Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin

Tong Li  Dan Baumberger  Scott Hahn

Intel Corporation
2111 NE 25th Ave., Hillsboro, OR, USA
{tong.n.li,dan.baumberger,scott.hahn}@intel.com

Abstract

Fairness is an essential requirement of any operating system scheduler. Unfortunately, existing fair scheduling algorithms are either inaccurate or inefficient and non-scalable for multiprocessors. This problem is becoming increasingly severe as the hardware industry continues to produce larger scale multi-core processors. This paper presents Distributed Weighted Round-Robin (DWRR), a new scheduling algorithm that solves this problem. With distributed thread queues and small additional overhead to the underlying scheduler, DWRR achieves high efficiency and scalability. Besides conventional priorities, DWRR enables users to specify weights to threads and achieve accurate proportional CPU sharing with constant error bounds. DWRR operates in concert with existing scheduler policies targeting other system attributes, such as latency and throughput. As a result, it provides a practical solution for various production OSes. To demonstrate the versatility of DWRR, we have implemented it in Linux kernels 2.6.22.15 and 2.6.24, which represent two vastly different scheduler designs. Our evaluation shows that DWRR achieves accurate proportional fairness and high performance for a diverse set of workloads.

Categories and Subject Descriptors D.4.1 [Operating Systems]: Process Management—Scheduling

General Terms Algorithms, Design, Experimentation, Performance, Theory

Keywords Fair scheduling, distributed weighted round-robin, multiprocessor, lag

1. Introduction

Proportional fair scheduling has long been studied in operating systems, networking, and real-time systems. The conventional approach is to assign each task a weight and the scheduler ensures that each task receives service time proportional to its weight [26]. Since perfect fairness requires infinitesimally small scheduling quanta, which is infeasible, all practical schedulers approximate it with the goal of obtaining small error bounds.

Though well-defined, proportional fairness has not been adopted in most general-purpose OSes, such as Mac OS*, Solaris*, Windows*, and Linux* prior to version 2.6.23. These OSes adopt an imprecise notion of fairness that seeks to prevent starvation and be “reasonably” fair. In these designs, the scheduler dispatches threads in the order of thread priorities. For each thread, it assigns the thread a time slice (or quantum) that determines how long the thread can run once dispatched. A higher-priority thread receives a larger time slice—how much larger is often determined empirically, not a proportional function of the thread’s priority. To facilitate fairness, the scheduler also dynamically adjusts priorities, for example, by allowing the priority of a thread to decay over time but boosting it if the thread has not run for a while [12, 18]. Similar to time slices, the parameters of these adjustments, such as the decay rate, are often empirically determined and are very heuristic.

The lack of precise definition and enforcement of fairness can lead to three problems. First, it can cause starvation and poor I/O performance under high CPU load. As an example, we ran 32 CPU-intensive threads on a dual-core system with Windows XP and Linux* kernel 2.6.22.15. In both cases, the windowing system was quickly starved and non-responsive. Second, the lack of precise fairness can cause poor support for real-time applications as proportional fair scheduling is the only known way to optimally schedule periodic real-time tasks on multiprocessors [1, 29]. Third, it leads to inadequate support for server environments, such as data centers, which require accurate resource provisioning. Unlike traditional data centers, which use discrete systems to serve clients, the trend of multi-core processors enables more services to be consolidated onto a single server, saving floor space and electricity. In these environments, one multiprocessor system services multiple client applications with varying importance and quality-of-service (QoS) requirements. The OS must be able to accurately control the service time for each application.

Many proportional fair scheduling algorithms exist, but none of them provides a practical solution for large-scale multiprocessors. Most algorithms are inefficient and non-scalable due to the use of global run queues. Accessing these global structures requires locking to prevent data races, which can cause excessive serialization and lock contention when the number of CPUs is high. Furthermore, writes to the global queues invalidate cache lines shared by other CPUs, which increases bus traffic and can lead to poor performance. Algorithms based on per-CPU run queues resolve these problems; however, all of the existing algorithms are either weak in fairness or slow for latency-sensitive applications. As multi-core architectures continue to proliferate, the OS must keep up with efficient and scalable designs for fair scheduling.

This paper presents Distributed Weighted Round-Robin (DWRR), a new scheduling algorithm with the following features:

* Accurate fairness. Using the Generalized Processor Sharing (GPS) model [26], DWRR achieves accurate proportional fair-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

\* P\*PP’09, February 14–18, 2009, Raleigh, North Carolina, USA.
Copyright © 2009 ACM 978-1-60558-397-6/09/02. . . $5.00

1
ness with constant error bounds, independent of the number of threads and CPUs in the system.

• **Efficient and scalable operation.** DWRR uses per-CPU run queues and adds low overhead to an existing OS scheduler, even when threads dynamically arrive, depart, or change weights.

• **Flexible user control.** DWRR assigns a default weight to each thread based on its priority and provides additional support for users to specify thread weights to control QoS.

• **High performance.** DWRR works in concert with existing scheduler policies targeting other system attributes, such as latency and throughput, and thus enables high performance as well as accurate fairness.

The remainder of this paper is organized as follows. Section 2 discusses background and related work. Section 3 describes the DWRR algorithm. We discuss our Linux implementation in Section 4 and experimental results in Section 5, which show that DWRR achieves accurate fairness with low overheads. In Section 6, we present a formal analysis of DWRR’s fairness properties and prove that it achieves constant error bounds compared to the idealized GPS system with perfect fairness. We conclude in Section 7.

2. Background on Fair Scheduling

**Generalized Processor Sharing (GPS)** is an idealized scheduling algorithm that achieves perfect fairness. All practical schedulers approximate GPS and use it as a reference to measure fairness.

### 2.1 The GPS Model

Consider a system with \( P \) CPUs and \( N \) threads. Each thread \( i \), \( 1 \leq i \leq N \), has a weight \( w_i \). A scheduler is perfectly fair if (1) it is work-conserving, i.e., it never leaves a CPU idle if there are runnable threads, and (2) it allocates CPU time to threads in exact proportion to their weights. Such a scheduler is commonly referred to as Generalized Processor Sharing (GPS) [26]. Let \( S_i(t_1, t_2) \) be the amount of CPU time that thread \( i \) receives in interval \([t_1, t_2]\). A GPS scheduler is defined as follows [26].

**Definition 1.** A GPS scheduler is one for which

\[
\frac{S_i(t_1, t_2)}{S_j(t_1, t_2)} = \frac{w_i}{w_j}, \quad j = 1, 2, \ldots, N
\]

holds for any thread \( i \) that is continuously runnable in \([t_1, t_2]\) and both \( w_i \) and \( w_j \) are fixed in that interval.

From this definition, two properties of GPS follow:

**Property 1.** If both threads \( i \) and \( j \) are continuously runnable with fixed weights in \([t_1, t_2]\), then GPS satisfies

\[
S_i(t_1, t_2) = S_j(t_1, t_2) \cdot \frac{w_i}{w_j}.
\]

**Property 2.** If the set of runnable threads, \( \Phi \), and their weights remain unchanged throughout the interval \([t_1, t_2]\), then, for any thread \( i \in \Phi \), GPS satisfies

\[
S_i(t_1, t_2) = S_i(t_1) \cdot \frac{w_i}{\sum_{j \in \Phi} w_j} (t_2 - t_1).
\]

Most prior research applied the GPS model to uniprocessor scheduling. For multiprocessors, some weight assignments can be infeasible and thus no GPS scheduler can exist [9]. For example, consider a two-CPU system with two threads where \( w_1 = 1 \) and \( w_2 = 10 \). Since a thread can run on only one CPU at a time, it is impossible for thread 2 to receive 10 times more CPU time than thread 1 unless the system is non-work-conserving. Chandra et al. [9] introduced the following definition:

**Definition 2.** In any given interval \([t_1, t_2]\), the weight \( w_i \) of thread \( i \) is infeasible if

\[
\frac{w_i}{\sum_{j \in \Phi} w_j} > \frac{1}{P},
\]

where \( \Phi \) is the set of runnable threads that remain unchanged in \([t_1, t_2]\) and \( P \) is the number of CPUs.

An infeasible weight represents a resource demand that exceeds the system capability. Chandra et al. [9] showed that, in a \( P \)-CPU system, no more than \( P - 1 \) threads can have infeasible weights. They proposed converting infeasible weights into their closest feasible ones. With this conversion, a GPS scheduler is well-defined for any multiprocessor system.

A GPS scheduler is idealized since, for Definition 1 to hold, all runnable threads must run simultaneously and be scheduled with infinitesimally small quanta, which is infeasible. Thus, all practical fair schedulers emulate GPS approximately and are evaluated from two aspects: fairness and time complexity. Log is the commonly-used metric for fairness [1]. Assume that threads \( i \) and \( j \) are both runnable and have a fixed weight in the interval \([t_1, t_2]\). Let \( S_i(t_1, t_2) \) and \( S_j(t_1, t_2) \) denote the CPU time that \( i \) and \( j \) receive in \([t_1, t_2]\) under some algorithm \( A \).

**Definition 3.** For any interval \([t_1, t_2]\), the lag of thread \( i \) at time \( t \in [t_1, t_2] \) is

\[
lag_i(t) = S_i(t, t) - S_i(t_1, t).
\]

A positive lag at time \( t \) implies that the thread has received less service than under GPS; a negative lag implies the opposite. All fair scheduling algorithms seek to bound the positive and negative lags—the smaller the bounds are, the fairer the algorithm is. An algorithm achieves strong fairness if its lags are bounded by small constants. On the other hand, fairness is poor and non-scalable if the lag bounds are \( O(N) \) function, where \( N \) is the number of threads, because the algorithm increasingly deviates from GPS as the number of threads in the system increases.

2.2 Previous Work

Fair scheduling has its roots in operating systems, networking, and real-time systems. Since the algorithms designed for one area are often applicable to another, we survey prior designs in all of the three areas and classify them into three categories.

2.2.1 Virtual-time-based Algorithms

These algorithms define a virtual time for each task or network packet. With careful design, they can achieve the strongest fairness with constant lag bounds. The disadvantage is that they rely on ordering tasks or packets and require \( O(\log N) \) or, in some cases, \( O(N \log N) \) time, where \( N \) is the number of tasks or network flows. Strong fairness also often relies on the use of centralized run queues, which limits efficiency and scalability of these algorithms. Next, we discuss some representative algorithms in this category.

In networking, Demers et al. [10] proposed Weighted-Fair Queuing (WFQ) based on packet departure times for single-link fair scheduling. Parekh and Gallager [26] showed that WFQ achieves a constant positive lag bound but \( O(N) \) negative lag bound, where \( N \) is the number of flows. WFQ [3] improves WFQ to achieve a constant bound for both positive and negative lag. Blanquer and Özden [4] extended WFQ and WFQ to multi-link scheduling. Other algorithms [13, 15, 16] use packet virtual arrival times and have similar bounds to WFQ.

In real-time systems, previous algorithms [1, 2] obtained constant lag bounds. Many studied adding real-time support to general-purpose OSes. Earliest Eligible Virtual Deadline First (EEVDF) [31] achieves constant positive and negative lag bounds, whereas Biased Virtual Finishing Time (BVFT) [24] obtains similar bounds.
to WFQ. For general-purpose OSes, Surplus Fair Scheduling (SFS) [9], Borrowed-Virtual-Time (BVT) [11], Start-time Fair Queuing (SFQ) [14], and the Completely Fair Scheduler (CFS) introduced in Linux 2.6.23 all have similar designs to WFQ and thus obtain a constant positive lag bound but $O(N)$ negative bound.

Many systems [19, 22, 30] use the Earliest Deadline First (EDF) or Rate Monotonic (RM) algorithm [21] to achieve fair scheduling. The Eclipse OS [5] introduced Move-To-Rear List Scheduling (MTR-LS). Though not using virtual time explicitly, these algorithms are all similar to WFQ in principle and thus have similar lag bounds and $O(\log N)$ time complexity.

2.2.2 Round-robin Algorithms

These algorithms extend Weighted Round-Robin (WRR) [23], which serves flows in round-robin order and transmits for each flow a number of packets proportional to its weight. Round-robin algorithms have $O(1)$ time complexity and thus are highly efficient. However, they have weak fairness with $O(N)$ lag bounds in general. Nevertheless, if task or flow weights are bounded by a constant, a reasonable assumption in practice, they can achieve constant positive and negative lag bounds. Thus, round-robin algorithms are perfect candidates for OSes to use to achieve efficient and scalable fair scheduling.

Unfortunately, most existing round-robin algorithms are non-scalable for multiprocessors because they use centralized queues or weight matrices, such as Group Ratio Round-Robin (GR') [6], Smoothed Round-Robin (SRR) [17], Virtual-Time Round-Robin (VTTR) [25], and Deficit Round-Robin (DRR) [28]. To the best of our knowledge, Grouped Distributed Queues (GDQ) [7] is the only general-purpose OS scheduling algorithm except DWRM that achieves constant positive and negative lag bounds, and uses distributed thread queues. However, GDQ requires significant changes to an existing scheduler and thus does not provide a practical solution. Since it is incompatible with existing OS scheduler policies, such as dynamic priorities and load balancing, which optimize for latency and throughput, GDQ can cause performance slowdowns. In contrast, DWR works in concert with these policies and retains high performance of the underlying scheduler.

2.2.3 Other Algorithms

Lottery scheduling [33] is a randomized algorithm with an expected lag bound $O(\sqrt{N})$ and worst-case bound $O(N)$. Stride scheduling [32] improves it to a deterministic $O(\log N)$ lag bound, but still has weak fairness. Both algorithms have time complexity $O(\log N)$. Petrou et al. [27] extended lottery scheduling to obtain faster response time, but did not improve its time complexity in general. Recently, Chandra and Shenoy [8] proposed Hierarchical Multiprocessor Scheduling (H-SMP) to support fair scheduling of groups of threads, which can be complementary to DWRM. H-SMP consists of a space scheduler, which assigns integer numbers of CPUs to thread groups, and an auxiliary scheduler, which uses any previous fair scheduler to provision the residual CPU bandwidth.

3. Distributed Weighted Round-Robin

This section gives an overview of DWRM, discusses its algorithm details, and illustrates its operation with an example.

3.1 Overview

DWRM works on top of an existing scheduler that uses per-CPU run queues, such as FreeBSD® 5.2, Linux® 2.6, Solaris® 10, and Windows Server® 2003. As its name suggests, DWRM is a distributed version of WRR. The problem with WRR is that it requires a global queue to maintain round-robin ordering—in each round, the scheduler scans the queue and schedules threads in the queue order. For DWRM, we observe that, to achieve fairness, threads do not need to run in the same order in each round—they can run in any order any number of times, as long as their total runtime per round is proportional to their weights.

DWRM maintains a round number for each CPU, initially zero. For each thread, we define its round slice to be $w \cdot B$, where $w$ is the thread’s weight and $B$ is a system-wide constant, round slice unit. A round in DWRM is the shortest time period during which every thread in the system completes at least one of its round slice. The round slice of a thread determines the total CPU time that the thread is allowed to receive in each round. For example, if a thread has weight two and $B$ is 30 ms, then it can run at most 60 ms in each round. The value of $B$ is an implementation choice. As we show in Section 6, a smaller $B$ leads to stronger fairness but lower performance, and vice versa.

When a thread uses up its round slice, we say that this thread has finished a round. Thus, DWRM removes it from the CPU run queue to prevent it from running again. When all threads on this CPU have finished the current DWRM round, DWRM searches other CPUs for threads that have not and move them over. If none is found, the CPU increments its round number and allows all local threads to advance to the next round with a full round slice.

3.2 Algorithm

This section describes DWRM in detail. On each CPU, DWRM performs round slicing to achieve local fairness; across CPUs, it performs round balancing to achieve global fairness.

3.2.1 Round Slicing

Besides the existing run queue on each CPU, which we call round-active, DWRM adds one more queue, round-expired. Though commonly referred to as a “queue” in scheduler nomenclature, the run queue can be implemented with any data structure. For example, many OSes implement it as an array of lists, where each list corresponds to one priority and contains all threads at that priority, whereas the recent CFS in Linux implements it as a red-black tree. Whatever the structure is, DWRM retains it in both round-active and round-expired. Figure 1 illustrates these data structures.

On each CPU, both round-active and round-expired are initially empty and the round number is zero. The scheduler inserts each runnable thread into round-active and dispatches threads from there, as it normally does. For all threads in round-active, the CPU’s round number defines the round in which they are running. DWRM places no control over threads’ dispatching order. For example, if the underlying scheduler dispatches threads based on priorities, DWRM retains that order. This feature is key to DWRM’s ability to keep similar fast response time to the underlying scheduler for latency-sensitive applications.

With any existing scheduler, a thread may run for a while, yield to another thread (e.g., due to quantum expiration), and run again. DWRM monitors each thread’s cumulative runtime in each round. Whenever it exceeds the thread’s round slice, DWRM preempts the thread, removes it from round-active, and inserts into round-
expired, all in O(1) time, i.e., a small constant time independent of
the number of threads and CPUs in the system. Thus, at any time,
DWRR maintains the invariant that if a CPU’s round number is
R, then all threads in its round-active queue are running in round
R and all threads in round-expired have finished round R and are
waiting to start round $R + 1$. Next, we discuss when a CPU can
advance from round $R$ to $R + 1$.

3.2.2 Round Balancing
To achieve fairness across CPUs, DWRR ensures that all CPUs in
the common case differ at most by one in their round numbers.
Section 6 describes this property precisely and proves how it leads
to strong fairness. Intuitively, this property enables fairness because
it allows threads to go through the same number of rounds (i.e.,
run for the same number of their respective round slices) in any
time interval. To enforce this property, whenever a CPU finishes
a round, i.e., its round-active queue becomes empty, it performs
round balancing to move over threads from other CPUs before
advancing to the next round.

To aid round balancing, DWRR keeps a global variable, highest,
which tracks the highest round number among all CPUs at any
time. Section 3.2.4 addresses scalability issues with this global
variable. Let round(p) be the round number of CPU p. Whenever
p’s round-active turns empty, DWRR performs round balancing as
follows:

Step 1: If round(p) equals highest or p’s round-expired is empty,
then
(i) DWRR scans other CPUs to identify threads in round highest
or highest − 1 and currently not running (excluding those that
have finished round highest). These threads exist in round-
active of a round highest CPU or round-active and round-
expired of a round highest − 1 CPU.
(ii) If step i finds a non-zero number of threads, DWRR moves X
of them to round-active of p. The value of X and from which
CPU(s) to move these X threads affect only performance, not
fairness, and thus are left as an implementation choice. Note
that after all X threads finish their round slices on p, p’s round-
active turns empty again. Thus, it will repeat Step 1 and can
potentially move more threads over.
(iii) If step i finds no threads, then either no runnable threads exist
or they are all currently running, so p is free to advance to the
next round. Thus, DWRR continues to step 2.

Step 2: If p’s round-active is (still) empty, then
(i) DWRR switches p’s round-active and round-expired, i.e., the
old round-expired queue becomes the new round-active and the
new round-expired becomes empty.
(ii) If the new round-active is empty, then either no runnable thread
exists or all runnable threads in the system are already running;
thus, DWRR sets p to idle and round(p) to zero. Else, it
increments round(p) by one, which advances all local threads
to the next round, and updates highest if the new round(p) is
greater.

Figure 2 summarizes this algorithm in a flowchart. These oper-
ations add little overhead to the underlying scheduler, since most
OSes already perform similar operations when a run queue is
empty. For example, Linux 2.6, Solaris’ 10, and Windows Server
2003 all search other CPUs for threads to migrate to the idle CPU
for load balancing. DWRR simply modifies that operation by con-
straining the set of CPUs from which threads can migrate. As a
proof-of-concept, we have modified Linux as follows.

Let p be a CPU whose round-active turns empty. When DWRR
scans other CPUs for threads in round highest or highest − 1, for

the first round highest − 1 CPU, $p_0$, it identifies, it moves $\lceil X/2 \rceil$
threads from $p_0$’s round-expired to p’s round-active, where $X$ is the
number of threads in $p_0$’s round-expired. Among all CPUs in round
highest or highest − 1, DWRR also finds the most loaded one
(counting only threads in round-active), $p_a$, where load is subject
to the definition of the underlying scheduler. It moves $Y$ threads
from $p_a$’s round-active to p’s round-active, where $Y$ is the number
of threads that, if moved to p, the load on p would equal the average
CPU load (i.e., total system load divided by the number of CPUs).

3.2.3 Dynamic Events and Infeasible Weights
Whenever the OS creates a thread or awakens an existing one,
DWRR locates the least loaded CPU among those that are either
idle or in round highest. It then inserts the thread into round-
active of the chosen CPU. If this CPU is idle, DWRR sets its round
number to the current value of highest. A thread’s departure (when
it exits or goes to sleep) affects no other thread’s weight and thus
requires no special handling in DWRR. If the user dynamically
changes a thread’s weight, DWRR simply updates the thread’s
round slice based on its new weight.

A unique feature of DWRR is that it needs no weight adjustment
for infeasible weights, an expensive operation that requires sorting
the weights of all threads in the system [9]. With DWRR, any thread
with an infeasible weight may initially share a CPU with other
threads. Since it has a larger weight, it remains in round-active
when other threads on the same CPU have exhausted their round
slices and moved to round-expired. Once its CPU’s round number
falls below highest, round balancing will move threads in round-
expired to other CPUs. Eventually, this thread becomes the only
one on its CPU, which is the best any design can do to fulfill an
infeasible weight.

Figure 2: Flowchart of DWRR’s round balancing algorithm.
3.2.4 Performance Considerations
The global variable highest presents a synchronization challenge. For example, suppose two CPUs, A and B, are both in round highest and each has one thread running. When a new thread, T, arrives, DWRR picks A as the least loaded CPU in round highest and assigns T to it. Suppose that, before DWRR inserts T into A’s round-active, the thread on B finishes its round and moves to B’s round-expired. CPU B then performs round balancing, but finds no thread to move over. Thus, it advances to the next round and updates highest. Now DWRR inserts T into A’s round-active, but A is no longer in round highest.

A simple solution to this race is to use a lock to serialize round balancing and thread arrival handling, which, however, can seriously limit performance and scalability. Instead, we found that this race does no harm. First, it affects only thread placement, not correctness. Second, as Section 6 shows, it does not impact fairness—DWRR achieves constant lag bounds regardless. Thus, we allow unprotected access to highest with no special handling.

Another concern is that DWRR could introduce more thread migrations and thus more cache misses. Our results in Section 5.2 show that migrations on SMPs have negligible performance impact. This is especially true when there are more threads than CPUs, which is when round balancing takes place, because a thread’s cache content is often evicted by peers on the same CPU even if it does not migrate. On the other hand, migrations can impact performance significantly on NUMA systems when threads migrate off their home memory nodes [20]. With DWRR, users can balance between performance and fairness by tuning the round slice unit B. A larger B value leads to less frequent migrations, but weaker fairness, as we show in Section 6.

3.3 Example
Figure 3 shows a simple example of DWRR’s operation. Assume two CPUs and three threads, A, B, and C, each with weight one and round slice of one time unit. At time 0, A and B are in round-active of CPU 0 and C in round-active of CPU 1. At time 1, both A and B have run half a time unit and C has run one time unit. Thus, C moves to round-expired on CPU 1. Since its round-active becomes empty, CPU 1 performs round balancing and moves B to its round-active, but not A because it is currently running. At time 1.5, both A and B have run for one time unit, so they move to round-expired of their CPUs. Both CPUs then perform round balancing, but find no thread to move over. Thus, they switch round-active and round-expired, and advance to round 1.

4. Implementation
DWRR can be easily integrated with an existing scheduler based on per-CPU run queues. To demonstrate its versatility, we have implemented DWRR in two Linux kernel versions: 2.6.22.15 and 2.6.24. The former is the last version based on the so-called Linux O(1) scheduler and the latter is based on CFS. Our code is available at http://triosched.sourceforge.net.

In the O(1) scheduler, each CPU run queue consists of two thread arrays: active and expired. They collectively form round-active in DWRR and we added a third array as round-expired. In CFS, each run queue is implemented as a red-black tree. We use this tree as round-active and added one more tree to act as round-expired. In our Linux 2.6.22.15 implementation, we assign each thread a default weight equal to its time slice divided by the system’s default time slice (100 ms). Linux 2.6.24 already assigns each thread a weight based on its static priority, so our implementation retains this weight as default. In both implementations, we added a system call that allows the user to flexibly control the weight of each thread by setting it to any arbitrary value.

The O(1) scheduler has poor fairness. CFS obtains accurate fairness within a CPU, but unbounded lag across CPUs. To support fast response time, the O(1) scheduler uses heuristics to dynami-
5.1 Fairness

We evaluate DWRR in terms of its fairness and performance.

5. Experimental Results

To evaluate DWRR’s fairness for different weights, we ran four threads of weights one, two, three, and four. Since our system has eight CPUs, we ran eight more threads of weight three in addition to the four foreground threads. For each sampling interval, it computes the lag of the thread for this time interval and its relative error, defined as lag divided by the ideal CPU time the thread would receive during this interval if under GPS. Figure 5 shows our results for each thread simulates a set of business transactions (details in Table 1). Each thread consists of 16 threads, five at nice level one (low priority) and the rest nice zero (default priority). The benchmark runs for 20 minutes and samples the CPU time of each thread every t seconds. For each sampling interval, it computes the lag of the thread for this time interval and its relative error, defined as lag divided by the ideal CPU time the thread would receive during this interval if under GPS. Figure 5 shows our results for t equal to 5, 30, 60, and 120 seconds. For a given sampling interval, each bar shows the maximum absolute lag value among all threads throughout the 20-minute run; above each bar is the maximum relative error. We see that Linux 2.6.24 (based on CFS) has a large relative error and, as the sampling interval increases, its lag increases linearly with no bound. In contrast, lag under DWRR is bounded by 0.5 seconds for all sampling intervals. Thus, DWRR achieves much better fairness.

To evaluate DWRR’s fairness for different weights, we ran four threads of weights one, two, three, and four. Since our system has eight CPUs, we ran eight more threads of weight three in background such that all weights are feasible. We ran for five minutes and sampled the cumulative CPU time of each thread every five seconds. Figure 6 plots our results, which show accurate correlation between thread weights and CPU times.

Finally, we ran SPECjbb2005 to demonstrate DWRR’s fairness under realistic workload. The benchmark is multithreaded where each thread simulates a set of business transactions (details in Ta-
Table 1: Pseudocode for microbenchmark evaluating migration overhead between two CPUs.

```
pin self to CPU 1
  // Warm up cache
touch_cache()
  // Measure cost with warm cache on CPU 1
start = current time
touch_cache()
stop = current time
t1 = stop - start
  // Measure cost with cold cache on CPU 2
migrate to CPU 2
start = current time
touch_cache()
stop = current time
t2 = stop - start
  // Difference in migration cost
migration cost = t2 - t1
```

5.2 Performance

This section evaluates DWRR’s performance by showing that it adds minimum overhead and enables performance similar to that of unmodified Linux 2.6.24.

5.2.1 Migration Overhead

Compared to an existing scheduler, DWRR’s overhead mainly comes from the extra thread migrations it might introduce. Migration overhead includes two components: the cost of moving a thread from one CPU to another and the cost of refilling caches on the new CPU. Our experience [20] shows that the latter often dominates and thus we focus on it. To evaluate cache refill costs, we constructed a microbenchmark. Table 1 shows its pseudocode, where touch_cache() accesses a memory buffer in a hard-to-predict way and the buffer size, i.e., the working set of the benchmark, is configurable. The benchmark calls touch_cache() first to warm up the cache, calls the function again and measures its runtime t1. Then, it migrates to a different CPU, calls the function once again, and measures its runtime t2 on the new CPU. The difference between t1 and t2 indicates the cost of refilling the cache and, consequently, the migration cost.

Figure 7 shows the migration costs for different working set sizes. In one case, the two CPUs reside in different sockets with separate caches; in the other case, they reside in the same socket with a shared L2 cache. With separate caches, the migration cost increases as the working set size increases, because it takes more time to refill caches on the new CPU. However, the cost is bounded by 1.8 ms and drops as the working set exceeds 4 MB, the L2 cache size in our system, because the benchmark incurs high cache misses regardless of migration and the initial cache refill cost turns into a negligible fraction of the total runtime. In the case of a shared cache, since the benchmark only needs to refill the L1 after migration, the migration cost decreases significantly to a maximum of 5.2 µs for the different working set sizes.

For both cases, the costs are far less than the typical quantum length of tens or hundreds of milliseconds. As the multi-core trend continues, we expect more designs with shared caches and thus low migration costs. These results are also conservative; in practice, the costs can be even smaller. As mentioned in Section 3.2.4, DWRR incurs extra migrations only when there are more threads than CPUs. In this case, a thread’s cache content is often already evicted by peers on the same CPU even if it does not migrate.

5.2.2 Overall Performance

Having discussed the individual cost of migrations, we now evaluate the overall performance of DWRR. Our goal is to show that DWRR achieves similar performance to unmodified Linux, but added advantage of better fairness. Table 2 describes our benchmarks. UT2004 represents applications with strict latency requirements, where any scheduling overhead can impact user experience (game scene rendering). Kernbench represents I/O workloads when there are more threads than CPUs. In this case, a thread’s cache content is often already evicted by peers on the same CPU even if it does not migrate.

Table 2: Benchmarks.

| UT2004: Unreal Tournament® 2004 | Unreal Tournament® 2004 is a single-threaded CPU-intensive 3D game. We use its botmatch demo with 16 bots in Assault mode on map AS-Convoy. We run 10 dummy threads in background (each an infinite loop) to induce migrations and expose DWRR’s overhead and use frame rate as the metric. |
| Kernbench | We use the parallel make benchmark. Kernbench v0.30, to compile Linux 2.6.15.1 kernel source with 20 threads (make -j 20) and measure the total runtime. |
| ApacheBench | We use Apache 2.2.6 web server and its ab program to simulate 1000 clients concurrently requesting an 8 KB document for a total of 200,000 requests. Our metric is mean time per request as reported by ab. |
| SPECjbb2005® | We use SPECjbb2005® V1.07 and BEA JRockit® 6.0JVM. Following the official run rule, we start with one warehouse (thread) and stop at 16, and report the average business operations per second (bops) from 8 to 16 warehouses. |
distribution is unattainable, we modeled this behavior to maximally expose DWRR's performance problems. Table 2 shows how we configure each benchmark. All threads have default weight one. Table 3 shows our results for unmodified Linux 2.6.24 and that extended with DWRR. All of the benchmarks achieve nearly identical performance under unmodified Linux and DWRR, demonstrating that DWRR achieves high performance with low overhead.

6. Analytical Results

In this section, we show the invariants of DWRR and, based on them, analyze formally its fairness properties.

6.1 Invariants

Let numThreads(p) denote the total number of threads in round-active and round-expired of CPU p. DWRR maintains the following invariants for any CPU p at any time.

**Invariant 1.** If numThreads(p) > 1, then round(p) must equal highest or highest − 1.

**Proof.** We prove by induction. For the base case, we show that the invariant holds when numThreads(p) > 1 is true for the first time on any CPU p. This occurs when p already has one thread in round-active or round-expired and the scheduler dispatches one more to it. The new thread can be either newly created, awakened, or one that migrated from another CPU due to round balancing. In all cases, DWRR ensures that round(p) must be highest to receive the new thread. Since we allow unsynchronized access to highest, some CPU could update highest after DWRR selects p to receive the new thread, but before it inserts the thread into p’s round-active. In this case, round(p) equals highest − 1 when numThreads(p) turns two, but the invariant still holds.

For the inductive hypothesis, we assume numThreads(p) > 1 and round(p) is highest or highest − 1 at an arbitrary time. We show that it continues to hold onwards. Consider the two cases in which highest can change. First, according to Step 2 of round balancing, when CPU p advances to the next round, if new round(p) > highest, then it updates highest. Thus, the new round(p) equals highest and the invariant holds. Second, if another CPU, p’, updates highest before p does, by the inductive hypothesis, round(p) must be highest or highest − 1 before p’ updates highest. If round(p) is highest, then, after p’ increments highest, round(p) equals highest − 1 and the invariant holds. If round(p) is highest − 1, round balancing ensures that all but the running thread on p migrate to p’ before p’ updates highest. Therefore, when p’ updates highest, numThreads(p) is one and the invariant holds.

**Invariant 2.** If 0 < round(p) < highest − 1, then

(i) numThreads(p) = 1, and
(ii) w is infeasible, i.e., w/W > 1/P, where w is the weight of the thread on p, W is the total weight of all runnable threads in the system, and P is the total number of CPUs.

**Proof.** If numThreads(p) is zero, p is idle and round(p) must be zero. If numThreads(p) > 1, by Invariant 1, round(p) ≥ highest − 1. Therefore, Invariant 2(i) holds.

For Invariant 2(ii), we show that if w is feasible, round(p) ≥ highest − 1 must hold. By Invariant 2(i), there is only one thread on p. Let T denote this thread and t be the time at which T runs for the first time on CPU p. To dispatch T to p, DWRR requires that round(p) be highest at time t. Since we allow unsynchronized access to highest, similar to the argument for Invariant 1, the actual value of round(p) can be highest or highest − 1 at t.

We prove inductively that, after time t, whenever highest increments, the up-to-date value of round(p) must always equal highest or highest − 1. Let tʰ denote the time at which highest increments for the first time after t and V be the total weight of all threads in the system except T, i.e., W = w + V.

For highest to increment at tʰ, round balancing ensures that all threads in the system have finished at least one round slice, which takes the least amount of time when the total weight on each CPU except p equals V/(P − 1), i.e., the load is perfectly evenly distributed to the P − 1 CPUs. Thus, we have

\[ tʰ \geq t + \frac{BV}{P - 1}. \]  

(1)

where B is the round slice unit as defined in Section 3.2.1.

Now, let \( t_p \) denote the time at which DWRR updates round(p) for the first time after time t. Since T is the only thread on p, round(p) increments after T finishes one round slice. Thus, \( t_p = t + wB \). Since w is feasible, by definition, we have

\[ \frac{w}{w + V} \leq \frac{1}{P} \Rightarrow w \leq \frac{V}{P - 1}. \]

Therefore,

\[ t_p \leq t + \frac{BV}{P - 1}. \]  

(2)

From (1) and (2), we have

\[ t_p \leq tʰ. \]

Thus, at time tʰ, when highest changes value to highest + 1, round(p) must have incremented at least once. Let \( h(t) \) denote the value of highest at time t and round(p, t) the value of round(p) at time t. We have

\[ round(p, t) \geq highest(t) - 1, \]

and

\[ round(p, tʰ) \geq round(p, t) + 1. \]

Thus,

\[ round(p, tʰ) \geq highest(t) = highest(tʰ) - 1. \]

Therefore, if w is feasible, \( round(p) = highest - 1 \) holds at all times, and hence Invariant 2(ii).

Given these invariants, we have the following corollary, which is the basis for our lag analysis in the next section.

**Corollary 1.** Let i and j be two arbitrary threads with feasible weights. Let m and \( m' \) be the number of rounds they go through in any interval \([t_1, t_2]\) under DWRR. The following inequality holds:

\[ |m - m'| \leq 2. \]

**Proof.** Let \( h(t) \) denote the value of highest at time t. According to Invariants 1 and 2, at time \( t_1, t \) and j must be on CPUs with round number \( h(t_1) \) or \( h(t_1) - 1 \). Similarly, at \( t_2 \), their CPUs must have round number \( h(t_2) \) or \( h(t_2) - 1 \). Thus, we have

\[ h(t_2) - h(t_1) - 1 \leq m \leq h(t_2) - h(t_1) + 1, \]

\[ h(t_2) - h(t_1) - 1 \leq m' \leq h(t_2) - h(t_1) + 1. \]

Therefore, \(-2 \leq m - m' \leq 2. \)
6.2 Fairness

This section establishes the lag bounds for DWRR. We focus on fairness among threads with feasible weights because DWRR guarantees that threads with infeasible weights receive dedicated CPUs. Let $w_{\text{max}}$ be the maximum feasible weight of all runnable threads in the interval $[t_1, t_2]$. $\Phi$ be the set of runnable threads in $[t_1, t_2]$, $P$ be the number of CPUs, and $B$ be the round slice unit.

**Lemma 1.** Under DWRR, if any thread with a feasible weight undergoes $m$ rounds in the interval $[t_1, t_2]$, then

$$\frac{(m - 2)B}{P} \sum_{i \in \Phi} w_i < t_2 - t_1 < \frac{(m + 2)B}{P} \sum_{i \in \Phi} w_i. \tag{5}$$

**Proof.** We first consider the worst case (in terms of the thread’s performance) in which the length of the interval $[t_1, t_2]$ obtains its maximum value. This occurs in the degenerate case when the system has $P - 1$ threads with infeasible weights. Thus, all threads with feasible weights run on one CPU and, by Corollary 1, the number of rounds they go through must differ from $m$ by at most two. Let $F$ denote this set of threads with feasible weights. For each thread $i \in F$, the CPU time it receives in $[t_1, t_2]$ satisfies

$$S_i(t_1, t_2) \leq (m + 2)w_iB.$$  

Since all threads in $F$ run on the same CPU and at least one goes through $m$ rounds, we have

$$t_2 - t_1 = \sum_{i \in F} S_i(t_1, t_2) < (m + 2)B \sum_{i \in F} w_i. \tag{6}$$

Since all threads in $F$ have feasible weights, their total weight must be feasible too. Thus

$$\sum_{i \in F} w_i = \frac{1}{P}. \tag{7}$$

Combining (5) and (7), we have

$$t_2 - t_1 < \frac{(m + 2)B}{P} \sum_{i \in \Phi} w_i. \tag{8}$$

We now consider the best case in which the length of the interval $[t_1, t_2]$ has its minimum value. This occurs when the system has no infeasible threads and the threads are perfectly balanced such that the total weight of threads on each CPU equals $\sum_{i \in \Phi} w_i / P$. For the thread that goes through $m$ rounds in $[t_1, t_2]$, let $p$ be the CPU on which it runs and $\Omega$ be the set of threads on CPU $p$. Following Corollary 1, we have

$$t_2 - t_1 > \sum_{i \in \Omega} (m - 2)w_i B = \frac{(m - 2)B}{P} \sum_{i \in \Phi} w_i.$$

Therefore, the lemma holds. \hfill $\Box$

From Lemma 1, we now show that DWRR has constant lag bounds.

**Theorem 1.** Under DWRR, the lag of any thread $i$ at any time $t \in [t_1, t_2]$ satisfies

$$-3w_{\text{max}}B < \text{lag}_i(t) < 2w_{\text{max}}B,$$

where $[t_1, t_2]$ is any interval in which thread $i$ is continuously runnable and has a fixed feasible weight.

**Proof.** Let $m$ be the number of rounds that thread $i$ goes through in the interval $[t_1, t]$ under DWRR. The CPU time that thread $i$ should receive in $[t_1, t]$ under GPS is

$$S_i,GPS(t_1, t) = \frac{w_i}{\sum_{j \in \Phi} w_j} (t_1 - t) P.$$

Applying Lemma 1, we have

$$(m - 2)w_i B < S_i,GPS(t_1, t) < (m + 2)w_i B. \tag{9}$$

The CPU time that thread $i$ receives in $[t_1, t]$ under DWRR satisfies

$$m w_i B \leq S_i,DWRR(t_1, t) < (m + 1)w_i B. \tag{10}$$

Based on (9) and (10), we have

$$-3w_{\text{max}}B < S_i,GPS(t_1, t) - S_i,DWRR(t_1, t) < 2w_{\text{max}}B.$$

Since $w_i \leq w_{\text{max}}$, the theorem holds. \hfill $\Box$

In practice, we expect $w_{\text{max}}$ to be small (e.g., less than 100) and $B$ on the order of tens or hundreds of milliseconds. The smaller $B$ is, the stronger fairness DWRR provides, but potentially lesser performance as round balancing would trigger more migrations.

7. Conclusion

Fairness is key to every OS scheduler. Previous scheduling algorithms suffer from poor fairness, high overhead, or incompatibility with existing scheduler policies. As the hardware industry continues to push multi-core, it is essential for OSes to keep up with accurate, efficient, and scalable fair scheduling designs. This paper describes DWRR, a multiprocessor fair scheduling algorithm that achieves these goals. DWRR integrates seamlessly with existing schedulers using per-CPU run queues and presents a practical solution for production OSes. We have evaluated DWRR experimentally and analytically. Using a diverse set of workloads, our experiments demonstrate that DWRR achieves accurate fairness and high performance. Our formal analysis also proves that DWRR achieves constant positive and negative lag bounds when the system limits thread weights by a constant. In our future work, we plan to extend the fairness model and DWRR to the scheduling of more types of resources, such as caches, memory, and I/O devices.

Acknowledgments

We are grateful to our colleagues at Intel, especially Jesse Barnes, Abhijit Davare, Barbara Hohlt, David Koufaty, and Huaio Liu for their help and support during this research. We thank James Anderson and John Calandrino (University of Carolina at Chapel Hill), Bill Huey (Novell, Inc.), Haifeng Yu (National University of Singapore), Peter Zijlstra (Red Hat, Inc.), and the anonymous reviewers for their comments and suggestions on early drafts of this paper.

References


* Intel, Intel Pentium, and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Other names and brands may be claimed as the property of others.