Providing Fairness in Heterogeneous Multicores with a Predictive, Adaptive Scheduler

Saeid Barati
University of Chicago
saeid@cs.uchicago.edu

Henry Hoffmann
University of Chicago
hankhoffmann@cs.uchicago.edu

***Abstract*—Multicore applications contend for resources — especially memory bandwidth — reducing both quality-of-service and overall system performance. Contention-aware schedulers have been proposed to provide fairness and predictable behavior through thread-level scheduling. Prior approaches have two drawbacks, however. First, many introduce overhead that reduces overall performance. Second, the emergence of heterogeneous multicores has made handling contention and providing fairness much more difficult as the scheduler must now account for both application interference and the performance affects of different core types.**

This paper proposes augmenting existing contention-aware approaches with *predictive* and *adaptive* components to provide fair memory access and performance improvements on heterogeneous multicores. The predictive component's closed-loop approach anticipates how different processes will perform with different core types, while the adaptive component dynamically tunes key scheduling parameters to the current workload. We implement and evaluate this approach on a real Linux/x86 system with a variety of memory and compute intensive benchmarks. We find that adding prediction improves fairness and performance by 38% and 4% (respectively) compared to a prior state-of-the-art contention-aware approach. The addition of adaptation allows users to select for fairness or performance optimization, providing an additional 24% improvement in fairness or a 9% improvement in performance beyond the predictive approach.

I. Introduction

Applications contending for shared resources in multicore systems leads slowdown and unpredictable performance. This unpredictability makes it difficult, or impossible, for applications to provide quality-of-service guarantees [30]. For multi-threaded data parallel applications, contention means threads with equal work require unequal time, leading to slowdown [4].

The dominant factor in contention, and thus unfairness, is main memory access – even if threads are scheduled on disjoint processing cores, they still must share main memory bandwidth and the on-chip interconnect that connects the cores to the memory [30]. Common contention-aware schedulers are implemented in software level and include three main components: (1) runtime progress monitoring, (2) performance prediction, and (3) online scheduling decisions [31]. Prior work has shown that the most important component is the second: predicting threads' future performance given interference (especially in memory access) from co-running threads [30].

While prior techniques offer notable fairness improvement for homogeneous multicores, they suffer from performance overhead (or in some cases, negligible performance improvement) [8, 28, 30]. Most schedulers for heterogeneous multicores focus on improving overall performance only and ignore fairness [3, 25]. Van Craeynest et al proposed the first approach to ensure fairness on a heterogeneous architecture, but it requires hardware support for the key phase of performance prediction [24]. As correctly predicting future behavior is an essential part of contention-aware scheduling, many prior approaches build elaborate, prediction models that require extensive off-line model building [20, 28] . Finally, static assignment of key scheduling parameters is not only a challenging problem, but incorrect assignment may limit achievable fairness and performance improvement [8]. Taking prior work into account, *there is a need for a contention-aware scheduler that can ensure fairness on a heterogeneous system and provide performance improvement without additional hardware support or extensive offline tuning*.

A. Dike: predictive, adaptive scheduler

To address this need we present Dike, a contention-aware scheduler for heterogeneous multicores that provides significant improvement to both fairness and performance compared to prior approaches without requiring hardware support[1]. Dike divides execution time into fixed-length *quanta*. At runtime, Dike measures the memory access rate of every thread during every quanta. Dike then predicts the potential effects of migrating threads to different cores. Dike's closed-loop prediction model is efficient to compute online, yet quickly converges to accurate estimates, allowing Dike to reduce the number of migrations required to maintain fairness.

We find that the quantum length and the number of threads to migrate per quantum are key scheduling parameters affecting both performance and fairness. Furthermore, the optimal value for these parameters varies depending on both the application workload and user preference for

[1]Dike is named after the Greek goddess of justice and fair judgment

fairness or performance. Therefore, Dike adaptively tunes these two parameters as the system executes to ensure that the scheduler is tuned to workload and user desires.

B. Summary of Results

We evaluate Dike on a Linux/x86 system with 40 cores (half running at maximum frequency and the other half running at minimum to form heterogeneous environment), one memory controller, and 32 GB of main memory. We make the source code, sample benchmarks and running scripts as open source so that others can evaluate or use Dike freely [2]. We compare Dike to both Linux default scheduler and Distributed Intensity Online (DIO) [30], a state-of-the-art contention-aware scheduler. According to our empirical results, Dike achieves:

- **Fairness and performance improvement**: We evaluate Dike with combinations of compute and memory intensive benchmarks and measure fairness and performance. By geometric mean, Dike improves fairness by 67% and 38% over Linux and DIO, respectively. For performance, Dike outperforms both Linux default scheduler and DIO by 8% and 4%. In addition, we set different targets for adaptive improvement. We find that by adapting key scheduling parameters to the current workload, Dike provides additional fairness and performance improvement of 24% and 9% respectively. (See Section IV-A.)
- **Low scheduling overhead**: As thread migrations are the mechanism for ensuring fairness in both DIO and Dike, minimizing the number of migrations reduces overhead. We find Dike reduces the average number of migrations by 64% compared to DIO. Adding the adaptive features results in an additional 23% to 29% reduction in migrations. (See Section IV-B.)
- **Runtime predictability**: Unpredictable behavior of applications under contention may violate QoS guarantees. We show that Dike can accurately predict threads' memory access rates regardless of core type and without a priori knowledge. The prediction error ranges from -9% to 10% at most. (See Section IV-C.)

C. Contributions

- Design and implementation of a software level contention-aware scheduler for heterogeneous multi-cores that requires no additional hardware support.
- Introduction of a lightweight, closed-loop predictor that accurately assesses the effects of thread migration.
- A methodology for adaptively tuning scheduling parameters to the current workload and to a user's preference for fairness or performance.
- Empirical evaluation of Dike's fairness and performance compared to the Linux default scheduler and a state-of-the-art homogeneous scheduler (DIO [30]).

[2]Available at https://github.com/saeidbarati157/dike

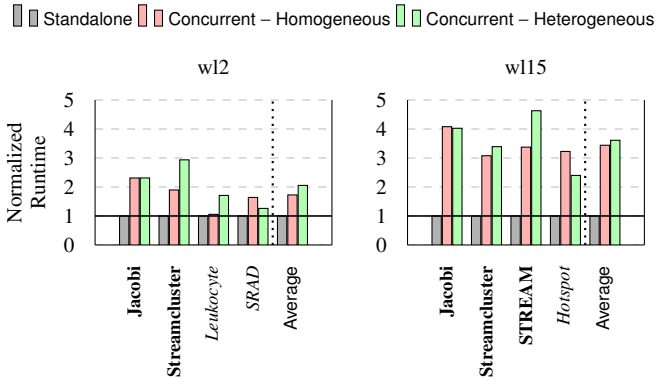


Figure 1: Performance variation of Standalone vs. Concurrent execution

The rest of paper is organized as follows. Section II presents background and motivation for contention-aware scheduling. Section III explains Dike's framework. Section IV illustrates the experimental results of our scheduler. Section V compares Dike to prior work. Finally, we conclude in Section VI.

II. Background and Motivation

This section motivates the need for Dike. We begin by briefly reviewing the effects of contention on application performance and discuss commonalities of prior contention-aware approaches. We then discuss *performance prediction*, a key component of any contention-aware approach. Finally, we demonstrate that key scheduling parameters vary with different applications and goals (fairness or performance).

Figure 1 shows the performance of various applications when run standalone (as the only application in the system) versus in a multi-application workload. The evaluation section contains a full description of application workloads (see Section IV). The figure shows that performance loss due to concurrent execution is significant, but it is not uniformly distributed. For example in workload 2 (wl2), the memory intensive Jacobi application experiences a 2.3× performance slowdown while the compute intensive SRAD application degrades by only 1.25×. The problem gets worse on a heterogeneous system. For example, the STREAM application in wl15 has a slowdown of 3.4× on the homogeneous system, but 4.6× slowdown on the heterogeneous system.

A body of work on contention-aware scheduling has arisen to address these slowdowns. These schedulers generally follow the same structure. First, a performance monitor records thread progress. Next, a predictor estimates performance degradation. Then, a decider chooses a thread-to-core mapping for improved fairness, and this mapping is finally enforced by a scheduler.

The key differences between approaches often comes down to the specific prediction mechanisms used. Many

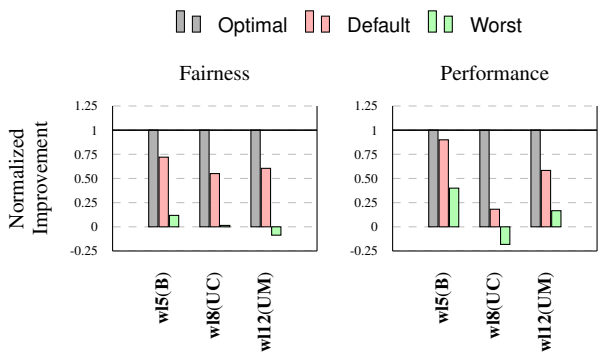


Figure 2: Comparing Fairness/Performance improvement of Optimal, Default and Worst scheduler configurations of Dike for selective workloads

prediction models require an extensive training phase before execution [8, 20, 28]. Other methods require hardware modification to collect specialized statistics [23, 25], but these mechanisms are not available on current hardware.

Prediction requires understanding how co-running applications affect each other. Prediction is already difficult on homogeneous systems (and the works referenced in the prior paragraph only deal with homogeneity), but it is even harder on heterogeneous systems where core types vary. With different core types, the predictor must anticipate both the effects applications will have on each other and the effects that different core types will have on different applications. One prior approach addresses predictive mechanisms for heterogeneous systems [24], but it requires hardware support that is not available on current systems. Thus, there is a need for a lightweight prediction mechanism for heterogeneous systems that works without offline training and does not require specialized hardware.

Almost all contention-aware schedulers work by intelligently migrating threads among cores. Therefore, two key scheduling parameters are the number of threads to migrate at once and the scheduling quantum. The values of these two variables define a scheduler configuration. Figure 2 compares the normalized fairness and performance improvement of the optimal, default and worst scheduler configuration where the optimal configuration provides highest fairness and performance between all possible configurations for three selected workloads (for more details, see Section IV). Poor scheduler configurations lead to notable fairness and performance loss. The optimal scheduler configuration, however, is a function of both the current application workload and user (or operator) preference to favor fairness or throughput. Furthermore, the optimal configuration may change as applications move through phases, new applications enter the system, or old applications exit. Thus, we propose that contention-aware schedulers should adapt these key parameters at runtime for optimal behavior.

In summary, these observations motivate the need for a contention-aware scheduler that has both lightweight prediction mechanisms and adaptive configurability.

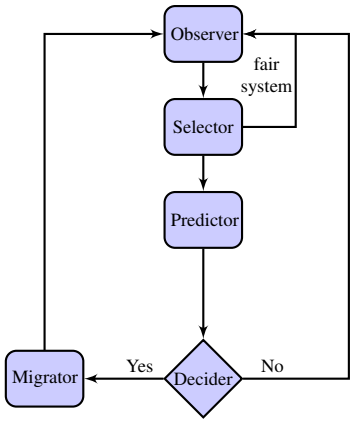


Figure 3: Overview of the Dike runtime during each quanta.

III. Dike Framework

Dike's primary goals are to (1) alleviate shared resource contention and (2) provide fairness among multi-threaded and multiprocess workloads. A secondary goal is enhancing overall performance and avoiding overhead (we could trivially provide fairness by making all threads extremely slow).

Without the fairness, unpredictable thread performance can introduce undesirable behavior which may violate QoS guarantees in some case. Therefore, fairness in an application means that threads' runtime are approximately close together. Fairness in a system means that applications are not unpredictably impeded by other applications.

To provide fairness without hardware modification, Dike is a software level scheduler that periodically re-maps threads to cores ensuring each thread gets a fair share of CPU and memory resources. Dike builds on the framework common to all existing contention-aware schedulers by augmenting it with an enhanced, closed-loop prediction model and an adaptation phase which dynamically tunes key scheduling parameters to the current workload.

Figure 3 illustrates Dike's high-level structure. At initialization, Dike has no knowledge of the current application workload. Execution time is divided into fixed length quanta. During each scheduling quantum, the **Observer** records the memory access rate per thread and categorizes threads as compute (C) or memory (M) intensive. Next, the **Selector** sorts threads based on access rate and forms pairs of threads. Dike uses coefficient of variation (standard deviation over mean) to quantify variation of memory access rate. If the system is in a fair state, where the coefficient of variation is less than a user-defined threshold (0.1 by default), no action is taken in the current quantum. Otherwise, the **Predictor** estimates possible changes in memory access rates if pairs of threads were swapped. Then, the **Decider** uses these predictions to ensure each swap benefits fairness or perfor-

mance. If a swap is beneficial, the **Migrator** actually swaps the threads, forcing each to migrate to the core currently occupied by the other.

In addition, Dike employs an **Optimizer** that adaptively updates the two key scheduling parameters *quantaLength* – indicating the time between scheduling decisions – and *swapSize* – indicating the number of threads to swap at a time. With adaptation, Dike can selectively optimize for both (1) the current application workload and (2) user preference for either fairness or throughput.

The remainder of this section provides details on each of the components illustrated in Figure 3. While many of these components are common to any contention-aware scheduler, we emphasize that Dike's unique features are in its prediction mechanism and optimizer.

A. Observer

The observer has two jobs: *thread classification* and *core identification*. Thread classification partitions threads into either memory intensive or compute intensive. Core identification partitions cores into higher and lower memory bandwidth.

The intention of *thread classification* is to identify thread demands, analyze current thread interference, and share resources more efficiently. Clearly, a memory intensive thread needs to access memory more often than a compute intensive one, therefore the scheduler will attempt to move memory intensive threads to high-bandwidth cores and compute intensive threads to low-bandwidth cores to ensure fair progress. Prior research studies the breakdown of shared resources contention [30] concluding that main memory is the dominant cause for contention and unpredictable performance behavior. Dike thus employs memory access rate to alleviate overall system contention.

The observer keeps track of memory access rate per thread by reading hardware performance counters. A high-access thread fetches most of its data from main memory, putting pressure on the memory controller and on-chip interconnect. Although memory access rate is a coarse heuristic approach, as it does not consider cache locality or reuse, it works well to approximate contention. The reason is it reflects contention of multiple shared resources such as memory controller and interconnect and therefore can function as suitable metric for scheduling decisions.

While many approaches employ instructions per clock (IPC) as a progress metric, we believe that memory access rate is a better contention metric. IPC fails to represent actual progress in heterogeneous systems where different cores could have different clock speeds. Additionally, IPC can be misleading in situations where threads are spinning to acquire a mutex lock. In these cases, despite executing instructions, threads are not making actual progress [10].

To distinguish between memory intensive and compute intensive threads, we rely on established boundaries from the literature – if a thread's LLC miss rate is more than 10%, it is considered memory intensive (M), otherwise it is compute intensive (C) [27]. Memory intensity of a thread dynamically changes as thread goes through execution phases. Accordingly, Dike reclassifies threads regularly to enforce best decisions.

Core identification distinguishes between higher and lower bandwidth cores allowing Dike to determine thread-core mappings that increase fairness and performance. Also, the observer stores the moving mean bandwidth for each core in the *CoreBW* variable and updates it every quanta. Moving mean represents average bandwidth of core throughout its execution. We later use this parameter to estimate the potential benefit of moving thread to such core. The predictor uses this variable to estimate future access rate if the thread were migrated. The core identification can change as the system evolves. A core may become low-bandwidth due to contention, or a core might become high-bandwidth if other sources of contention clear up. Thus, *CoreBW* is a dynamic property of the cores and the current system usage.

B. Selector

The selector sorts threads based on memory access rate and selectively pairs them. At first, the selector analyzes current fairness. Coefficient of variation (standard deviation over mean) is used to depict variation of memory access. Smaller value indicates higher tendency towards average, which means a fair environment.

The system is defined as *fair* if the standard deviation (σ) over mean of threads memory access rate is less than a user-defined threshold. If the system is *fair*, we skip to next quanta, otherwise we should select potential pairs of threads to migrate. An ideal mapping has high-access threads bound to high bandwidth cores and low-access threads bound to low bandwidth cores. The selector finds threads that are not running on a proper core type and performs re-mapping accordingly. We call the ideal mapping one that obeys a *placement* rule where the minimum number of threads are breaking ideal mapping; *i.e.,* the smallest possible number of threads are running on the wrong core type.

In some cases this rule is not achievable. For example, if all threads are of the same type (C/M) or the number of threads for each type is not equal to number of cores of each type (high BW/low BW), several threads break the *placement* rule. When the rule cannot be met in one quanta, Dike will naturally migrate threads so that the rule is obeyed, on average, across several quanta.

Algorithm 1 illustrates the Selector's pair forming procedure. The inputs are a *threads* array that includes all running threads and *swapSize* which is the number of threads to swap – statically assigned or decided by the optimizer in adaptive mode. *head* and *tail* are the pointers

Algorithm 1 Forming pairs of threads in the Selector

Require: $threads$ ▷ array of threads
Require: $adaptationGoal$ ▷ Fairness / Performance
Require: $swapSize$ ▷ given by optimizer

$fairness = getSystemFairness()$ ▷ calculates current fairness
if $fairness < \theta_f$ **then** ▷ System is fair
  return
end if
$n = size(threads)$
$pair = \{\langle t_l, t_h\rangle \mid t_l, t_h \in threads\}$ ▷ pair of low/high access threads
$sort(threads)$ ▷ sorts threads based on access rate
$head = 0, tail = n - 1$ ▷ pointers to beginning/end of threads array
$counter = 0$
if all threads are same type (C/M) **then**
  for $k = 0$ to $swapSize$ **do**
    $pairs[k] = \langle threads[k], threads[n-k]\rangle$
  end for
  return
end if
while $counter < swapSize$ or $head < tail$ **do**
  for $i = head$ to n **do** ▷ starting from lowest access rate
    if $threads[i]$ is violator **then** ▷ violation of *placement* rule
      $t_l = threads[i]$
      Break
    end if
  end for
  $head = i$
  for $j = tail$ downto 0 **do** ▷ starting from highest access rate
    if $threads[j]$ is violator **then** ▷ violation of *placement* rule
      $t_h = threads[j]$
      Break
    end if
  end for
  $tail = j$
  $pairs[counter++] = \langle t_l, t_h\rangle$
end while

return $pairs$ ▷ pairs of threads for swap operation

which indicate the lowest and highest access rate threads respectively. If all of threads have same type (C/M), pairs are generated from both ends regardless of the *placement* rule. Unless pointers cross each other or sufficient pairs have been chosen, the selector checks *placement* of pointing threads. If a thread violates the rule, we select it to be paired, otherwise the corresponding pointer moves to next thread in *threads* array. Each pair is a combination of thread with low-access thread (t_l) and a high-access thread (t_h) , and is represented as $< t_l, t_h >$. In the end, we transfer pairs of threads to the predictor. Sometimes the pointers cross each other which means the number of violating threads are less than $swapSize$, and thus no more threads are available to be chosen.

C. Predictor

After preparing thread pairs for swap – and before performing the migration – Dike ensures the swap will improve either fairness or performance depending on a predefined improvement target. Hence, the predictor estimates the memory access rate of each pair of threads in next quanta. We develop a closed-loop model for predicting a single thread pair's future access rates. For a thread pair of $< t_l, t_h >$, the $profit$ of swapping low-access thread (t_l) is defined as:

$$profit_{t_l} = CoreBW_{t_h} - AccessRate_{t_l} - Overhead_{t_l} \quad (1)$$

$profit_{t_l}$ is the expected memory access rate change from swapping thread t_l. We assume that if a thread migrates to a new core, it consumes the new core's entire memory bandwidth. Thereupon, backed up by empirical results, we use the new core's bandwidth – kept in parameter $CoreBW_{t_h}$ – as the thread's new access rate. $CoreBW_{t_h}$ is provided by the observer earlier and keeps the moving mean of bandwidth for thread t_h's current core. If thread t_l stays on same core, we expect it to keep the same access rate. Hence, $AccessRate_{t_l}$ is the access rate of thread t_l in the next quanta. To conclude, $CoreBW_{t_h}$ and $AccessRate_{t_l}$ are the expected access rates of threads if the swap happens or not, respectively. Further, the parameter $Overhead_{t_l}$ expresses the reduction in memory access rate due to the context switch overhead.

$$Overhead_{t_l} = \frac{swapOH}{quantaLength} * AccessRate_{t_l} \quad (2)$$

$swapOH$ is the average time that a thread spends during a swap. However, overhead depends on hardware and concurrent thread types, it can be simply obtained by common system profilers. To run Dike online and avoid a preprocessing phase, we consider $Overhead_{t_l}$ as the precision error of our model. This is an advantage of a closed loop systems, as these types of errors are inherently accounted for in the process of collecting feedback [9, 15].

Once each thread's $profit$ has been calculated individually using Eqn. 1, the total profit of a swap operation is defined as sum of profits of both threads.

$$totalProfit = profit_{t_l} + profit_{t_h} \quad (3)$$

Total profit is forwarded to the decider for further actions. In some cases, $profit_{t_l}$ (or $profit_{t_h}$) could be a negative number which represents a reduction in memory access rate. For instance, when a memory intensive thread with high-access rate migrates to a slow bandwidth core, memory access rate drops and as a result, thread performance degrades as well.

Our prediction model is simple yet efficient. Prior work shows heterogeneous multicore schedulers must be aware of core types and thread characteristics to achieve high overall throughput [24]. Dike requires no a prior knowledge, but dynamically determines both core behavior (using $CoreBW$ parameter) and thread access rates (stored in $AccessRate$ parameter) making it a suitable lightweight scheduler for heterogeneous architectures.

D. Decider

Once the prediction has been made, Dike decides whether to perform each individual swap independently. To prevent excessive overhead on a thread, Dike does not swap a thread in consecutive quanta. Further increasing the number of inactive quanta for migrated threads results in less fairness

with no substantial gain in performance. If any member of a thread pair has been swapped in last quanta, scheduler skips the pair. Also, the decider ignores pairs with negative $totalProfit$.

E. Migrator

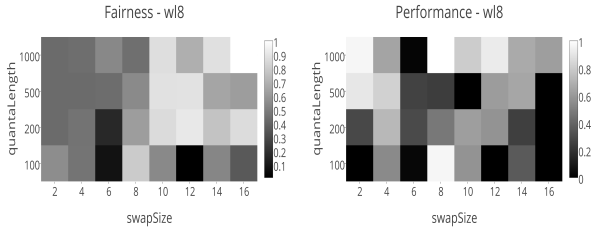
While some prior work employs thread suspension as scheduling enforcement, Dike uses thread migration instead. Although suspending threads does not produce context switch overhead, it slows down performance significantly as fast threads are idle waiting for the slowest threads to catch up. The migrator simply manipulates thread-to-core affinity mappings to swap a thread pair's cores. The order of migration during the swap procedure has not been found to make an empirical difference to either fairness or performance. In detail, Dike does not use a third core to operate swap. Therefore, at some point one core will briefly host two threads, and the other one is threadless. Our findings show no substantial change in fairness or performance by choosing the slow or fast core to host both threads for this short amount of time.

F. Optimizer

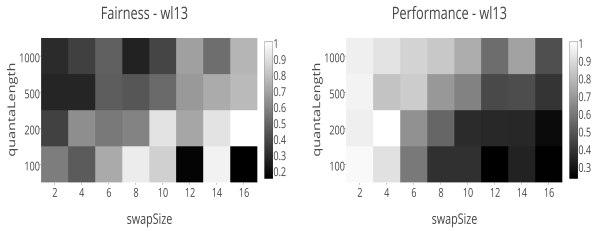
A broad range of prior work focuses on either fairness or performance and ignores the other goal. Dike, in contrast, can be tuned to favor improving fairness over performance or vice versa.

Dike has two key parameters that dramatically affect fairness and performance: *quantaLength*, the time between scheduling decisions, and $swapSize$, the number of threads to swap in one quanta. Longer *quantaLength* hurts fairness as migrations happen less frequently, but this property improves performance. With increasing $swapSize$, threads get migrated repeatedly, leading to higher fairness but more performance overhead. Empirically, we find that the best value of these key parameters can vary as a function of both workload (the applications running) and the user's goals (favoring fairness or throughput). Additionally, we expect application workload to vary as a function of time as threads will enter and leave the systems. Thus, rather than fix these parameters, Dike supports adaptively tuning them.

Based on values for each scheduling parameter, we have 32 possible configurations for every workload. Figure 4 illustrates the effects of different configurations on fairness and performance for two workloads (details about workloads can be found in Section IV). In each subplot, fairness and performance of each configuration is normalized to best configuration which has the highest fairness or performance. Every cell in the heatmap represents a configuration, where the x-axis is $swapSize$ and the y-axis is *quantaLength*, brighter represents better fairness or performance. The figure shows that (1) for a given workload, the best configuration is different for fairness or performance and (2) for a given metric the best configuration varies among workloads.



(a) Fairness - wl8 (b) Performance - wl8



(c) Fairness - wl13 (d) Performance - wl13

Figure 4: Normalized fairness/performance of every configuration for selective workloads

Therefore, there is no unique global configuration that can provide the best fairness or performance for each workload. Selecting a best configuration is a non-convex optimization, so we rely on a heuristic approach for efficiency.

In the Dike implementation, *quantaLength* is drawn from $[100, 200, 500, 1000]$ milliseconds, while $swapSize$ is any even number from 2 to half the total number of running threads. Empirical results show that by swapping more than half of threads in each quanta, overhead is substantial and it is impossible for the scheduler to achieve any performance improvement.

As optimal parameters vary per workload, we classify workloads based on the number of compute and memory intensive threads. Specifically, we group workloads into three classifications: balanced (B), where the number of memory and compute intensive threads are equal; unbalanced, compute (UC) where the compute intensive threads outnumber the memory intensive ones; and unbalanced, memory (UM), where memory intensive threads outnumber compute intensive ones. We then use a different set of heuristics to optimize for different classifications of workload.

We propose heuristic solution for each workload type rather than individual workload. Specifically, we select top configurations that provide 75% or more of best configuration with highest fairness and performance for each workload type. Figure 5 depicts contour plot of normalized fairness and performance for each workload type. In subplots, the x-axis and y-axis are $swapSize$ and *quantaLength* respectively, and higher intensity in each region represents improvement in fairness or performance.

Given this data, we derive optimization rules based on local extrema of contour plots. For example, *Fairness - UC*

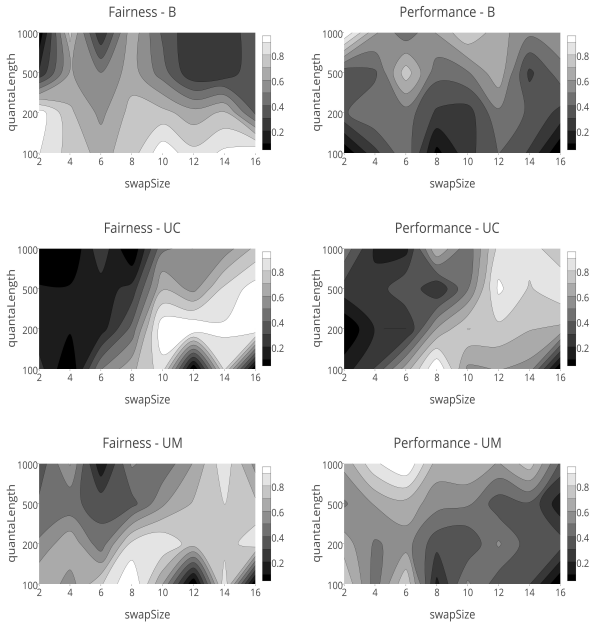


Figure 5: Optimization space of scheduler configuration for B, UC and UM workloads

subplot shows higher intensity in the center right, indicating higher fairness is achievable by increasing *swapSize* and decreasing *quantaLength* until reaches it reaches 200ms. In contrast, *Performance - UC* subplot suggests an increase in both *swapSize* and *quantaLength* to produce higher performance. Similar observations have been made for other types of workloads from the other plots shown in Figure 5.

In non-adaptive mode, Dike assigns median of ranges for each scheduling parameter as default. In our experiments, default scheduling configuration is $< 8, 500 >$. In adaptive mode, Dike uses optimization rules drawn from the contour plots. Algorithm 2 summarizes these optimization rules. Initially, optimizer starts from default configuration and updates scheduling parameters periodically according to type of workload. In every step, the optimizer is allowed to change scheduling parameter for one unit. For instance, updating *quantaLength* from 100 to 1000 milliseconds requires calling optimizer for 3 times.

IV. Experimental Evaluation

We evaluate Dike on a real system with one memory controller and two sockets, each with an Intel Xeon-E5 CPU. System configuration details are available in Table I. We set one socket to the minimum CPU frequency, and on the other we enable TurboBoost to build a large-scale heterogeneous structure. Both the fast and slow cores have hyperthreading enabled, meaning contention can occur from threads sharing a single virtual core, from threads on different cores on the same chip, or from threads scheduled on the two separate sockets. This platform is thus a real system that has the type

Algorithm 2 Optimizing configurations of workloads

Require: θ_f ▷ fairness threshold by user
Require: *adaptationGoal* ▷ Fairness / Performance
Require: $< swapSize, quantaLength >$ ▷ current configuration

$fairness = getSystemFairness()$ ▷ calculates current fairness
if $fairness < \theta_f$ **then** ▷ System is fair
  return
end if
$workloadType = getWorkloadType()$ ▷ Identify workload type
if *adaptationGoal* is Fairness **then**
  switch *workloadType* **do**
    case B :
      decrease *quantaLength*
      $quantaLength = Math.Max(quantaLength, 100)$
    case UC :
      $swapSize = Math.Min(swapSize + 2, 16)$
      decrease *quantaLength*
      $quantaLength = Math.Max(quantaLength, 200)$
    case UM :
      $swapSize = Math.Min(swapSize + 2, 16)$
      decrease *quantaLength*
      $quantaLength = Math.Max(quantaLength, 500)$
end if
if *adaptationGoal* is Performance **then**
  switch *workloadType* **do**
    case B :
      increase *quantaLength*
      $quantaLength = Math.Min(quantaLength, 1000)$
    case UC :
      $swapSize = Math.Min(swapSize + 2, 16)$
      increase *quantaLength*
      $quantaLength = Math.Min(quantaLength, 1000)$
    case UM :
      increase *quantaLength*
      $quantaLength = Math.Min(quantaLength, 1000)$
end if

return $< swapSize, quantaLength >$ ▷ new configuration

of large scale heterogeneity anticipated for future high-end computing systems [1].

The experimental workloads are from the Rodinia OpenMP benchmark suite [5] shown in Table II. Each workload is four benchmarks with differing numbers of compute and memory intensive benchmarks with 8 threads for 32 total threads (4 benchmarks by 8 threads). In our experiments, each core (including the hyperthreaded virtual cores) always runs one thread. Workloads are classified into three types, Balanced (B), Unbalanced Compute Intensive (UC) and Unbalanced Memory Intensive (UM) based on number of compute and memory intensive threads. While we know this mix from our own observation, the schedulers are not given any a priori knowledge. Additionally, each workload includes the KMEANS benchmark with 8 threads which further increases contention as KMEANS produces excessive inter-thread communication.

The experimental evaluation is divided into four parts. First we compare Dike's fairness and performance to prior work and Linux default scheduler. Next, we show how the number of thread migrations changes for each scheduling method. Then, we demonstrate Dike's prediction accuracy.

Table I: System Configuration

Component	Details
Hardware	Intel (r) Xeon-E5 , 10 Cores (2.33 GHz), 10 Cores (1.21 GHz) , 25 MB shared LLC , 32 GB RAM
Operating System	Ubuntu 14.04

Table II: Workloads used for experiments. Memory Intensive benchmarks are displayed bold

Workload	**B**: Balanced (2 M / 2 C)			
WL1	**jacobi**	**needle**	leukocyte	lavaMD
WL2	**jacobi**	**streamcluster**	leukocyte	srad
WL3	**streamcluster**	**needle**	hotspot	lavaMD
WL4	**jacobi**	**streamcluster**	lavaMD	heartwall
WL5	**streamcluster**	**needle**	leukocyte	hotspot
WL6	**jacobi**	**needle**	heartwall	srad
	UC: Unbalanced-Compute Intensive (1 M / 3 C)			
WL7	**jacobi**	lavaMD	leukocyte	srad
WL8	**needle**	hotspot	leukocyte	heartwall
WL9	**streamcluster**	heartwall	leukocyte	srad
WL10	**jacobi**	hotspot	leukocyte	heartwall
WL11	**needle**	lavaMD	hotspot	srad
	UM: Unbalanced-Memory Intensive (3 M / 1 C)			
WL12	**jacobi**	**needle**	**streamcluster**	lavaMD
WL13	**jacobi**	**needle**	**stream_omp**	leukocyte
WL14	**streamcluster**	**needle**	**stream_omp**	lavaMD
WL15	**jacobi**	**streamcluster**	**stream_omp**	hotspot
WL16	**jacobi**	**needle**	**streamcluster**	srad

A. Fairness and Performance

Recent work on contention-aware scheduling is built on the Distributed Intensity Online (DIO) approach [30]. In this section, we compare DIO to different Dike policies in terms of fairness and performance. We use Linux's default scheduler - completely fair scheduler (CFS) - as a baseline. CFS tries to equalize allocated CPU time. In DIO, the scheduler measures last level cache miss rates of at runtime, sorts them from highest to lowest, and then pairs threads by choosing one from top of the list (highest miss rate) and one from bottom of the list (lowest miss rate) and swaps them.

We examine three different instantiations of Dike: a non-adaptive version with a fixed *swapSize* and *quantaLength* of 8 and 500 (called Dike), an adaptive version favoring fairness (Dike-AF), and an adaptive version favoring performance (Dike-AP). The default Dike shows how prediction improves fairness and performance compared to a prior state-of-the-art approach. The adaptive versions show the additional improvement from dynamically tuning scheduling parameters.

We introduce a *Fairness* metric. In a fair environment, homogeneous threads are expected to finish execution close together with minimum difference from average execution time. Thus, we employ the coefficient of variation (which is standard deviation over mean) for measuring dispersion of threads' runtime. For each benchmark, Dike computes coefficient of variation of threads execution time and takes an average for all benchmarks. For a workload with n benchmarks:

$$Fairness = 1 - \frac{\sum_{i=1}^{n} cv_i}{n} \quad (4)$$

where cv_i is the coefficient of variation of homogeneous threads execution time in the benchmark i. In an ideal fair system, homogeneous threads have same execution time; *i.e.,* cv_i would be zero for each benchmark and thus the maximum *Fairness* is 1. Higher *Fairness* implies more predictability as threads' execution times are approximately equal.

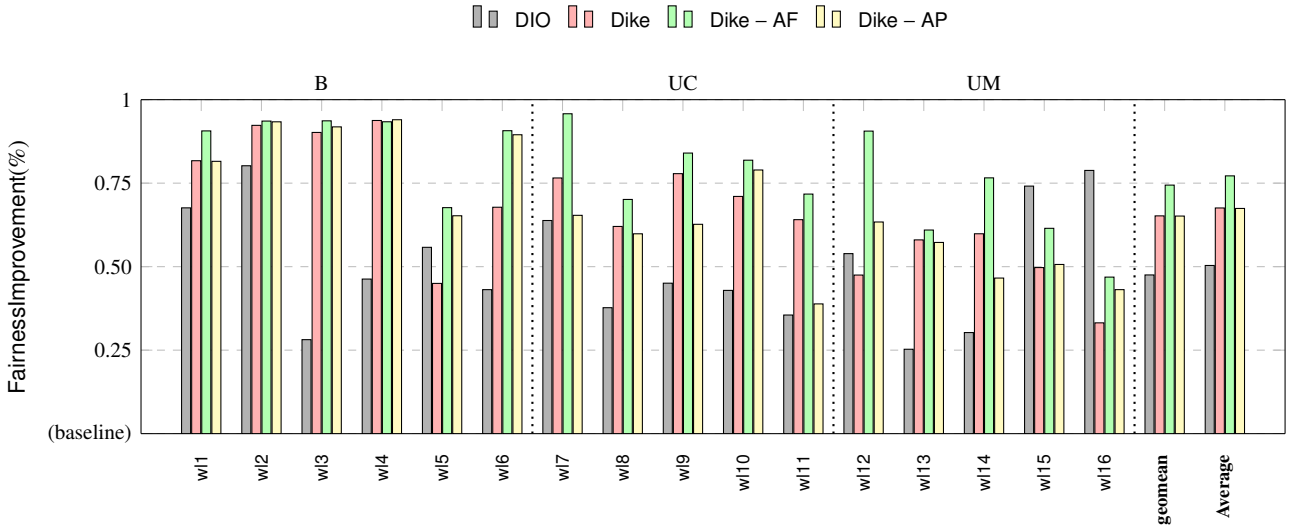
Prior works have proposed the ratio of maximum slowdown over minimum slowdown as a fairness metric[8, 13]. This ratio fails to address fairness completely as it only considers best and worst cases. Choosing coefficient of variation allows Dike to examine all threads behavior.

Figure 6a demonstrates how fairness changes by employing different scheduling methods for each workload individually and averaged on 16 workloads over the baseline (Linux Default Scheduler - CFS). As threads require different shares of resources, time-sharing resources hurts the performance of those who require more. For instance, assigning the same memory bandwidth to both compute and memory intensive threads slowdowns performance of memory intensive threads, leading to unfairness. Figure 6a shows the improvement in fairness over the baseline, so the baseline is zero. The chart is divided up into four regions. Each workload class is grouped together and the final region shows both the average and geometric mean improvement.

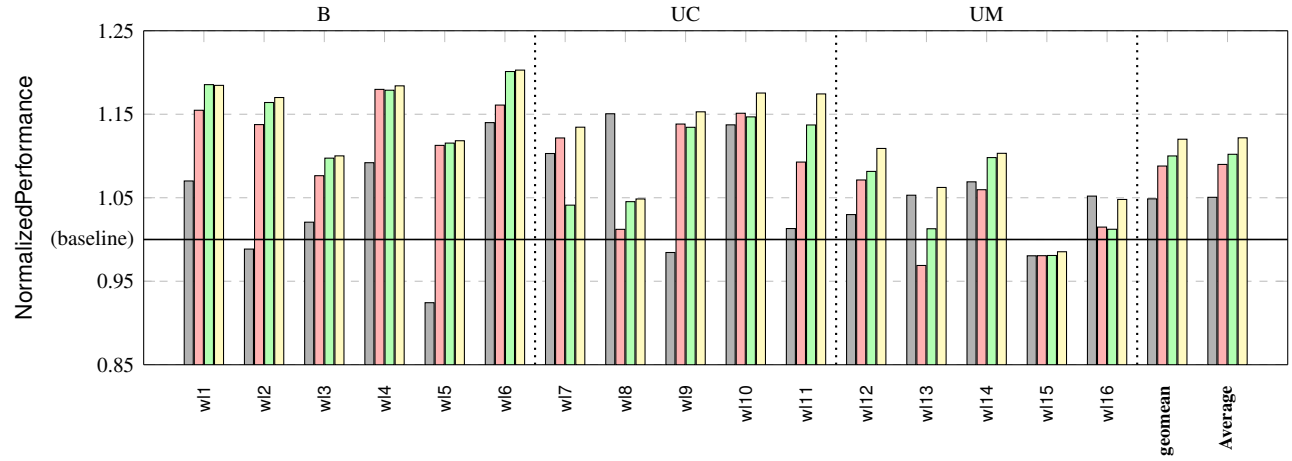
By geometric mean, Dike improves fairness by 65% compared to the baseline and by 38% compared to DIO, which is, itself, 47% above the baseline. The difference between Dike and DIO is the closed-loop prediction mechanism that allows Dike to reach the ideal thread-core mapping with minimal migration overhead. In contrast, DIO swaps all threads in every quanta ignoring the overhead of thread migrations.

Dike-AF increases the fairness gains by a further 14% compared to Dike bringing the total improvement over the baseline to 75%. This adaptation is especially beneficial for the majority of the unbalanced workloads. Not surprisingly, Dike-AP does not improve fairness compared to Dike as it optimizes for performance; however, it is important to note that this approach does not hurt fairness.

Figure 6b shows each workload's speedup over baseline for four the scheduling methods. The vertical bold line on the y-axis represents the baseline performance of the workload under CFS (which is 1). Since Dike is a fairness oriented scheduler, it forces homogeneous threads of a benchmark to have similar execution time. Therefore benchmark runtime is not delayed by the slowest thread and consequently performance improves in return. Comparing the average speedup over baseline, Dike surpasses DIO by 4% in performance



(a) Fairness improvement of DIO, Dike, Dike-AF and Dike-AP over Linux default scheduler



(b) Relative Performance of DIO, Dike, Dike-AF and Dike-AP to Linux Default Scheduler

Figure 6: % Fairness and Performance improvement

improvement. While Dike-AF focuses on fairness rather than performance, it still improves performance more over the non-adaptive mode. Dike-AP provides the best performance, which brings 12% geometric mean speedup compared to baseline.

We consider both the fairness and performance results together. Dike improves upon DIO in both fairness and performance by 38% and 4% respectively. Adaptation secures either better fairness or performance while giving the option to emphasize one or the other. In some workloads (such as wl5, wl12, wl15 and wl16), DIO has a better fairness, but Dike compensates by providing particularly better performance. The same pattern applies when DIO has slightly better performance (*e.g.,* wl13, wl14 and wl16), and Dike delivers higher fairness in return. Dike-AF and Dike-AP outdo DIO in terms of fairness and performance respectively for almost every workload individually and on average as well. Finally, analyzing non-adaptive mode versus Dike-AF and Dike-AP shows that if we prioritize a target, the adaptation provides better results than the non-adaptive mode. In other words, for every workload in Figure 6a, Dike-AF provides better fairness than the non-adaptive mode. Likewise, Dike-AP outperforms the non-adaptive mode for each workload according to Figure 6b. The reason is that in every step of the adaptation, the optimizer ensures changing scheduling parameters does not harm the desired behavior.

The performance of wl15 is a special case where the workload is extremely sensitive to do thread migration. Essentially any migration is going to hurt performance for this workload. Also due to numerous drastic changes in memory access rate of threads, Dike's predictor fails to estimate threads behavior correctly. In this situation, both DIO and

Dike come short of providing performance improvement comparing to Linux's default scheduler.

B. Number of swap operations

Above we argue that the performance improvement of Dike compared to DIO is largely due to its prediction mechanism which prevents needless migrations. In fact, a major objective of Dike is to achieve fairness in the minimum number of thread migrations; *i.e.,* Dike trades prediction overhead to reduce swap overhead. In this section we provide further evidence for this argument by evaluating the number of migrations that occur under various scheduling policies.

Table III shows the swap (a pair of migrations) counts for each workload and scheduling policy. Dike has third of the swaps on average compared to DIO. Many benchmarks have a memory intensive phase in the beginning to fetch data and instructions. The memory access rate may drop after a short period or continue at an even higher rate. Hence, it is necessary to maintain fairness and prevent overuse of shared resources in early stages by swapping more frequently. After time, some threads may finish or change phases, and the swap rate could decrease. At this moment, adaptation could be useful by updating scheduling parameters to the best combination depending on workload type in order to reduce number of swaps further. Dike-AF makes certain fairness boosts after each update while on the other side, Dike-AP tries to enhance performance even more by reducing number of swaps aggressively while ensuring fairness does not diminish significantly. Comparing Dike-AF and Dike-AP to the non-adaptive mode, the average number of swaps is cut down by 69% and 89% respectively.

C. Prediction Error

An accurate memory access rate prediction allows a scheduler to move threads across the cores without harming fairness or incurring unnecessary thread movement. Figure 7 displays maximum, average and minimum of prediction error across all threads for each workload. Zero error implies perfect prediction while negative and positive error represent underestimation and overestimation respectively. Prediction error is the average difference between predicted and actual memory access of the running threads. Dike's average prediction error ranges from 0 to 3%, while lower and upper bounds are -9% and +10%. UM workloads are simpler to estimate as threads are accessing memory in steady rate. In contrast, predicting UC workloads is more difficult since memory access patterns in compute intensive threads fluctuates vastly. These threads experience short periods of intensive memory access and then long periods with few memory accesses. Unanticipated rise and fall in access rate results in incorrect estimation.

To inspect prediction error in detail, we select workloads with higher prediction errors and examine how error changes

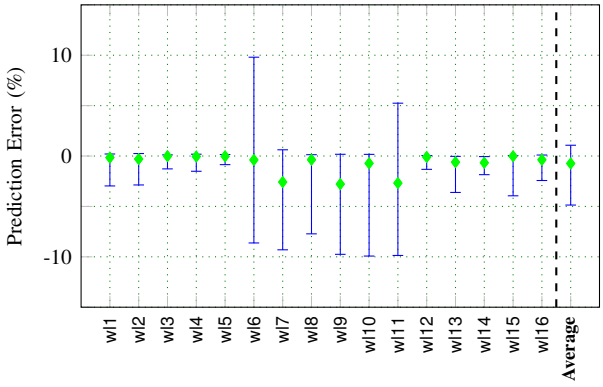


Figure 7: Prediction Error of Dike

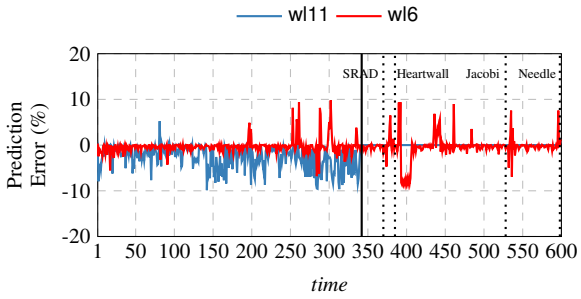


Figure 8: Prediction Error of selective workloads

at runtime. Figure 8 illustrates trend of prediction error for wl6 and wl11. The bold horizontal line depicts the end of wl11 execution. Dotted lines illustrate the finish time of benchmarks in wl6 (SRAD, Heartwall, Jacobi and Needle). Phase changes with notable change in memory access rate (more probable in compute intensive threads) cause spikes in prediction error as Dike estimates memory access using the moving mean. For instance, spikes at time 248 (or 305) are result of sudden change in memory access rate of threads. Also completion of threads affects the execution environment by freeing memory bandwidth and therefore prediction error fluctuates afterward (the spikes after dotted lines). Despite complications, Dike keeps prediction within 10% of the actual value.

V. Related Work

This section discusses prior work that addresses contention (1) in the last level cache (LLC), (2) in the memory controller, (3) through OS scheduling policies, (4) in heterogeneous multicores, and (5) through adaptive policies.

Shared LLC contention can degrade performance notably, and many solutions have been proposed for cache contention exclusively. Tam et al suggest page coloring where cache portion is determined by a thread's access rate [22]. Qureshi et al partition cache space to minimize LLC misses [19]. Wang et al manage process's cache requirement by re-

Table III: Comparing swap count in DIO, Dike, Dike-AF and Dike-AP

Workload Type	**B**						**UC**					**UM**					
	wl1	wl2	wl3	wl4	wl5	wl6	wl7	wl8	wl9	wl10	wl11	wl12	wl13	wl14	wl15	wl16	**Average**
DIO	1980	2120	1952	1964	2364	2068	1624	1720	2220	2568	2144	1872	1776	1812	3564	2120	2116.75
Dike	9	14	12	9	12	270	2288	1594	2197	2500	981	317	1015	806	312	37	773.3125
Dike-AF	10	9	19	9	12	16	207	398	216	634	206	404	450	1081	260	696	289.1875
Dike-AP	8	8	8	8	14	14	293	333	254	513	323	168	310	405	95	304	191.125

adjusting scheduling order [26]. Fedorova et al introduce cache-aware scheduling that grants more CPU time to threads which are hurt more by contention [7]. All these approaches focus on LLC only and do not address other sources of contention – like main memory and on-chip interconnect bandwidth.

Another crucial shared resource is the memory controller, and several new memory controller designs have been proposed. Nesbit et al describe Fair Queueing Memory (FQM) where threads with earlier virtual time have higher priority [16]. Inspired by FQM, Kim et al introduce Atlas that favors threads with the least attained service [13]. Ebrahimi et al estimate unfairness with a feedback loop and slow down the cores executing especially demanding threads [6]. These approaches require hardware modification. In addition, they limit the worst case behavior (*i.e.,* the thread that receives the worst service) but they do not ensure equal progress among all threads.

Contention-aware schedulers are a promising solution for current hardware as they only employ OS thread/process scheduling. Zhuralev et al provide fairness by relying on LLC miss rate heuristics and dynamically balancing threads' progress [30]. While LLC miss rate effectively distinguishes compute and memory intensive threads, it does not always indicate contention[23]. Xu et al allocates more CPU time to threads that suffer more slowdown by estimating the standalone IPC of each thread[28]. Feliu et al estimates performance by periodically creating low-contention co-schedulers [8]. These approaches rely on either a complex prediction model or offline training to achieve acceptable fairness/performance improvement.

Heterogeneity is increasingly prevalent in architectures, where it may arise due to design time decisions [12], runtime configurability [29], or as a side effect when a logically homogeneous processor behaves like a heterogeneous processor due to the physical layout of memory [11]. Most heterogeneous multicore schedulers utilize the different core types to reach the highest overall throughput [24]. Suleman et al use fast cores to accelerate critical sections of code [21], while Annavaram et al prioritize serial code segments [2]. Lakshminarayana et al assign threads with larger remaining execution times to faster cores [14]. While these approaches successfully maximize overall throughput, they fail to guarantee fairness. This becomes an issue for a barrier-synchronized multi-threaded workload and for multi-application workloads where some applications require quality-of-service guarantees. Van Craeynest et al address this need by ensuring equal work is done on each core type [24]. While this was the first approach to provide fairness in heterogeneous multicore scheduling it requires hardware support for its prediction model that does not currently exist on real machines. Adaptive solutions can further reduce contention. FACT, the framework for adaptive task migration, minimizes inter-workload interference [18]. Pricopi intelligently reconfigures and allocates cores to applications to form a heterogeneous architecture and minimize makespan [17]. Both these approaches adapt at the hardware level.

In summary, Dike is a contention-aware scheduler for heterogeneous architecture that considers thread classification and core type for scheduling decisions. Dike has a lightweight predictive model requiring no offline training and an adaptive optimization that improves fairness and performance simultaneously. Dike can be deployed on current hardware.

VI. Conclusion

Prior contention-aware schedulers employ complex prediction models often requiring extensive training phases. Additionally, they favor either fairness or performance, but cannot handle both goals simultaneously. To meet the need for a simple, but effective contention-aware scheduler that can address with user performance, this paper presents Dike. We evaluate Dike with various combinations of compute and memory intensive benchmarks on a real machine and achieve 65% and 8% improvement in fairness and performance respectively compared to Linux default CFS scheduler. Using adaptation, fairness and performance improvements raise to 78% and 12%. We release Dike's source code and configuration scripts as open source to further development and reproducible results.

Acknowledgments: We are grateful to the anonymous reviewers whose suggestions improved the paper. The effort on this project is funded by the U.S. Government under the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award.

References

[1] J. Ang et al. “Abstract Machine Models and Proxy Architectures for Exascale Computing”. In: *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*. 2014, pp. 25–32.

[2] M. Annavaram et al. “Mitigating Amdahl’s law through EPI throttling”. In: *ISCA*. 2005.

[3] C. Augonnet et al. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *Concurr. Comput. : Pract. Exper.* 23.2 (2011).

[4] A. Bhattacharjee and M. Martonosi. “Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors”. In: *ISCA*. 2009.

[5] S. Che et al. “Rodinia: A benchmark suite for heterogeneous computing”. In: *IISWC*. 2009.

[6] E. Ebrahimi et al. “Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems”. In: *ASPLOS*. 2010.

[7] A. Fedorova et al. “Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler”. In: *PACT*. 2007.

[8] J. Feliu et al. “Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler”. In: *IPDPS*. 2015.

[9] A. Filieri et al. “Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees”. In: *ICSE*. 2014.

[10] H. Hoffmann et al. “A generalized software framework for accurate and efficient management of performance goals”. In: *EMSOFT*. 2013.

[11] H. Hoffmann et al. “Remote Store Programming”. In: *High Performance Embedded Architectures and Compilers* (2010), pp. 3–17.

[12] B. Jeff. “Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration”. In: *DAC*. 2012.

[13] Y. Kim et al. “ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers”. In: *HPCA*. 2010.

[14] N. Lakshminarayana et al. “Age based scheduling for asymmetric multiprocessors”. In: *SC*. 2009.

[15] M. Maggio et al. “Power Optimization in Embedded Systems via Feedback Control of Resource Allocation”. In: *IEEE TCST* 21.1 (2013).

[16] K. J. Nesbit et al. “Virtual Private Caches”. In: *ISCA*. 2007.

[17] M. Pricopi and T. Mitra. “Task Scheduling on Adaptive Multi-Core”. In: *Computers, IEEE Transactions on* 63.10 (2014).

[18] K. K. Pusukuri et al. “FACT: A Framework for Adaptive Contention-aware Thread Migrations”. In: *CF*. 2011.

[19] M. K. Qureshi and Y. N. Patt. “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches”. In: *MICRO*. 2006.

[20] D. Shelepov et al. “HASS: A Scheduler for Heterogeneous Multicore Systems”. In: *SIGOPS Oper. Syst. Rev.* 43.2 (2009).

[21] M. A. Suleman et al. “Accelerating Critical Section Execution with Asymmetric Multi-core Architectures”. In: *ASPLOS*. 2009.

[22] D. K. Tam et al. “RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations”. In: *SIGPLAN Not.* 44.3 (2009).

[23] L. Tang et al. “Contentiousness vs. Sensitivity: Improving Contention Aware Runtime Systems on Multicore Architectures”. In: *EXADAPT*. 2011.

[24] K. Van Craeynest et al. “Starchart: hardware and software optimization using recursive partitioning regression trees”. In: *PACT*. 2013.

[25] K. Van Craeynest et al. “Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE)”. In: *ISCA*. 2012.

[26] Y. Wang et al. “Reducing Shared Cache Contention by Scheduling Order Adjustment on Commodity Multi-cores”. In: *IPDPSW*. 2011.

[27] Y. Xie and G. Loh. “Dynamic classification of program memory behaviors in CMPs”. In: *CMMSI*. 2008.

[28] D. Xu et al. “On Mitigating Memory Bandwidth Contention Through Bandwidth-aware Scheduling”. In: *PACT*. 2010.

[29] Y. Zhou and D. Wentzlaff. “The Sharing Architecture: Sub-core Configurability for IaaS Clouds”. In: *ASPLOS*. 2014.

[30] S. Zhuravlev et al. “Addressing Shared Resource Contention in Multicore Processors via Scheduling”. In: *ASPLOS*. 2010.

[31] S. Zhuravlev et al. “Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors”. In: *ACM Comput. Surv.* 45.1 (2012), 4:1–4:28.