SLITS: Sparsity-Lightened Intelligent Thread Scheduling

Wangkai Jin Xiangjun Peng[‡]

Abstract

A diverse set of scheduling objectives (e.g., resource contention, fairness, priority, etc.) breed a series of objective-specific schedulers for multi-core architectures. Existing designs incorporate thread-to-thread statistics at runtime, and schedule threads based on such an abstraction (we formalize thread-to-thread interaction as the Thread-Interaction Matrix). However, such an abstraction also reveals a consistently-overlooked issue: the Thread-Interaction Matrix (TIM) is highly sparse. Therefore, existing designs can only deliver sub-optimal decisions, since the sparsity issue limits the amount of thread permutations (and its statistics) to be exploited when performing scheduling decisions.

We introduce *Sparsity-Lightened Intelligent Thread Scheduling* (SLITS), a general scheduler design for mitigating the sparsity issue of TIM, with the customizability for different scheduling objectives. SLITS is designed upon the key insight that: the sparsity issue of the TIM can be effectively mitigated via advanced Machine Learning (ML) techniques. SLITS has three components. First, SLITS profiles Thread Interactions for only a small number of thread permutations, and form the TIM using the run-time statistics. Second, SLITS estimates the missing values in the TIM using *Factorization Machine* (FM), a novel ML technique that can fill in the missing values within a large-scale sparse matrix based on the limited information. Third, SLITS leverages *Lazy Reschedule*, a general mechanism as the building block for customizing different scheduling policies for different scheduling objectives. We also show how SLITS can be (1) customized for different scheduling objectives, including resource contention and fairness; and (2) implemented with only negligible hardware costs.

We evaluate two SLITS variants against four state-of-the-art scheduler designs. We highlight that, averaged across 11 benchmarks, SLITS achieves an average speedup of 1.08X over the *de facto* standard for thread scheduler - the *Completely Fair Scheduler*, under the 16-core setting for a variety of number of threads (i.e., 32, 64 and 128). Our analysis reveals that the benefits of SLITS are credited to significant improvements of cache utilization. In addition, our experimental results confirm that SLITS is scalable and the benefits are robust when of the number of threads increases. We also perform extensive studies to (1) break down SLITS components to justify the synergy of our design choices, (2) examine the impacts of varying the estimation coverage of FM, (3) justify the necessity of *Lazy Reschedule* rather than periodic rescheduling, and (4) demonstrate the hardware overheads for SLITS implementations can be marginal ($<$1% chip area and power).

The design of SLITS breed a few implications on the limitations of multi/many-core architectures, and how our design can be generally applicable to a variety of similar scenarios on resource scheduling and managements. We first identify the potential of SLITS for other types of resource scheduling and management in the current processor-centric computer systems. Then, we discuss the implications from SLITS in terms of future research on emerging computing paradigms, and highlight important research questions in a broad context of resource scheduling (and managements).

[*]Note that [‡]refers the affiliation to *The Chinese University of Hong Kong.*

Contents

1 Introduction **3**

2 Background & Motivation **6**
- 2.1 Existing Scheduler Designs for Different Scheduling Objectives . . . 6
- 2.2 Motivation: Sparsity of the Thread-Interaction Matrix - The Hidden Issue . . . 6
- 2.3 A Quantitative Characterization on the Sparsity Issue of the Thread-Interaction Matrix among Mainstream Schedulers . . . 7

3 SLITS: Design Overview **9**
- 3.1 SLITS Workflow . . . 9
- 3.2 Step 1: Profiling Statistics for Thread-Interaction Matrix . . . 9
- 3.3 Step 2: Estimating Missing Statistics via Factorization Machines . . . 10
- 3.4 Step 3: *Lazy Reschedule* . . . 11

4 SLITS: Detailed Designs **13**
- 4.1 Thread Interaction Statistics Cache (TIS-Cache) . . . 13
- 4.2 Using Factorization Machines for Thread-Interaction Matrix . . . 13
- 4.3 Customizing SLITS for Different Objectives . . . 14
  - 4.3.1 SLITS-(Contention): Mitigating Resource Contention . . . 14
  - 4.3.2 SLITS-(Fairness): Preserving Thread Fairness . . . 15
  - 4.3.3 Customizing SLITS for More Scheduling Objectives . . . 15

5 Experimental Methodology **16**
- 5.1 Experimental Infrastructure . . . 16
- 5.2 Comparison Points . . . 16
- 5.3 Workload Configurations . . . 17
- 5.4 Performance Metrics . . . 17

6 Experimental Results **18**
- 6.1 SLITS versus Other Schedulers . . . 18
- 6.2 Breakdown Analysis of SLITS . . . 19
- 6.3 Breakdown Analysis of Different SLITS Components . . . 20
- 6.4 Different Coverage of FM Module . . . 21
- 6.5 Suitability of *Lazy Reschedule* . . . 22
- 6.6 Overhead Analysis . . . 23

7 Related Works **24**
- 7.1 Scheduler Designs for Fixed Scheduling Objectives . . . 24
- 7.2 Heuristic/Statistical-Approach-based Schedulers Designs . . . 24
- 7.3 Exploiting Machine Learning Techniques to Assist the Schedulers . . . 24

8 Conclusions and Implications **25**
- 8.1 Conclusion from this work . . . 25
- 8.2 Implications: Generality and Applicability . . . 25
- 8.3 Implications on Future Research and Practice . . . 26
  - 8.3.1 Major Implications on Research . . . 26

1 Introduction

Thread scheduling on multi-core architectures is essential to maximize the utilization of hardware resources. To better satisfy the needs of thread scheduling, a variety of scheduling objectives are formed[1]. Regardless of which specific scheduling objective is targeted, existing designs can be categorized into two types. ❶ Early designs schedule threads with fixed rules[2], which does not rely on and/or correlate with workload characteristics. However, fixed-rule schedulers can not satisfy the needs of different scheduling objectives. ❷ The demands for different scheduling objectives breed more recent efforts, to take scheduling objectives into account. These designs target a specific scheduling objective (e.g., resource content, fairness, etc.), by (1) collecting run-time information of threads and their correlations quantitatively (e.g., Cache Miss Count [30, 46], Thread IPC [81], dynamic priority requirements like Earliest Deadline First [5, 57, 23]); and (2) perform scheduling decisions based on thread-to-thread interactions.

We first observe that it is feasible to formalize these two types of scheduler designs in a unified manner, by centering our focus on Thread-Interaction statistics. For the first time, we define Thread-Interaction Matrix (TIM), which stores the statistics of thread-to-thread interaction. The types of these statistics can be any type of run-time statistics regarding the thread-to-thread pairs (e.g., Cache Miss Count [30, 46], Thread IPC [81], dynamic priority requirements like Earliest Deadline First [5, 57, 23])[3]. Based on TIM, we can specialize the designs of scheduling policies, by specifying the rules of thread rescheduling (e.g., rescheduling conditions and rescheduling strategies). The combination of the above two parts provides a sufficient formalization of all existing thread scheduler designs. Hence, such a formalization provides an essential foundation to rethink a general scheduler design, which can be (1) customized for different scheduling objectives; and (2) co-designed between Thread-Interaction Matrix and the scheduling policies in a synergistic manner.

Our work is first motivated by the **key observation** that the TIM is highly sparse, because (1) the actual executions can only explore a very small subset of all possible thread permutations (i.e., thread-to-core mapping); and (2) the interaction statistics of workload threads may vary greatly in different execution stages. Hence, the sparsity of the TIM is expected to restrict thread schedulers to only making sub-optimal scheduling decisions, since the decision-making procedure only relies on a large-yet-(mostly)-unavailable search space for all possible thread permutations[4]. We experimentally characterize five mainstream scheduler designs, and report the best results of them (i.e., top two from the explored permutations during executions) in terms of the exploitation from the thread scheduling search space (i.e., Round-Robin Scheduler and Random Scheduler), to examine the severity level of the sparsity within the TIM. Our characterizations reveal that the sparsity of the TIM is highly severe: for all workloads included in our experiments, the best results of existing mainstream schedulers can only enumerate up to 2.1% of all possible thread permutations under the 32-thread configuration, and they can only enumerate up to 0.23% of all possible thread permutations under the 64-thread configuration.

[1] We name several examples regarding scheduling objectives: mitigating hardware resource contention in different levels for high throughput [4, 32], allocating evenly-distributed resources for co-running threads for fairness [14, 15], scheduling threads following priority-based policies for prioritization [22], etc.

[2] Representative examples from fixed-rule schedulers include Round-Robin or Work-Stealing [27, 52, 12, 62, 47], priority [22], and etc.

[3] Note that, for fixed-rule scheduling, we can simply view that the TIM contains any static values since the statistics do not affect the scheduling decisions at any stage.

[4] One straightforward example to back up this claim is the scheduler designs to minimize the resource contention: though the scheduler is expected to allocate available threads for the minimization of the resource contention, the scheduler is incapable to approximate the minimization since the run-time statistics of most thread permutations are not available.

Our characterizations reveal that: the sparsity issue of TIM is affected significantly by both scheduling policies and workload characteristics. Hence, it is critical to mitigate the sparsity issue of TIM, since the root cause of such a sparsity issue is not deterministic. Moreover, the ambiguity of the sparisty issue within TIM also makes it challenging, to co-design an effective scheduling policy. **Our goal** in this work is to deliver a general design, to allow effective and customizable co-designs between TIM and scheduling policies, with the central focus on mitigating the sparsity issue of TIM.

To this end, we introduce *Sparsity-Lightened Intelligent Thread Scheduling* (SLITS), a general scheduler design for mitigating the sparsity issue of TIM, with the customizability for different scheduling objectives. SLITS consolidates the key insight that: the sparsity issue of the TIM can be effectively mitigated via advanced Machine Learning (ML) techniques. SLITS has three components. First, SLITS profiles Thread Interaction for only a small number of available threads based on their run-time statistics. Second, SLITS estimates the missing values from the TIM using *Factorization Machine* (FM), a novel ML technique that can exploit a limited amount of available information for a comprehensive estimation within a large-scale sparse matrix. Third, SLITS leverages *Lazy Reschedule*, a general mechanism as the building block, when customizing different scheduling policies for different scheduling objectives.

We then complement SLITS design with detailed implementations, and demonstrate potential extensions of SLITS. There are three aspects. First, we propose *Thread Interaction Statistics Cache* (TIS-Cache), to facilitate the need for fast access to the TIM. TIS-Cache can be implemented directly by repurposing the recently-discussed eDRAM cache solely for storing, updating and exploiting TIM. Second, we demonstrate the customizability of SLITS with two concrete variants, including SLITS-(Contention) (i.e., for resource contention) and SLITS-(Fairness) (i.e., for fairness), by only re-configuring the metrics for the TIM and adjusting the scheduling policies. These two variants of SLITS serve as proof-of-concepts that SLITS can be customized for different scheduling objectives.

Results Overview. We compare two SLITS variants[5] against four mainstream schedulers designs, including Global Round-Robin Scheduler (G-RRS), Per-core Round-Robin Scheduler (P-RRS), Work-Stealing Scheduler (WSS) and Completely Fair Scheduler (CFS) in multiple workloads and threads configurations. Our results show that, averaged across 11 benchmarks, SLITS-(Contention) achieves 1.27X, 1.18X, 1.12X, and 1.09X speedup compared to G-RRS, P-RRS, WSS and CFS under the 16-core setting with 32, 64 and 128 workload threads; and SLITS-(Fairness) achieves 1.25X, 1.14X, 1.09X and 1.06X speedup compared to G-RRS, P-RRS, WSS and CFS under the same settings. The above results show that SLITS always improves the performance over mainstream thread schedulers, regardless of which scheduling objective SLITS aims for. Moreover, we also justify that the benefits of SLITS are robust and scalable when the number of working threads increases. Our analysis reveals that the benefits of SLITS are credited to significant improvements of cache utilization. This is because SLITS mitigate the sparsity issue of the Thread-Interaction Matrix by incorporating a novel ML technique; and leveraging a viable scheduling policy to achieve different scheduling objectives in a customized manner.

We also perform an extensive amount of studies to understand the benefits of SLITS from different modules (i.e., the FM module and the *Lazy Reschedule* module) and the hardware costs. ❶ We examine the cache behaviors of SLITS and mainstream schedulers. The results show that SLITS can increase 21.45% L1 D-Cache hits while reducing 10.05% cycles spent in the L3 cache. ❷ We perform a breakdown analysis by module isolation, to understand the source of performance benefits from the FM module and *Lazy Reschedule*. The results show that disabling either of the two modules can lead to a considerable amount of performance degradation. ❸ We vary the estimation

[5]Note that, since the fairness can be viewed as scheduling with dynamic priorities, we do not deliver additional examples of SLITS for Priority-oriented thread scheduling.

range of the FM module to better understand how FM-driven estimations help with the scheduling. The results justify the effectiveness of FM-driven estimation, to improve overall performance. ❹ We demonstrate the necessity of *Lazy Reschedule* mechanism, by comparing it against periodic rescheduling policies under various configurations of the fixed-time window. The results confirm that *Lazy Reschedule* module is useful in controlling trigger conditions for thread rescheduling. ❺ We analyze the hardware overhead of SLITS, based on our experimental settings. Our results show that SLITS design and implementation incur only negligible overheads in terms of chip area and power on commercial processors (i.e., $<1\%$).

We make the following key contributions:

- We deliver a new formalization for general thread scheduler designs, and characterize the sparsity issue under our formalization. Our characterization reveals that the sparsity issue have been overlooked but is considerably severe for thread scheduling, which is caused by both (1) scheduler designs and (2) workload characteristics.

- We propose *Sparsity-Lightened Intelligent Thread Scheduling* (SLITS), a general thread scheduler design to mitigate the sparsity issue of the Thread-Interaction Matrix, via a novel Machine Learning technique, *Factorization Machines*.

- We provide detailed implementations of SLITS, by (1) repurposing the recently-discussed L4 eDRAM cache for the *Thread Interaction Statistics Cache*; and (2) concretely demonstrating how SLITS can be customized for different scheduling objectives, including resource contention and fairness.

- We evaluate the performance of SLITS against four mainstream thread schedulers and the results suggest great potential for SLITS in practice. We also perform sensitivity studies to understand the benefits and reveal potential issues of SLITS.

2 Background & Motivation

In this section, we provide the background information of this work. We first brief main categories of thread scheduling designs, and distill two general components of these designs: the Thread-Interaction Matrix and the scheduling policy (Section 2.1). Then, we reveal the hidden issues in existing scheduling techniques: the sparsity of the Thread-Interaction Matrix (Section 2.2). Finally, we conduct a quantitative characterization study on the sparsity issue of the Thread-Interaction Matrix, to quantitatively reveal the severity of this issue (Section 2.3).

2.1 Existing Scheduler Designs for Different Scheduling Objectives

Thread scheduling is ubiquitous and paramount in modern computing systems, to efficiently manage hardware resources for different scheduling objectives. Mainstream scheduling objectives can be categorized into three types, which are Resource Contention (e.g., Cache Contention [46, 20, 17, 4, 32]), Fairness (e.g., P-Fairness and its variants [14, 15, 8, 53, 9, 10, 11, 73], Fair Share [81, 44, 36] and Priority (e.g., Earliest Deadline First [51, 5, 57, 23]). Accordingly, several scheduler policies are proposed for different scheduling objectives, which can be categorized into two types. The first type of scheduler leverages fixed rules for thread scheduling (e.g., Round-Robin Scheduler, pre-defined-priority Scheduler [22], Work-Stealing Scheduler [27, 12, 47, 62, 52]), which view thread-to-thread interactions statically. The other type of schedulers leverages dynamic inter-thread statistics (e.g., IPC, Cache Miss [46], Fairness [81], dynamic priority [5], etc.). Both types of scheduling designs[6] can be formalized into two parts. One is the Thread-Interaction Matrix, which stores the inter-thread statistics of corresponding scheduling objectives[7]. The other is the scheduling policy, which schedules candidate threads based on the available values in the Thread-Interaction Matrix.

2.2 Motivation: Sparsity of the Thread-Interaction Matrix - The Hidden Issue

Different from prior work, our work focuses on the sparsity of the Thread-Interaction Matrix, and how we can deliver better scheduling policies against such an issue. The sparsity issue is triggered by two facts. First, existing scheduling policies are limited by a small amount of available information about inter-thread interactions, given the high sparsity of the Thread-Interaction Matrix. Second, the profiling of certain statistics for adaptive scheduling (e.g., cache contention) relies on previous phases of the execution, regardless of which metrics are used. This can incur expensive costs with the increase of the number of workload threads, and is often rapidly changing due to the variance of inter-thread interactions at different stages of the execution. Existing schedulers are restricted by such an issue, which fail to explore a sufficient amount of possible thread permutations: such a failure results in a limited amount of explorations, and can only lead to sub-optimal solutions when accounting for the whole search space (i.e., all possible thread permutations)[8]. Note that such a sparsity issue consistently exists in the Thread-Interaction Matrix, regardless of different scheduling objectives: this is because reconfiguring scheduling objectives can not affect scheduling policies in the current formalization; therefore, it can not make schedulers to explore more search space.

[6] Namely, scheduling policies are treated as the central consideration point for our categorization.

[7] The collected statistics are generally from hardware performance counters that can reflect the inter-thread relationships in terms of the hardware usages. The values are collected for the assistance to the schedulers for better decision-making, which are the same mindset at the design-time for these schedulers.

[8] This conjecture is based on the reasoning that: scheduling decisions are made without the knowledge of all inter-thread relationships (e.g., potential hardware resource conflicts), by solely focusing on the scheduled threads and neglecting the rest.

2.3 A Quantitative Characterization on the Sparsity Issue of the Thread-Interaction Matrix among Mainstream Schedulers

To better understand the severity of the sparsity issue, we conduct a characterization study using five mainstream schedulers, and report the lowest two sparsity levels among the five schedulers. The reported results are obtained by Random Scheduler (RS) and Global Round-Robin Scheduler (G-RRS). Compared with the rest three, these two schedulers have lower sparsity, because they can explore more kinds of thread permutations due to their scheduling policy designs. All schedulers are experimented on PARSEC Benchmarks [18], and the number of workload threads is configured as 32 and 64 respectively. Our characterization setup is consistent with our experimental infrastructure (16-core), which is detailed in Table 2. For each scheduler in our characterization, we record all permutations of different running threads on each core, and represent the sparsity in each scheduler design by the ratio of the recorded permutations to the number of all possible thread permutations[9]. Note that, for thread permutations, we consider both thread combinations and placements (i.e., which core). The time interval for our recording is set as 100 nanoseconds, which is a finer granularity compared to that of a context switch in our experimental settings. Figure 1 shows the normalized results regarding the number of thread permutations explored by RS and G-RRS for each benchmark; and we make two key observations.

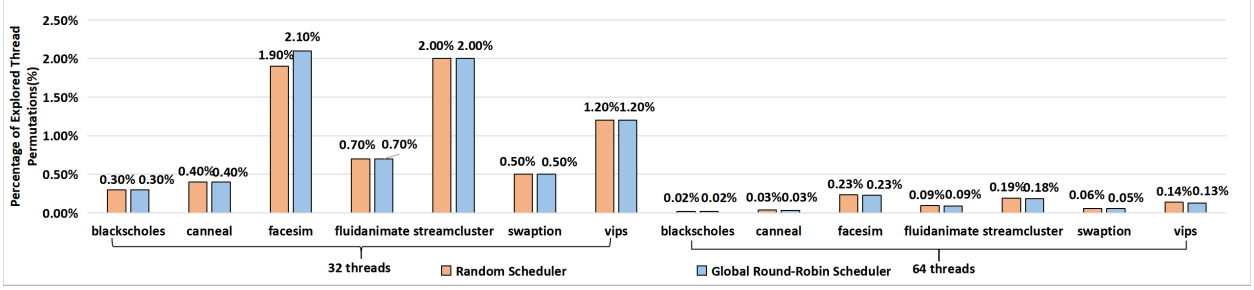

Figure 1: A quantitative evaluation on the sparsity of the Thread-Interaction Matrix in percentage of explored thread permutations (lower means sparsity is more severe), for Random scheduler and Global Round-Robin scheduler.

• **Observation 1: the sparsity is highly severe within the Thread-Interaction Matrix.** Our results reveal that the sparsity of the Thread-Interaction Matrix is extremely severe: across all benchmarks, the explored number of thread permutations only occupies up to 2.10% of all possible thread permutations. Therefore, building sophisticated heuristics upon such a coverage can mostly lead to a sub-optimal solution, because there are still a large number of unexplored thread permutations. Furthermore, the sparsity issue becomes more severe, when the number of threads increases. For 64 threads, the explored number of thread permutations can only occupy up to 0.23% of all possible thread permutations.

• **Observation 2: the sparsity is impacted by both scheduling policies and workload characteristics.** Our results reveal that there exist varied trends of sparsity, when using different schedulers for different benchmarks. Particularly, we observe that IO-intensive workloads make the sparsity issue of schedulers less severe than compute-intensive workloads (e.g., blackscholes, canneal). This is because IO-intensive workloads can incur more context switches, and the explored number of possible thread permutations are higher due to the increased amount of context switches.

[9]For 32 threads, there are $C\binom{32}{16} = 6,010,080,390$ possible thread permutations; and for 64 threads, there are $C\binom{64}{16} = 488,526,937,079,580$ possible thread permutations.

Our characterizations confirm that existing scheduler designs can cause an extremely-high level of sparsity in the Thread-Interaction Matrix. Therefore, building sophisticated heuristics, upon such a sparse matrix, makes the schedulers consider only a tiny subset of all possible thread permutations, and such scheduling decisions shall only be considered sub-optimal since the whole search space is not well-explored. Different from previous studies, we take a different view on the design of thread schedulers: rather than improving the mechanism with a limited amount of available information regarding thread interaction; can we effectively estimate all missing information from the Thread-Interaction Matrix, and directly perform scheduling decisions based on such estimations? Therefore, **our goal** in this work is to deliver a customizable and scalable solution for thread scheduling, which can effectively estimate the missing values from the Thread-Interaction Matrix for diverse scheduling objectives; and properly exploit such estimations for better scheduling.

3 SLITS: Design Overview

We introduce *Sparsity-Lightened Intelligent Thread Scheduling* (SLITS), a general thread scheduler design on multi-core architectures, by mitigating the sparsity issue of the Thread-Interaction Matrix. We first introduce the workflow of SLITS (Section 3.1). Then we elaborate three key components of SLITS, including Profiling Thread Interactions (Section 3.2), Estimating Missing Values in the Thread-Interaction Matrix (Section 3.3) and *Lazy Reschedule* (Section 3.4).

3.1 SLITS Workflow

Figure 2 presents an overview of the design and implementations of SLITS. There are three main components in SLITS to work serially, which include (1) profiling thread characteristics,(2) estimating missing characteristics via Factorization Machine, and (3) *Lazy Reschedule* via customized rescheduling policies. Here, we introduce each step in detail. As shown in Figure 2, first, in Step ❶, the Thread-Interaction Matrix about different levels of inter-thread interactions is profiled; then, in Step ❷, the sparsity in the primary Thread-Interaction Matrix will then be mitigated by leveraging Factorization Machines (FM) to estimate the missing values; and finally, in Step ❸. the estimation results will be used for *Lazy Reschedule*, where SLITS performs customized rescheduling policies under a uniform trigger condition[10].

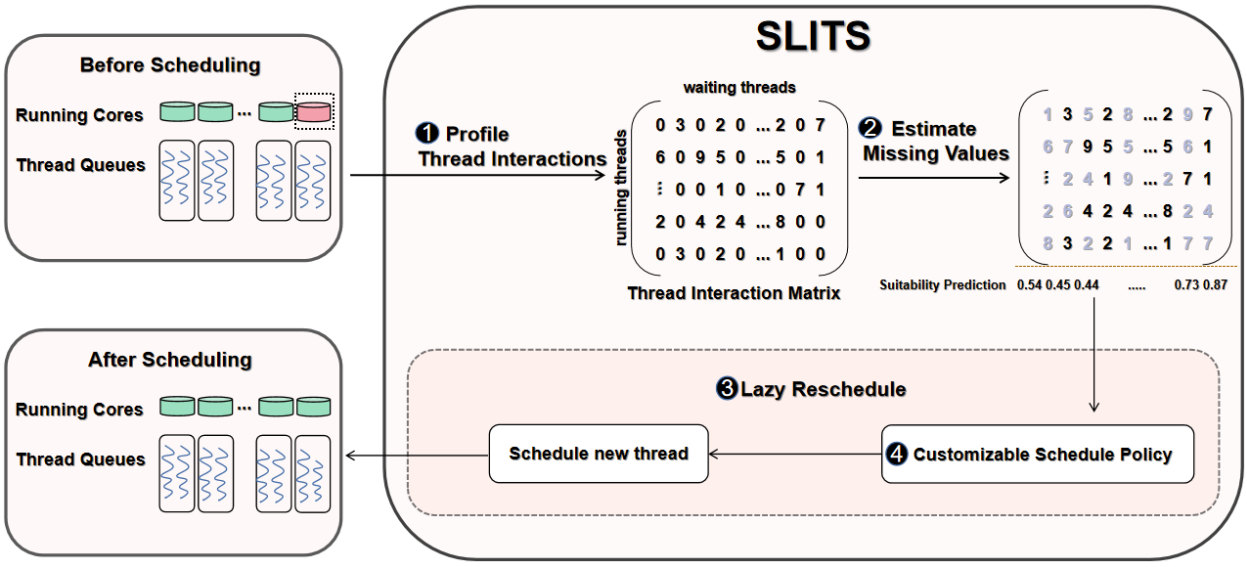

Figure 2: SLITS Workflow

3.2 Step 1: Profiling Statistics for Thread-Interaction Matrix

Profiling thread-interaction statistics is to provide quantitative information, regarding the inter-relationships among different threads. SLITS leverages the inter-thread statistics to form the Thread-Interaction Matrix, by tracking the performance counters. In this work, we focus on generally-accessible statistics on commercial processors (e.g., branch mispredictions, cache misses), that are frequently updated by hardware performance counters with low overhead [38, 7]: these statistics are easily-accessible by incorporating interfaces such as *perf_ events*. Note that SLITS can

[10]Note that Step ❹ refers to the customized scheduling policy in SLITS, which we cover more details in Section 4.3.

also benefit from the profile of specialized statistics, with the support of specialized hardware monitors (e.g., load misses, sharing misses [28])[11]. With these supports, SLITS can utilize a diverse set of low-level statistics without additional hardware overheads, to obtain the inter-thread interactions among different threads.

3.3 Step 2: Estimating Missing Statistics via Factorization Machines

Deploying matrix-oriented approaches (e.g., Collaborative Filtering, denoted as CF), for resource management and scheduling, are already discussed (e.g., Paragon [26] and CuttleSys [48]). However, the outstanding issue from CF-based approaches is that: the sparsity issue is highly severe when constructing the matrix. Such an issue in CF has already been evidenced in commercial recommender systems, which the original design goal of CF targets [6]. We elaborate this issue in more detail. CF delivers User-Item Recommendation in the sparse User-Item matrix, which analyzes the patterns of co-occurrence from the past behaviors of users[12]. Since the analysis of user/item similarities usually demands correlation statistics, the sparsity issue prevents CF to deliver proper recommendations with a large-yet-mostly-unavailable matrix.

To address the sparsity issue when estimating missing statistics, *Factorization Machines* (FM) are proposed and proved to be very effective for online recommender systems [66]. As the second step of SLITS, we utilize *Factorization Machines* to effectively lighten the sparsity in the Thread-Interaction Matrix, so that scheduling policies can properly exploit a sufficient amount of information, in terms of inter-thread relationships, in a quantitative manner. Such a design choice allows SLITS to be combined with a variety of scheduling policies, to potentially satisfy different scheduling objectives (if needed/required at design time).

The power of *Factorization Machines* is delivered by its focus on Feature Interaction. Originally derived from Matrix Factorization, *Factorization Machines* are a novel class of machine learning models, that can estimate the interactions between unobserved variables by using factorized parameters, with generality and linear training-time complexity on large-yet-sparse datasets. FM achieves better generality by using real-valued feature vectors to represent, learn and infer the unobserved variable interactions. Hence, we consider FM as a suitable candidate for the estimator of the Thread-Interaction Matrix, since every positively-defined matrix can be transformed into real-valued vectors. Through FM, SLITS can effectively complement the extremely-sparse Thread-Interaction Matrix via estimation. To ensure the consistency of understanding, we brief *Factorization Machines* below. We denote that FM models the interactions between two variables as follow[13].

$$\hat{y}(\mathbf{x}) := w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \mathbf{v}_i, \mathbf{v}_j \rangle \, x_i x_j \tag{1}$$

$$w_0 \in R, \quad \mathbf{w} \in R^n, \quad \mathbf{V} \in R^{n \times k} \tag{2}$$

where n is the number of features, k is the number of factors. A row vector $\mathbf{v}_i$ depicts the k embedding latent factors for the ith feature.

[11] It indicates that: the profiling process can be better supported with advanced studies on performance characterizations (e.g., [43, 82]), and these studies can be performed (e.g., via compiler-assisted approaches [45], etc.).

[12] It is usually done by quantifying user/item similarities (e.g., Pearson Correlation Coefficient [61]).

[13] In Equation 1: w_0 is the global bias, w_i represents the weight of *ith* feature; Equation 2 describes the domain set of model parameters, to complement Equation 1; and Equation 3 depicts details from the inner product of two vectors of length k and $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ is the weight that profile the interactions between $\mathbf{v}_i$ and $\mathbf{v}_j$.

$$\langle \mathbf{v}_i, \mathbf{v}_j \rangle := \sum_{f=1}^{k} v_{i,f} \cdot v_{j,f} \tag{3}$$

The generality of FM is credited to the weighted values of feature combination of x_i and x_j, which is the inner product $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$. For each feature, FM learns a row vector of size k, which can be abstracted as single feature embedding. Therefore, for features say x_n and x_m that have never appeared together in the history, which their correlations are expected to be unknown, can now be learned and inferred by FM. As long as x_n has feature combinations with any other features before, FM can learn its own corresponding embedding vector. Thus, the feature combinations of x_n and x_m can be used by leveraging their own corresponding embedding vectors learned from the historical records and their inner product that represents the weight of this new combination, achieving the estimation on a large sparse dataset based on limited knowledge. Also, FM has linear time complexity. The primary FM model equation (Equation 1) which is in $O(kn^2)$ time complexity, can be simplified to $O(kn)$ time complexity by reformulating the equation using factorization (for detailed proof, we refer readers to the original paper [66]).

3.4 Step 3: *Lazy Reschedule*

The trigger condition (for rescheduling) and the rescheduling policy are two major design points, that can impact the overall costs of rescheduling. The final step of SLITS first exploits proper trigger conditions to reschedule threads, based on different scheduling policies. Since SLITS can be customized for different scheduling objectives (detailed in Section 4.3), we deliver a general condition to trigger rescheduling. We propose *Lazy Reschedule*, to effectively balance the tradeoffs between rescheduling costs and performance gains from different scheduling policies. Prior work (e.g., Round-Robin Scheduler) set the trigger condition statically, by periodically rescheduling new threads. However, this approach has two constraints that can cause significant performance overheads, which we brief them as follow.

1. The time interval for periodic scheduling needs to be manually set, which demands extra knowledge about the workloads. The assumption does not apply to real-world scenarios, where the workload threads may change in terms of either quantity or categories (i.e., compute/IO-intensive). Therefore, a suitable determination of time interval in a fixed manner is too hard to be determined proactively.

2. When the trigger condition is satisfied, the scheduler needs to pause the running threads and schedule new threads, the costs of context switches hereby can potentially cause great performance overheads, which can outweigh the benefits from thread rescheduling.

Lazy Reschedule, as the trigger condition for thread rescheduling, can balance the costs of rescheduling and the benefits of better thread permutations. The idea of *Lazy Reschedule* is simple: setting the trigger condition to reschedule based on whenever any running thread stalls/exits. The rationale comes from two simple-yet-effective considerations, as described below:

1. Compared to fixed periodic rescheduling polices, scheduling new threads at the time of thread stalls/exits can cause less interference of workload thread executions. This is evidenced by our experimental results from the breakdown analysis (as detailed in Section 6.5).

2. There are no additional hardware overheads to enable such trigger conditions for thread rescheduling. Therefore, the setting of such trigger conditions allows SLITS to be easily implemented, reconfigured and customized furthermore.

Note that the rationale behind this method also consider the situation that CPUs are overloaded: *Lazy Reschedule* can also minimize the competition between workload threads and FM module, and our evaluation results (detailed in Section 5) also take this into the consideration for a quantitative comparison.

4 SLITS: Detailed Designs

Based on SLITS design, we present detailed designs of SLITS to enhance its practicality. We first propose Thread Interaction Statistics (TIS) Cache, to facilitate the storage of the Thread-Interaction Matrix while guaranteeing fast accesses (Section 4.1). Then we detail how SLITS utilizes the Thread-Interaction Matrix in *Factorization Machines*, with a particular focus on Thread Interaction Statistics (Section 4.2). Finally, we demonstrate the feasibility of SLITS customization, via two concrete examples of SLITS variants for different scheduling objectives including Resource Contention and Fairness (Section 4.3).

4.1 Thread Interaction Statistics Cache (TIS-Cache)

To store, update and utilize statistics from Thread-Interaction Matrix efficiently, it is essential to provide a separate module to store such a matrix, while guaranteeing fast access. In SLITS design, we propose Thread Interaction Statistics Cache (TIS-Cache), by exploiting variants based on recently-discussed L4 eDRAM caches (e.g., [13, 54, 63]). Among a large body of studies on large L4 caches [54, 63, 69, 41, 40, 77, 37, 76], we follow the latency-optimized Alloy Cache design for TIS-Cache design [63]. Similar to Alloy, TIS-Cache stores a tag and data pair within the same DRAM row (page) to access them with a single DRAM command. TIS-Cache also uses a direct-mapped organization to reduce cache hit latency and to better exploit spatial locality by mapping consecutively-addressed cache lines to the same DRAM row. The consequent impact of reduced associativity is relatively minor. Following the most recent trends, we use on-package eDRAM instead of ordinary DRAM chips to further reduce access latency.

The consideration of how large the TIS-Cache shall be is critical in SLITS design. Instead of analyzing the required capacity of TIS-Cache under the scope of the current SLITS design, we take the current design of L4 Cache to reason about how large a Thread-Interaction Matrix can be stored. The current L4 Cache design can reach up to 128MB capacity [34, 74]. Then, we estimate the size of a Thread-Interaction Matrix of the workload threads, since it can be expressed as the square of the number of working threads (in bytes): given a 128MB TIS-Cache, the Thread-Interaction Matrix can contain the statistics for 11,585 working threads. Hence, the current design of TIS-Cache is feasible, and sufficient to handle Thread-Interaction Matrix for a single chiplet (i.e. under 16-core for our evaluation settings, which are also common on current multi-core products). Table 1 shows the number of working threads and the required capacity for TIS-Cache, under our experimental settings. Our analysis confirms that TIS-Cache can be shrunk to a relatively small size, to facilitate the need for a limited area and power budget.

Table 1: The Number of Working Threads and the Required TIS-Cache Capacity accordingly.

Threads No.	32	64	128
TIS-Cache	1KB (0.001MB)	4KB (0.004MB)	16KB (0.016MB)

4.2 Using Factorization Machines for Thread-Interaction Matrix

The FM estimation is executed by SLITS on the stalled/idle cores, which do not affect other running cores. After accessing statistics from the Thread-Interaction Matrix, the scheduler thread can perform the estimation and further conduct a customized thread scheduling policy based on the estimation outputs. To ease the understanding of how such a procedure works, we hereby

elaborate on how the FM module of SLITS abstracts and exploits input vectors from the Thread-Interaction Matrix, to address the sparsity issue via a concrete example. We first denote the profiled statistics O in the Thread-Interaction Matrix in the format of (rt_i, wt_j, M_{ij}) which stands for (*RunningThreadID*, *WaitingThreadID*, *InterThreadStat*[14]), and represent thread interactions that are useful to estimate the unavailable statistics between rt_1 and wt_{32} as below.

$$\begin{aligned}\mathrm{O} = \{&(\mathrm{rt}_1, wt_{33}, 453), (rt_1, wt_{62}, 140), (rt_{34}, wt_{32}, 655),\\ &(\mathrm{rt}_{34}, wt_{62}, 670), (rt_{61}, wt_{33}, 115), (rt_{61}, wt_{62}, 640)\}\end{aligned}$$

To estimate the missing statistics between rt_1 and wt_{32} using the above statistics, factorized parameters $\langle \mathbf{v}_{rt_1}, \mathbf{v}_{wt_{32}} \rangle$ can achieve this purpose. Note that the execution can simultaneously perform on multiple vectors and we elaborate the rationale of this approach using the following three steps. First, rt_{34} and rt_{61} has similar factorized vectors $\mathbf{v}_{rt_{34}}$ and $\mathbf{v}_{rt_{62}}$ since they obtain similar statistics when co-running with thread 62. This means $\langle \mathbf{v}_{rt_{34}}, \mathbf{v}_{wt_{62}} \rangle$ and $\langle \mathbf{v}_{rt_{61}}, \mathbf{v}_{wt_{62}} \rangle$ have to be similar. The factorized vector for rt_1 is different from that of rt_{61} as they have different statistics with threads 62 and 33. Next, the factorized vectors of wt_{62} and wt_{32} are inclined to be similar as rt_{34} has similar trends when co-running with these two threads separately. Therefore, it indicates that the dot product of rt_1 and wt_{32} will be similar to the one of rt_1 and wt_{62}, which will also cause high similarity in inter-thread interaction statistics.

4.3 Customizing SLITS for Different Objectives

SLITS can be customized based on different scheduling objectives, by properly co-designing Thread-Interaction Matrix and the scheduling policy. Hereby, we propose two SLITS variants to address two scheduling objectives (i.e., Resource Contention and Fairness) respectively, as the proof-of-concept of the customizability of SLITS[15]. We first describe SLITS-(Contention), a variant to improve the performance by mitigating resource contention (Section 4.3.1); and we introduce SLITS-(Fairness), a variant to improve the performance by preserving the fairness among all threads (Section 4.3.2).

4.3.1 SLITS-(Contention): Mitigating Resource Contention

SLITS-(Contention), as a motivating design for mitigating Resource Contention, needs to (1) configure Thread-Interaction Matrix to use a particular type of statistics for resource contention; and (2) design the scheduling policy to minimize the level of the contention on such a resource. For Thread-Interaction Matrix, SLITS-(Contention) uses Last-Level-Cache (LLC) Misses as the metric to measure thread contention. This design choice is built upon a series of prior work on Resource Contention [39, 72, 35, 80], which demonstrates that: LLC Misses is one of the most representative metric to abstract resource contention for threads on multi-/many-core architectures. As for the scheduling policy, SLITS-(Contention) simply champions the thread with the least amount of (estimated) contention for minimizing resource contention. There are three steps. ❶ SLITS-(Contention) filters out threads that are not available to run on the idle core from all waiting threads. ❷ the estimation results from FM are averaged based on running threads, and the results on waiting threads allow SLITS-(Contention) to rank them and select the most suitable thread(s) to run, which has the maximum value from the average results of the FM estimation. ❸ The selected thread is rescheduled to the idle core, and the corresponding auxiliary features (e.g., last co-run threads on different cores) are updated for further inter-thread interaction profiling and FM estimations.

[14]Hereby, we use LLC Misses as an example. There are no additional challenges to use other types of the metrics.

[15]Indeed, SLITS can be designed with more sophisticated types of profiled statistics and detailed scheduling policies, and how these design choices can bring trade-offs in SLITS. We believe this would an interesting direction for future exploration of novel scheduler designs.

4.3.2 SLITS-(Fairness): Preserving Thread Fairness

Similar to the design methodology above, SLITS-(Fairness) requires properly co-design of these two components by (1) configuring the Thread-Interaction Matrix to use the information for quantitative reflections of thread fairness; and (2) utilizing a scheduling policy to ensure fairness across all threads. For Thread-Interaction Matrix, SLITS-(Fairness) applies Instruction per Cycle (IPC) to quantitatively examine the fairness of each thread. This design choice is built upon a line of prior work [44, 14, 15], which demonstrates that: IPC is proper to examine the fairness by ensuring the distribution of workload threads across cores, since the fairness is determined by the allocated hardware resources for each thread (i.e., CPU cycles). As for scheduling policy, SLITS-(Fairness) utilizes a simple-yet-effective approach: by grouping all threads with the IPC below the median, SLITS-(Fairness) randomly selects a thread with such a subset of threads. This is because SLITS-(Fairness) only needs to exploit partial order information to ensure the fairness, rather than always championing the most/least one.

4.3.3 Customizing SLITS for More Scheduling Objectives

SLITS can be customized for more scheduling objectives, following the methodology described above. We clarify that we are well aware that priority is also an important thread scheduling objective (e.g., [68, 83]). However, from the perspective of SLITS formalization, we can assume priority-oriented Scheduling objectives share the core spirits as fairness-oriented ones: they are fundamentally similar since scheduling for fairness can be treated as a dynamic priority problem. Hence, due to the limited space, we do not cover the design for SLITS-(Priority) but such a design goal is certainly feasible under our formalization and methodology.

5 Experimental Methodology

We first describe the experimental infrastructure in Section 5.1. Then we enumerate all comparison points in our experimental studies in Section 5.2. Next we elaborate the configurations of all workloads in Section 5.3. Finally, we describe performance metrics during our study in Section 5.4.

5.1 Experimental Infrastructure

We use an extended version of SNIPER [21], a fast and accurate x86 simulator that supports detailed simulations for the multi-threaded workload. Table 2 provides detailed configurations of our simulation infrastructure. We use Pin tool [56] as the frontend. To integrate Factorization Machine (FM) with SNIPER, we exploit the FM module from the xLearn library [3], the state-of-the-art package for large-scale Machine Learning system developments and deployments. To obtain realistic modeling of hardware scheduling, we account for all overheads on storing, updating and computing of the FM module, by extending the SNIPER simulator.

Table 2: Detailed configurations of the simulated system.

Components	**Specifications**
Processor	16-core Intel Xeon X5550 Gainestown, 2.6GHZ, 128-entry instruction window
Branch Misprediction Penalty	8 cycles
Branch Mispredictor Size	1024
Reorder Buffer Size	128
L1 I-Cache/D-Cache	32/32KB, 4/8-way assoc, LRU Replacement Policy, SRAM
L2 Cache	256KB, 8-way assoc, LRU Replacement Policy, SRAM
L3 Cache	8MB, 16-way assoc, SRAM
TIS Cache (L4 Cache)	16KB, 1 assoc, eDRAM
DRAM	4 controllers, 4 DIMMs/controller, 8 chips/DIMM, 45ns access time

5.2 Comparison Points

To provide a comprehensive coverage of comparison points, we select four general-purpose thread schedulers for examining the effectiveness of two variants of SLITS. Therefore, comparison points during our experimental studies include:

1. **Global Round-Robin Scheduler (denoted as G-RRS)**: G-RRS is selected as baseline for our experiment, which schedules waiting threads, grouped in a global queue, to each idle core in a Round-Robin manner.

2. **Per-Core Round-Robin Scheduler (denoted as P-RRS)**: P-RRS groups threads into multiple waiting queues for different cores, and schedule them within each core via a Round-Robin manner.

3. **Work-Stealing Scheduler (denoted as WSS)**: Work-stealing scheduler periodically sets a random core as the thieve and “steals" one waiting thread from a random victim core’s waiting queues if the thief has empty waiting queues. Otherwise, all the cores schedule their own threads in a Round-Robin manner.

4. **Completely Fair Scheduler (denoted as CFS)**: CFS is the *de facto* design for thread scheduling, which is currently implemented as the Linux default scheduler[16]. CFS assigns each thread with equal costs for fairness, and schedules thread from per-core waiting queues, where threads are sorted by Red-Black Trees in a temporal order.

5. **SLITS-(Contention)**: SLITS variant for mitigating resource contention, by using LLC Misses as the metric (as described in Section 4.3.1).

6. **SLITS-(Fairness)**: SLITS variant for preserving the fairness (as described in Section 4.3.2).

5.3 Workload Configurations

We select representative benchmarks, which contain both computation-intensive and IO-intensive workloads, from the PARSEC Benchmark Suite [18] and SPLASH-2 Benchmarks [79] including High Performance Computing applications across multiple domains. More specifically, we configure *sim-large* input size for *blackscholes*, *canneal*, *facesim*, *fluidanimate*, *swaptions* and *vips* from PARSEC benchmarks and *ocean.cont*, *ocean.ncont*, *water.nsq*, *fft_O1* and *raytrace* from SPLASH-2 benchmarks. All experiments are run on 16-core settings with diverse workload thread configurations (i.e., 32/64/128 threads, with 2/4/8 threads assigned to each core for WSS, P-RRS, CFS and two variants of SLITS).

5.4 Performance Metrics

We measure the system performance using the speedup over the baseline schedulers (i.e., Global Round-Robin Scheduler). To minimize the impacts of performance variations, we run 1,000 times for each evaluated workload and report the average results and measure the end-to-end execution time for each run. Note that it is infeasible to examine the effectiveness of SLITS, by using the accuracy metrics directly, since the statistics of thread interaction can not be formed as the ground truth with the current system formalization, due to the non-determinism of thread interactions.

[16]It is notable to mention that, [55] identifies a serious performance bug of CFS. Quoted from [55]: "The scheduler unintentionally and for a long time leaves cores idle while there are runnable threads waiting in runqueues". We do not disadvantage CFS in our experimental studies.

6 Experimental Results

In this section, we report the evaluation results by comparing two SLITS variants against all general-purpose schedulers, in a diverse set of thread-to-core configurations. Specifically, we first report the experimental results when configuring SLITS to reduce Resource Contention; and to guarantee Fairness during thread scheduling under different scalability settings (Section 6.1). Then we present our extensive studies, including (1) a breakdown analysis, to examine the cache behaviors of SLITS (Section 6.2); (2) a isolation study by disabling the FM module and *Laze Reschedule* mechanism respectively (Section 6.3); (3) a sensitivity study by varying the estimation coverage of SLITS, to justify the effectiveness of estimating missing values (Section 6.4); (4) an ablation study to showcase the merits of the proposed *Lazy Reschedule* mechanism, over the periodic scheduling policies (Section 6.5); and (5) an analysis on the overhead of SLITS in terms of chip area, power and latency on commercial processors (Section 6.6).

6.1 SLITS versus Other Schedulers

We report the performance results between SLITS using Resource Contention/Fairness as the scheduling objective and all the selected schedulers across all the benchmarks. Figure 3 reports the results of SLITS (i.e., SLITS-(Contention) for scheduling for minimum resource contention and SLITS-(Fairness) for Fairness) when comparing the performance with the baseline scheduler in terms of the speedup. We make three observations.

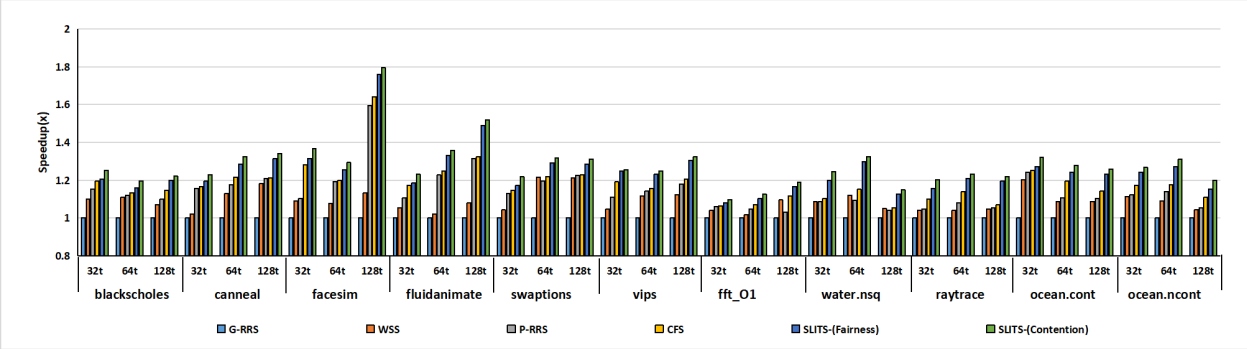

Figure 3: Speedup over G-RRS under 16-core and 32/64/128-thread configurations on PARSEC and SPLASH2 benchmark suites, with 2/4/8 threads assigned to each core. Note that SLITS-(Contention) and SLITS-(Fairness) refers to SLITS using cache misses and IPC for abstracting the Thread-Interaction Matrix.

First, SLITS variants always deliver considerable performance benefits, regardless of which different scheduling objective is customized for (i.e., mitigating Resource Contention or preserving Fairness) over all general-purpose schedulers. When customized for reducing Resource Contention, SLITS achieves 1.27X, 1.18X, 1.12X, and 1.09X speedup, compared with G-RRS, WSS, P-RRS, and CFS respectively, averaged across all workloads. When customized for Fairness, SLITS achieves 1.25X, 1.14X, 1.09X, and 1.06X speedup, compared with G-RRS, WSS, P-RRS, and CFS respectively, averaged across all workloads. Our results show that SLITS design can greatly improve the overall system performance. Moreover, the consistency between performance benefits, obtained from two SLITS variants, demonstrates that the merits of SLITS customizability: SLITS can be customizable to different scheduling objectives, and configuring different scheduling objectives can be easily achieved by changing metrics; and they can achieve the performance benefits consistently.

Second, SLITS exhibits great scalability, by outperforming all general-purpose schedulers with different settings of the thread amount. For most of the selected benchmarks (e.g., *canneal*, *fluidanimate*, *fft_O1*, *raytrace*), the performance benefits of SLITS increase gradually from 32 threads to 128 threads, and peaks at the setting of 128 workload threads. The relatively-low performance benefits in 32-thread settings are due to the narrowed optimization space, since only two threads are assigned to each core. Though, for some other workloads (e.g., *facesim* in 64-thread setting, *ocean.ncont* in 128-thread setting), SLITS achieves sub-optimal performance, compared with the benefits on the same workload under different scalability settings: we argue that it is because the baseline scheduler (i.e., G-RRS) can obtain better performance in these cases. SLITS still achieves an increasing trend, in terms of the speedup, from the 32-thread setting to the 128-thread setting over other schedulers (e.g., speedup over CFS by 1.07X, 1.08X, and 1.09X on *facesim* in 32, 64, and 128-thread settings). Therefore, this justifies that SLITS has great potential to be configured for different scheduling objectives in multi-threaded workloads, where the number of cores and threads can be up to thousands.

Third, the benefits of SLITS correlate with the workload characteristics. SLITS obtains more speedup on workloads with more instructions or synchronization primitives. For instance, among all the selected PARSEC benchmarks, SLITS achieves more speedup on *facesim*, *fluidanimate* and *vips* than on *canneal* and *blackcholes*, compared with the results from all general-purpose schedulers. This is because the former type of benchmarks has more instructions or synchronization primitives than the latter one [79, 18]. Such characteristics benefit SLITS in the following two ways: (1) the profiling of run-time inter-thread interactions can be more accurate when more instructions are executed, which can further benefit the estimation of FM; and (2) there are more re-scheduling opportunities for optimizing scheduling decisions, when a large quantity of synchronization primitives exist during the execution, and *Lazy Reschedule* can achieve more benefits, given a longer period of the execution. Similar observations also apply to all the selected benchmarks from SPLASH-2, where the speedup of SLITS on *water.nsq* and *raytrace* overrides the speedup on *fft_O1*, *ocean.cont* and *ocean.ncont*.

6.2 Breakdown Analysis of SLITS

To better understand the source of the performance benefits from SLITS, we reveal the cache behaviors of SLITS and the baseline schedulers under the same evaluation settings (as described in Section 6.1). Specifically, we measure the cache hit counts and the normalized Cycles Per Instruction (CPI) for each cache level. We configure each benchmark with 64 workload threads. Note that we only report the best cache utilization results from all baseline schedulers; and the worst results of SLITS among SLITS-(Fairness) and SLITS-(Contention), based on our studies from Section 6.1.

Figure 4 reports the experimental results. We make two observations from the experimental results. We observe that, in terms of the cycle stack usages, SLITS achieves an average reduction of 10.05% cycles on the L3 cache. This reveals that the latency penalty, for fetching data from the L3 cache, is greatly mitigated by SLITS. Therefore, we conclude that the performance benefits from SLITS are credited to the better utilization of on-chip caches.

Figure 5 reports the detailed results in terms of the L1 D-Cache Hits. We confirm that SLITS benefits from a great increase of the cache hits. On average, SLITS obtains 21.45% L1 D-Cache hits improvement across all the selected benchmarks, compared with the baseline schedulers. This is because the FM module can effectively estimate inter-thread relationships, which leads to the better utilization of on-chip caches.

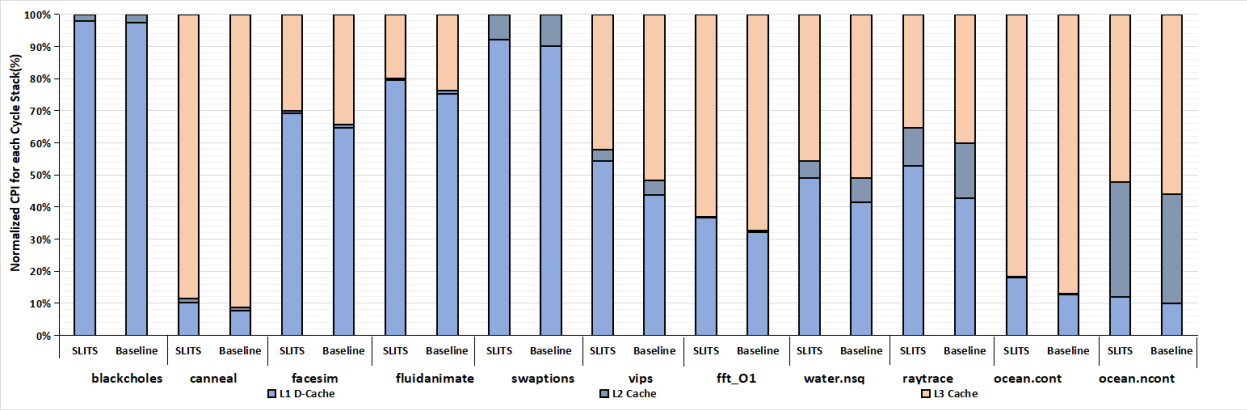

Figure 4: Normalized CPI percentage in L1 D-Cache, L2 cache and L3 cache of SLITS and baseline schedulers. We report the best results among all baseline schedulers and the worst results of SLITS.

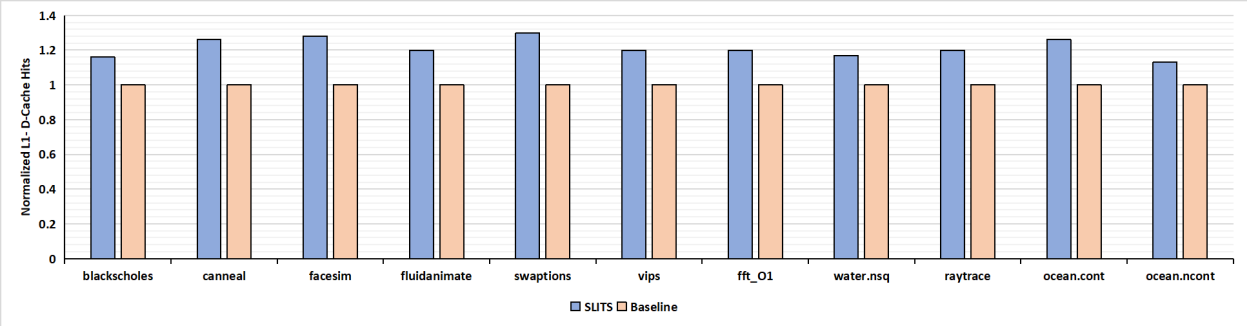

Figure 5: Normalized L1 D-Cache hits of SLITS and baseline schedulers.We report the best results among all baseline schedulers and the worst results of SLITS.

6.3 Breakdown Analysis of Different SLITS Components

To better understand the benefits of each core module of SLITS, we conduct a breakdown analysis, to differentiate the extent of benefits from the FM module and *Lazy Reschedule* mechanism respectively. We replace these two modules separately with the *de facto* scheduling designs, and compare their performance with the full design of SLITS on all the selected benchmarks. More specifically, for isolating the FM module, we disable the *Lazy Reschedule* mechanism by setting the trigger condition of SLITS into preemptive ones, which allocates each workload thread a pre-defined time quota (i.e., the same time quota as P-RRS's); and periodically triggers SLITS rescheduling once the quota is met for the running thread on each core; as for isolating the *Lazy Reschedule* mechanism, we disable the FM module and directly make it schedule threads in a Round-Robin manner on each core, except that there is no pre-defined quota for workload threads (i.e., non-preemptive rescheduling). We run these two experiments under the settings of 16 cores and 64-thread configurations. The rest of the setup is consistent with other experiments.

Figure 6 reports the experimental results of running individual module on the selected benchmarks. We make two observations. First, both of these two modules contribute greatly to the overall speedup. By disabling either of these two modules, SLITS encounters about 24.23%-32.73% performance degradation, compared with SLITS with both FM module and *Lazy Reschedule* mechanism. This justifies the feasibility of both modules. Second, disabling the FM module causes more loss of the speedup, compared with the loss caused by disabling *Lazy Reschedule*. We assume that

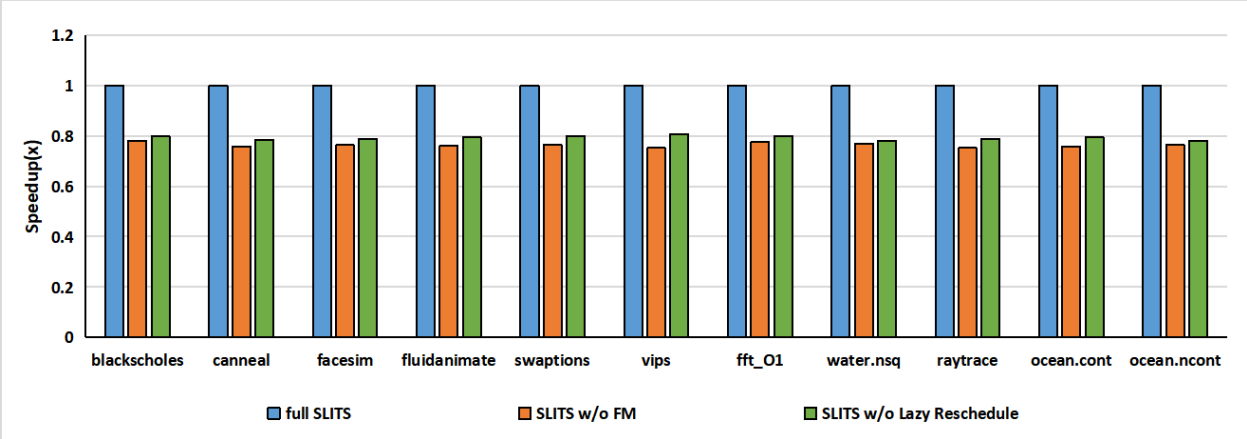

Figure 6: Performance slowdown compared with full SLITS by disabling FM module and *Lazy Reschedule* separately. SLITS w/o FM indicates SLITS scheduling without FM module and SLITS w/o *Lazy Reschedule* represents SLITS scheduling without *Lazy Reschedule*.

this phenomenon is because the feasibility of FM, to address the sparsity in the Thread-Interaction Matrix by estimating missing values, plays a key role in bringing performance benefits (compared with solely changing the trigger conditions to improve the performance).

6.4 Different Coverage of FM Module

To examine the sensitivity of the impacts from Factorization Machines (FM), we compare the performance of SLITS by setting different levels of FM coverage (when estimating missing values), which refers to the ratio of the Thread-Interaction Matrix using FM estimation. We set the FM coverage ratio in SLITS as 0%, 25%, 50%, 75% and 100%, and the default setting of SLITS is 100% FM coverage. Note that SLITS with R% FM coverage ($R \in \{25, 50, 75\}$) means that r% of the threads are scheduled scheduled using FM; and the rest is scheduled using Round-Robin policy./

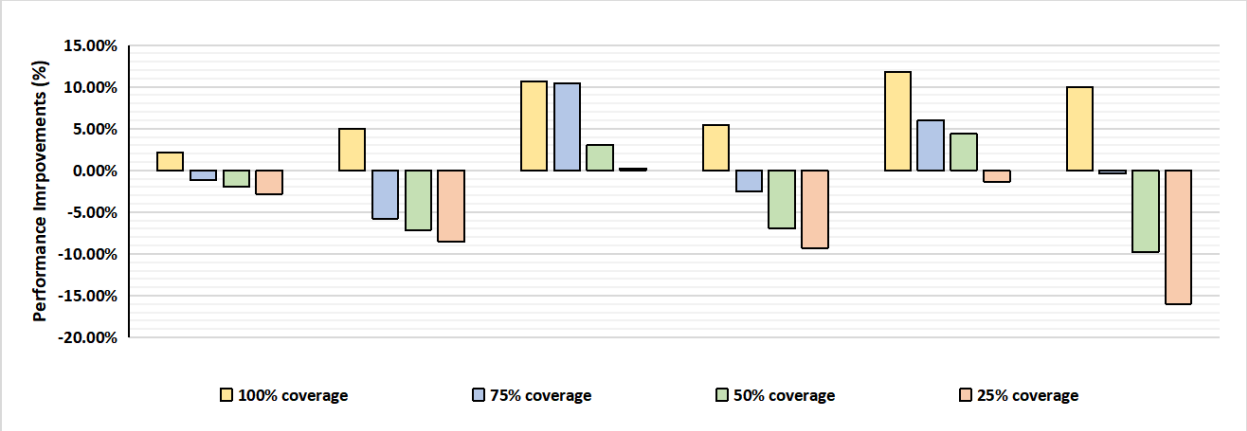

Figure 7: Speedup of SLITS with different levels of FM coverage over SLITS with 0% FM coverage. N% FM coverage indicates N% of the threads are scheduled using FM; and the rest are scheduled using Round-Robin policy.

Figure 7 reports the speedup of schedulers with different FM coverage, compared with P-RRS on six PARSEC benchmarks. We make two observations. First, the speedup exhibits a positive correlation with the increasing levels of FM coverage. On all workloads, SLITS with 100% FM coverage achieves the best speedup, compared with the baseline settings; and the performance degrades (close to) linearly, with the decrease of the level of FM coverage. This justifies the effectiveness of FM-driven estimation in the design of SLITS. Second, on most of the benchmarks, SLITS variants, without the full coverage of FM-driven estimation, can cause a significant slowdown, compared with SLITS with no coverage from FM. For instance, SLITS with 0% coverage outperforms SLITS with 25% coverage on all the benchmarks. This is because SLITS with limited coverage can only utilize a random subset for sparsity mitigation, which makes scheduling decisions sub-optimal; and such impacts are amplified with the decrease of the FM coverage.

6.5 Suitability of *Lazy Reschedule*

To examine the suitability of the proposed *Lazy Reschedule* mechanism, we rigorously compare the scheduler performance of SLITS against various periodic scheduling policies (in a fixed time-window setting)[17] on the representative benchmarks from our experimental settings. We configure the time interval for different periodic scheduling policies as 0.5ms, 1ms and 1.5ms, which periodically reschedule the running thread on a core while the determined time interval is reached[18]. We select benchmarks from the previous experiments that have a large number of instructions and synchronization primitives, to balance the number of rescheduling conducted by *Lazy Reschedule* from SLITS and periodic scheduling policies.

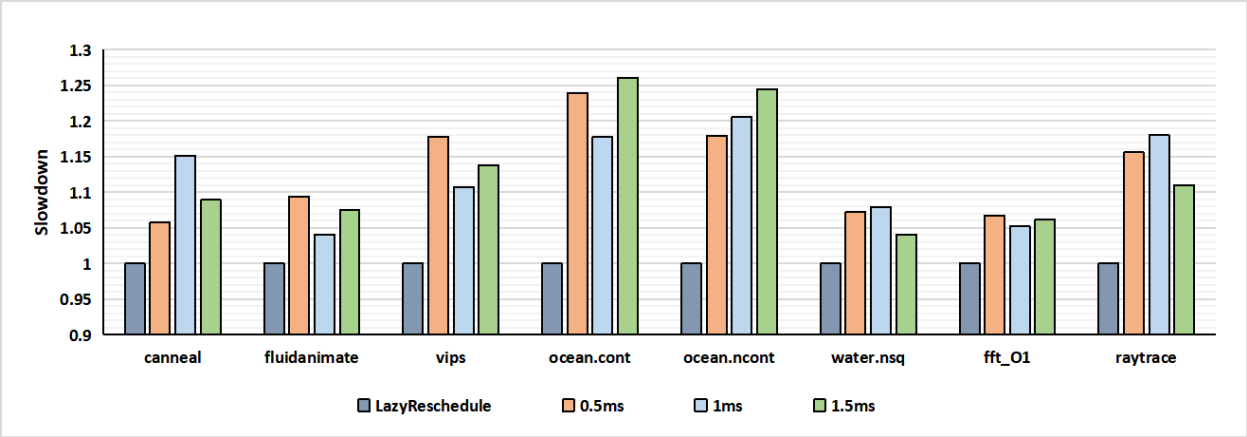

Figure 8: Slowdown of several periodic (i.e., fixed-window) scheduling policies, with different settings of time-interval size, over *Lazy Reschedule*.

Figure 8 shows the experimental results of the *Lazy Reschedule* and the comparisons against periodic scheduling policies. We make two observations. First, the *Lazy Reschedule* mechanism outperforms all the periodic scheduling policies on the selected benchmarks. Averaged across all the benchmarks, the *Lazy Reschedule* mechanism achieves 11.29%, 10.82% and 10.93% performance benefits, over periodic scheduling policies with time intervals configured at 0.5ms, 1ms and 1.5ms respectively. The underlying rationale for such benefits is that: periodic rescheduling incurs expensive-yet-unnecessary context switches, as well as resource waste for schedulers. Second, different periodic

[17]We use periodic scheduling to refer to fixed time-window scheduling in the rest of this section, for the consistency.
[18]The methodology for core selection is in a Round-Robin manner.

scheduling policies have varied performances on the selected benchmarks. For instance, a scheduling policy with a fixed-time window of 0.5ms achieves the best performance on PARSEC *canneal* benchmark (excluding *Lazy Reschedule*); whereas a scheduling policy with a fixed-time window of 1.5ms achieves the best results on SPLASH2 *water.nsq* benchmark. This consolidates our hypothesis that periodic scheduling policies, such as fixed-window scheduling, lack generality and practicality: since it requires prior knowledge to manually set up the time window of the scheduler to achieve a considerably-good performance. Rather than the inflexibility of periodic scheduling policies, the *Lazy Reschedule* mechanism of SLITS relaxes such constraints, and it can be complemented with various types of customized scheduling policies.

6.6 Overhead Analysis

We accurately estimate the chip and power overheads of SLITS (namely the overheads of TIS-Cache), based on Alloy Cache [63]. To contain the Thread-Interaction Matrix from our experimental settings (i.e. up to 128 threads for 16 cores), the size of TIS-Cache is required to be 16KB. In this case, TIS-Cache consumes 0.21 mm^2 of area and 35.27 mW of power in each core[19]. Then, we show the relative overhead of SLITS on two commercial processors (i.e., 18/28-Core Skylake processors [1, 2]) in terms of chip area and power in Table 3, where TIS-Cache requires $<$0.85% chip area and incurs $<$0.48% overhead on these processors. Note that, as we covered previously, the TIS-Cache design can be scaled to a much larger size if needed (i.e., up to 128MB eDRAM Cache in current products [34, 74]. If needed, the overheads from TIS-Cache are expected to be similar to recent proposals (e.g., [13]). Therefore, we conclude that SLITS incurs negligible hardware overheads.

Table 3: Area and Power Overhead of SLITS on Commercial Processors.

Processors	Area	Power
18-core Skylake 6150, 165W TDP [1]	0.79%	0.38%
28-core Skylake 8180M, 205W TDP [2]	0.85%	0.48%

[19]We replicate Thread-Interaction Matrix on each core, so that all cores can deliver fast access to this information.

7 Related Works

To the best of our knowledge, SLITS is the first work to ❶ provide a unified formalization of existing thread scheduler designs; ❷ concretely identify, address and mitigate the sparsity in multi-core thread scheduling; ❸ propose a Machine-Learning-based thread scheduler to improve the performance of multi-core architectures. In this section, we brief relevant works to justify the novelty.

7.1 Scheduler Designs for Fixed Scheduling Objectives

Most existing schedulers are designed with definite scheduling objectives, which can be broken into inter-thread statistics and further be effectively profiled using the Thread-Interaction Matrix. A major part of these schedulers aims to mitigate hardware contention in multi-core architectures to improve the performance, by leveraging run-time inter-thread statistics of resource contention in the Thread-Interaction Matrix and adaptively minimizing resource contention [46, 20, 17, 4, 32, 75]. Other scheduling policies are for Fairness and Priority, which we group together since Fairness can be viewed as dynamic priority. The values within Thread-Interaction Matrix can be constant in static-priority scheduling and the scheduling policies are usually designed as strict rules under certain conditions (e.g., pre-defined priority [22], Deadline-First Scheduling [51, 5, 57, 23], Deterministic Task Dependencies [67, 33], Round-Robin schedulers, Work-Stealing scheduling policy and its variants [27, 12, 47, 62, 52]). As for dynamic scheduling objectives, Pfair scheduling algorithms [14, 15] and its variants [10, 11] schedule both periodic [8, 53] and sporadic tasks [9, 73] on multiprocessors, where the value types from Thread-Interaction Matrix can be represented via corresponding metrics for dynamic priority. The novelty of SLITS, compared with the above works is two-fold: (1) SLITS can be customized with different scheduling objectives by configuring user-defined metrics and adjusting scheduling policies; and (2) SLITS is a Machine-Learning-based thread scheduler that effectively addresses the sparsity issue, which are overlooked by prior work.

7.2 Heuristic/Statistical-Approach-based Schedulers Designs

There are also a few proposals focusing on improving thread schedulers by employing more sophisticated heuristic/statistical methods. As covered above, all schedulers for mitigating resource contention can be classified as heuristic schedulers, since they always champion the optimal thread to reschedule (based on their limited subset of available statistics) [46, 20, 17, 4, 32, 75]. Besides heuristic methods, there are also statistical methods for better scheduler designs [65, 64]. Compared with these proposals, SLITS addresses the overlooked sparsity issue of the Thread-Interaction Matrix, and leverages *Factorization Machines* to mitigate this issue. Moreover, the above efforts can be potentially synergistic with SLITS, by coupling them as (parts of) scheduling policies of SLITS.

7.3 Exploiting Machine Learning Techniques to Assist the Schedulers

Exploiting Machine Learning (ML) techniques also receive some attention. One line of works aims to deliver accurate thread-interaction estimation for applications on modern multi-core architectures on resource contention [58, 50, 49]. The other line of works exploits ML techniques for coarse-grained task scheduling (i.e., process), such as [59, 60]. Compared with these prior work, SLITS (1) is the first machine learning-based scheduler by leveraging *Factorization Machines* as its core module, to provide the customizability for different scheduling objectives; (2) allows fine-grained scheduling (i.e. thread) and on-the-fly reconfiguration of scheduling objectives; and (3) mitigate the sparsity issue in the Thread-Interaction Matrix.

8 Conclusions and Implications

8.1 Conclusion from this work

We introduce SLITS (Sparsity-Lightened Intelligent Thread Scheduling), the first intelligent thread scheduling design to mitigate the inherent sparsity in the Thread-Interaction Matrix. SLITS exploits *Factorization Machines* to estimate the threads while maintaining customizability for different scheduling objectives, with a uniform mechanism called *Lazy Reschedule* to effectively balance the costs between rescheduling costs and performance gains. We introduce hardware modifications to support SLITS and concretely demonstrate how SLITS can be customized with different scheduling objectives. We extensively analyze the benefits of SLITS in terms of performance and scalability; and perform a detailed breakdown analysis to understand the source benefits of SLITS.

8.2 Implications: Generality and Applicability

Generality. Credited to the generality of SLITS design, it is potentially applicable to a variety of thread scheduling scenarios. SLITS can be applied for processors with Simultaneous Multi-Threading (SMT)/Hyper-Threading enabled, where multiple logical cores share the physical cores[20]. Therefore, it is highly demanded to schedule threads on the same physical core with the minimum level of resource contention. A line of works attempts to address this issue for better performance (e.g., Symbiosis Scheduler and its derivatives [71, 29, 31, 19]. In this manner, the key to delivering proper scheduling decisions is to exploit the contention statistics of different threads. SLITS can significantly benefit such scheduling by addressing the missing statistics on resource contention, namely the sparsity issue in the Thread-Interaction Matrix (configured for resource contention)).

Though the current design is in single-chiplet, we envision SLITS can be beneficial to multi-chiplet scheduling due to the following reasons: ❶ our proposed TIS-Cache can host a large scale of sparse TIM, generated by the huge number of threads among multi-chiplet packages, following a 256MB cost-effective L4 eDRAM cache for multi-chiplet packages [63]; and ❷ by co-designing the TIM and scheduling policy SLITS can improve the complex resource management on multi-chiplet, which are generally heterogeneous chiplets within a package in the current trend. There are also a number of works in this direction (e.g., PIE [25]).

Applicability. The outstanding difference between our design and prior arts is that: **SLITS can enable temporal variations for different scheduling objectives.** Since the customizability of SLITS only requires (1) reconfiguration of types of statistics within the Thread-Interaction Matrix; and (2) different scheduling policies based on different scheduling objectives. Therefore, it shall be feasible to change the deployments of different SLITS variants on the fly, if properly migrated.

Incorporating temporal variations in schedulers can be beneficial. There are several attempts for new scheduler designs at the Operating System level, since the emergence of multi-/many-core architectures (e.g., Barrelfish [16], fos [78], Tessellation [24], etc.). Also, there are several recent proposals on distributed scheduling for large-scale datacenter systems, which breed opportunities for customization (e.g., Syrup [42], DBOS [70], etc.). These efforts also envision that more sophisticated analysis can be applied for proper scheduler designs so that hardware resources can be well-utilized. SLITS also contributes to this line of works by ❶ identifying and addressing the sparsity issue in the Thread-Interaction Matrix via *Factorization Machine* - a novel machine learning technique; and ❷ maintaining the customizability for different scheduling objectives.

[20]SLITS is expected to have a constant chip area usage and power consumption, since the amount of threads for the scheduler queue is usually fixed. If SMT is enabled, one simple way is to double the size of TIS-Cache and we have showed that the overhead is still much lower (where TIS-Cache can be several GB, while doubling the TIS-Cache only require it to be 32KB).

8.3 Implications on Future Research and Practice

Scheduling plays a central role in resource managements. However, the roadmap to improve the effectiveness from scheduling is limited in the current scope. They are mostly likely to remain as huge obstacles for effective distributions of resources. By taking thread scheduling as a motivating example, our work demonstrates a new methodology to address the ineffectiveness of resource management in modern computing stacks; and our work is expected to breed more research opportunities in a broad context of resource scheduling (and managements).

8.3.1 Major Implications on Research

Our work breeds three major implications on research, which will likely have long-term impact on both industry and academia:

1. Sparsity in Element-Interaction Matrix: our work formalizes the scheduling problems by abstracting interactions between elements in a matrix manner. Such a formalization can be exploited for a variety of scenarios for resource managements. Our work also clears the roadmap on how to mitigate the sparsity issue of Element-Interaction Matrix, and how to exploit the results: by leveraging the (mostly)-inaccurate estimations of interactions among elements, it is expected to be feasible to distribute the resources effectively. This is because the central focus of resource managements lies on the order of preferred elements (regardless of which objectives to be satisfied), rather than the exact quantity of the metrics for these objectives.

2. A New Solution Direction to Enable Effective Resource Distributions: our work demonstrates the first customizable scheduler for resource distributions, to enhance the practicality of scheduling. Our work also reveals the feasibility to reconfigure scheduling objectives on the fly: this motivates the needs to rethink the formation of scheduling objectives, and how these objectives can be properly emerged and/or quantified. By a successful demonstration of on-the-fly reconfiguration for different scheduling objectives, it is feasible to incorporate time dimension within the design (and implementation) of schedulers. Therefore, it is essential to examine whether it is needed for active/reactive control/adjustments for newly-quantified metrics of different scheduling objectives

3. The Dynamism between Quantification and Mechanism: our work is the first to reveal the issue of *the dynamism between quantification and mechanism* by using scheduling as an example. The ill-defined formalization of schedulers breed the chicken-egg argument between the quantification of elements' interactions; and the mechanism to distribute resources for these elements. The dynamism, caused by interactions between the quantification and mechanism, is expected to be an important research question for further exploration. To better identify the issue, we leverage the quantification-mechanism argument and leave the four elementary principles: the divergence of quantified metrics; the convergence of quantified metrics; the reverberation of the mechanism under quantified metrics; and the execution after the interactions from the quantified metrics and the mechanism. Interestingly, these four principles corresponds to four principal types of neural circuits, to our current knowledge.

Acknowledgments

The original version of this work consists of (1) a published conference paper entitled “SLITS: Sparsity-Lightened Intelligent Thread Scheduling", as a part of the proceedings of ACM on Measurement and Analysis of Computing Systems, SIGMETRICS 2023; and (2) an extended abstract of the published conference paper, entitled “SLITS: Sparsity-Lightened Intelligent Thread Scheduling". This work reorganizes the above materials, and supplements additional implications.

References

[1] "Intel Xeon Gold 6150," https://en.wikichip.org/wiki/intel/xeon_gold/6150.

[2] "Intel Xeon Platinum 8180M," https://en.wikichip.org/wiki/intel/xeon_platinum/8180m.

[3] Xlearn. [Online]. Available: https://github.com/aksnzhy/xlearn

[4] I. Akturk and O. Ozturk, "Adaptive Thread Scheduling in Chip Multiprocessors," *International Journal of Parallel Programming*, 2019.

[5] Z. Al-bayati, H. Zeng, M. Di Natale, and Z. Gu, "Multitask Implementation of Synchronous Reactive Models with Earliest Deadline First Scheduling," in *SIES*, 2013.

[6] S. Aldrich, "Recommender Systems in Commercial Use," *AI Magazine*, vol. 32, pp. 28–34, 09 2011.

[7] AMD, "BIOS and Kernel Developer's Guide for AMD Family 15h processors," 2013. [Online]. Available: https://www.amd.com/system/files/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf

[8] J. H. Anderson and J. M. Calandrino, "Parallel Real-Time Task Scheduling on Multicore Platforms," in *RTSS*, 2006.

[9] J. H. Anderson and A. Srinivasan, "Pfair Scheduling: Beyond Periodic Task Systems," in *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*, 2000.

[10] J. Anderson and A. Srinivasan, "Mixed Pfair/ERfair Scheduling of Asynchronous Periodic Tasks," *Journal of Computer and System Sciences*, 2004.

[11] J. Anderson and A. Srinivasan, "Early-release Fair Scheduling," in *Euromicro RTS*, 2000.

[12] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread Scheduling for Multiprogrammed Multiprocessors," in *SPAA*, 1998.

[13] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory Hierarchy for Web Search," in *HPCA*, 2018.

[14] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate Progress: A Notion of Fairness in Resource Allocation," in *STOC*, 1993.

[15] S. Baruah, J. Gehrke, and C. Plaxton, "Fast Scheduling of Periodic Tasks on Multiple Resources," in *IPSS*, 1995.

[16] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *SOSP*, 2009.

[17] N. Beckmann and D. Sanchez, "Jigsaw: Scalable Software-Defined Caches," in *PACT*, 2013.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, 2008.

[19] J. R. Bulpin and I. A. Pratt, "Hyper-Threading Aware Process Scheduling Heuristics," in *USENIX ATC*, 2005.

[20] J. M. Calandrino and J. H. Anderson, "On the Design and Implementation of a Cache-Aware Multicore Real-Time Scheduler," in *Euromicro RTS*, 2009.

[21] T. E. Carlson, W. Heirman, and L. Eeckhout, "SNIPER: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations," in *SC*, 2011.

[22] R. Cayssials, J. Orozco, J. Santos, and R. Santos, "Rate Monotonic Scheduling of Real-time Control Systems with the Minimum Number of Priority levels," 1999.

[23] H. S. Chwa, J. Lee, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin, "Global EDF Schedulability Analysis for Parallel Tasks on Multi-Core Platforms," *IEEE TPDS*, 2017.

[24] J. A. Colmenares, S. Bird, G. Eads, S. A. Hofmeyr, A. Kim, R. Poddar, H. Alkaff, K. Asanovic, and J. Kubiatowicz, "Tessellation Operating System: Building a Real-Time, Responsive, High-Throughput Client OS for Many-core Architectures," in *IEEE HotChips*, 2011.

[25] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narváez, and J. S. Emer, "Scheduling Heterogeneous Multi-cores through Performance Impact Estimation (PIE)," in *ISCA*, 2012.

[26] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," 2013.

[27] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang, "BWS: Balanced Work Stealing for Time-Sharing Multicores," in *EuroSys*, 2012.

[28] S. Eyerman and L. Eeckhout, "Per-Thread Cycle Accounting in SMT Processors," in *ASPLOS*, 2009.

[29] S. Eyerman and L. Eeckhout, "Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling," 2010.

[30] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," in *PACT*, 2007.

[31] J. Feliu, J. Sahuquillo, S. Petit, and L. Eeckhout, "Thread Isolation to Improve Symbiotic Scheduling on SMT Multicore Processors," *IEEE Trans. Parallel Distrib. Syst.*, 2020.

[32] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "L1-bandwidth Aware Thread Allocation in Multicore SMT Processors," in *PACT*, 2013.

[33] X. Geng, G. Xu, D. Wang, and Y. Shi, "A Task Scheduling Algorithm based on Multi-core Processors," in *MEC*, 2011.

[34] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. G. Hallnor, H. Jiang, M. G. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The Fourth-Generation Intel Core Processor," *IEEE Micro*, 2014.

[35] R. Hemani, S. Banerjee, and A. Guha, "Easy and Expressive LLC Contention Model," in *HPCS*, 2016.

[36] G. J. Henry, "The UNIX system: The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal*, 1984.

[37] C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache," in *PACT*, 2014.

[38] Intel, "Intel 64 and IA-32 Architecture Software Developer Manual," 2014.

[39] P. N. Jain and S. K. Surve, "A Review on Shared Resource Contention in Multicores and its Mitigating Techniques," *IJHPSA*, 2020.

[40] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *MICRO*, 2014.

[41] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *ISCA*, 2013.

[42] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis, "Syrup: User-Defined Scheduling Across the Stack," in *SOSP*, 2021.

[43] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks, "Profiling a Warehouse-Scale Computer," *IEEE Micro*, 2016.

[44] J. Kay and P. Lauder, "A Fair Share Scheduler," *CACM*, 1988.

[45] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt, "Profile-assisted Compiler Support for Dynamic Predication in Diverge-Merge Processors," in *CGO*, 2007.

[46] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *PACT*, 2004.

[47] R. Kuchumov, A. S. Sokolov, and V. Korkhov, "Staccato: Shared-Memory Work-Stealing Task Scheduler with Cache-aware Memory Management," *IJWGS*, 2019.

[48] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi, "CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores," in *MICRO*, 2020.

[49] C. V. Li, V. Petrucci, and D. Mossé, "Predicting Thread Profiles across Core Types via Machine Learning on Heterogeneous Multiprocessors," in *SBESC*, 2016.

[50] C. V. Li, V. Petrucci, and D. Mossé, "Exploring Machine Learning for Thread Characterization on Heterogeneous Multiprocessors," *ACM OSR*, 2017.

[51] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global EDF Scheduling for Parallel Real-Time Tasks," in *Springer RTS*, 2015.

[52] C. Lin, T. Huang, and M. D. F. Wong, "An Efficient Work-Stealing Scheduler for Task Dependency Graph," in *ICPADS*, 2020.

[53] D. Liu and Y. Lee, "Pfair Scheduling of Periodic Tasks with Allocation Constraints on Multiple Processors," in *IPDPS*, 2004.

[54] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches," in *MICRO*, 2011.

[55] J. Lozi, B. Lepers, J. R. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The Linux Scheduler: A Decade of Wasted Cores," in *EuroSys*, 2016.

[56] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.

[57] C. Mattihalli, "Designing and Implementing of Earliest Deadline First Scheduling Algorithm on Standard Linux," in *CPSCom*, P. Zhu, L. Wang, F. Xia, H. Chen, I. McLoughlin, S. Tsao, M. Sato, S. Chai, and I. King, Eds., 2010.

[58] N. Mishra, J. D. Lafferty, and H. Hoffmann, "ESP: A Machine Learning Approach to Predicting Application Interference," in *ICAC*, 2017.

[59] A. Negi and P. K. Kumar, "Applying Machine Learning Techniques to Improve Linux Process Scheduling," in *TENCON*, 2005.

[60] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. S. Unsal, and A. Cristal, "A Machine Learning Approach for Performance Prediction and Scheduling on Heterogeneous CPUs," in *SBAC-PAD*, 2017.

[61] K. Pearson, "Note on Regression and Inheritance in the Case of Two Parents," *Royal Society of London*, 1895.

[62] Y. Peng, S. Wu, and H. Jin, "Robinhood: Towards Efficient Work-Stealing in Virtualized Environments," *IEEE TPDS*, 2016.

[63] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *MICRO*, 2012.

[64] P. Radojkovic, V. Cakarevic, M. Moretó, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Optimal Task Assignment in Multithreaded Processors: A Statistical Approach," in *ASPLOS*, 2012.

[65] P. Radojkovic, P. M. Carpenter, M. Moretó, V. Cakarevic, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Thread Assignment in Multicore/Multithreaded Processors: A Statistical Approach," *IEEE TC*, 2016.

[66] S. Rendle, "Factorization Machines," in *ICDM*, 2010.

[67] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core Real-Time Scheduling for Generalized Parallel Task Models," in *RTSS*, 2011.

[68] J. H. Saltzer and M. F. Kaashoek, "Chapter 6 - Performance," in *Principles of Computer System Design*, 2009.

[69] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *MICRO*, 2012.

[70] A. Skiadopoulos, Q. Li, P. Kraft, K. Kaffes, D. Hong, S. Mathew, D. Bestor, M. Cafarella, V. Gadepally, G. Graefe, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, L. Suresh, and M. Zaharia, "DBOS: A DBMS-Oriented Operating System," *PVLDB*, 2021.

[71] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," in *ASPLOS*, 2000.

[72] S. Srikanthan, S. Dwarkadas, and K. Shen, "Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems," in *ATC*, 2015.

[73] A. Srinivasan and J. H. Anderson, "Optimal Rate-based Scheduling on Multiprocessors," *JCSS*, 2006.

[74] W. J. Starke, J. Stuecheli, D. Daly, J. S. Dodson, F. Auernhammer, P. Sagmeister, G. L. Guthrie, C. F. Marino, M. S. Siegel, and B. Blaner, "The Cache and Memory Subsystems of the IBM POWER8 Processor," *IBM JRD*, 2015.

[75] D. K. Tam, R. Azimi, and M. Stumm, "Thread Clustering: Sharing-aware Scheduling on SMP-CMP-SMT Multiprocessors," in *EuroSys*, 2007.

[76] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies," in *ISCA*, 2008.

[77] S. Volos, D. Jevdjic, B. Falsafi, and B. Grot, "Fat Caches for Scale-Out Servers," *IEEE Micro*, 2017.

[78] D. Wentzlaff, C. G. III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. E. Miller, and A. Agarwal, "An Operating System for Multicore and Clouds: Mechanisms and Implementation," in *SoCC*, 2010.

[79] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*, 1995.

[80] C. Xu, x. Chen, R. Dick, and Z. Mao, "Cache Contention and Application Performance Prediction for Multi-core systems," in *ISPASS*, 2010.

[81] D. Xu, C. Wu, P.-C. Yew, J. Li, and Z. Wang, "Providing Fairness on Shared-Memory Multiprocessors via Process Scheduling," *SIGMETRICS Perform. Eval. Rev.*, 2012.

[82] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in *ISPASS*, 2014.

[83] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors," *ACM CSUR*, 2012.