PROVIDING FAIRNESS IN HETEROGENEOUS MULTICORES WITH A PREDICTIVE, ADAPTIVE SCHEDULER

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER’S

DEPARTMENT OF COMPUTER SCIENCE

BY
SAEID BARATI

CHICAGO, ILLINOIS
2015
# TABLE OF CONTENTS

LIST OF FIGURES ......................................................... iv

LIST OF TABLES ............................................................ v

ACKNOWLEDGMENTS ....................................................... vi

ABSTRACT ................................................................. vii

1 INTRODUCTION .......................................................... 1
  1.1 Summary of Results ................................................... 3
  1.2 Contributions ........................................................ 4

2 MOTIVATION .............................................................. 5

3 SIMCEPTION: FAST FLEXIBLE MULTICORE SIMULATOR ................. 10

4 DIKE: PREDICTIVE, ADAPTIVE SCHEDULER .......................... 14
  4.1 Observer ............................................................... 15
  4.2 Selector ............................................................... 17
  4.3 Predictor .............................................................. 18
  4.4 Decider ............................................................... 19
  4.5 Migrator ............................................................... 19
  4.6 Optimizer ............................................................. 20

5 EVALUATION .............................................................. 26
  5.1 Fairness and Performance .......................................... 26
  5.2 Number of swap operations ...................................... 29
  5.3 Prediction Error .................................................... 30

6 RELATED ................................................................. 33

7 CONCLUSION AND FUTURE WORK ................................. 35
  7.1 Conclusion ........................................................... 35
  7.2 Future Work ........................................................ 35
LIST OF FIGURES

2.1 Data Parallel Multi-threaded applications ............................................. 6
2.2 Performance variation of Standalone vs. Concurrent execution ................ 6
2.3 Tile64Pro Topology .................................................................................... 7
2.4 Performance variation of STREAM on Tile64Pro manycore system ............ 7
2.5 Memory Access Rate of w116 .................................................................... 8
2.6 Improvement of Optimal, Default and Worst scheduler configuration over Fairness/Performance .......................................................... 8
3.1 Highlevel architecture of Simception ........................................................ 11
3.2 Statistical comparison of Tile64Pro and Simception .................................... 12
4.1 Overview of the Dike runtime. .................................................................. 15
4.2 Normalized fairness/performance of every configuration for selective workloads .......................................................................................... 21
4.3 Optimization space of scheduler configuration for B, UC and UM workloads .......................................................... 22
5.1 Fairness and Performance improvement .................................................... 31
5.2 Prediction Error of Dike ............................................................................ 32
5.3 Prediction Error of selective workloads ................................................... 32
LIST OF TABLES

5.1 System Configuration ................................................................. 26
5.2 Workloads used for experiments. Memory Intensive benchmarks are displayed bold . 27
5.3 Comparing swap count in DIO, Dike, Dike-AF and Dike-AP ............................. 29
ABSTRACT

On multicore processors applications contend for resources – especially memory bandwidth – re-
ducing both fairness and overall system performance. To address this problem, contention-aware
schedulers have been proposed to provide fairness and predictable behavior through resource man-
agement. These approaches have two limitations: 1) Many introduce overhead that reduces per-
formance. 2) The emergence of heterogeneous multicores has made the handling contention and
providing fairness much more difficult.

This thesis made two major contributions. First we present Simception, a fast and flexible mul-
ticore simulator that can be used to design and experiment different contention-aware scheduling
policies. We verify Simception with performance output of TilePro64 manycore system. Further-
more, we propose augmenting existing contention-aware approaches with lightweight predictive
and adaptive components named as Dike to provide fair memory access and performance improve-
ments on heterogeneous multicores. The predictive component’s closed-loop approach anticipates
how different processes will perform with different core types, while the adaptive component dy-
namically tunes key scheduling parameters to the current workload.

We implement and evaluate this approach on a real Linux/x86 system with a variety of mem-
ory and compute intensive benchmarks. We find that adding prediction improves fairness and
performance by 30% 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 optimiza-
tion, providing an additional 24% improvement in fairness or a 9% improvement in performance
beyond the predictive approach.
CHAPTER 1
INTRODUCTION

Competing for shared resources in multicore systems leads into contention and consequently slowdown and unpredictable performance. This unpredictability makes performance tuning very difficult as performance varies at runtime as a function of what other applications are alive in the system. Multi-threaded data parallel applications are a perfect example of the need for fair performance to ensure equivalent runtime of parallel threads [3]. The most common shared resources are last level caches (LLC), the memory bus or interconnects, main memory controllers and prefetchers. These shared resources are thread-unaware, they treat requests regardless of type of thread as they are managed at hardware level. 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 [25].

Contention-aware scheduling seems as promising solution to shared resources contention. Common contention-aware schedulers are implemented in software and include runtime progress monitoring, performance prediction and online scheduling decisions [26]. The main challenge is predicting threads’ future performance given interference (especially in memory access) from co-running threads. Due to reasons such as limited available debugging information or difficult trace of full system behavior on a real machine, implementing prediction models is significantly irritating. Also experimenting various prediction models on cycle accurate simulators is a time consuming procedure. Motivated by development difficulties ahead, we present Simception, a simple simulator that employs the common structure of a contention-aware scheduler and provides high simulation speed for a faster development.

While prior techniques offer notable fairness improvement for homogeneous multicores, they suffer from performance overhead (or in some cases, negligible performance improvement) [8, 24, 25]. Most previous methods in scheduling heterogeneous multicores focused on improving overall
performance only and ignored fairness [2, 21]. Van Craeynest et al proposed the first approach to ensure fairness on a heterogeneous architecture, but it requires hardware support to correctly estimate the current fairness and predict the fairness impacts of future actions [20]. Indeed, correctly predicting future behavior is an essential part of contention-aware scheduling. Many approaches build elaborate, accurate prediction models, but they require extensive off-line model building [16, 24]. Finally, static assignment of key scheduling parameters limits achievable fairness and performance improvement [8]. Taking this prior work into account, *there is a need for a contention-aware scheduler that can ensure fairness on a heterogeneous system without additional hardware support or extensive offline tuning.*

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. Dike divides execution time into fixed-length chunks named as *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 determine the minimum 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 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.

---

1. Dike is named after the Greek goddess of justice and fair judgment
1.1 Summary of Results

We implement Simception as a Linux software which can be customized to provide various multi-core systems in terms of memory controller and cores. We verify Simception’s performance results with Tile64Pro manycore system. The most recent version of Simception is available online for download. 2 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. 3 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 30 % over Linux’s default scheduler and Distributed Intensity Online (DIO) [25] respectively. Similarly for performance, Dike outperforms both Linux and DIO by 8 % and 4 % as well. In addition, we study setting different targets for adaptive improvement. Subsequently, Dike provides additional fairness and performance improvement of 24% and 9% respectively. (See Section 5.1.)

- **Low scheduling overhead**: As thread migrations are the mechanism for ensuring fairness, 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 and additional 23% and 29% reduction in migrations. (See Section 5.2.)

- **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 5.3.)

2. Available at https://github.com/saeidbarati157/simception
3. Available at https://github.com/saeidbarati157/dike
1.2 Contributions

- Implementing simple and fast simulator that allows rapid simulation of multicore system in order to experiment new contention-aware scheduling policies.

- Design and implementation of a software level contention-aware scheduler for heterogeneous threads 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 [25]).

The rest of thesis is organized as follows. 2 presents background and motivation for contention-aware scheduling. Chapter 3 describes details of Simception. Chapter 4 explains Dike. Chapter 5 illustrates the experimental results of our scheduler. Chapter 6 compares Dike to prior work. Finally, we conclude in Chapter 7.
CHAPTER 2
MOTIVATION

This section motivates the need for Simception and Dike. We begin by stating importance of fairness in data parallel applications. Next, we highlight sources of contention and its effects on fairness and performance in homogeneous and heterogeneous systems. Afterwards, we depict contention results of multi-threaded benchmark on Tile64Pro manycore processor. 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 2.1 illustrates iterative execution of a barrier-synchronized multi-threaded workload. In each iteration, threads start at the same time and those who finish first, should wait stalling until last thread reaches barrier. Because of contention, imbalanced runtime of threads results into excessive execution time and subsequently slowdown system. These idle cores hurts performance even more in heterogeneous systems because of different performance of big vs little cores. Ensuring equivalent progress of threads (or in other words, making sure threads hurts from contention equally) decreases stall time and thus provides performance boost.

Figure 2.2 shows the performance of various applications when run standalone (as the only application in the system) versus in a multi-application workload. 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 only by 1.25 ×. Also an application in different workload combination can experience different slowdowns which makes it harder for progress prediction. For instance, in workload 2 (wl2), Jacobi slowdowns by 2.3× comparing to workload 15 (wl15) with 4.1× performance degradation. These unpredictable runtimes complicates delivering QoS guarantees.

The TILEPro64 is a 64-core tiled architecture processor whose tiles are connected by a 8×8
Then we run a STREAM benchmark (memory intensive) with 64 threads. STREAM is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s). Figure 2.4 presents performance variation of threads. Contention of shared resources force standard deviation over mean of threads performance to be 15% which implies unfairness between identical threads. In an ideally fair system, standard deviation over mean of threads is zero as same performance is expected. On the other hand, higher standard deviation denotes less fairness of system. This shows the need for scheduler that can decrease contention effects and increase fairness in return.

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
To ensure equal progress of threads, scheduler should predict progress in presence of contention. Main memory is considered as dominant cause of contention. Therefore, memory access rate can be a representative for progress of benchmark in presence of contention. Figure 2.5 illustrates memory access rate of workload wl16 during runtime. Contention exacerbate trend of memory access rate significantly, however inter-workload interference and execution phase changes are effective as well. These noticeable fluctuations provokes the need for an accurate prediction model.

Schedulers are supposed to predict progress of individual threads. The key differences between approaches often comes down to the specific prediction mechanisms used to estimate the effects of
Figure 2.5: Memory Access Rate of w116

Figure 2.6: Improvement of Optimal, Default and Worst scheduler configuration over Fairness/Performance
different scheduling choices. Many prediction models require an extensive training phase before
execution [8, 16, 24]. Other methods require hardware modification to collect specialized statistics [19, 21], but these mechanisms are not available on current hardware. Prediction is already
difficult on homogeneous systems (and the citations thus far only deal with homogeneity), but it
is even harder on heterogeneous systems where core types vary. One prior approach addresses
predictive mechanisms for heterogeneous systems [20], but it also relies on 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 2.6 compares the fairness and performance of the optimal, default and worst scheduler configuration. 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.
CHAPTER 3

SIMCEPTION : FAST FLEXIBLE MULTICORE SIMULATOR

Shared resource contention is an inevitable incident in today’s multicore systems as threads are competing to exploit resources as much as possible for a higher performance. Due to contention, unpredictable performance may violate QoS guarantees. Employing contention-aware scheduling is necessary when we are dealing with a barrier-synchronized application or a real time system. Contention-aware schedulers uses thread level scheduling to manage contention.

Since we are looking to ensure fairness, we need to estimate and compare progress of threads. Therefore, prediction model is the main component of a contention-aware scheduler. Developing various prediction models on a real system is difficult due to limited performance information, complexity of implementation and possibility to crash system. On the other hand, cycle-accurate simulators can provide further additional data on progress of each threads, but they suffer from significant longer run times. Thus, there was a need for a framework with fast simulation speed, providing common structure of contention-aware scheduler, supporting wide range of systems (by customizing system specification such as number of memory controllers and cores) and simple retrieval of performance counters.

Previously, LinSched [4] has been proposed that host a Linux scheduler in an isolated environment. LinSched has identical behavior comparing to Linux kernel, and can be used for rapid prototyping of new Linux scheduling policies. Drawback is LinSched is running as user space process which makes scheduler unable to receive progress of individual threads. Also, it provides the platform for time sharing scheduling policies mostly while contention-aware schedulers rather space-shares the machine. Another work is StarPU [2] which is runtime system that provides unified execution model on heterogeneous systems. Although StarPU provides progress monitoring of threads (reporting IPC), it fails to measure events of shared resources such as LLC or memory controller.

Primary objective Simception is fast simulation with flexible customization of system in ad-
dition to simple retrieval of performance information. Simception has a modular design and offers large set of architecture parameters such as speed and number of cores, unified/non-unified memory controllers and routing options in interconnect. An overview of Simception is shown on Figure 3.1. Note that as main memory is the dominant factor of contention, our measurement and metrics are pointed directly to memory system. Thus, we ignore contention effect of LLC for now. However, simulator can be easily extended to support LLC further.

We now list the features of Simception:

- **Customizable System Structure**: From a system simulation perspective, Simception is flexible enough to represent various systems. Simception is able to customize system’s components such as speed and number of cores, memory controllers and interconnect. Detailed customization of components allows scheduler to be applicable on wide range of systems (both homogeneous and heterogeneous systems).

- **Performance Counters**: Estimating progress of each thread is necessary for contention-aware scheduler. Simception provides memory access rate, memory controller response time and cycle count to calculate progress of thread.

- **Workload Specification**: We employ different combination of workloads, trying to cover as many as cases that may happen in a real system. Based on memory access rate, benchmarks are divided into CPU intensive and memory intensive. Inspired by scientific benchmarks,
Simception has sample benchmarks that covers memory access behavior of running threads. 

- **Modular Design**: Each component of contention-aware scheduler is implemented as a function in Simception. Therefore, implementing and debugging new policies is convenient and does not affect rest of system.

There have been multiple cycle-accurate simulators that can be used as development platform of a contention-aware scheduler. The main reason of building Simception is to have a fast simulator that can approximately produce results of a real machine. If we use a cycle-accurate simulator, noticeably long runtime would be troublesome. We trade simulation accuracy for a faster simulation. Thus, we evaluate Simception statistically rather than direct comparison.

In this part, we explain some features of Simception in detail. Fairness is trivially achievable if you slowdown all threads significantly. Efficient contention-aware scheduler should enforce fairness with minimum performance overhead. Therefore, measuring performance

Since we are looking to ensure fairness by reducing standard deviation of threads performance, keeping progress of threads close to each other is preferred. Thereupon, we use number of threads in each normalized performance quarter to measure performance tendency between Tile64pro and Simception. If we have \( n \) threads in an experiment, lowest quarter (Q1), includes \( \frac{n}{4} \) threads starting from the lowest performance thread. Same pattern applies to rest of performance quarters. For instance, Q3 represents threads that has normalized performance in range (50%, 75%). We compare number of threads in each quarter in both tile64Pro manycore systems versus Simception.
Figure 3.2 demonstrates quarterly comparison between tile64Pro and Simception. Difference between number of threads in each quarter is less than two. Thus Simception provide sufficient accurate approximation of system behavior that can be used further. Again, we mention Simception is designed to comparing relative performance behavior of threads rather than absolute performance numbers. We trade in accuracy for faster simulation in order to reduce algorithm development overhead.
CHAPTER 4

DIKE: PREDICTIVE, ADAPTIVE SCHEDULER

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).

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 4.1 illustrates Dike’s high-level structure. At initialization, Dike has no knowledge of the current workload. During each scheduling quanta, the Observer records the memory access rate per thread and categorizes threads as compute (C) or memory intensive (M). Next, the Selector sorts threads based on access rate and forms pairs. Dike then computes the mean and standard deviation of memory access rate across all threads. If the system is in a fair state where the standard deviation divided by the mean is less than a user-defined threshold (0.1 by default), no action is taken in the current quanta. 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 performance. If a swap is beneficial, the Migrator actually swaps the threads, forcing each to migrate to the core currently occupied by the other.

As supplementary info, we explained workflow of Dike with pseudocode shown in Algorithm 1. Improvement target and thresholds are set by user. By default, \( \text{swapSize}, \text{quantaLength} \) are set to \( < 8, 500 > \). \( \text{swapSize} \) represent number of threads involving in every quanta while \( \text{quantaLength} \) containing frequency of scheduling in milliseconds. Until all threads are finished, observer records memory access rate of threads in array structure. Then selector forms \( \text{pairs} \) for further swaps. If \( \text{pairs} \) is null, it means that system is fair at present. Otherwise, for each pair,
predictor estimate \textit{totalProfit} from swap. If decider verifies getting benefit from swap, migrator
swaps the pair. Based on $\theta_0$, optimizer updates scheduler configuration.

In addition, Dike employs an \textbf{Optimizer} that adaptively updates the two key scheduling pa-
rameters \textit{quantaLength} – indicating the time between scheduling decisions – and \textit{swapSize} – in-
dicating the number of pairs 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 Fig-
ure 4.1. 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.

4.1 Observer

The observer has two jobs: \textit{thread classification} and \textit{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 \textit{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-
Algorithm 1: Finding pairs of threads in the Selector

```
Require: \( \theta_f \) \quad \triangleright \text{fairness threshold by user}
Require: \( \theta_o \) \quad \triangleright \text{frequency of running optimizer}
Require: \( \text{adaptationGoal} \) \quad \triangleright \text{Fairness / Performance}

1: while NOT Finished() do \quad \triangleright \text{Check if all threads have finished}
2:     \( \text{threads} = \text{observer()} \) \quad \triangleright \text{Array of threads’ memory access rate}
3:     \( \text{pairs} = \text{selector(threads, swapSize, adaptationGoal)} \)
4:     if pairs is NULL then \quad \triangleright \text{if system is fair}
5:         continue
6:     end if
7:     for \( p \) in pairs do \quad \triangleright \( p = < t_l, t_h > \)
8:         \( \text{totalProfit} = \text{predictor}(p) \)
9:         if decide\(r(p, \text{totalProfit}) \) then
10:            migrator\( (p) \)
11:         end if
12:     end for
13:     \text{sleep(quantaLength)}
14:     \text{currentTime} = \text{getSystemTime()}
15:     if currentTime - lastOptimizationTime == \( \theta_o \) then \quad \triangleright \text{if } \theta_o \text{ has passed}
16:         < swapSize, quantaLength >= \text{optimizer}(\theta_f, \text{adaptationGoal})
17:     lastOptimizationTime = \text{getSystemTime()}
18: end if
19: end while
```

bandwidth cores to ensure fair execution time.

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. Thus memory access rate represents contention of multiple shared resources within the memory hierarchy and can function as a 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. Also imagine the case where a thread is spinning waiting to acquire the lock, executing instructions but no progress is making. Therefore IPC fails to cover contention situations. 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) [23].

16
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. The predictor uses this variable to estimate future access rates 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.

4.2 Selector

The selector sorts threads based on memory access rate and selectively pairs them. At first, the selector analyzes current fairness. The system is defined as fair if the standard deviation ($\sigma_i$) over mean of threads 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 2 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 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.

### 4.3 Predictor

After preparing thread pairs for swap — and before performing the migration — Dike ensures the swap will improve either fairness or performance. 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:

$$\text{profit}_{t_l} = \text{CoreBW}_{t_h} - \text{AccessRate}_{t_l} - \text{Overhead}_{t_l}$$  \hspace{1cm} (4.1)

profit is the expected memory access rate gain 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 $\text{CoreBW}_{t_h}$ – as the thread’s new access rate. $\text{CoreBW}_{t_h}$ is provided by the observer 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, $\text{AccessRate}_{t_l}$ is the access rate of thread $t_l$ in the next quanta. To conclude, $\text{CoreBW}_{t_h}$ and $\text{AccessRate}_{t_l}$ are the expected access rates of threads if the swap happens or not, respectively. Further, the parameter $\text{Overhead}_{t_l}$ expresses the reduction in memory access rate due to the context switch overhead.

$$\text{Overhead}_{t_l} = \frac{\text{swapOH}}{\text{quantaLength}} * \text{AccessRate}_{t_l}$$  \hspace{1cm} (4.2)
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.

Once each thread’s profit has been calculated individually using Eqn. 4.1, the total profit of a swap operation is defined as sum of profits of both threads.

\[
totalProfit = profit_l + profit_h
\]  

Total profit is forwarded to the decider for further actions. In some cases, \(profit_l\) (or \(profit_h\)) 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 [20]. Dike requires no a priori 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.

### 4.4 Decider

Once the prediction has been made, Dike decides whether to perform each swap individually. To prevent excessive overhead on a thread, Dike does not swap a thread in consecutive quanta. If any member of thread pair has been swapped in last quanta, scheduler skips the pair. Also, the decider ignores pairs with negative totalProfit respectively.

### 4.5 Migrator

The migrator simply manipulates thread-to-core affinity mappings to swap a thread pair’s cores. We employed swap operation rather than suspension for the following reasons. First, suspension slowdowns the performance significantly as many of cores may end up blocking and waiting for
slowest thread to finish. Second, overhead of swap is not notably higher that suspension, even it involves two threads performance.

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 thread-less. 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.

4.6 Optimizer

Dike has two key parameters that dramatically affect fairness and performance: \textit{quantaLength}, the time between scheduling decisions, and \textit{swapSize}, the number of threads to swap in one quanta. 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.

Each workload has 32 possible scheduling configurations. Figure 5.1 illustrates the effects of different configurations on fairness and performance for two workloads (details about workloads can be found in Section ??). In each subplot, fairness and performance of each configuration is normalized to best configuration. Every cell in heatmap represents a configuration, where x-axis is \textit{swapSize} and y-axis is \textit{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. 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, \textit{quantaLength} is drawn from [100, 200, 500, 1000] milliseconds,
Figure 4.2: Normalized fairness/performance of every configuration for selective workloads while \textit{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 4.3 depicts contour the plot of normalized fairness and performance for each workload type. In subplots, the x-axis and y-axis are \textit{swapSize} and \textit{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* subplot shows higher intensity in the center right, indicating higher fairness is achievable by increasing `swapSize` and decreasing `quantaLength` until 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 4.3.

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 3 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.
Algorithm 2 Finding pairs of threads in the Selector

Require: \( \text{threads} \) \( \triangleright \) array of threads
Require: \( \text{adaptationGoal} \) \( \triangleright \) Fairness / Performance
Require: \( \text{swapSize} \) \( \triangleright \) given by optimizer

1: \( \text{fairness} = \text{getSystemFairness}() \) \( \triangleright \) calculates current fairness
2: if \( \text{fairness} < \theta_f \) then
3: \hspace{1em} return \( \triangleright \) System is fair
4: end if
5: \( n = \text{size(threads)} \) \( \triangleright \) pair of low/high access threads
6: \( \text{pair} = \{ (t_l, t_h) \mid t_l, t_h \in \text{threads} \} \)
7: \( \text{sort(threads)} \) \( \triangleright \) sorts threads based on access rate
8: \( \text{head} = 0, \text{tail} = n - 1 \) \( \triangleright \) pointers to beginning/end of threads array
9: \( \text{counter} = 0 \)
10: if all threads are same type (C/M) then
11: \hspace{1em} for \( k = 0 \) to \( \text{swapSize} \) do
12: \hspace{2em} \( \text{pairs}[k] = \langle \text{threads}[k], \text{threads}[n-k] \rangle \)
13: \hspace{1em} end for
14: \hspace{1em} return
15: end if
16: while \( \text{counter} < \text{swapSize} \) or \( \text{head} < \text{tail} \) do
17: \hspace{1em} for \( i = \text{head} \) to \( n \) do \( \triangleright \) starting from lowest access rate
18: \hspace{2em} if \( \text{threads}[i] \) is violator then \( \triangleright \) violation of placement rule
19: \hspace{3em} \( t_l = \text{threads}[i] \)
20: \hspace{3em} Break
21: \hspace{1em} end if
22: \hspace{1em} end for
23: \hspace{1em} \( \text{head} = i \)
24: \hspace{1em} for \( j = \text{tail} \) downto \( 0 \) do \( \triangleright \) starting from highest access rate
25: \hspace{2em} if \( \text{threads}[j] \) is violator then \( \triangleright \) violation of placement rule
26: \hspace{3em} \( t_h = \text{threads}[j] \)
27: \hspace{3em} Break
28: \hspace{1em} end if
29: \hspace{1em} end for
30: \hspace{1em} \( \text{tail} = j \)
31: \hspace{1em} \( \text{pairs}[\text{counter}++] = \langle t_l, t_h \rangle \)
32: \hspace{1em} end while

return \( \text{pairs} \) \( \triangleright \) pairs of threads for swap operation
Algorithm 3 Optimizing configurations of workloads

Require: $\theta_f$  
Require: adaptationGoal  
Require: $<\text{swapSize, quantaLength}>$  

1: fairness = getSystemFairness()  
2: if fairness < $\theta_f$ then  
3:     return  
4: end if  
5: workloadType = getWorkloadType()  
6: if adaptationGoal is Fairness then  
7:     switch workloadType do  
8:     case $B$ :  
9:         decrease quantaLength  
10:        quantaLength = Math.Max(quantaLength, 100)  
11:     case $UC$ :  
12:         swapSize = Math.Min(swapSize + 2, 16)  
13:         decrease quantaLength  
14:        quantaLength = Math.Max(quantaLength, 200)  
15:     case $UM$ :  
16:         swapSize = Math.Min(swapSize + 2, 16)  
17:         decrease quantaLength  
18:        quantaLength = Math.Max(quantaLength, 500)  
19:     end if  
20: if adaptationGoal is Performance then  
21:     switch workloadType do  
22:     case $B$ :  
23:         increase quantaLength  
24:        quantaLength = Math.Min(quantaLength, 1000)  
25:     case $UC$ :  
26:         swapSize = Math.Min(swapSize + 2, 16)  
27:         increase quantaLength  
28:        quantaLength = Math.Min(quantaLength, 1000)  
29:     case $UM$ :  
30:         increase quantaLength  
31:        quantaLength = Math.Min(quantaLength, 1000)  
32:     end if  

return $<\text{swapSize, quantaLength}>$  

▶ fairness threshold by user  
▶ Fairness / Performance  
▶ current configuration  
▶ calculates current fairness  
▶ System is fair  
▶ Identify workload type  
▶ new configuration
CHAPTER 5

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 5.1. We set one socket to the minimum CPU frequency, and on the other we enable TurboBoost to build a large-scale heterogeneous structure. The experimental workloads are from the Rodinia OpenMP benchmark suite [5] plus STREAM benchmark [11] shown in Table 5.2. Each workload is four benchmarks with 8 threads or 32 total threads. 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.

5.1 Fairness and Performance

Recent work on contention-aware scheduling is built on the Distributed Intensity Online (DIO) approach [25]. 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

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Intel (r) Xeon-E5 , 10 Cores (2.33 GHz), 10 Cores (1.21 GHz) , 25 MB shared LLC , 32 GB RAM</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 14.04</td>
</tr>
</tbody>
</table>
Table 5.2: Workloads used for experiments. Memory Intensive benchmarks are displayed bold

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

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 \textit{swapSize} and \textit{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 quantify fairness of a workload by introducing \textit{Fairness} metric. First we measure the standard deviation of thread run times within a benchmark. Then, get the average of deviations over all of benchmarks. For a workload with \( n \) benchmarks:

\[
Fairness = 1 - \frac{\sum_{i=1}^{n} \sigma_i}{n}
\]

where \( \sigma_i \) is threads standard deviation of execution time of in benchmark \( i \). The higher the \textit{Fairness}, the more predictable the system. In ideal fair system, all threads have same execution time, \( \sigma_i \) is zero and therefore optimal \textit{Fairness} is 1.

Figure 5.1a demonstrates how fairness changes by employing different scheduling methods
for each workload individually and averaged on 16 workloads over the baseline (CFS). This chart 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.

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. In average, Dike-AF achieves 75% of optimal fairness.

Figure 5.1b shows each workload’s speedup over baseline for four the scheduling methods. The horizontal bold line on the y-axis represents the baseline performance of the workload under CFS (1). Comparing the average speedup over baseline, Dike surpasses DIO by 4% in performance 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. Optimal performance is achievable by employing the best known scheduler configuration. Comparing to optimal, Dike-AP improves performance by 71% of optimal performance in geometric mean.

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 and wl12), 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
Table 5.3: Comparing swap count in DIO, Dike, Dike-AF and Dike-AP

<table>
<thead>
<tr>
<th>Workload Type</th>
<th>B</th>
<th>UC</th>
<th>UM</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w11</td>
<td>w12</td>
<td>w13</td>
<td>w14</td>
</tr>
<tr>
<td>DIO</td>
<td>1980</td>
<td>2120</td>
<td>1952</td>
<td>1964</td>
</tr>
<tr>
<td>Dike</td>
<td>9</td>
<td>14</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>Dike-AF</td>
<td>10</td>
<td>9</td>
<td>19</td>
<td>9</td>
</tr>
<tr>
<td>Dike-AP</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

and w14), and Dike delivers higher fairness in return.

Dike-AF and Dike-AP outdo DIO in terms of fairness and performance respectively for each 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 5.1a, Dike-AF provides better fairness than the non-adaptive mode. Likewise, Dike-AP outperforms the non-adaptive mode for each workload according to Figure 5.1b. The reason is that in every step of the adaptation, the optimizer ensures changing scheduling parameters does not harm the desired behavior.

### 5.2 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. In this section we provide further evidence for this argument by evaluating the number of migrations that occur under various scheduling policies.

Table 5.3 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.

5.3 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 5.2 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. 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 at runtime. Figure 5.2 illustrates trend of prediction error for wl6 and wl11. Phase changes with notable change in memory access rate (more probable in compute intensive threads) or completion of threads cause spikes in prediction error as Dike estimates memory access using the moving mean. Vertical dotted lines illustrates the time of finished threads during running (wl6) workload. Despite complications, Dike keeps prediction within 10% of the actual value.
Figure 5.1: Fairness and Performance improvement

(a) Fairness improvement of DIO, Dike, Dike-AF and Dike-AP

(b) Relative Performance of DIO, Dike, Dike-AF and Dike-AP to Linux Default Scheduler
Figure 5.2: Prediction Error of Dike

Figure 5.3: Prediction Error of selective workloads
CHAPTER 6

RELATED

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 [18]. Qureshi et al partition cache space to minimize LLC misses [15]. Wang et al manage process’s cache requirement by re-adjusting scheduling order [22]. 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 [12]. Inspired by FQM, Kim et al introduce Atlas that favors threads with the least attained service [9]. 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 [25]. While LLC miss rate effectively distinguishes compute and memory intensive threads, it does not always indicate contention [19]. Xu et al allocates more CPU time to threads that suffer more slowdown by estimating the standalone IPC of each thread [24]. 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.

Most heterogeneous multicores schedulers utilize the different core types to reach the highest overall throughput [20]. Suleman et al use fast cores to accelerate critical sections of code [17], while Annavaram et al prioritize serial code segments [1]. Lakshminarayana et al assign threads with larger remaining execution times to faster cores [10]. 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 [20]. 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 [14]. Pricopi intelligently reconfigures and allocates cores to applications to form a heterogeneous architecture and minimize makespan [13]. 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.
CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 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 thesis made two major contributions. First designing Simception, a fast flexible multicore contention scheduler that has support for contention-aware scheduling. Simception is used to build prediction model before implementing on a real machine. Second, Dike is a predictive, adaptive contention-aware scheduler for heterogeneous multicores which has been evaluated on Intel Xeon E5 processors. We evaluate Dike with various combinations of compute and memory intensive benchmarks and achieve 65% and 8% improvement in fairness and performance respectively compared to Linux’s 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.

7.2 Future Work

Future work of Dike can be followed in different ways. Currently, Dike uses memory access rate to estimate the progress of running threads. However, some applications that highly rely on LLC for accessing the data, may be ignored by Dike. These type of applications occupy huge chunk of LLC with minimal access rate to memory. Thus, contention occurs in LLC while Dike is not aware. Considering other shared resources such as LLC and interconnect in measuring contention seems a promising path to continue this work.

At present, all threads are forced to use a specific memory controller. Consequently, Dike focuses on that memory controller to reduce the contention. If multiple memory controllers exist
in a system, Dike fails to ensure fairness. With multiple memory controllers, Dike should consider two tasks. First, find and measure the contention that can happen in any of memory controllers. Second, keep track of contention as re-mapping threads to cores may reduce the contention or transfer it from one memory controller to the other. In future, Dike should be developed to have the ability to manage multiple memory controllers.

In addition to major extensions that mentioned earlier, minor improvements can be made in Dike. For example, while swap forces overhead on two threads, suspension only affects a singe thread. Therefore, employing suspension in some scenarios can boost performance further. Simultaneous use of both suspension and swap operations extends the room for more improvement. Another tweak to Dike can be made in prediction model. Optimizing scheduler configuration is a non-convex problem, and various heuristic approach can be tested for more accurate optimization of scheduler.
BIBLIOGRAPHY


