# Understanding and Reaching the Performance Limit of Schedule Tuning on Stable Synchronization Determinism


Qi Zhao*
Department of Computer Science
North Carolina State University
Raleigh, NC, United States
qzhao6@ncsu.edu

Zhengyi Qiu
Department of Computer Science
North Carolina State University
Raleigh, NC, United States
zqiu2@ncsu.edu

Shudi Shao
Department of Computer Science
North Carolina State University
Raleigh, NC, United States
sshao@ncsu.edu

Xinning Hui
Department of Computer Science
North Carolina State University
Raleigh, NC, United States
xhui@ncsu.edu

Hassan Ali Khan
Department of Computer Science
North Carolina State University
Raleigh, NC, United States
hakhan@ncsu.edu

Guoliang Jin
Department of Computer Science
North Carolina State University
Raleigh, NC, United States
guoliang_jin@ncsu.edu



## ABSTRACT

Deterministic MultiThreading (DMT) systems eliminate nondeterminism from the dynamic executions of multithreaded programs. They can greatly simplify multithreaded programming and ease the deployment of systems that rely on replication. We first categorize and compare existing DMT system designs along three axes, incorporating the most recent advances in DMT systems. From our study, we conclude that stable synchronization determinism is the most cost-effective design, and it is thus the focus of our work.

To reduce the overhead of enforcing stable synchronization determinism, previous work has explored scheduling-based methods that tune the synchronization schedule. However, it is not clear how low the performance overhead can be through schedule tuning and how to reach the performance limit. To answer these questions, we then follow an iterative process of understanding the performance limit of schedule tuning on stable synchronization determinism and designing new scheduling policies to reach the performance limit. Through this process, we identify two types of scheduling-oblivious overheads that cannot be eliminated by schedule tuning alone. In addition, we also design a group of new policies and implement them in minSMT.

Our evaluation shows that minSMT successfully reaches the performance limit of stable synchronization determinism on 107 out of 108 benchmarks after excluding the impact of scheduling-oblivious overheads, and this also results in significant performance improvements compared with state-of-the-art stable synchronization-determinism systems on 9 benchmarks. Our results also suggest that, to further improve the performance of stable synchronization determinism, future research should focus on addressing the two types of scheduling-oblivious overheads with approaches other than schedule tuning.



## CCS CONCEPTS

• **Software and its engineering** → **Software performance**; **Scheduling**; **Synchronization**; **Multithreading**; Software reliability; • **Computing methodologies** → Parallel computing methodologies.

## KEYWORDS

stable synchronization determinism, performance limit, synchronization scheduling, scheduling-oblivious overheads, totally-ordered synchronization, workload-length imbalance

**ACM Reference Format:**
Qi Zhao, Zhengyi Qiu, Shudi Shao, Xinning Hui, Hassan Ali Khan, and Guoliang Jin. 2022. Understanding and Reaching the Performance Limit of Schedule Tuning on Stable Synchronization Determinism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3559009.3569669


## 1 INTRODUCTION

Multithreaded programs are prevalent and critical for utilizing the computing power of multicore processors. However, multithreaded programs are nondeterministic by default, where threads can interleave differently in different executions as long as the resulting interleavings are compatible with program synchronization.

While such flexibility benefits performance, multithreaded programs can exhibit nondeterministic behaviors even with the same inputs, and such nondeterminism is undesirable from many perspectives. For example, the vast number of possible interleavings that an execution could run into nondeterministically makes it infeasible to guarantee all interleavings are free of concurrency bugs. As a result, concurrency bugs are common in many widely used multithreaded programs [46], and some have caused real-world disasters [42, 57]. As another example, nondeterminism makes it difficult for techniques that rely on replication, e.g., State Machine Replication (SMR), which is a powerful fault tolerance technique [40, 56], to take advantage of parallel hardware and scale to multi-core servers.

*Qi Zhao is also a full-time employee at Google at the time of publication, and this work is done at North Carolina State University where he is pursing his Ph.D. degree.


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.


*PACT '22, October 10–12, 2022, Chicago, IL, USA*


© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.


ACM ISBN 978-1-4503-9868-8/22/10…$15.00
https://doi.org/10.1145/3559009.3569669

The complexity of existing systems that tolerate nondeterministic executions [31, 39] imposes a significant barrier for their adoption in practice.

*Deterministic MultiThreading (DMT)* systems [15, 16, 18, 19, 26–30, 34, 44, 45, 48–50, 52, 63] address this nondeterminism issue by eliminating or reducing two main *internal* sources of nondeterminism, i.e., nondeterministic synchronization-operation orders and nondeterministic shared-memory-access orders. DMT systems can greatly simplify debugging, testing, verification, record-replay, and fault-tolerant replication of multithreaded programs [17, 22, 25, 26, 61]. As the benefits of DMT systems come at a performance cost, which could sometimes be significant, researchers have proposed different DMT system designs to explore the tradeoff between determinism guarantees and performance.

Depending on whether and how they eliminate the two internal sources of nondeterminism, different DMT systems provide different types of determinism guarantees and incur different performance overheads. As we will detail the categorization and comparison of existing DMT systems in the next section, stable synchronization determinism is the most cost-effective design among existing DMT systems, where *synchronization determinism* means that the DMT systems enforce a deterministic order only on synchronization operations, and *stable* means the DMT systems use the same *schedule*, i.e., the order of synchronization operations, on many different inputs and upon minor program changes [62].

With schedule stability, the synchronization schedule for one input will remain the same as long as synchronization code is not changed, and the synchronization schedules for various inputs will be the same as long as they make threads execute the same sequence of synchronization operations. Such a stability guarantee will allow programmers to make minor changes, such as adding `printf()` statements, for debugging without changing the schedule and ease the overall understanding of multithreaded programs. More importantly, schedule stability makes synchronization determinism effective by itself without enforcing the costly memory-access determinism [26, 63], making it the most cost-effective design.

Although stable synchronization determinism does not need extra heavyweight mechanisms associated with enforcing memory-access determinism, achieving good performance while enforcing stable synchronization determinism is still not trivial. Specifically, existing DMT systems that provide schedule stability all use round robin as the base of their scheduling policies, where threads are scheduled in a round-robin fashion to execute synchronization operations, but a vanilla round-robin policy can limit concurrency and make threads all serialized in the worst case [26, 63].

To solve the aforementioned problem and achieve good performance, state-of-the-art stable synchronization determinism systems differ in their round-robin-based scheduling policies. Parrot [26], the pioneering stable synchronization determinism system, provides two scheduling annotations that programmers can manually add to hint DMT systems on how to schedule certain code regions to achieve better performance. One recent work, QiThread [63], proposes scheduling policies leveraging synchronization semantics without requiring extra annotations. While these proposals improve the performance of stable synchronization determinism, it is not clear whether they have reached the performance limit of schedule tuning and any further improvement can be made.

In this paper, we aim to understand the performance limit of schedule tuning on stable synchronization determinism and build a stable synchronization determinism system that can reach the limit. Specifically, we strive to understand what types of overheads are *oblivious* to scheduling that cannot be addressed by tuning the schedule of synchronization operations. Meanwhile, we seek to build a stable synchronization determinism system that can reach the performance limit modulo the impact of scheduling-oblivious overheads by resolving as many remaining scheduling-dependent overheads as possible. We expect our results to help researchers determine whether they should keep spending more effort on addressing scheduling-dependent overheads as done in existing work or instead focus on addressing scheduling-oblivious overheads.

To fulfill our goal, we follow an iterative approach in design, implementation, and evaluation. We use all the 108 programs used for evaluating Parrot [26] and QiThread [63], which is the largest set of benchmarks used for evaluating DMT systems in the literature, and the set includes 55 real-world programs and 53 programs from four widely used benchmark suites. Among these benchmarks, we start from ones with a high overhead on state-of-the-art stable synchronization determinism systems, and we iteratively form hypotheses about possible types of scheduling-oblivious overheads and design new scheduling policies to address scheduling-dependent overheads. We stop when evaluation shows the new scheduling policies can help us reach the performance limit after excluding the impact of scheduling-oblivious overheads.

To this end, we identify two types of scheduling-oblivious overheads, namely *totally-ordered synchronization* and *workload-length imbalance*. Although Merrifield et al. proposed techniques to enforce a partial order on synchronization operations, the resulting DMT system is deeply coupled with memory-access determinism and is more costly than a stable DMT system enforcing synchronization determinism only. We will elaborate on why it is challenging to decouple partially-ordered synchronization with memory-access determinism and argue why we consider totally-ordered synchronization one type of scheduling-oblivious overhead. For workload-length imbalance, it is a new source of overhead that has not been discussed in the literature before, and it cannot be addressed by schedule tuning due to the requirement of providing schedule stability. We also design techniques to exclude their impact while evaluating the overhead of stable synchronization determinism systems. In the meantime, we develop several new policies and annotations to address the remaining scheduling-dependent overheads, and we implement them in minSMT.

Our evaluation using the large set of 108 benchmarks shows that minSMT reaches the performance limit of schedule tuning on stable synchronization determinism on all but one of the benchmarks. We also identify challenges that prevent us from determining whether minSMT reaches the performance limit on that remaining one. Comparison between minSMT and state-of-the-art stable synchronization determinism systems shows that minSMT results in significant performance improvement on 9 benchmarks without hurting the performance of any benchmarks significantly, and this also shows that state-of-the-art stable synchronization determinism systems fail to reach the performance limit on these 9 benchmarks.

Our approach iterates between the understanding of scheduling-oblivious overheads and the elimination of scheduling-dependent

overheads, and we consider this iterative approach the key enabler for us to design and implement a more performant stable synchronization determinism system that can directly benefit users, which is otherwise not achieved by state-of-the-art systems that consider schedule tuning alone. Our work will also be useful for the research community to decide where to spend research effort to further improve DMT systems. As minSMT still exhibits very high overheads on some benchmarks where it reaches the performance limit of schedule tuning, future research should focus on approaches beyond schedule tuning to address the two types of scheduling-oblivious overheads identified in this work.

Overall, the paper makes the following contributions:

- We present a detailed categorization of existing DMT systems based on three axes. To the best of our knowledge, this is the most up-to-date categorization of DMT systems beyond just the separation between strong determinism and weak determinism but also includes the most recent development on DMT systems.
- We use an iterative methodology to understand and reach the performance limit of schedule tuning on stable synchronization determinism, and we identify two types of scheduling-oblivious overheads and design minSMT to eliminate most of the remaining overheads after excluding scheduling-oblivious overheads. Our methodology allows us to gain insights and achieve performance improvement beyond the state-of-the-art.
- We conduct a thorough evaluation of our finding of the two types of scheduling-oblivious overheads and our design of minSMT on a diverse set of 108 benchmarks. Our results show that minSMT is more performant and reaches the performance limit on more benchmarks than state-of-the-art stable synchronization determinism systems. Our results also suggest that future research should focus on addressing scheduling-oblivious overheads.
- Our artifacts, including minSMT implementation and migrated Parrot, QiThread, and benchmarks for updated Linux kernel and gcc, are available on GitHub at https://github.com/chyiz/minSMT/.

## 2 RELATED WORK

We will discuss related work on DMT systems in the next section, where we provide a categorization of existing DMT runtime systems. In this section, we first discuss some related work that does not itself introduce a runtime DMT system.

Segulja and Abdelrahman analyzed the sources of nondeterminism while enforcing synchronization determinism and the cost imposed by the deterministic order while enforcing synchronization determinism [59]. Their goal was to measure the performance overhead that comes from the deterministic order itself but not from the mechanism that generates and enforces the order, and they used a schedule-record-replay framework to achieve their goal. Their work did not propose any solutions for performance improvement. We have a different goal and different focus. Our goal is to understand the performance limit of generating and enforcing synchronization determinism that is stable, but we do not separate the overhead of the schedule itself and the overhead of generating/enforcing the schedule. We further propose techniques to reach the performance limit, and our study and evaluation use 3× more benchmarks.

Other than DMT systems, researchers also proposed deterministic programming languages [20, 21, 23, 24, 32, 51, 55] and record-replay systems [14, 33, 35–37, 41, 43, 47, 53, 60] to address nondeterminism. While new languages are useful, DMT systems can support existing programs. Compared with record-replay systems, DMT systems actively control the execution of programs but not just record, and stable DMT systems can significantly shrink the schedule space, which is valuable for multithreaded programs [62].

## 3 CATEGORIZATION AND COMPARISON OF EXISTING DMT SYSTEMS

In this section, we categorize and compare existing DMT systems. While determinism can be enforced at different system levels, including hardware [29, 30, 34, 58], runtime systems [16, 19, 26, 28, 44, 45, 48, 49, 52], programming languages [23, 24, 51], and operating systems [15, 18, 38], we focus on DMT systems that enforce determinism at the runtime level, as such systems are the simplest to deploy and lessons learned from runtime DMT systems can also benefit DMT systems in other system levels.

### 3.1 Three Axes for Categorization

Based on whether and how runtime DMT systems eliminate the two internal sources of execution nondeterminism, i.e., nondeterministic orders on synchronization operations and shared-memory accesses, we categorize them along the following three axes. Note that the first axis is a classic one and the last two are the results of recent developments on DMT system concepts and designs.

***(1) Synchronization determinism only or with memory-access determinism.*** DMT systems were first categorized into just two types depending on whether they only enforce synchronization determinism or further enforce memory-access determinism. DMT systems that only enforce synchronization determinism are referred to as providing *weak determinism*, as programs with data races can still run into different interleavings given the same input. DMT systems that not only enforce synchronization determinism but also further enforce a deterministic order on shared-memory accesses, i.e., *memory-access determinism*, are referred to as providing *strong determinism*, as they guarantee the same schedule even for programs with data races.

However, the improvement of determinism guarantee in strong determinism systems comes with a much higher performance overhead compared with synchronization determinism systems, as enforcing memory-access determinism requires some costly mechanism, e.g., memory-address space isolation for threads [44, 48, 49].

***(2) Stable or not.*** Later, Yang et al. proposed the concept of *schedule stability* [62]. A DMT system with schedule stability makes schedules stable upon input changes, meaning that many inputs will share the same schedule, and it also makes schedules stable upon minor program changes, allowing programmers to add `printf()` statements for the purpose of debugging without changing the schedule. They argued and demonstrated using their prototype system, Parrot [26], that races are no longer a critical issue for DMT systems only enforcing synchronization determinism once they are made stable. They further built an SMR system, Crane [25], atop Parrot to show the feasibility of leveraging stable synchronization

**Table 1: The five types that existing DMT runtime systems can be categorized to and their representatives**

| | Total Order | | Partial Order | |
|---|---|---|---|---|
| | Stable | Non-Stable | Stable | Non-Stable |
| Synchronization Determinism Only | ①: Parrot [26], QiThread [63] | ②: Kendo [52] | - | - |
| With Memory-Access Determinism | ③: Dthreads [44] | ④: Consequence [48], CoreDet [16] | - | ⑤: LazyDet [50] |

determinism for implementing SMR systems. With these results, they established the usefulness of stable synchronization determinism and suggested that the level of schedule-space reduction is a better criterion to evaluate DMT systems [62].

All existing DMT systems use the same *turn-based mechanism* to enforce synchronization determinism, and the stability of schedules only depends on the *scheduling policy*. With the turn-based mechanism, only one thread owns the turn at any given time, and only the thread owning the turn is allowed to execute a synchronization operation. If a thread that does not own the turn wants to execute a synchronization operation, the thread will be blocked until the turn is passed to it. On top of the turn-based mechanism, the scheduling policy, which determines how the turn passes around threads, is based either on logical clock [52] or round robin. Logical-clock-based policies do not provide schedule stability, while round-robin-based policies provide schedule stability [26, 63].

Note that the combination of mechanism and policy described above is not only used by DMT systems that just enforce synchronization determinism, but it is also used by DMT systems that further enforce memory-access determinism. Although early strong determinism systems serialize threads on quantum boundaries, recent work [45, 48, 49] has demonstrated that memory-access determinism can be enforced without extra program serialization on top of the serialization caused by synchronization determinism enforcement [59, 63]. Therefore, synchronization determinism and memory-access determinism can be considered as two orthogonal tasks, and progresses made in synchronization determinism can also benefit strong determinism systems [63]. The schedule stability of such strong determinism systems only depends on how they enforce synchronization determinism. Although some early strong determinism systems are stable, they did not distill the concept of schedule stability. The early weak determinism system, Kendo [52], is not stable.

***(3) Total order or partial order.*** More recently, Merrifield et al. proposed to enforce a *partial order*, which only orders synchronization operations if they operate on some common synchronization variables, but not a *total order* regardless of variables, and implemented such a system, LazyDet [50]. Their motivation is to reduce the overhead of DMT systems for programs using fine-grained locks with frequent lock operations. Before LazyDet, all DMT systems enforced a total order on synchronization operations, and this can lead to a high performance overhead for such programs. With a partial order but not a total order on synchronization operations, the performance overhead can be reduced.

LazyDet enforces a partial order on synchronization operations through speculation, and it is enabled by thread-level memory-address isolation [50]. Since thread-level memory-address isolation also enables strong determinism, the partial order on synchronization operations is deeply coupled with strong determinism. Evaluation of LazyDet on a microbenchmark using a lot of fine-grained locks shows that it can reduce the overhead to be lower than only enforcing synchronization determinism with a total order. However, the execution time is still at least 16× of nondeterministic executions, the cost of which is partly due to the fact that LazyDet is deeply coupled with thread isolation and strong determinism. On real-world benchmarks, the performance benefit of LazyDet is marginal, as these programs use much fewer or no fine-grained locks and the speculation mechanism leads to extra overhead. Therefore, the performance overhead is often similar to a totally-ordered strong determinism system and higher than a DMT system that only enforces synchronization determinism on real-world benchmarks.

## 3.2 Categorization and Comparison

Table 1 categorizes existing DMT systems along the three axes. Only one system, i.e., LazyDet [50], enforces a partial order on synchronization operations. While it is generally not possible to predict future synchronization operations executed by other threads, LazyDet bypasses this challenge with speculation as mentioned earlier, which leverages the same thread-level address-space isolation mechanism for memory-access determinism. Therefore, the only existing DMT system that enforces a partial order on synchronization operations is deeply coupled with memory-access determinism, and there is no known approach that can enforce synchronization determinism only following a partial order without incurring the overhead associated with memory-access determinism. In Section 5.1, we will also argue in detail why it is challenging to design and implement a synchronization-determinism-only system with a partial order.

Figure 1 shows the relative comparison on cost and schedule-space reduction of these five types of DMT systems. Their relative positions on cost are based on performance numbers reported in their original papers, as well as how easy it is to deploy them. DMT systems of types ① and ② have similar performance costs as they both only enforce synchronization determinism, and they only need to change the thread library for deployment. DMT systems of types ③, ④, and ⑤ have similar performance costs as they all enforce strong determinism, and they require more system-level changes in the operating system or compiler for deployment.

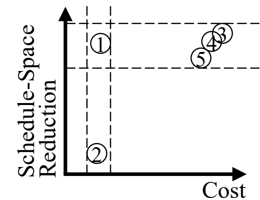



**Figure 1: The relative comparison on cost and schedule-space reduction**

The relative positions of different types of DMT systems on schedule-space reduction in Figure 1 are based on the following reasoning. Strong determinism reduces the schedule space further compared with non-stable synchronization determinism, as strong determinism deterministically resolves each race. On the other hand, stable synchronization determinism is comparable with strong determinism in its capability of schedule-space reduction.

This has been argued by Yang et al. [62]. The major takeaway is that although stable synchronization determinism does not necessarily provide the mapping from one input to one schedule while facing races as a strong determinism system does, it can still shrink the number of possible schedules by using a small number of schedules on all inputs. Further, synchronization determinism already greatly constrains the interleaving space and significantly reduces the number of remaining data races, e.g., Peregrine [28] reported at most 10 races in millions of shared memory accesses within an execution after enforcing synchronization determinism, and these races will not expand the number of possible schedules much. With schedule stability, the extra benefit of further enforcing memory-access determinism is marginal and not worth the cost.

From the categorization and comparison results above, we conclude that stable totally-ordered synchronization determinism is the most cost-effective DMT system design.

# 4 BENCHMARKS, PLATFORM, AND METHODOLOGY

In this section, we describe the benchmarks and platform we use, and we then present our methodology. As our work uses existing stable synchronization determinism systems, Parrot [26] and QiThread [63], as the starting point, we first provide the necessary background below.

## 4.1 Background on Parrot and QiThread

Parrot and QiThread share the same turn-based mechanism but differ in their scheduling policies for addressing the thread serialization problem caused by a vanilla round-robin policy. Below, we present the system architecture of Parrot, which is inherited by QiThread, and describe Parrot annotations and QiThread policies that are used to address the serialization problem. We only discuss the parts that are relevant for understanding the scheduling policies. For details on other aspects, please refer to the original papers of Parrot and QiThread.

*4.1.1 Overall Architecture.* Figure 2 shows the DMT system architecture shared by Parrot and QiThread. The major components include a deterministic user-space scheduler and a set of wrapper functions for intercepting pthreads synchronization, network, and timeout operations and handling scheduling annotations. The deterministic user-space scheduler only schedules synchronization operations, and it delegates everything else to the underlying nondeterministic OS-space scheduler. The wrapper functions interpose function calls to a library dynamically loaded through `LD_PRELOAD`, “trap” the function calls into the deterministic user-space scheduler, and then delegate the actual implementation to pthreads or the OS. Different scheduling policies are implemented in the deterministic user-space scheduler. We also leverage this DMT system architecture to implement minSMT.

*4.1.2 Parrot Scheduling Annotations.* Parrot provides two scheduling annotations that programmers can use to improve performance, i.e., a *soft barrier* and a *performance critical section.* The deterministic user-space scheduler treats these scheduling annotations as *performance hints*, as they are added for performance only and can be ignored without hurting program correctness.

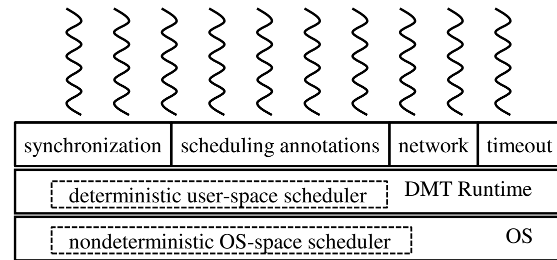



**Figure 2: The DMT system architecture shared by Parrot and QiThread**

The soft barrier is to help the deterministic scheduler to avoid serializing parallel computations. In general, the soft barrier should be used to align high-level, time-consuming computations that should be executed in parallel, and it encourages the scheduler to co-schedule a group of threads at the program point where the annotation is added. The soft barrier operates like a barrier with a deterministic timeout. Note that the timeout in the soft barrier is made deterministic in Parrot by counting the number of turns.

The performance critical section (PCS) annotation is to annotate a performance bottleneck when its overhead is so significant that it is worthwhile to trade determinism for performance. Once a thread enters a section annotated with PCS, the runtime system will delegate the thread to the nondeterministic OS scheduler. Since the PCS annotation trades determinism for performance, it should be used with caution.

*4.1.3 QiThread Scheduling Policies.* QiThread addresses the potential serialization problem of the vanilla round-robin policy with heuristic-based scheduling policies, and these policies leverage synchronization semantics to schedule various synchronization operations differently. The performance goal of QiThread is to achieve similar performance as Parrot with the soft barrier but without the PCS annotation. Evaluation shows that QiThread achieved its goal without adding Parrot annotations and sometimes outperforms Parrot with soft barrier only, as QiThread can better align threads beyond just those high-level, time-consuming computations. Specifically, QiThread introduced the following five semantics-aware scheduling policies:

**BoostBlocked** When a thread unblocks some other threads during a turn, the threads that were just unblocked will get the turn before the threads that are currently running.

**CreateAll** A thread executing `pthread_create()` in a loop will be able to finish the whole loop in one turn if the loop does not contain other synchronization.

**CSWhole** A critical section will be scheduled as a whole, i.e., a thread entering a critical section with the turn will pass the turn only when it leaves the critical section.

**WakeAMAP** A thread executing an unblocking operation on condition variables or semaphores will keep the turn and continue executing until there are no more threads to be woken up on the same condition variable or semaphore, or until the unblocking thread gets blocked itself.

**BranchedWake** This is an annotation that can be added in conditional statements where only some branches contain synchronization operations to balance the number of synchronization operations in all branches.

## 4.2 Benchmarks and Platform

We use all the 108 programs used to evaluate Parrot and QiThread in our work, which is the largest set of benchmarks used for evaluating DMT systems in the literature. This set covers a good range of parallel programming models and idioms, and it uses complete software or benchmark suites to avoid potential biases in the evaluation. This set of programs includes 55 real-world programs, and they are aget [1], Berkeley DB [8], MPlayer MEncoder [6], OpenLDAP [11], PBZip2 [5], pfscan [2], Redis [12], all 14 parallel image processing utilities implemented in OpenMP from the ImageMagick software suite [9], and all 33 parallel C++ STL algorithm implementations [7] also implemented in OpenMP. Since compression and decompression in PBZip2 take very different code paths, PBZip2 is counted twice. The other 53 programs are from four widely used benchmark suites, and they are 10 benchmarks in NPB [10]; 15 in PARSEC [4]; 14 in Phoenix [54], where each algorithm has two implementations, one using pthreads directly and the other using a map-reduce library built atop pthreads; and 14 in SPLASH-2x [3]. The individual names of the 108 programs can be found in Figure 5.

We use a Dell Precision 5810 Tower workstation with a single Intel Xeon E5-2695 V4 CPU and 64 GB memory. The CPU has 18 physical cores and 36 hyper-threading cores. Our machine configuration is comparable with the ones used for evaluating Parrot and QiThread. On the software side, while we inherit many settings, e.g., program workload, number of threads, and optimization level, made available through Parrot and QiThread artifacts, our machine is running Ubuntu 20.04 with Linux kernel version 5.4.0, which requires us to migrate the public versions of Parrot and QiThread available on GitHub to Ubuntu 20.04. During this process, we also upgrade the PARSEC benchmark suite used for evaluation from 2.0 to 3.0 and migrate necessary annotations added by Parrot. All these artifacts, including the migrated Parrot and QiThread and our own minSMT, are made available on GitHub [13].

## 4.3 Methodology

We follow an iterative process to understand overheads that are oblivious to schedule tuning and design techniques to address the remaining, scheduling-dependent overheads to reach the performance limit of schedule tuning.

We first run programs in three environments, i.e., nondeterministic execution, under Parrot, and under QiThread. Starting from programs with a significant overhead under a DMT system compared with nondeterministic executions, we compare their synchronization schedules to form hypotheses of possible scheduling-oblivious overheads that cannot be eliminated by schedule tuning. During this process, we also leverage our understanding of existing DMT systems and insights from our categorization and comparison results. To verify the hypotheses, we develop techniques to exclude the impact of scheduling-oblivious overheads while evaluating the performance overhead. Meanwhile, we try to address the remaining overhead with new scheduling policies. If new policies are not sufficient, it is possible that we are missing some scheduling-oblivious overheads or need better policies.

We repeat this iterative process until our new scheduling policies are able to achieve similar performance as nondeterministic executions after excluding scheduling-oblivious overheads. We consider a program is a resolved case if our DMT system with new policies has an extra overhead of no more than 3% after excluding scheduling-oblivious overheads. To this end, we find two types of scheduling-oblivious overheads with our new scheduling policies and annotation implemented in minSMT.

In the next two sections, we present the scheduling-oblivious overheads and our new scheduling policies and annotations.

# 5 SCHEDULING-OBLIVIOUS OVERHEADS

Through our iterative process, we find two types of scheduling-oblivious overheads while enforcing stable synchronization determinism. Below, we describe these two types of overheads, discuss why they cannot be eliminated by schedule tuning, and design techniques to exclude their impact while evaluating performance.

## 5.1 Totally-Ordered Synchronization

We consider totally-ordered synchronization as a scheduling-oblivious overhead for stable synchronization determinism. From Section 3, we can see that the only DMT system that enforces a partial order is deeply coupled with memory-access determinism. To decouple partially-ordered synchronization with memory-access determinism, a DMT system will not be able to adopt the speculation strategy as done by LazyDet, but it needs the capability of predicting future synchronization operations that may be executed by other threads.

We next detail the need for such a predicting capability under both logical-clock-based and round-robin-based policies. In the case of a logical-clock-based policy, according to the partial order, a thread could proceed to execute a synchronization as long as it knows that it has the smallest logical clock among threads using the same lock, but it is not necessary for the thread to have the globally smallest logical clock given the goal to enforce a partial order. In order to know this, the scheduler needs to predict what other threads may execute from the time the thread wants to execute a synchronization operation to the point when its logical clock becomes the globally smallest. In the case of a round-robin-based policy, each lock should be managed independently with its own queue. When a DMT system passes a lock-specific turn to the next thread, it needs to predict which thread will execute a synchronization operation on the lock bound to the queue. As pointed out by the authors of LazyDet, the capability of predicting future synchronization operations that may be executed by other threads is infeasible [50]. To put this argument more formally, we leverage the halting problem.

Considering a program with three threads, where all three threads acquire/release the same lock, we put the halting problem before the lock-acquisition statement in the second thread. At the moment when the first thread executes the lock-release statement, if a stable synchronization determinism scheduler that enforces partial order wants to determine which thread to pass the turn to, the scheduler needs to solve the halting problem to decide whether the turn

should be given to the second or the third thread. In this way, we reduce the problem of predicting future synchronization operations in other threads to the halting problem.

With the discussion above, we conclude that a general scheduler that can predict what synchronization operations other threads may execute and decide which thread to pass the turn to does not exist, and we consider totally-ordered synchronization as an overhead that is oblivious to schedule tuning. With totally-ordered synchronization, all synchronization operations will execute serially with a total order. Reflecting this to DMT system implementations, this is because of the turn-based mechanism, where only one thread is allowed to execute synchronization code at any given time.

Although we cannot eliminate the performance impact of totally-ordered synchronization through schedule tuning, we can instead measure this performance impact. Specifically, we develop a runtime library that executes each synchronization operation in a critical section protected by the same global lock, under which all synchronization operations are Totally-Ordered but with a NonDeterministic order, and we name it as the TONDSync mode. During performance evaluation, the overhead of programs running under the TONDSync mode will be subtracted from the overhead of running under a stable synchronization determinism system. In this way, we exclude the performance impact of totally-ordered synchronization, as the subtracted performance overhead represents the impact of totally-ordered but nondeterministic synchronization, and the delta of performance overheads between a stable synchronization determinism system and the TONDSync mode is caused by the different schedules of synchronization operations.

Fine-grained locks will further enlarge the performance impact of totally-ordered synchronization, regardless of how lock acquisition and release statements are scheduled. Under QiThread, which applies the CSWhole policy by default, all critical sections protected by different fine-grained locks are essentially now protected by one single lock, as only one critical section can execute at any given time. However, even if we turn off the CSWhole policy, the performance may not get much better. Without CSWhole, a thread that has acquired one fine-grained lock has to wait for the turn to pass around all other threads before it can execute the lock-release statement. As critical sections protected by fine-grained locks are usually very short, the lock-release statement will then need to wait for quite some time to get the turn, which is again a significant overhead. To exclude the performance impact of totally-ordered synchronization enlarged by fine-grained locks, we provide an annotation, Critical Section protected with Fine-grained lock (CSFine), to mark critical sections protected with fine-grained locks. Lock acquisition and release statements inside critical sections marked by CSFine can be viewed as not intercepted by the runtime.

## 5.2 Workload-Length Imbalance

Another type of scheduling-oblivious overhead for stable synchronization determinism is workload-length imbalance. While the papers of Kendo, Parrot and QiThread have used the term "load (im)balance," they used it to refer to the phenomenon that different threads execute different numbers of tasks. While such load imbalance can be addressed by schedule tuning, as exemplified by the aforementioned papers, the overhead caused by workload-length imbalance cannot, and hence a scheduling-oblivious overhead. Workload-length imbalance in the context of stable synchronization determinism is also different from CoreDet's "quantum length imbalance" problem, where a "quantum" is a time slice of arbitrary length determined by CoreDet [16], while workload length refers to the code region between two synchronization operations.

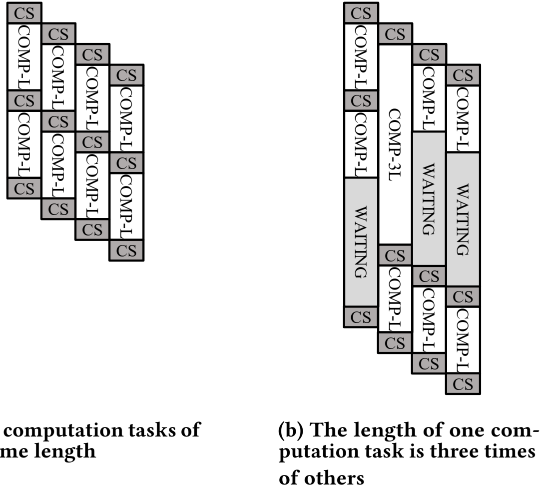
```
1  void *thread_entry(void *args)
2  {
3    int id;
4    long local_sum = 0;
5    while (1) {
6      pthread_mutex_lock(&mutex);
7      if (task_idx < task_total)
8        id = task_idx++;
9      else {
10       pthread_mutex_unlock(&mutex);
11       break;
12     }
13     pthread_mutex_unlock(&mutex);
14
15     long iter_num = task_len[id];
16     for (int i = 0; i < iter_num; i++)
17       local_sum += i*i*i;
18   }
19   ...
20   return NULL;
21 }
```

**Figure 3: An example illustrating workload-length imbalance**

Figure 3 shows an example to illustrate the concept of workload-length imbalance. In this example, each thread first enters a critical section (lines 6 to 13) to get an index and store it in variable `id` (line 8), which is then used to index into the `task_len` array to get `iter_num` (line 15). With QiThread's CSWhole policy, the two synchronization operations for lock acquisition and release will be scheduled in one turn, and threads in the program will execute the critical section in a round-robin fashion. After getting a task from the array, each thread then executes its computation, the length of which is specified by the value of `iter_num`.

The `task_len` array is initialized based on an input file, which essentially stores some pre-initialized task lengths. When all task lengths are initialized with the same value $L$, the overhead of running the program configured with 32 threads under QiThread on our platform compared with nondeterministic execution under vanilla pthreads is 13.06%. This overhead is not close to 0% as the execution time of the **for** loop (lines 16 and 17) still varies even with the same `iter_num`. We refrain from using more complex code to achieve the exact same execution time for the simplicity of our example. When we initialize the task lengths with some imbalance, the overhead will increase. For example, if we initialize each task length with two different values $L$ and $3L$ with a probability of 15/16 and 1/16 respectively, the overhead is 136.04%.

Figure 4 uses four threads to illustrate why the overhead increases with workload-length imbalance. Note that in our measurement the length of the computation and the length of the critical section is comparable. Since QiThread schedules each critical section as a whole in a round-robin fashion, the first computation task in the second thread, which is longer than others, leads to extra waiting in the other three threads.




(a) All computation tasks of the same length

(b) The length of one computation task is three times of others

**Figure 4: Illustration of how workload-length imbalance leads to a larger overhead**

This overhead is scheduling oblivious under the context of stable synchronization determinism, and that is because if the synchronization scheduler takes execution time into account, the resulting DMT system will be non-stable. Previous logical-clock-based DMT systems [16, 52] discussed this overhead, and argued that logical-clock-based policies would reduce overhead when encountered execution patterns similar to this workload-length imbalance. Indeed, if we use logical-clock-based scheduling policies on the example in Figure 4, and the logical-clock is well-established so that it tracks the real execution time closely, we may not suffer from the problem of workload-length imbalance, but they will allow all permutations of thread orders and result in many different schedules with different program inputs. To provide schedule stability, we would like the DMT system to use the same schedule under different inputs as long as the synchronization operations executed by each thread remain the same. Therefore, a stable synchronization determinism system will use the same schedule for different inputs for the example in Figure 4. This results in an overhead caused by workload-length imbalance, but the number of schedules becomes one and is significantly reduced. Therefore, we can consider schedule stability as a trade-off between performance and reliability, and workload-length imbalance is a scheduling-oblivious overhead as the result of providing schedule stability.

Essentially, as stable synchronization determinism aims to use the same schedule for many different inputs, it will naturally encounter inputs with different levels of workload-length imbalance, which requires different strategies to optimize. As programmers generally aim to divide workload evenly, workloads without length imbalance are the most general case. If we hard-code an optimized schedule for the input with workload-length imbalance, the schedule will then lead to extra overhead for inputs without workload-length imbalance. Together with the challenge of recognizing imbalanced workload length patterns, both Parrot and QiThread currently optimize for workloads without length imbalance, making workload-length imbalance a scheduling-oblivious overhead of stable synchronization determinism.

To exclude the performance impact of workload-length imbalance while evaluating the overhead of stable synchronization determinism systems, we will first use the synchronization logs generated by the runtime system to recognize major computation tasks leveraging the timestamps of synchronization operations. We then profile the length of these computation tasks to get the maximum length for each computation task. Finally, we use a runtime library to add delays to the end of each dynamic computation-task instance and make all instances of the same task the same length, i.e., 1.1× of the maximum length previously profiled. During evaluation, such delays will be added to all execution environments being compared when we need to exclude the impact of workload-length imbalance.

For the example in Figure 3, the profiled maximum length will be different when the two different inputs are used. In either case, once we insert delay as described above, the program running under QiThread has a negligible, close to 0% overhead compared with nondeterministic execution. Note that the delay will be added for both nondeterministic execution and deterministic execution.

# 6 MINSMT DESIGN AND IMPLEMENTATION

While trying to understand the performance overheads after excluding the two scheduling-oblivious ones, we find a few benchmarks not reaching the performance limit under QiThread. This guides our efforts in analyzing these benchmarks and identifying more opportunities to better align threads with one more scheduling policy and one more scheduling annotation. We also identify a minor optimization over QiThread. Next, we present our proposed policies and annotations and describe our implementation, minSMT.

## 6.1 New Policies and Annotation

***WakeAMAP+: An Improved WakeAMAP.*** As described in Section 4.1.3, the WakeAMAP policy in QiThread allows a thread that performs a wake-up operation on a condition variable or semaphore to continue holding the turn and wake up as many blocked threads on the same synchronization variable as possible. This policy is useful, especially in programs that use producer-consumer queues. In our study, we find that WakeAMAP is not able to cover the following two scenarios while aligning consumer threads, and our improved policy, WakeAMAP+, fixes such deficiencies.

First, different consumers could wait on different synchronization variables, i.e., there is an array of condition variables, and each of them is used to synchronize one consumer with the producers. In a program where these consumers are blocked on the same static statement but on different synchronization variables, WakeAMAP+ will allow the producers to keep the turn and continue execution to wake up as many of those blocked consumers as possible. The multiple consumers being woken up together are better aligned.

Secondly, there could be multiple producers in a producer-consumer queue, and WakeAMAP+ will let all the producers take turns to wake up consumers. Then, once a producer executes a woken-up operation, WakeAMAP+ will pass the turn around producers. Once there is no more consumer that can be woken up with the same static statement executed by different producer threads, the turn will then be passed around all threads as normal.

In both scenarios described above, to recognize all producers, minSMT will use the child thread entry function pointer passed

to `pthread_create()` as the role of the thread being created. In the case where a thread pool is used, we provide an annotation to recognize the address of the function being executed in a thread. With the role information, producers are threads sharing the same role as a thread that just previously executed a woken-up operation.

***An Extended-Turn Annotation.*** An extended turn identifies a code region containing multiple synchronization operations, the group of which when scheduled as a whole can improve performance. In Figure 3, each thread contains a loop that first gets a task and processes the task in each iteration. Now imagine a different program with the same structure but the logic to get a task involves a varied number of synchronization operations. Then the computation serialization problem could happen. With the extended-turn annotation, we ask the scheduler to execute the code logic that gets a task in one turn, and this will solve the serialization problem.

The extended-turn annotation essentially groups multiple synchronization operations that are executed in a high-level operation, e.g., getting a task from a synchronized queue, and treats the high-level operation as one synchronization operation regardless of how many synchronization operations are actually executed. This annotation is very useful to align threads executing code that contains a computation-intensive loop and each iteration in the loop contains a mixture of computation and synchronization.

***NoChildNoDMT.*** As its name suggests, the NoChildNoDMT policy turns on determinism enforcement only after the main thread creates the first child thread. While this is a straightforward policy, it is not included in Parrot and QiThread, and some of the 108 benchmarks can benefit from this policy significantly. These benchmarks have a very large number of synchronization initialization statements that are intercepted before any child threads are created, and their execution times are very short.

### 6.2 Implementation

minSMT is implemented on top of QiThread, which is in turn implemented on top of Parrot. As QiThread is implemented on top of Parrot and keeps all the functionalities of Parrot, QiThread can be viewed as a superset of Parrot, and minSMT also keeps all the functionalities of Parrot and QiThread. Options for turning on and off different policies and functionalities can be controlled through a configuration file.

The delta between minSMT and QiThread includes three major pieces: (1) the new WakeAMAP+ policy, which is implemented following QiThread's WakeAMAP policy but with extra code to recognize thread roles and track unblocking and blocking threads on condition variables and semaphores; (2) the extended-turn annotation provides two APIs, `ext_turn_begin()` and `ext_turn_end()`, to mark the code region to be executed in the extended turn, and it also has extra code in `get_turn()` and `put_turn()`, which are two interfaces inherited from Parrot, when the extended-turn annotation is encountered; (3) turning on determinism synchronization enforcement on the first `pthread_create()`.

For the purpose of excluding scheduling-oblivious overheads, minSMT also includes the CSFine annotation for marking critical sections protected by fine-grained locks and three more modes: (1) TONDSync mode, where the runtime only provides totally-ordered nondeterministic synchronization by executing each synchronization operation in a critical section protected by the same global lock; (2) a profiling mode to profile task lengths; and (3) a delaying mode where each task will be padded with delay to exclude the performance impact of workload-length imbalance.

## 7 EVALUATION

As described in Section 4.3, we follow an iterative process to understand and reach the performance limit of stable synchronization determinism with the 108 benchmarks described in Section 4.2.

For each benchmark, we measure the execution times under five different environments, nondeterministic executions under vanilla pthreads, TONDSync, Parrot, QiThread, and minSMT. We run a benchmark 30 times under each environment and take the average. Following the state-of-the-practice in performance evaluation of DMT systems, we do not intentionally control program execution to trigger nondeterministic behavior, under which nondeterministic behavior is possible but rare as in the real-world scenario. For Parrot performance, we add all soft barriers as designated in the Parrot reproducing package. Following the practice of QiThread evaluation [63], we do not add the PCS annotations from Parrot as they introduce nondeterminism. For QiThread performance, we turn on all five policies that come with QiThread, as QiThread evaluation [63] shows that it reaches the best performance with all policies turned on. For minSMT performance, we turn on all new policies and add the extended-turn annotations if applicable. Currently, the extended-turn annotation is added at one location in splash2x cholesky, two locations in parsec freqmine-openmp, three locations in parsec x264, and six locations in parsec ferret. We normalize all execution times to nondeterministic execution times under vanilla pthreads.

Below, we first present our detailed results on evaluating whether and how well minSMT can reach the performance limit of schedule tuning on stable synchronization determinism, and then we compare minSMT performance with Parrot and QiThread, focusing on benchmarks that minSMT reduces overhead significantly. Lastly, we present the complete set of normalized execution times under different execution environments.

### 7.1 Evaluation of Performance Limit

With the scheduling-oblivious overheads and design of minSMT presented in the previous sections, we consider that minSMT reaches the performance limit of schedule tuning if it has no more than 3% extra overhead after excluding the impact of scheduling-oblivious overheads. On the 108 benchmarks used for evaluation, we show that the performance limit is reached on all but one of them.

To determine whether minSMT can reach the performance limit on a specific benchmark, we first compare the minSMT performance overhead number with TONDSync performance overhead number, where performance overhead numbers are normalized against nondeterministic execution times under vanilla pthreads, and we conclude that minSMT reaches the performance limit if the minSMT performance overhead number is no more than TONDSync performance overhead number plus 3%. Among these 108 benchmarks, 86 of them are considered as reaching the performance limit after comparing their minSMT and TONDSync performance overhead

**Table 2: Detailed evaluation results of excluding the impact of scheduling-oblivious overheads on 22 benchmarks. All numbers are performance overhead compared with nondeterministic executions under vanilla pthreads.**

| Name | TONDSync | minSMT | Delta | Add CSFine | | | Add delay | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | # CSFine | TONDSync | minSMT | # Delays | TONDSync | minSMT |
| npb ua-l | 11470.57% | 20048.41% | 8577.84% | 27 | 104.56% | 105.62% | - | - | - |
| parsec fluidanimate | 7215.59% | 8985.36% | 1769.78% | 5 | 75.88% | 78.03% | - | - | - |
| parsec ferret | 168.63% | 993.16% | 824.53% | - | - | - | 4 | 0.12% | 0.09% |
| splash2x raytrace | 78.70% | 481.13% | 402.43% | 1 | 3.13% | 199.17% | 1 | 0.58% | 1.19% |
| parsec x264 | 75.22% | 442.61% | 367.39% | - | - | - | 2 | 0.29% | 2.80% |
| splash2x fmm | 41.68% | 371.64% | 329.96% | 10 | 3.48% | 3.03% | - | - | - |
| parsec freqmine-openmp | 0.30% | 128.81% | 128.51% | - | - | - | 1 | 0.03% | 1.84% |
| parsec vips | -0.95% | 67.48% | 68.43% | 5 | -0.32% | 67.39% | 1* | 0.13% | 1.22% |
| parsec dedup | 11.83% | 76.33% | 64.50% | 1 | 10.24% | 63.77% | 3 | 1.12% | 3.64% |
| splash2x volrend | 42.94% | 92.92% | 49.98% | 2 | -6.39% | -15.35% | - | - | - |
| phoenix string_match | 15.98% | 44.65% | 28.66% | - | - | - | 1 | 10.13% | 6.88% |
| phoenix kmeans | 4.51% | 32.93% | 28.41% | - | - | - | 1 | 4.76% | 4.11% |
| phoenix word_count | 14.93% | 39.35% | 24.41% | - | - | - | 3 | 0.81% | 2.05% |
| phoenix pca | 12.82% | 33.72% | 20.90% | - | - | - | 3 | 25.62% | 27.44% |
| splash2x cholesky | 51.22% | 72.10% | 20.88% | 2 | 48.57% | 70.09% | 1 | 15.68% | 12.14% |
| parsec facesim | -4.33% | 15.81% | 20.14% | 5 | -7.87% | -9.02% | - | - | - |
| splash2x barnes | 183.37% | 202.64% | 19.28% | 1 | 3.53% | 3.68% | - | - | - |
| phoenix histogram | 33.91% | 53.04% | 19.14% | - | - | - | 1 | 1.84% | 1.17% |
| parsec streamcluster | -6.21% | 6.84% | 13.05% | - | - | - | 2 | 7.03% | 8.58% |
| phoenix linear_regression | 22.38% | 32.79% | 10.40% | - | - | - | 1 | 0.84% | 0.13% |
| pbzip2 'decompress' | 2.49% | 10.52% | 8.03% | - | - | - | 1 | 6.91% | 8.45% |
| splash2x water_nsquared | 0.32% | 6.41% | 6.09% | 3 | -0.11% | -0.23% | - | - | - |

numbers, and we do not further exclude the impact of scheduling-oblivious overheads caused by fine-grained locks and workload-length imbalance on these cases.

For the remaining 22 benchmarks, Table 2 shows the results of further excluding scheduling-oblivious overheads caused by fine-grained locks and workload-length imbalance. For each benchmark whose minSMT overhead is more than TONDSync overhead plus 3%, we list its TONDSync overhead, minSMT overhead, and their delta, i.e., minSMT overhead minus TONDSync overhead. Our goal is to fully exclude the impact of the scheduling-oblivious overheads and have minSMT fully eliminate the remaining scheduling-dependent overheads through schedule tuning. In the ideal case, the performance overheads of TONDSync and minSMT should be very close after excluding the impact of the scheduling-oblivious overheads with necessary CSFine annotations and delays added. Currently, we first add CSFine annotations and then delays, and we stop further excluding the impact of scheduling-oblivious overheads after seeing minSMT has an extra overhead of less than 3% compared with TONDSync.

Among these 22 benchmarks in Table 2, 11 of them use fine-grained locks, and we annotate critical sections protected by fine-grained locks with the CSFine annotation. We list the numbers of CSFine annotations being added, and we report the overhead numbers under TONDSync and minSMT after these CSFine annotations are added. After adding CSFine annotations, 7 more benchmarks are considered as reaching the performance limit, as the extra overhead of minSMT is less than 3% compared with TONDSync.

For the remaining 15 benchmarks, we further add delays to exclude the performance impact of workload-length imbalance. We manually examine the synchronization logs generated by the DMT runtime system to recognize computation tasks. If workload-length imbalance is found, we add delays to such tasks. In 14 benchmarks, it is easy to identify the task boundaries and add delays. In the remaining case, i.e., parsec vips, computation is expressed using recursion with synchronization in the middle. For this case, we have to first mark a critical section protected by a global lock inside recursion with the CSFine annotation, and then we can mark task boundaries and add delays. We list the numbers of locations where delays are added, and we report the overhead numbers under TONDSync and minSMT after delays are added. After adding delays, the 14 benchmarks are considered as reaching the performance limit. Although minSMT can also reach an extra overhead less than 3% on parsec vips, we do not consider it a case where we can determine whether minSMT reaches the performance limit. This is because the CSFine annotation is added on a critical section not protected by fine-grained locks but a global lock, and we mark the number of delays for parsec vips with '*' to indicate the exceptional use of CSFine.

From the analysis above, we conclude that the performance limit is reached, e.g., no more than 3% extra overhead after excluding all scheduling-oblivious overheads, on all but one of the benchmarks. For the 107 programs reaching the performance limit under minSMT, 57 programs have less than 3% overhead compared with vanilla pthreads execution, 28 have an overhead between 3% and 10%, 14 have an overhead between 10% and 100%, and 8 have more than 100% overhead. For the remaining benchmark, parsec vips, the current performance overhead under minSMT is 67.48%, and we identify the programming pattern that is challenging for us to determine whether minSMT reaches the performance limit.

## 7.2 Performance Comparison

We next discuss the performance of minSMT without excluding the impact of scheduling-oblivious overheads and compare it with the performance of Parrot and QiThread on all the 108 benchmarks. The results are shown in Figure 5. All performance numbers are normalized to nondeterministic execution time.

QiThread has already established that it can achieve similar or better performance compared with Parrot with soft barrier but without PCS annotations in most cases [63], and our reproduced

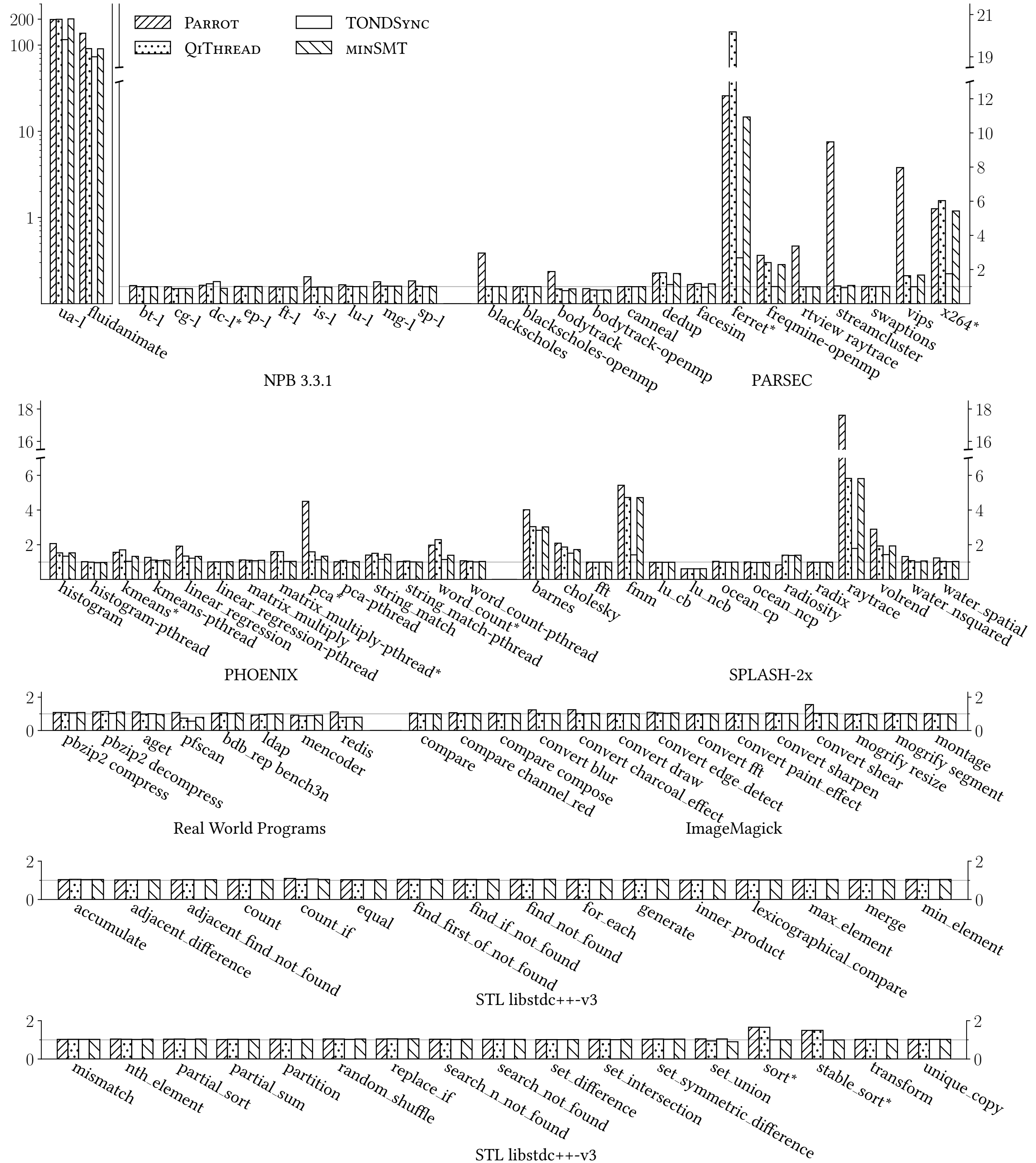



**Figure 5: Execution times of 108 benchmarks under Parrot, QiThread, TONDSync, and minSMT. All numbers are normalized based on nondeterministic execution times. Benchmarks that benefit significantly from minSMT compared with QiThread are marked with '*'. The charts for NPB ua-l and PARSEC fluidanimate use a logarithmic scale, and others use a linear scale.**

**Table 3: 9 benchmarks that minSMT performs significantly better than QiThread. All numbers are performance overhead numbers compared with nondeterministic executions under vanilla pthreads.**

| Name | Parrot | QiThread | minSMT | What helps |
|---|---|---|---|---|
| npb dc-l | 7.78% | 16.54% | -10.37% | WakeAMAP+ |
| parsec ferret | 1117.28% | 1917.98% | 993.16% | Extended-Turn |
| parsec x264 | 455.90% | 503.40% | 442.61% | Extended-Turn |
| phoenix kmeans | 56.23% | 70.43% | 32.93% | WakeAMAP+ |
| phoenix matrix_multiply-pthread | 60.09% | 60.14% | 4.36% | NoChildNoDMT |
| phoenix pca | 349.96% | 59.15% | 33.72% | WakeAMAP+ |
| phoenix word_count | 97.65% | 129.24% | 39.35% | WakeAMAP+ |
| stl sort | 65.97% | 65.60% | 0.07% | NoChildNoDMT |
| stl stable_sort | 49.26% | 49.52% | -0.45% | NoChildNoDMT |

results can confirm it. Therefore, we focus on the comparison between minSMT and QiThread, i.e., how many benchmarks can benefit from new policies in minSMT, and whether the performance of any benchmarks is hurt.

To understand how many benchmarks can benefit from new policies and annotations in minSMT, we normalize minSMT execution times to QiThread execution times. On average, minSMT has a speedup of 2.80% over QiThread. 9 benchmarks have minSMT execution times less than 90% of QiThread execution times, and the average speedup is 29.33%. We consider that minSMT reaches a significant performance improvement compared with QiThread in these cases. Table 3 lists the performance overheads of Parrot, QiThread, and minSMT of these 9 cases compared with nondeterministic executions under vanilla pthreads.

On these 9 benchmarks, we can see from the numbers in Table 3 that minSMT greatly reduces the performance overhead of stable synchronization determinism compared with both QiThread and Parrot. Three benchmarks, npb dc-l, stl sort, and stl stable_sort, now even have less than 1% overhead. For each case, we also list the minSMT design that helps to achieve the speedup. We can see that each new design in minSMT is beneficial. For these 9 benchmarks, numbers in Table 3 suggest that Parrot and QiThread cannot reach the performance limit as minSMT does.

Note that npb dc-l in Table 3, together with 6 other programs, enjoys more than 10% speedup against vanilla pthreads execution. This type of speedup is also observed in QiThread and Parrot results on a similar set of programs. The reason is that Parrot includes a more efficient wait implementation utilizing a spin-lock phase. Deterministic schedules can also reduce contention and context-switches, and they may improve affinity. As our minSMT library is based on Parrot, we inherit these performance benefits.

Using the same normalized execution times, we also check how many benchmarks are hurt by new policies in minSMT. In no benchmarks do minSMT execution times exceed 110% of QiThread execution times. Therefore, we consider that minSMT does not hurt any benchmarks significantly compared with QiThread. Only two benchmarks, pfscan and mplayer encoder, have minSMT execution times between 103% and 110% of QiThread execution times. Specifically, pfscan is at 106.27% and mplayer encoder is at 105.12%. This is due to the extra overhead from the WakeAMAP+ policy when tracking the caller addresses of all pthread_cond_wait/signal calls, which is currently implemented using a costly stack unwinding to avoid changing the benchmark source code, and this overhead outweighs the benefits for these two programs. However, both QiThread and minSMT lead to speedup but not slowdown in these two benchmarks. A more performing implementation that instruments the source code to pass the caller address to the library at each pthread_cond_wait/signal call would likely reduce or eliminate these slowdowns. In addition, 10 benchmarks have their minSMT execution times within 100% to 103% of QiThread execution times, and we consider these differences insignificant.

From these results, we conclude that minSMT can generally improve performance compared with state-of-the-art stable synchronization determinism systems.

### 7.3 Discussion

Based on our results, we conclude that our identified scheduling-oblivious overheads and our proposed minSMT together successfully achieve the goal of understanding and reaching the performance limit of stable synchronization determinism on almost all benchmarks we evaluate. The challenge of the remaining benchmark is caused by one specific code pattern of using synchronization inside recursive function calls. Future work may design code refactoring tools to make programs with such code pattern more suitable for stable synchronization determinism.

Our results show that minSMT can still exhibit high performance overhead on some benchmarks where we consider that minSMT reaches the performance limit of stable synchronization determinism. To further improve the performance of stable synchronization determinism in those cases, future work needs to propose solutions to tackle the scheduling-oblivious overheads through approaches other than schedule tuning.

To eliminate the overhead of totally-ordered synchronization, one possibility to enable partial-order enforcement without speculation is to provide an annotation system, which will allow programmers or program analyses to communicate with the scheduler on what synchronization operations will be executed by threads in the future. If such a system is built, we need to further evaluate its determinism guarantee and schedule stability. To eliminate workload-length imbalance, one may build some code refactoring tools to make workload length more balanced. We could also use a previously proposed idea, i.e., schedule memorization and relaxation [27, 28], but we now need to further consider the workload length while forming schedule reuse conditions. We leave these ideas for future work.

## 8 CONCLUSION

In this paper, we first categorize and compare existing DMT runtime systems along three axes and conclude that stable synchronization determinism is the most cost-effective design. We then focus on understanding and reaching the performance limit of schedule tuning on stable synchronization determinism. We identify two types of scheduling-oblivious overheads and design a new system, minSMT, to address most of the remaining scheduling-dependent overheads. Evaluation on 108 benchmarks shows that minSMT can reach the performance limit of schedule tuning on stable synchronization determinism in all but one of the cases. Compared with state-of-the-art stable synchronization determinism systems, minSMT can improve the performance in many benchmarks significantly.

## REFERENCES


[1] 2009. Aget. http://www.enderunix.org/aget/.
[2] 2010. pfscan. http://freshmeat.sourceforge.net/projects/pfscan.
[3] 2012. SPLASH-2x. http://parsec.cs.princeton.edu/parsec3-doc.htm.
[4] 2012. The PARSEC Benchmark Suite. http://parsec.cs.princeton.edu/.
[5] 2016. Parallel BZIP2 (PBZIP2). https://launchpad.net/pbzip2.
[6] 2017. MPlayer. http://www.mplayerhq.hu/design7/news.html.
[7] 2017. STL Parallel Mode. http://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html.
[8] 2018. Berkeley DB. https://www.oracle.com/database/technologies/related/berkeleydb.html.
[9] 2018. ImageMagick. http://www.imagemagick.org/script/index.php.
[10] 2018. NASA Parallel Benchmarks. http://www.nas.nasa.gov/software/npb.html.
[11] 2018. OpenLDAP. http://www.openldap.org/.
[12] 2018. Redis. http://redis.io/.
[13] 2022. minSMT. https://github.com/chyiz/minSMT/.
[14] Gautam Altekar and Ion Stoica. 2009. ODR: Output-deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/1629575.1629594
[15] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient System-enforced Deterministic Parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, Berkeley, CA, USA, 193–206. http://dl.acm.org/citation.cfm?id=1924943.1924957
[16] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/1736020.1736029
[17] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. 2011. The Deterministic Execution Hammer: How Well Does it Actually Pound Nails?. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming* (Newport Beach, California, USA) *(WODET '11)*.
[18] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010. Deterministic Process Groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, Berkeley, CA, USA, 177–191. http://dl.acm.org/citation.cfm?id=1924943.1924956
[19] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe Multithreaded Programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 81–96. https://doi.org/10.1145/1640089.1640096
[20] Guy E. Blelloch. 1993. *NESL: A Nested Data-Parallel Language (Version 2.6)*. Technical Report. Pittsburgh, PA, USA.
[21] Robert L. Bocchino and Vikram S. Adve. 2011. Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks. In *Proceedings of the 25th European Conference on Object-Oriented Programming* (Lancaster, UK) *(ECOOP'11)*. Springer-Verlag, Berlin, Heidelberg, 306–332. https://doi.org/10.1007/978-3-642-22655-7_15
[22] Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel Programming Must Be Deterministic by Default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism* (Berkeley, California) *(HotPar'09)*. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1855591.1855595
[23] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 97–116. https://doi.org/10.1145/1640089.1640097
[24] Robert L. Bocchino, Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe Nondeterminism in a Deterministic-by-default Parallel Language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 535–548. https://doi.org/10.1145/1926385.1926447
[25] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. Paxos Made Transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 105–120. https://doi.org/10.1145/2815400.2815427
[26] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 388–405. https://doi.org/10.1145/2517349.2522735
[27] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. 2011. Efficient Deterministic Multithreading Through Schedule Relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 337–351. https://doi.org/10.1145/2043556.2043588
[28] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. 2010. Stable Deterministic Multithreading Through Schedule Memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, Berkeley, CA, USA, 207–221. http://dl.acm.org/citation.cfm?id=1924943.1924958
[29] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/1508244.1508255
[30] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: A Relaxed Consistency Deterministic Computer. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 67–78. https://doi.org/10.1145/1950365.1950376
[31] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. 2014. Rex: Replication at the Speed of Multi-Core. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 11, 14 pages. https://doi.org/10.1145/2592798.2592800
[32] Stephen T. Heumann, Vikram S. Adve, and Shengjie Wang. 2013. The Tasks with Effects Model for Safe Concurrency. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/2442516.2442540
[33] Nima Honarmand and Josep Torrellas. 2014. RelaxReplay: Record and Replay for Relaxed-consistency Multiprocessors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 223–238. https://doi.org/10.1145/2541940.2541979
[34] Derek R. Hower, Polina Dudnik, Mark D. Hill, and David A. Wood. 2011. Calvin: Deterministic or Not? Free Will to Choose. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 333–334. http://dl.acm.org/citation.cfm?id=2014698.2014870
[35] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 207–216. https://doi.org/10.1145/1882291.1882323
[36] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/2491956.2462167
[37] Shiyou Huang, Bowen Cai, and Jeff Huang. 2017. Towards Production-run Heisenbugs Reproduction on Commercial Hardware. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 403–415. http://dl.acm.org/citation.cfm?id=3154690.3154729
[38] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. 2013. DDOS: Taming Nondeterminism in Distributed Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 499–508. https://doi.org/10.1145/2451116.2451170
[39] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 237–250.
[40] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563
[41] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. 2010. Respec: Efficient Online Multiprocessor Replayvia Speculation and External Determinism. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association

for Computing Machinery, New York, NY, USA, 77–90. https://doi.org/10.1145/1736020.1736031

[42] N. G. Leveson and C. S. Turner. 1993. An Investigation of the Therac-25 Accidents. *Computer* 26, 7 (July 1993), 18–41. https://doi.org/10.1109/MC.1993.274940

[43] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. IReplayer: In-Situ and Identical Record-and-Replay for Multithreaded Applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3192366.3192380

[44] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587

[45] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. 2014. Efficient Deterministic Multithreading Without Global Barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 287–300. https://doi.org/10.1145/2555243.2555252

[46] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[47] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 693–708. https://doi.org/10.1145/3037697.3037751

[48] Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. 2015. High-performance Determinism with Total Store Order Consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 31, 13 pages. https://doi.org/10.1145/2741948.2741960

[49] Timothy Merrifield and Jakob Eriksson. 2013. Conversion: Multi-version Concurrency Control for Main Memory Segments. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 127–139. https://doi.org/10.1145/2465351.2465365

[50] Timothy Merrifield, Sepideh Roghanchi, Joseph Devietti, and Jakob Eriksson. 2019. Lazy Determinism for Faster Deterministic Multithreading. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 879–891. https://doi.org/10.1145/3297858.3304047

[51] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic Galois: On-demand, Portable and Parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 499–512. https://doi.org/10.1145/2541940.2541964

[52] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/1508244.1508256

[53] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 177–192. https://doi.org/10.1145/1629575.1629593

[54] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. IEEE Computer Society, Washington, DC, USA, 13–24. https://doi.org/10.1109/HPCA.2007.346181

[55] Martin C. Rinard and Monica S. Lam. 1998. The Design, Implementation, and Evaluation of Jade. *ACM Trans. Program. Lang. Syst.* 20, 3 (May 1998), 483–545. https://doi.org/10.1145/291889.291893

[56] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[57] SecurityFocus. 2004. Software Bug Contributed to Blackout. http://www.securityfocus.com/news/8016.

[58] Cedomir Segulja and Tarek S. Abdelrahman. 2012. Architectural Support for Synchronization-Free Deterministic Parallel rogramming. In *Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPCA.2012.6169038

[59] Cedomir Segulja and Tarek S. Abdelrahman. 2014. What is the Cost of Weak Determinism?. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 99–112. https://doi.org/10.1145/2628071.2628099

[60] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/1950365.1950370

[61] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. 2012. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 205–216. https://doi.org/10.1145/2254064.2254090

[62] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. 2014. Making Parallel Programs Reliable with Stable Multithreading. *Commun. ACM* 57, 3 (March 2014), 58–69. https://doi.org/10.1145/2500875

[63] Qi Zhao, Zhengyi Qiu, and Guoliang Jin. 2019. Semantics-aware Scheduling Policies for Synchronization Determinism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 242–256. https://doi.org/10.1145/3293883.3295731

# A ARTIFACT APPENDIX

## A.1 Abstract

Our artifact can be downloaded through Github. It provides the source code of our minSMT library and other libraries used for comparison. It also provides scripts to download all the benchmarks used in our paper, automatically build them, and conduct experiments. All libraries, benchmarks, and datasets needed for evaluation are either already included in our artifact or can be downloaded using our scripts.

## A.2 Artifact Check-List (Meta-Information)

- **Program:** DMT runtime libraries, including minSMT and others for comparison, and benchmark suites and real-world programs used for evaluation.
- **Compilation:** Compilation scripts are provided, and they use GCC 9 from Ubuntu 20.04.
- **Run-time environment:** Ubuntu 20.04 and GCC 9 with OpenMP support. Alternatively, the Docker environment provided in this artifact, which can run on any Linux distribution with a kernel version 5.4.
- **Hardware:** x86_64 CPU with at least 16 logical cores, at least 16GB of memory, and 70GB of disk space.
- **Execution:** Evaluation scripts are provided, and they can be customized if necessary.
- **Metrics:** Execution time of programs under non-deterministic, TONDSync, QiThread, and minSMT will be reported, as well as selected programs with added CSFine annotations, and the calculated overhead of each configuration.
- **Output:** Results are saved as a `.csv` file, and scripts are provided to turn it into figures.
- **Experiments:** We provide evaluation scripts and configurations to run experiments, collect results, and generate figures. Due to hardware differences, the actual results may vary. However, the general trend comparing different DMT systems should be similar.
- **How much disk space is required (approximately)?:** Around 70GB.
- **How much time is needed to prepare workflow (approximately)?:** It takes about two hours to download and compile all libraries and programs.
- **How much time is needed to complete experiments (approximately)?:** It will take several days to produce all data with the default configuration of taking the average from 50 repeated runs. We also provide evaluation configurations with fewer runs or on a curated set of programs.
- **Publicly available?:** Yes. https://github.com/chyiz/minSMT.
- **Code licenses:** Custom copyright license. Please refer to the LICENSE file in the repository.

## A.3 Description

*A.3.1 How to Access.* Our artifact is publicly accessible on Github at https://github.com/chyiz/minSMT. You may clone it using git:

```
git clone https://github.com/chyiz/minSMT.git
```

*A.3.2 Hardware Dependencies.* We conducted our experiments on a machine with a Xeon E5-2695 V4 processor and 64GB memory. While our code should be runnable on any multi-core machines, we recommend a CPU with at least 16 logical cores and a similar memory configuration.

Building all benchmarks takes about 70GB of disk space, so make sure you have enough free disk space on the volume you use.

*A.3.3 Software Dependencies.* We used Ubuntu 20.04 64-bit to compile minSMT and all our benchmarks, and the scripts provided in this repository have been tested on this OS version. We also provide a Docker environment to help you use our artifact under other Linux distributions.

## A.4 Installation

There are two ways of building and running our artifact: natively or using Docker. We recommend using Docker for easier deployment.

*A.4.1 Docker Installation.* We prepared a Ubuntu 20.04 Docker environment to help you compile and run our experiments. After cloning this repository onto your local system, and while in the repository root directory, run the following command:

```
sudo DOCKER_BUILDKIT=1 docker build \
  --build-arg UID=$(id -u) \
  --build-arg GID=$(id -g) \
  -t minsmt-ae .
```

to build a docker image with all necessary packages to compile minSMT and all benchmarks. This step should take about five minutes with good internet speed.

After that, a docker image will be available with tag `minsmt-ae`. While still in the project root directory, run the following command:

```
sudo docker run -it \
  -v `pwd`:/home/minsmt/minsmt-ae minsmt-ae:latest
```

You will enter a BASH shell at `/home/minsmt/minsmt-ae`, and the repository on your host machine is mounted on that path in the container. If you ever exit the docker environment, simply go back to the repository root directory and run the above command again.

Now, run `./buildall-docker.sh` to compile libraries and benchmarks. This step will take about two hours. During this process, make sure you have a stable internet connection, since the build script will also download benchmarks and input files from their official websites.

*A.4.2 Native Installation.* Alternatively, you may build the project directly. The build script is tested under Ubuntu 20.04 and may not work on other Linux distributions.

First, run `source env.sh` in the repository root directory to set the environment variables.

Then, simply run `./buildall.sh`, which will build all libraries and download and build all programs used for evaluation.

## A.5 Experiment Workflow

To run the experiments, first switch to the evaluation directory while you are in the repository root directory with `cd eval`. In the evaluation directory, run:

```
./eval_policy.py all-compare-with-qithread.cfg
```

to test all benchmarks. This will run all benchmarks 50 times with QiThread, TONDSync, and minSMT configurations, and take the

average of each configuration. Evaluating all benchmarks will take a couple of days (around five on our platform) to finish.

Alternatively, you may replace the full-scale configuration file `all-compare-with-qithread.cfg` in the above command with `all-compare-with-qithread-small.cfg` to run each benchmark for 10 times, and the script should finish in a day or two. You may also use `minSMT-subset-compare-with-qithread.cfg` to run a curated set of benchmarks that we have discussed in the paper.

The evaluation results will be saved to a directory named

```
<config_file_name><date_and_time>_<git_commit>
```

A symbolic link `current` is also modified to always point to the latest evaluation results.

Then, to extract all results into a comma separated values (`.csv`) file, run:

```
./get_all_results.sh current/ > results.csv
```

Replace `current` with the directory name that contains evaluation results if you are exporting old evaluations. The results will be in a `.csv` file that can then be imported to a spreadsheet.

At last, to generate the graphs presented in our paper, use Jupyter Notebook to open `generate-figure.ipynb` in the `eval` directory and run all cells. It will read the `results.csv` file in the same directory by default.

## A.6 Evaluation and Expected Results

After finishing experiments with the full-scale configuration file `all-compare-with-qithread.cfg` and extracting results, your `.csv` file should contain all the data needed to recreate Figure 5 or the first seven columns of Table 2.

Due to hardware differences, you will not get exactly the same results. However, you should observe a similar trend when comparing different libraries and configurations.

## A.7 Experiment Customization

*A.7.1 Running an Individual Benchmark.* A configuration (`.cfg`) file consists of one or more sections, and each section represents a benchmark program to be evaluated. You may extract a single section and save it to an individual configuration file to test one benchmark. Then, you can supply the configuration file to the same evaluation script `eval_policy.py`.

*A.7.2 Selecting Policies.* Each benchmark section in a configuration file has a `RUN_CONFIGS` parameter. It is used to select a set of policies to run. In `all-compare-with-qithread.cfg`, three configurations are used: `qithread` means all the policies introduced in the QiThread paper but nothing else, `null-policy` is the TONDSync mode introduced in this paper, and `all-policies` are all policies available in minSMT, including the ones from QiThread. Prefix `pcs-` can be added before `null-policy` and `all-policies` to enable CSFine annotations.

Other available options are: `no-hint` for pure round-robin scheduling, `hinted` for Parrot paper configurations, and `no-pcs-hint` for Parrot paper configurations without the PCS hint.

*A.7.3 Tuning Parameters.* You may fine tune minSMT parameters like disabling individual features using the `local.options` file, which will be read by the evaluation script from the same directory as the benchmark binary.

To start, you run a single benchmark to get the results directory. Inside the results directory, you will see a subdirectory named after the benchmark, and the subdirectory contains the benchmark binary, inputs it used, outputs it generated during evaluation, one or more `.options` files the evaluation script generated corresponding to the `RUN_CONFIGS` designated in the configuration file, and a `local.options` file, i.e., the last options file used.

You can rename an existing options file to `local.options` so that it can be picked up by the next execution. For the meaning of parameters, please refer to the comments in the `default.options` file under the repository root folder.

After adjusting the `local.options` file, you may run the benchmark using the following command:

```
LD_PRELOAD=$minSMT_ROOT/dync_hook/interpose.so \
  ./<benchmark_binary> <benchmark_parameters>
```

*A.7.4 Applying Delays.* To apply delays to benchmarks, first follow the above instructions to run a benchmark once, and get the results directory. Then, change directory into the benchmark sub-directory inside the generated results directory, create a file named `app.time`, and edit `local.options` to set `enforce_delays = 1`. Now, run the benchmark like below:

```
LD_PRELOAD=$minSMT_ROOT/dync_hook/interpose.so \
  ./<benchmark_binary> <benchmark_parameters>
```