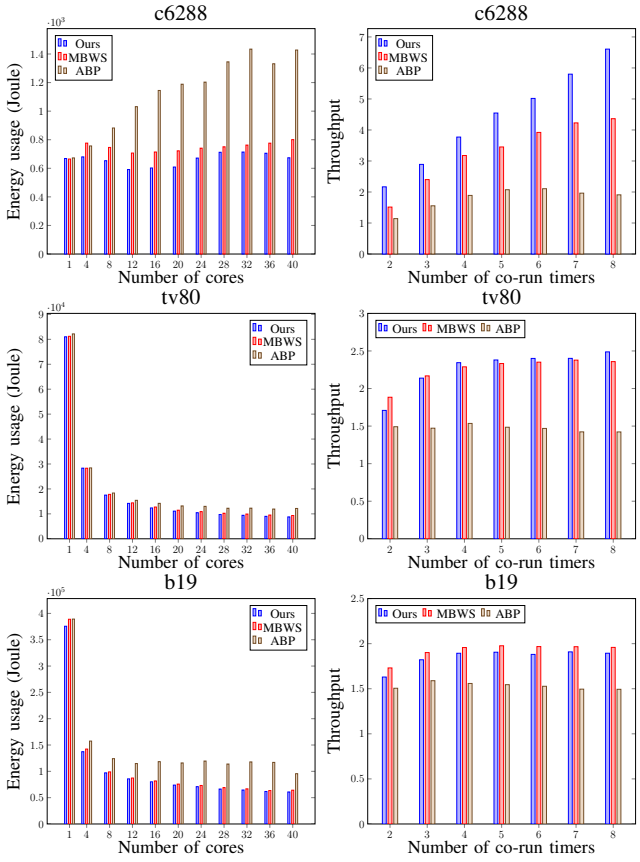

Figure 5. Energy usage and throughput of ours, MBWS, and ABP on OpenTimer.

on 40 cores the runtime of ours is 5% and 15% less than MBWS and ABP, respectively, and the CPU utilizations are 22.7 (ours), 21.7 (MBWS) and 38.5 (ABP). This experiment shows that our scheduler has competitive performance and it can utilize the CPU resource efficiently under a large-scale workload.

Next we demonstrate the energy usage of each scheduler with OpenTimer. Intel has provided the Running Average Power Limit (RAPL) [26] interface for power management on recent processors. We use `perf`, which can access RAPL to measure the energy consumed by the packages during the execution (The command is `perf stat -e power/energy-pkg/ -a`), and we let `perf` report the average value of five runs.

Figure 5 is the average energy usage (left) reported by `perf` and the throughputs (right). For energy usage, ABP is the highest in most cases. MBWS is very close to ours with ours performs slightly better in most cases. On smallest circuit c6288 the energy usage of ABP increases along with the number of cores even the performance does not scale. For example, the energy usage of ABP is 2x of ours at 40 cores but the runtime of ABP is only 8% less than ours. For other circuits, the energy usage of all schedulers decreases after adding more cores as those circuits have good scalability. However, ABP's energy usage is still much higher than others. For example, in the largest circuit b19 ABP's energy usage is 1.57x of ours and 1.48x of MBWS when using 40 cores.

For throughput, we simulate the real working environment which is typically shared by multiple users such as servers or cloud computing platforms. In those environments users can run multithreaded applications concurrently, and applications might request computing resources more than their actual parallelism. In this experiment, we run multiple OpenTimers simultaneously on the same circuit and every timer can use all cores (40 on our machine). The number of OpenTimers in the co-runs ranges from 2 to 8 and we use the `time` command to measure the runtime (wall clock time) of each timer. We repeat each co-run five times and use the average to derive the throughput. For each scheduler, we take the runtime of its solo-run as the baseline and compute the throughput using the weighted-speedup method [4] [27]. The weighted-speedup method sums the speedup of each process in the co-runs, where the speedup

of a process is defined as $T_{baseline}/T_{proc}$. As shown in Figure 5 (right), ABP has the lowest throughput in all co-runs regardless of the circuit size. For example, when co-running 8 OpenTimers, the throughputs of ABP are 1.49 and 1.9 on b19 and c6288, respectively, while ours is 1.89 and 6.6 and MBWS is 1.95 and 4.36. The throughputs of MBWS and ours are quite close except at c6288 where ours is much higher.

IV. Conclusion

In this paper, we have introduced a work-stealing scheduler for executing task dependency graph. We have designed an efficient worker management method that can adapt the number of working threads to the available task parallelism. This method not only prevents resource from being underutilized but also mitigates resource waste. We have evaluated the scheduler on a set of micro-benchmarks and a VLSI timing analyzer. The results show our scheduler achieved comparable performance to existing approaches with effective resource utilization. For instance, in a real workload, our scheduler achieved 15% less runtime with 36% less energy usage than the baseline method. We have also demonstrated our scheduler is very energy-efficient and can maintain good throughput when co-running multithreaded applications.

References

[1] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720748, September 1999.

[2] Robert D. Blumofe et al. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[3] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments. Technical report, Austin, TX, USA, 1998.

[4] Xiaoning Ding et al. BWS: Balanced work stealing for time-sharing multicores. In *EuroSys*, pages 365–378. ACM, 2012.

[5] Nimar S. Arora et al. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pages 119–129. ACM, 1998.

[6] G. Contreras and M. Martonosi. Characterizing and improving the performance of Intel Threading Building Blocks. In *2008 IEEE International Symposium on Workload Characterization*, pages 57–66, Sep. 2008.

[7] Kunal Agrawal et al. Adaptive work stealing with parallelism feedback. In *PPoPP*, pages 112–120. ACM, 2007.

[8] Yi Guo et al. Work-first and help-first scheduling policies for async-finish task parallelism. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009.

[9] Olivier Tardieu et al. A work-stealing scheduler for X10's task parallelism with suspension. In *PPoPP*, pages 267–276. ACM, 2012.

[10] Umut A. Acar et al. The data locality of work stealing. In *SPAA*, pages 1–12. ACM, 2000.

[11] Kyle Singer et al. Proactive work stealing for futures. In *PPoPP*, pages 257–271. ACM, 2019.

[12] Y. Guo et al. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *IEEE IPDPS*, pages 1–12, April 2010.

[13] Alexey Kukanov et al. The foundations for scalable multi-core software in Intel Threading Building Blocks, November 2007.

[14] Matteo Frigo et al. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[15] Shivali Agarwal et al. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA*, pages 229–240. ACM, 2007.

[16] K. Agrawal et al. Executing task graphs using work-stealing. In *IEEE IPDPS*, pages 1–12, April 2010.

[17] Daan Leijen et al. The design of a task parallel library. In *OOPSLA*. ACM SIGPLAN, September 2009.

[18] Jaana B. Dogan. Go's work-stealing scheduler. https://rakyll.org/scheduler/.

[19] Haris Ribic and Yu David Liu. Energy-efficient work-stealing language runtimes. ASPLOS '14, pages 513–528, New York, NY, USA, 2014. Association for Computing Machinery.

[20] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA*, pages 21–28. ACM, 2005.

[21] Nhat Minh Lê et al. Correct and efficient work-stealing for weak memory models. In *PPoPP*, pages 69–80. ACM, 2013.

[22] http://eigen.tuxfamily.org/index.php?title=Main_Page, Eigen.

[23] T. Huang et al. Cpp-Taskflow: Fast task-based parallel programming using modern C++. In *IEEE IPDPS*, pages 974–983, May 2019.

[24] Tsung-Wei Huang and Martin D. F. Wong. OpenTimer: A high-performance timing analysis tool. In *Proceedings of the IEEE/ACM ICCAD*, pages 895–902, 2015.

[25] J. Hu et al. TAU 2015 contest on incremental timing analysis. In *IEEE/ACM ICCAD*, pages 882–889, Nov 2015.

[26] Intel 64 and IA-32 architectures software developer manuals. https://software.intel.com/en-us/articles/intel-sdm.

[27] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 234–244, New York, NY, USA, 2000.