LETTER

A Load-Balanced Deterministic Runtime for Pipeline Parallelism

Chen CHEN[†,††a)], **Kai LU**[†,††], **Xiaoping WANG**[†,††], ***Nonmembers***, **Xu ZHOU**[†,††], ***Member***, ***and*** **Zhendong WU**[†,††], ***Nonmember***

SUMMARY Most existing deterministic multithreading systems are costly on pipeline parallel programs due to load imbalance. In this letter, we propose a Load-Balanced Deterministic Runtime (LBDR) for pipeline parallelism. LBDR deterministically takes some tokens from non-synchronization-intensive threads to synchronization-intensive threads. Experimental results show that LBDR outperforms the state-of-the-art design by an average of 22.5%.

key words: *deterministic runtime, pipeline parallelism, round-robin scheduling, load balance*

1. Introduction

Nondeterminism makes parallel programming a daunting task. For the same input, a parallel program is unlikely to repeat the same thread interleaving between any two executions. Different interleavings may not only lead to different outputs, but also make the bug reproducing a challenging task. Recognizing this fact, researchers have recently developed a range of deterministic multithreading (DMT) systems that enforce deterministic thread interleavings.

Pipeline parallelism is one of the most important parallel patterns. There are two pipeline parallel programs in the PARSEC benchmark suite [1]. Pipeline parallelism can be applied to many emerging applications, such as streaming workloads, due to its ability to extract parallelism from loops which are difficult to parallelize otherwise.

Unfortunately, state-of-the-art DMTs are much costly on pipeline parallelism than on data partition parallelism. One of the main reasons is, unlike data partition parallelism, pipeline parallelism assigns unbalanced work among threads. In pipeline parallelism, threads are split into stages of a pipeline. Threads from different stages execute different codes, which make DMTs hard to achieve load-balanced scheduling among threads.

Currently, there are two ways to guarantee deterministic order of synchronization operations. One way is to order synchronization operations according to the number of instructions each thread has executed [2], [3]. Taking instruction count as logical clock is an effective way to guarantee load balance. However, as pointed out by [4], such approaches are unstable, as instruction count is sensitive to minor input or code changes.

Another way is round-robin scheduling [4]–[6]. Round-robin scheduling is a simple and stable way to guarantee determinism. However, round-robin scheduling may serialize computations. To compensate for this weakness, *soft barriers* [4] are added to align time-consuming computations, preventing them from serialization.

The combination of round-robin scheduling and *soft barrier* is effective on data partition parallelism, in which worker threads execute the same computation on different data partitions. However, when it comes to pipeline parallelism, *soft barrier* performs poorly. It is difficult to statically align threads from different stages.

In this paper, we present a Load-Balanced Deterministic Runtime (LBDR) for pipeline parallelism. Like the method in [4], LBDR orders synchronization operations in a round-robin manner by default. To address the load imbalance caused by default scheduler, we introduce two techniques. The first technique forces a thread to give up its tokens when entering time-consuming synchronization-free sections. The second technique eliminates token operations (token acquisitions and releases) when a thread is about to pass token to itself.

2. Overview

This section first illustrates the poor performance of round-robin scheduling on pipeline parallelism using an example, and then gives an overview of our approaches.

Parrot is a state-of-the-art round-robin deterministic runtime. We study the behavior of Parrot on *ferret*, a pipeline parallel program from the PARSEC benchmark suite [1]. Figure 1 shows the simplified version of *ferret*. We run *ferret* with 4 threads in each parallel stage. Figure 2 shows part of the schedule we observed. Thread *T1* is assigned with *t_load* stage, while thread *T2* is assigned with *t_seg* stage. There are two sources of inefficiency in Fig. 2. First, *T2* calls a time-consuming synchronization-free function *image_segment()* while holding the token. At the same time, *T1* is blocked at the synchronization point waiting for the token. To address this inefficiency, we add a programmer annotation at line 10. Line 10 informs LBDR that the thread is entering a *Synchronization-Free Section* (SFS). During this SFS, the thread will give up the token

Manuscript received August 22, 2014.
Manuscript publicized October 21, 2014.

[†]The authors are with National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, PR China.
[††]The authors are with the School of Computer, National University of Defense Technology, Changsha, PR China.
a) E-mail: chenchen2011@nudt.edu.cn

DOI: 10.1587/transinf.2014EDL8171

```
1 : void* t_load(char* dir) {
2 :     foreach image in dir do {
3 :         data n = read_image(image);
4 :         queue_enqueue_wait(&q_load_seq,n);
5 :     }
6 : }
7 : void* t_seg() {
8 :     while(1) {
9 :         queue_dequeue_wait(&q_load_seg, &seg);
10:         give_up_turn(0); /* performance hint */
11:         image_segment(&seg);
12:         queue_enqueue_wait(&q_seg_extract, seg);
13:     }
14: }
```

Fig. 1 Simplified *ferret*. It contains 6 stages, 2 of which are shown. *t_load* is the input stage, which reads image from disk and then enqueues the items into *q_load_seq* queue. *t_seg* stage dequeues items from *q_load_seq* queue, segments and then enqueues them into queue *q_seq_extract*.

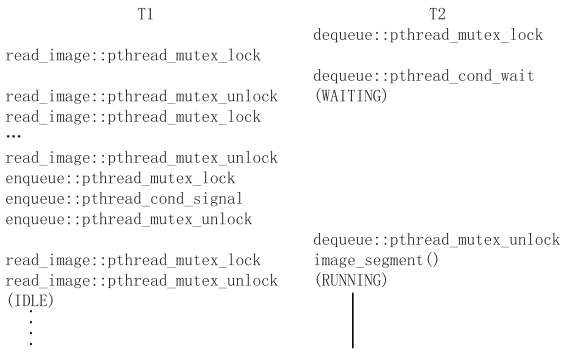

Fig. 2 A round-robin schedule. *T2* holds the token when executes *image_segment*, results to *T1* being blocked at *pthread_mutex_unlock*.

assigned to it.

The second source of inefficiency in Fig. 2 is that *T2* is blocked at *pthread_cond_wait* waiting for *T1* to enqueue an item. Compared to *T2*, Thread *T1* is slow because function *read_image()* makes intensive use of locks. What's worse, the slowdown is magnified by token operations (every synchronization is wrapped by a token acquisition and a token release). We present an optimization called *Single-Threaded Section* (STS) to eliminate unnecessary token operations. STS indicates that there is only one active thread. When a STS is dynamically detected, our runtime bypasses token operations to reduce overhead. Determinism is preserved since the bypassing does not change synchronization scheduling.

After using the above two techniques on baseline round-robin scheduler, the schedule changes to the one in Fig. 3, in which there are one SFS and two STSs. When *T2* enters the SFS, *T2* gives up the token so that *T1* successfully executes synchronization operation without blocking. Furthermore, *T1* becomes the only candidate for the token, and then enters a STS, even though *T2* is running (assuming that other threads are all blocked). Another STS is detected when *T2* is blocked at *pthread_cond_wait* and *T1* is the only candidate for the token.

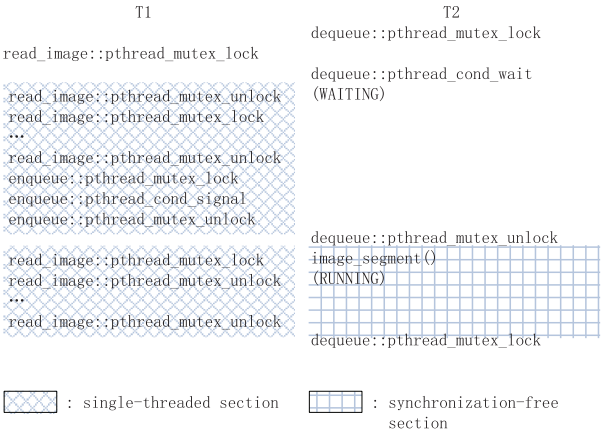

Fig. 3 A schedule after two techniques enforced.

3. Load-Balanced Deterministic Runtime

In this section we present our techniques and illustrate how these techniques improve the load balance of round-robin scheduling.

3.1 Synchronization-Free Section

We provide a performance hint for developers to start *Synchronization-Free Section* (SFS). SFS is used to prevent thread from holding token for a long time without executing any synchronization operation. Thread must hold the token when starting a SFS. Then it will give up the turn and quit the queuing up for token. When a thread enters a SFS, token scheduler ignores the thread, even it is running. The interface is like below:

```
void give_up_turn(int hint_id);
```

hint_id is the identity of the SFS. Each SFS should be assigned with an unique identity.

Developers should generally add *give_up_turn* hints at the beginning of time-consuming synchronization-free computations. One way to determine the starting point of a SFS is performance profiling. We execute programs with a set of representative inputs and log the round-robin schedules. From the logs, we identify the most time-consuming sections during which tokens are held by a thread without executing any synchronization operation. The starting points of these sections are determined as the insertion points of *give_up_turn* hints.

The exit points of SFSs are conducted by LBDR dynamically. It is error-prone for developers to decide the exit points of SFSs. These exit points should cover all the execution paths started from the entry of the SFS. Furthermore, all the paths should be synchronization-free, which is hard to determine statically in the presence of external libraries.

Instead of using explicit exit points, LBDR maintains a budget for each SFS. A SFS's budget is the amount of synchronization operations the other threads should execute

during this SFS. Once the budget is exhausted, the SFS exits. Thus, budgets must be deterministic. A simple way to guarantee determinism is assigning same budget for every SFS. However, setting up a reasonable budget is critical to performance. If a budget is too small, the thread may still hold the token without executing synchronization operation. If a budget is too large, the thread will block at the end of the SFS waiting for other threads to exhaust the budget.

LBDR computes budgets in a deterministic and adaptive manner. Figure 4 shows the algorithm. Theoretically, a thread T's budget represents the amount of synchronization operations the other threads execute when T executes one synchronization operation (line 10). However, this budget is often small, as SFSs are likely to be longer than average interval of two synchronization points. We design an adaptive supplementary method for budgets (line 11). At the exit of a SFS, LBDR increases the supplement if the budget is small (line 15-17). The increasing of supplement is deterministic. LBDR employs deterministic performance counters to determine if a budget is small or not. From the start to the end of a SFS, if the SFS thread executes more instructions than any other active threads, namely other threads are blocked early, then we determine that the budget is small.

3.2 Single-Threaded Section

We define *Single-Threaded Section* (STS) as an execution section that there is only one thread available for token passing, namely the current token-owner thread will pass token to itself repeatedly. In STSs, LBDR directly executes synchronization operations without acquiring for token.

STSs are detected by LBDR dynamically. LBDR maintains a *run* queue and a *wait* queue for threads. Once a thread is blocked, it is transferred from *run* queue to *wait* queue. When there is only one thread in the *run* queue, a STS is determined. A STS is ended whenever a thread is backed into *run* queue.

STSs do not violate determinism. First, STSs are started when only one thread is available for token. Since threads are always blocked at synchronization points, so the logical time when a STS starts is deterministic. Second, a STS ends when a thread is signaled by some synchronization operations, so the timing only depends on logical time. Third, removing token passing in STSs only reduce the real time, not the logical time. When a synchronization operation is executed in a STS, LBDR also increases the logical time. STSs do not change the result of token passing, because token operations are eliminated when provably redundant.

```
1: SyncFreeStart (int h_id) {
2:     //total num of sync opts executed by currently
3:     //active threads
4:     sumSync = 0;
5:     //the stage set of currently active threads
6:     activeStageSet.clear();
7:     foreach active thread t
8:         sumSync += t->logicClock;
9:         activeStageSet.insert(t->stage_id);
10:    budget = sumSync / self()->logicClock +
11:             supplement[h_id][activeStageSet];
12:}
13:
14:SyncFreeEnd (int h_id) {
15:    if budget is small
16:        supplement[h_id][activeStageSet] +=
17:            budget / 10;
18:}
```

Fig. 4 Entry and exit of the synchronization-free section.

4. Experiment and Analysis

We evaluated LBDR on two pipeline parallel programs: *ferret* and *dedup* from the PARSEC benchmark suite [1]. We excluded a pipeline program *bzip2* [7] because LBDR dose not yet support Intel Threading Building Blocks (TBB). Our evaluation machine was a 2.60GHz dual-socket hex-core Intel Xeon with 32 hyper-threading cores and 126GB memory running Linux 2.6.32. The *native* inputs for PARSEC were used.

4.1 Performance Hints

The program *ferret* only needs one *give_up_turn* hint while *dedup* needs two. Table 1 shows the locations at where hints been added. It is straightforward to add *give_up_turn* hints in pipeline parallelism. Even without performance profiling, developers can easily add *give_up_turn* hints at following locations.

- Right before the return of enqueue().
- Right before the return of dequeue().
- Right before the calling of time-consuming functions.

In our experiments, adding *give_up_turn* hints at wrong locations incurs negligible runtime overhead. For example, developers may add *give_up_turn* hints before the section which is neither synchronization-free nor time-consuming. Since the budget of a SFS is dynamically computed, fake SFSs always get small budgets.

4.2 Performance

We compare the performance of LBDR to Parrot, default round-robin scheduling and *pthreads*. Figure 5 presents the results (normalized to *pthreads*). For *ferret*, LBDR outperforms Parrot and round-robin scheduling by an average of 26.5% and 48.1%, respectively. For *dedup*, LBDR outperforms Parrot and round-robin scheduling by an average of 18.5%. Although Parrot reduces the overhead of round-robin scheduling from 4.6x to 3.0x by inserting

Table 1 Locations of *give_up_turn* hints.

benchmark	location
ferret	queue.h:129, Right before the return of *dequeue()*.
dedup	enqueue.c:85, Right before the return of *enqueue()*.
dedup	enqueue.c:136, Right before *compress()*.

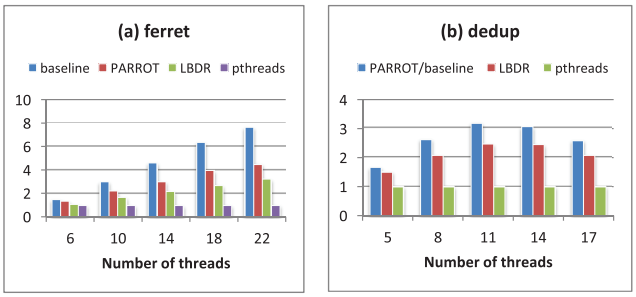

Fig. 5 Normalized execution time with respect to *pthreads*. PARROT uses no performance hint on *dedup*, so its performance on *dedup* is the same as round-robin scheduling. (a) ferret; (b) dedup.

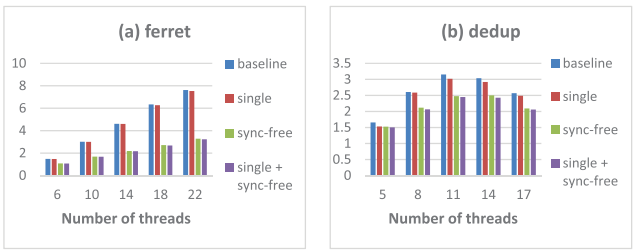

Fig. 6 Normalized execution time with different sets of optimizations. (a) ferret; (b) dedup.

Table 2 Proportion of synchronization operations executed in *single-threaded sections* and the execution time reduced by *single-threaded sections*.

benchmark	proportion	time saving (s)
ferret	98.3%	3.0
dedup	23.9%	1.6

a *soft barrier* in *ferret*, *soft barriers* cannot alleviate dynamic load imbalance. LBDR achieves good load balance by taking token from non-synchronization-intensive threads to synchronization-intensive threads. Note that *soft barrier* is orthogonal to the optimizations presented in this article. Although currently LBDR has not employ *soft barrier*, we believe that PARROT and LBDR are complementary.

We also evaluated how two optimizations reduce LBDR's overhead. Figure 6 shows the effects of different sets of optimizations. *Synchronization-free section* is very effective at reducing overhead. *Single-threaded section* does not help *ferret* much, but it does help for *dedup*. Table 2 shows the proportion of synchronization operations executed in *single-threaded sections*. Although LBDR accelerates 98.3% of synchronization operations, the overhead of token passing is not the main source of inefficiency on *ferret*.

4.3 Scalability

To measure the scalability of LBDR, we ran our benchmarks with different core counts and different workload scales. LBDR scales better than PARROT for all workload scales. PARROT's *soft barrier* needs global synchronization, which limits the scalability of PARROT. Threads in LBDR exit from *synchronization-free sections* based on their own history information, thus LBDR does not need global synchronization.

5. Conclusion

In this paper, we propose LBDR, a load-balanced deterministic runtime for pipeline parallelism. By default, LBDR schedules synchronization in a round-robin fashion. To address the load imbalance of default scheduling, LBDR employs two critical improvements. *Synchronization-free section* prevents thread from holding token for a long time without executing synchronization operation. *Single-threaded section* removes token passing when a thread is about to pass token to itself. Experiments show that LBDR outperforms prior deterministic runtime on pipeline parallelism.

Acknowledgments

This work is partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA01A301 and 2012AA010901, by program for New Century Excellent Talents in University and by National Science Foundation (NSF) China 61272142, 61103082, 61170261, 61103193 and 614024992.

References

[1] C. Bienia, S. Kumar, J.P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," Proc. Intl Conf. Parallel Architectures and Compilation Techniques, PACT, pp.72–81, 2008.
[2] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient deterministic multithreading in software," Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.97–108, 2009.
[3] K. Lu, X. Zhou, T. Bergan, and X. Wang, "Efficient deterministic multithreading without global barriers," Proc. PPOPP, pp.287–300, 2014.
[4] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G.A. Gibson, and R.E. Bryant, "Parrot: A practical runtime for deterministic, stable, and reliable threads," Proc. Twenty-Fourth ACM Symposium on Operating Systems Principles, pp.388–405, 2013.
[5] T. Liu, C. Curtsinger, and E.D. Berger, "DTHREADS: Efficient deterministic multithreading," Proc. 22nd ACM Symposium on Operating Systems Principles, pp.327–336, 2011.
[6] T. Merryfield and J. Eriksson, "Conversion: Multi-version concurrency control for main memory segments," Proc. EuroSys, pp.127–139, 2013.
[7] Intel, Source code for Intel threading building blocks, http://www.threadingbuildingblocks.org/, 2009.