Computer Sciences Department

Uncovering CPU Load Balancing Policies with Harmony

Joe T. Meehean
Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau
Miron Livny

Technical Report #1707

December 2011

University of Wisconsin - Madison

Uncovering CPU Load Balancing Policies with Harmony

Joe T. Meehean, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny
Department of Computer Sciences, University of Wisconsin-Madison

Abstract

We introduce Harmony, a system for extracting the multiprocessor scheduling policies from commodity operating systems. Harmony can be used to unearth many aspects of multiprocessor scheduling policy, including the nuanced behaviors of core scheduling mechanisms and policies. We demonstrate the effectiveness of Harmony by applying it to the analysis of the load-balancing behavior of three Linux schedulers: O(1), CFS, and BFS. Our analysis uncovers the strengths and weaknesses of each of these schedulers, and more generally shows how to utilize Harmony to perform detailed analyses of complex scheduling systems.

1 Introduction

The era of multicore computing is upon us [6], and with it come new challenges for many aspects of computing systems. While there may be debate as to whether new [7] or old [10] kernel architectures are necessitated by the move to multicore processors, it is certain that some care will be required to enable operating systems to run well on this new breed of multiprocessor.

One of the most critical components of the OS in the multicore era is the scheduler. Years of study in single-CPU systems have led to sophisticated and highly-robust single-CPU scheduling algorithms (e.g., the multi-level feedback queue found in Solaris, Windows, and BSD variants [11, 40]); although studied for years in the literature [8, 15, 21, 41, 43, 44, 47], there is little consensus as to the best multiprocessor approach.

An excellent example of this multiprocessor confusion is found in Linux, perhaps one of the most fecund arenas for the development of modern schedulers. At least three popular choices exist: the O(1) scheduler [32], the Completely-Fair Scheduler (CFS) [31], and BFS [27]. Each is widely used and yet little is known about their relative strengths and weaknesses. Poor multiprocessor scheduler policies can (unsurprisingly) result in poor performance or violation of user expectations [16, 29], but without hard data, how can a user or administrator choose which scheduler to deploy?

In this paper, we address this lack of understanding by developing *Harmony*, a multiprocessor scheduling behavior extraction tool. The basic idea is simple: Harmony creates a number of controlled workloads and uses a variety of timers and in-kernel probes to monitor the behavior of the scheduler under observation. As we will show, this straightforward approach is surprisingly powerful, enabling us to learn intricate details of a scheduler's algorithms and behaviors.

While there are many facets of scheduling one could study, in this paper we focus on what we believe is the most important to users: *load balance*. Simply put, does the system keep all the CPUs busy, when there is sufficient load to do so? How effectively? How efficiently? What are its underlying policies and mechanisms?

We apply Harmony to the analysis of the three aforementioned Linux schedulers, O(1), CFS, and BFS, and discovered a number of interesting and previously to our knowledge undocumented behaviors. While all three schedulers attempt to balance load, O(1) pays the strongest attention to affinity, and BFS the least. O(1) uses global information to perform fewer migrations, whereas the CFS approach is randomized and slower to converge. Both O(1) and CFS take a long time to detect imbalances unless a CPU is completely idle. Under uneven loads, O(1) is most unfair, leading to notable imbalances while maintaining affinity; CFS is more fair, and BFS is even more so. Finally, under mixed workloads, O(1) does a good job with load balance, but (accidentally) migrates scheduling state across queues; CFS continually tries new placements and thus will migrate out of good balances; BFS and its centralized approach is fair and does well. More generally, our results hint at the need for a tool such as Harmony; simply reading source code is not likely to uncover the nuanced behaviors of systems as complex as modern multiprocessor schedulers.

The remainder of the paper is organized as follows. §2 provides background, and a detailed overview of Harmony is provided in §3. We then analyze the load-balancing behavior of the three schedulers in §4, discuss related work in §5, and conclude in §6.

2 Background

Before delving into the details of Harmony, we first present some background information on scheduling architectures. We then describe the Linux schedulers of interest – O(1), CFS, BFS – in more detail.

2.1 Scheduling Architectures

There are two basic approaches to multiprocessor scheduling. In the first architecture, a *global run queue* is shared amongst all of the processors in the system [14, 16, 19, 23]. Each processor selects a process to run from this global queue. When a process finishes its quantum, or is preempted, it is returned to this queue and another is selected. This scheme is conceptually simple; the scheduling policy is centralized allowing each processor access to the full global state. It is also naturally work conserving as any eligible process can be selected by an idle processor. One drawback of this approach is that access to the global run queue must be synchronized amongst processors. As the number of processors increases, this can result in lock and cache-line contention, hence limiting scalability [15, 22]. Another shortcoming of this scheme is that it requires a separate mechanism to manage processor affinity. Without processor affinity, a process may not be rescheduled on the same processor that ran it previously, which can degrade performance [26].

The second approach to multiprocessor scheduling is to provide each processor with its own run queue [9, 28, 30, 39]. In this *distributed run queue* scheme, each processor executes processes only from its own run queue; new processes are assigned to an individual processor by a load-balancing mechanism. If processor run queues become unbalanced, the load balancing mechanism migrates processes between processors. A distributed run queue approach requires only limited synchronization (during migration) and simplifies managing processor affinity. The major drawback of this scheme is that it requires a load balancing mechanism and attending policy. A poorly designed policy, or one that simply does not match an application's preferred policy, results in performance degradation [24]. The distributed run queue approach also requires extra effort to be work conserving; if a processor has no eligible processes in its local run queue it may need to steal processes from another processor.

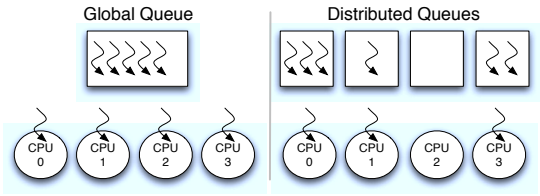


Figure 1: **Global vs. Distributed Queues.** *The figure depicts the two basic architectures employed by modern multiprocessor schedulers. On the left is the single, global queue approach; on the right, the distributed queues approach.*

2.2 Linux Schedulers

This paper uses Harmony to analyze three Linux schedulers: O(1), CFS, and BFS. Linux is an ideal choice for this evaluation because it is commonly deployed in server environments that execute multitasking applications with multiple processors. Over 41% of web servers [4] and 91% of "TOP 500" most powerful computer systems in world [5] run Linux. Despite being so popular, Linux has very little documentation about its multiprocessor scheduling policy (e.g., CFS is distributed without any such documentation [31]).

The most stable is the O(1) [9] scheduler found in kernel versions 2.6 through 2.6.22. This scheduler has been distributed in Red Hat Enterprise Linux since 2005 and is used internally by Google [17]. The O(1) scheduler is implemented using the distribute queue technique. Periodically, each processor checks to ensure that the load is evenly balanced. If the load is imbalanced, an underloaded processor migrates processes from an overloaded processor. The documentation states that it should also be work conserving and that processes "should not bounce between CPUs too frequently" [32].

The Completely Fair Scheduler (CFS) [34] is a proportional-share scheduler currently under active development in the Linux community. This scheduler is found in kernel versions 2.6.23 through the present, and has been distributed under Fedora and Ubuntu for several years. It is also implemented using a distributed queue architecture. Similar to O(1), each processor periodically compares its load to the other processors. If its load is too small, it migrates processes from a processor with a greater load. The documentation provides no description of its multiprocessor policy [31].

The final scheduler is BFS, a proportional-share scheduler. BFS is the default scheduler for the ZenWalk [2] and PCLinuxOS [1] distributions, as well as the CyanogenMod [3] aftermarket firmware upgrade for Android. Unlike O(1) and CFS, this scheduler uses a global queue architecture. BFS documentation provides details about its processor affinity mechanism [27]; however, it is unclear how these low-level details translate into a high-level policy.

3 Harmony

The primary goal of the Harmony project is to enable developers and researchers to extract multiprocessor scheduling policies with an emphasis on load-balancing behavior. We now describe the details of our approach.

3.1 An Empirical Approach

In building Harmony, we decided to take a black-box approach, in which we measure the behavior of the scheduler under controlled workloads, and analyze the outcomes to characterize the scheduler and its policies. We generally do not examine or refer to source code for the "ground truth" about scheduling; rather, we believe that the behavior of the scheduler is its best measure.

This approach has two primary advantages. First, schedulers are highly complex and delicate; even though they are relatively compact (less than 10k lines of code), even the smallest change can enact large behavioral differences. Worse, many small patches are accrued over time, making overall behavior difficult to determine (see [36] for a typical example); by our count, there were roughly 323 patches to the CFS scheduler in 2010 alone.

Second, our approach is by definition portable and thus can be applied to a wide range of schedulers. We do require a few in-kernel probes in order to monitor migrations and queue lengths (discussed further below); however, many systems support such probes (e.g., DTrace [13] or systemtap [20]).

3.2 Using Harmony

The main goal of Harmony is to extract policies from the scheduler under test. To help answer these questions, Harmony provides the ability to easily construct workloads and monitor low-level scheduler behavior; however, the user of Harmony must still design the exact experiments in order to analyze the particular properties of the system the user is interested in.

The Harmony user-level workload controller can be used to start, stop, and modify synthetic processes to create the individual workload suites. This controller must be able to introduce run queue imbalances into the system, and these imbalances should be created instantly rather than slowly accrued to increase precision of the results obtained. Use of process groups and binding to specific CPUs enables us to carefully control where and when load is placed upon the system.

The low-level monitoring component of Harmony records three simple aspects of multiprocessor scheduling behavior: the run queue lengths for each processor, the CPU allocation given to each Harmony process, and the CPU selected to run each Harmony process. Our Linux implementation of Harmony relies on the systemtap kernel instrumentation tool [20]. Harmony's kernel instrumentation records each time a processor is selected to run a Harmony process, and it also samples the run queue lengths every millisecond. Harmony also uses the `/proc/` virtual file system to collect a variety of information about its processes, including CPU allocations and scheduling priorities.

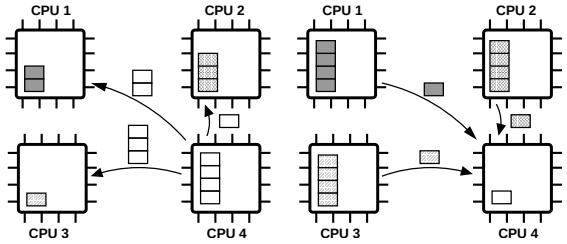


Figure 2: **Single-source and Single-target.** *On the left, CPU 4 is the source of processes (initial: $[2, 3, 1, 10]$); two processes migrate to CPU 1, one to CPU 2, and three to CPU 3 (final: $[4, 4, 4, 4]$). On the right, CPU 4 is underloaded and becomes the target ($[5, 5, 5, 1]$); a single process is migrated from each CPU to CPU 4 ($[4, 4, 4, 4]$).*

3.3 Experiment Types

Although Harmony can be used to setup a variety of experiments, our analysis of Linux schedulers and their load-balancing behavior relies on a few specific experiment types. The first is a **single source** experiment type, in which a single processor is overloaded and the remaining processors are underloaded. This overloaded processor becomes the single source of processes to migrate to the other underloaded processors. The second is a **single target** experiment, in which the imbalance consists of a single underloaded processor, the target; this processor steals processes from each of the remaining overloaded processors. See Figure 2 for an example.

For simplicity, we refer to the initial and final conditions of an experiment with the following notation, $[a, b, c, d]$, which means the first CPU has a processes, the second b, and so forth. Thus, a single-source experiment with idle targets and m processes on the source would have the following initial configuration: $[m, 0, 0, 0]$; the starting configuration of a single-target experiment with a busy target would instead be represented as $[m, m, m, n]$, where $m > n$.

3.4 Hardware and Software Environment

For all experiments in this paper we used a machine with a quad-core Intel Xeon processor; we feel that this size system is a "sweet spot" for the multicore era and thus worthy of intense study. The specific operating systems used in these experiments are Red Hat Enterprise Linux 5.5 (kernel version 2.6.18-194.3.1.el5), Fedora 12 (kernel version 2.6.32.21-168.fc12.x86_64), and Linux kernel 2.6.32 patched with BFS (2.6.32-bfs.313). Each operating system is configured to deliver scheduling interrupts once per millisecond.

4 Multiprocessor Scheduling Policies

We now describe our results of using Harmony to uncover the scheduling policies of the O(1), CFS, and BFS Linux schedulers.

4.1 Load balancing versus Affinity?

We begin with the most basic question for a multiprocessor scheduler: does it perform load balancing across processors and contain mechanisms for maintaining affinity between processes and processors? We begin our examination with a workload that should be straightforward to balance: eight identical 100% CPU-bound processes running on a single source with three idle targets (expressed as $[8, 0, 0, 0]$).

This basic scenario allows us to determine the tradeoffs the underlying scheduler makes between load balancing and affinity. If the multiprocessor scheduler does not have a load balancing mechanism, then all eight processes will remain on the single target. At the other extreme, if the multiprocessor scheduler does not attempt to maintain any affinity, then the processes will be continuously migrated over the lifetime of the experiment. Finally, if the multiprocessor scheduler attempts to achieve a compromise between load balance and affinity, then initially the processes will be migrated across cores and then after some period the processes will each remain on its own core (or migrated less frequently).

Figure 3 shows the number of process migrations over time for the three Linux schedulers. Both O(1) and CFS have an initial burst of process migrations (6 and 30 respectively) and then zero migrations afterward. This indicates that O(1) and CFS perform load balancing with processor affinity, matching their known implementation of using a separate local queue per core. On the other hand, BFS has a sustained rate of roughly 13 migrations per second. This indicates that BFS does not attempt to maintain affinity, and matches well with its known global-queue implementation.

This basic experiment raises many questions. Given that the O(1) and the CFS schedulers achieve the same final balanced allocation of two processes per core, how do the two schedulers each arrive at this allocation? Our initial experiment illustrated that the O(1) scheduler arrives at this balance with fewer total migrations than the CFS scheduler; how does each scheduler determine the *number of processes* that should be migrated? We investigate this question in Section 4.2.

Other questions that are raised are related to *time*. As an example, Figure 4 shows the behavior of the O(1) scheduler over time for this workload; specifically, it illustrates the length of the four run queues for a single experiment starting with $[8, 0, 0, 0]$. This figure shows that when the experiment begins at time 24.05 s, Core 1 has a run queue containing 8 processes while the other three

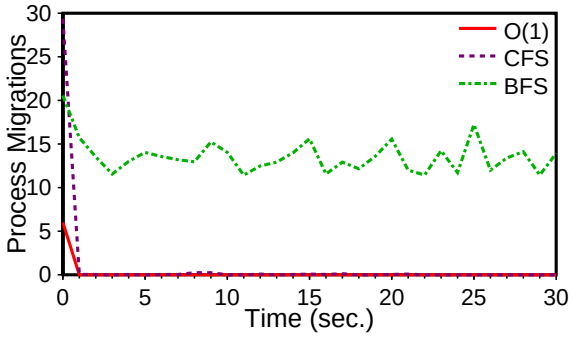


Figure 3: **Timeline of Process Migrations for O(1), CFS, and BFS Schedulers.** *The figure shows the average number of processes migrated over 25 runs with a starting load of eight processes on 1 CPU: $[8, 0, 0, 0]$. Only the first 30s of the experiment is shown; the remainder is similar.*

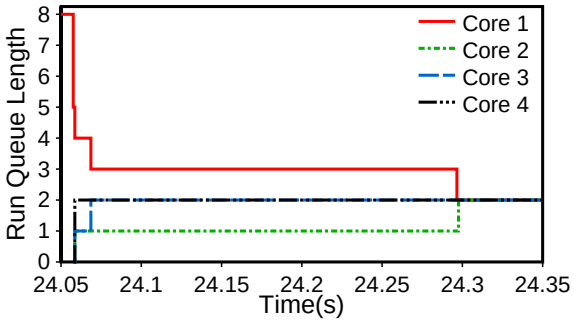


Figure 4: **Timeline of Run Queues for O(1).** *The figure shows the length of each of the four run queues over time for a single experiment initially configured as $[8, 0, 0, 0]$. At time 24.05, Cores 2, 3, 4 are able to start migrating processes away from Core 1; eventually, at time 24.3, all four cores have two processes each.*

CPUs each have zero processes. As time progresses, the load on Core 1 drops in distinct increments from 8 processes to 2, while the load on the other cores increases from 0 to 2. In this case, it takes 250 ms for each CPU to have exactly two processes; furthermore, migrations occur at different points in time on each CPU. We would like to know how long it takes each scheduler to react to load imbalance. Do schedulers react more rapidly when a CPU is idle, when there is a large imbalance, or when there has been an imbalance recently? These questions are addressed in Section 4.3.

The final set of questions are related to *which* processes are migrated by the scheduler. As another example, the three graphs in Figure 5 show the percentage of CPU given to each of the eight identical processes for the O(1), CFS, and BFS schedulers. The figure illustrates that O(1) and BFS allocate a fair percentage of the CPU to every process: each of the eight processes obtains half of a CPU. However, CFS does not always allocate a fair

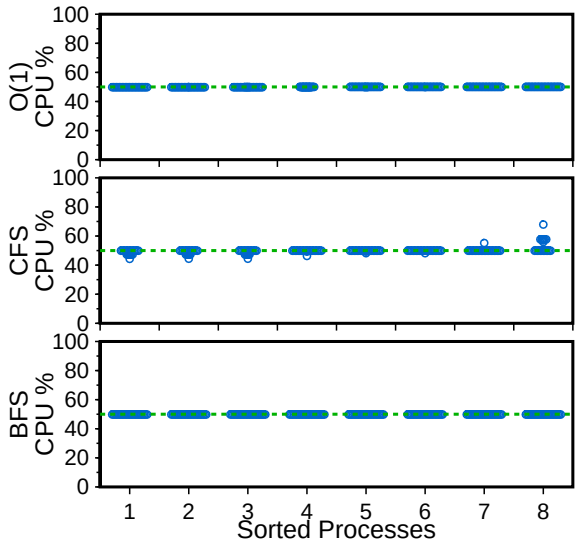


Figure 5: **CPU allocations.** *The 3 graphs show the percentage of CPU given to eight processes running on four CPUs using O(1), CFS, and BFS; the initial allocation is* $[8, 0, 0, 0]$*. In the figure, the processes are sorted by allocation and each point is an allocation from one of 25 runs; the dashed line is the expected allocation for a perfect balance.*

share to every process: in some runs of this workload, some of the processes receive less and some correspondingly more than their fair share. If this inequity occurs for the simplest of workloads, what does this imply for more complex workloads? Thus, we would like to know how each scheduler picks a particular process for migration. Specifically, which processes share a CPU when the workload cannot be divided evenly across processes? Which processes share CPUs when some have different CPU requirements or priorities? We address these questions in Sections 4.4 through 4.6.

4.2 How Many Processes are Migrated?

Our motivational experiments in the previous section lead us to next determine the number of processes each scheduler migrates in order to transform an imbalanced load into a balanced one. We focus on the O(1) and CFS schedulers since they explicitly move processes from one queue associate with one core to another; in contrast, BFS contains a single queue with no default affinity.

Balancing load across multiple processors is challenging because the scheduler is attempting to achieve a property for the system as a whole (e.g., the number of processes on each CPU is identical) with a migration between pairs of CPUs (e.g., migrating process A from CPU 1 to 2). Thus, the scheduler contains a policy for using a series of pairwise migrations to achieve balance.

We hypothesize that there are two straight-forward policies for achieving a global balance. In the first, the scheduler performs a series of *pairwise* balances while ensuring that the final number of processes is evenly divided between the one pair of CPUs. For example, on a four core system with $[30, 30, 30, 10]$, a pairwise balance migrates 10 processes from CPU 1 to CPU 4 so that both have 20; then 5 processes are migrated from CPU 2 to CPU 4 so that both have 25; then, 2 processes are migrated from CPU 3 to CPU 4 to leave the system with the load $[20, 25, 28, 27]$. Pairwise balances must then be repeated again until the system converges. Pairwise balances are simple, but potentially require many cycles of migrations to to achieve a system-wide load balance.

In the second policy, the scheduler performs a *poly-balance* by calculating the number of processes each processor should have when the system is finally balanced (e.g., the number of processes divided by the number of processors). When migrating processes, a poly-balance moves only a source processor's excess processes (those that exceed the system average) to the target. Using the example load of $[30, 30, 30, 10]$, the desired final balance is 25 processes per processor; thus, the poly-balance migrates 5 processes from each of the first three CPUs to the fourth CPU. A poly-balance balances the system quickly, but requires information sharing between processors to calculate the total number of processes.

To determine whether a scheduler uses a pairwise or poly-balance, we measure the number of processes migrated between the first source processor and the target. We examine workloads in which a single target must migrate processes from multiple sources; each source processor has from 20 to 90 more processes than the target and each workload is repeated 10 times. Figure 6 shows the number of migrations performed between the first two CPUs to perform a balance; the graphs on the left and right show the results for the O(1) and the CFS schedulers, respectively.

The graph on the left illustrates that the O(1) scheduler appears to be performing a poly-balance. In most cases, the first migration performed by O(1) matches the number that is exactly needed for a fair global balance; these results hold even as the imbalance (and the resulting number of processes that must be migrated) is varied. In a few cases, significantly greater or fewer numbers of processes are migrated, but these anomalies occur at unpredictable points. We infer that the O(1) scheduler must be using global information across all processors to determine the correct number of processes to migrate.

The graph on the right illustrates that CFS migrates a wide, unpredictable range of processes, usually more than are required for a poly-balance. Thus, the first migration is usually too large and leaves the first source processor with too small of a load; the underloaded source processor must then migrate processes from other processors to complete the global balance. This result corroborates our initial result shown earlier in Figure 3 in

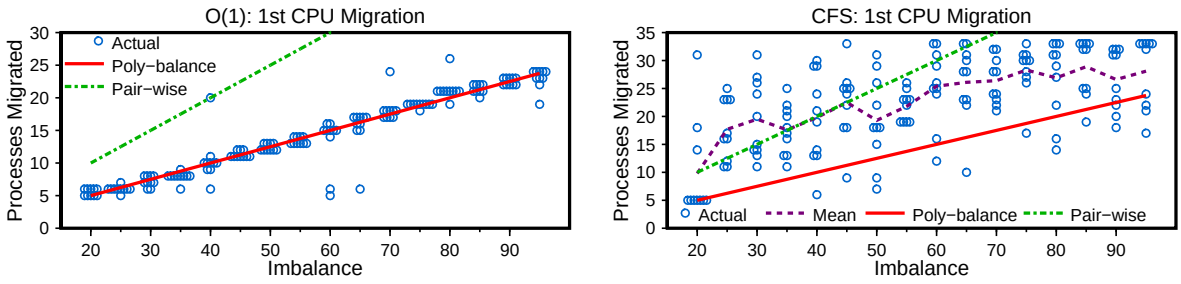


Figure 6: **First Migration: O(1) and CFS.** *The first graph shows the O(1) scheduler and the second the CFS scheduler. Each graph shows the number of processes migrated between the first source and the target processor. The imbalance between the three loaded sources and the single target is varied along the x-axis; that is, [*$10 + x, 10 + x, 10 + x, 10$*].*

which CFS performed 30 total migrations compared to 6 by the O(1) scheduler. Thus, we conclude that CFS is not performing a correct poly-balance.

4.3 Time to Resolve and Detect?

Our next questions revolve around how long it takes a scheduler to detect and respond to a load imbalance. We start with a macro experiment that measures how long the scheduler takes to completely resolve an imbalance and move to micro experiments that measure how long the scheduler takes to detect an imbalance.

The setup for our macro experiment is identical to those in the previous section in which we vary the amount of imbalance between multiple sources and single target. We now measure how long it takes the scheduler to create a balance that is within 15% of optimal. For example, given an ideal balance of $[25, 25, 25, 25]$, the balance $[28, 22, 28, 22]$ is acceptable because the length of each run queue is within 15% of optimal. From the previous results, we expect O(1) will quickly find a balance using a poly-balance and CFS will likely take longer using pairwise balances.

Figure 7 reports the amount of time the O(1) and CFS schedulers take to find acceptable balances given a range of initial imbalances. As expected, O(1) finds a balance within seconds, even for very large imbalances. In comparison, CFS is quite slow to find a stable balance, requiring nearly 9 s on average and 26 s for some workloads.

The large difference in time for O(1) versus CFS to find a stable balance leads us to ask if the difference is due to a better balancing policy or to faster imbalance detection in O(1). Therefore, our next set of experiments investigate how long it takes each scheduler to detect that an imbalance exists and to begin reacting.

Our first micro experiment is designed to determine whether a scheduler is work-conserving: is a processor idle only if there are no eligible processes in the system? In a work-conserving system, a newly-idle processor should immediately steal processes from busy CPUs.

To determine if the scheduler is work conserving, we

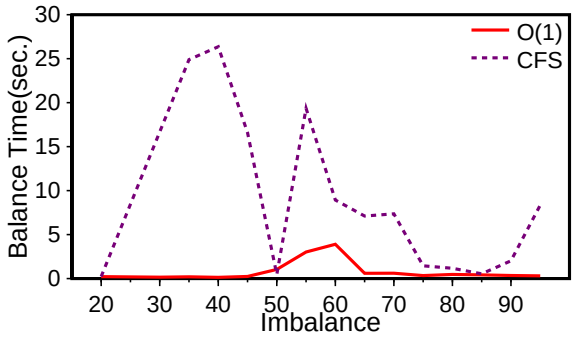
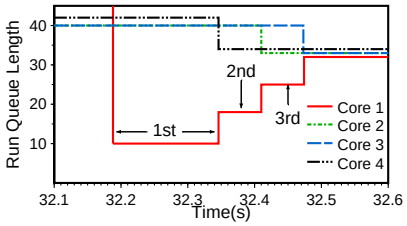


Figure 7: **Time to Resolve Imbalance.** *An imbalance is considered resolved when each run queue is within 15% of optimal. The imbalance between the three loaded sources and the single target is varied along the x-axis; that is, [*$10+x, 10+x, 10+x, 10$*]*

construct a workload with heavily loaded sources and a single target ($[40, 40, 40, 0]$); the target becomes idle after a uniformly random interval. This experiment is repeated 25 times, each time measuring the interval before the target processor steals its first process. A work-conserving policy will immediately migrate processes and non-work conserving schedulers will not.

Harmony shows that, for the O(1) scheduler, the idle target processor begins migrating processes after a single millisecond of idle time; for CFS, migration always begins in less than 1 ms. From these results we infer that both CFS and O(1) are work conserving.

Our next experiment examines how long the system takes to respond when one processor becomes relatively underloaded, but not idle. Underloaded processors effectively reduce the performance of all the processes that are assigned to more heavily loaded processors. A multiprocessor scheduler detects that a processor is underloaded by performing a *balance check* between two processors. Because each balance check incurs some cost, schedulers are likely to need some heuristic about when to perform this operation. We specifically want to know the frequency of balance checks.



O(1)	Median(Max)	Predictable
1st Interval	248 (11458)	No
2nd Interval	64	Yes
3rd Interval	64	Yes
CFS	Median(Max)	Predictable
1st Interval	210.5 (9419)	No
2nd Interval	256	Yes
3rd Interval	64	Yes

Figure 8: **Imbalance Detection.** *The figure illustrates the definition of the three detection intervals: interval 1 occurs between when an imbalance is introduced and when the first migration occurs; interval 2 is time between the first migration and the second; interval 3 is between the second migration and the third. The table reports those intervals (in milliseconds) for O(1) and CFS.*

The setup of this experiment is identical to the previous one, except the load on the target processor is reduced instead of completely eliminated ($[40, 40, 40, 10]$). Because the target processor is imbalanced with respect to each of the three source processors, it must migrate processes from all three and there are three corresponding balance checks. The definition of these intervals is illustrated in Figure 8.

We measure all three intervals to infer the detection policy. If the balance check is based on an event related to process activity (e.g., the check is performed whenever a process exits the run queue), then we expect the first interval to a be a small, fixed amount. On the other hand, if the balance check is performed at some periodic, fixed interval (e.g., the check is performed every 5 seconds), then we expect the measured first interval to appear random, since the load is decreased at a random point in time. The longest recorded first interval should be close to the period of the balance checks.

Figure 8 shows the median and maximum duration of the first interval for the O(1) and CFS schedulers. These results show that, for both schedulers, the first interval is not fixed relative to the time at which the processor became underloaded; thus, we infer that both schedulers perform a balance check relative to some external timer. The maximum duration we observed for this interval for O(1) and CFS were 11 and 9 seconds, respectively.

The results for the second and third intervals are shown in Figure 8 as well. For O(1), the second and third intervals are fixed at 64 ms; for CFS, the second interval usually follows after 256 ms and the third after 64 ms. The implication of these results is that each scheduler performs a periodic balance check, where the period is effected by the likelihood of the imbalance. For a policy like this, the first interval is not predictable, but the second and third intervals (measured from the last migration) are. The policy in CFS appears to be slightly more sophisticated in that the period continues to shorten as more imbalances are detected.

In summary, both O(1) and CFS are work-conserving and perform a periodic balance check. In both, idle processors are assigned eligible processes in about a millisecond. Imbalances that do not involve newly-idle processors may not be detected for long periods of time (roughly 10s); however, once an imbalance is detected, both systems check the next processor relatively quickly (in 64 or 256 ms). We find that O(1) often resolves imbalances within seconds, whereas CFS takes much longer (nearly 9 seconds on average). Because CFS and O(1) have similar detection latencies, we attribute CFS's longer balance time to its process migration strategy.

4.4 Resolution of Intrinsic Imbalances?

Our next questions revolve around how load balancing interacts with the general processor scheduling policy. For example, a proportional-share scheduler should provide the same CPU allocation to each process with the same scheduling weight; unfortunately, this can be difficult to achieve when there exists an *intrinsic imbalance* (i.e., when the number of processes does not divide evenly by the number of processors).

We begin by using Harmony to examine how O(1), CFS, and BFS resolve intrinsic imbalances. One way to achieve a *fair balance*, or an even division of resources across processes, is to frequently migrate processes. However, fair balancing conflicts with providing processor affinity, since frequent migrations mean fewer consecutive process executions on the same processor.

To stress the decisions of each of the three schedulers given workloads with intrinsic imbalances, we introduce five identical processes for four processors; the experiment is started with the load of $[5, 0, 0, 0]$. Thus, if each process is allocated 80% of a processor, the policy is fair.

Figure 9 shows the average allocation each process receives over a 60 second interval for each of the three different schedulers. Figure 10 reports the corresponding rate of migrations over time. The two figures show that the three different schedulers behave significantly different given intrinsic imbalances.

The O(1) scheduler gives a strong preference to affinity over fairness. As shown in the top graph of Figure 9, three processes are allocated an entire processor and the remaining two are each allocated half a processor. Figure 10 supports the observation that few migrations are

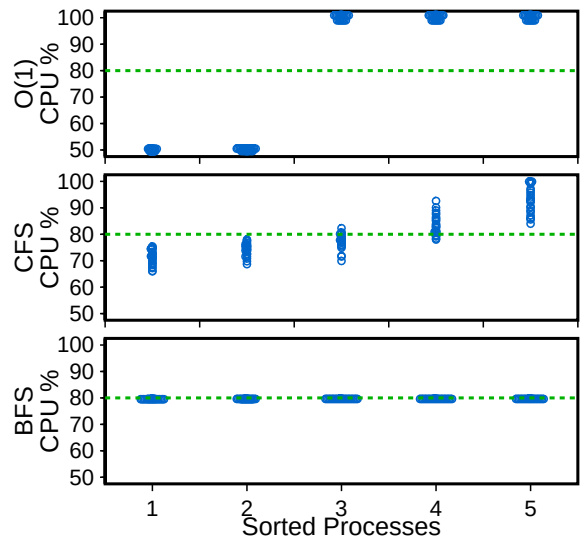


Figure 9: **Allocations with Intrinsic Imbalances.** *The three graphs report the percentage of CPU allocated by O(1), CFS, and BFS to each of five processes running on four processors. Each point represents a process's average CPU allocation over one of the 25 runs of this experiment. The dashed line represents the expected allocation given a perfect fair balance.*

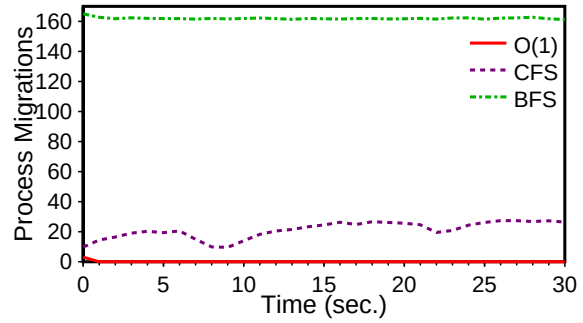


Figure 10: **Migration Timeline with Intrinsic Imbalances.** *The graph shows the average number of processes migrated per second over the lifetime of the experiment for the O(1), CFS, and BFS schedulers.*

performed after finding this acceptable balance.

The BFS scheduler strongly ranks fairness above processor affinity. As shown in the bottom graph of Figure 9, in all 25 runs of this experiment, each process receives within 1% of the exact same allocation. This perfect fair balance comes at the cost of 163 migrations per second.

Finally, the behavior of the CFS scheduler falls between that of the O(1) and BFS schedulers. As shown in the middle graph of Figure 9, CFS allocates each process between 65 to 100% of a CPU; as shown in Figure 10, processes are migrated at a rate of approximately 23 migrations per second.

We now delve deeper into the cause of these allocations by CFS. Figure 12 reports the amount of CPU allocated to five processes on four CPUs in one particular run. The figure shows that processes E and D are each allocated their own CPU for a long period of time (between 35 and 65 seconds) while processes A, B, C share the two other CPUs; then after 65 seconds, CFS migrates process D, at which point processes A and E are each allocated their own CPU. Across many runs, we have found that CFS allocates, for long periods of time, two CPUs to two processes and divides the remaining two CPUs between three processes. In general, CFS is more likely to migrate processes that have recently been migrated. While this technique provides a nice compromise between processor affinity and fair balancing, some processes are migrated quite often: once a process begins migrating it may continue for tens of seconds. These oft-migrated processes suffer both in lost processor affinity and in reduced allocations.

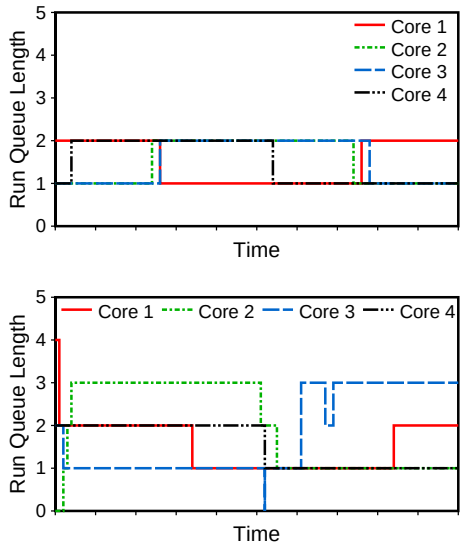


Figure 11: **Run Queue Timelines for Mixed CPU Workloads.** *Each graph shows the run queue length for each of the four cores given a workload with four heavy and four light processes. The top graph illustrates the case where the processes are statically balanced; the bottom graph illustrates a case with dynamic balancing.*

To summarize, given intrinsic imbalances, the O(1) policy strongly favors processor affinity over fairness. BFS has the exact opposite policy: intrinsic imbalances are resolved by performing a process migration every 6ms on average. CFS's policy falls somewhere in the middle: it attempts to resolve intrinsic imbalances while honoring processor affinity. This policy results in 85% less migrations than BFS, but unfairly divides processors amongst processes.

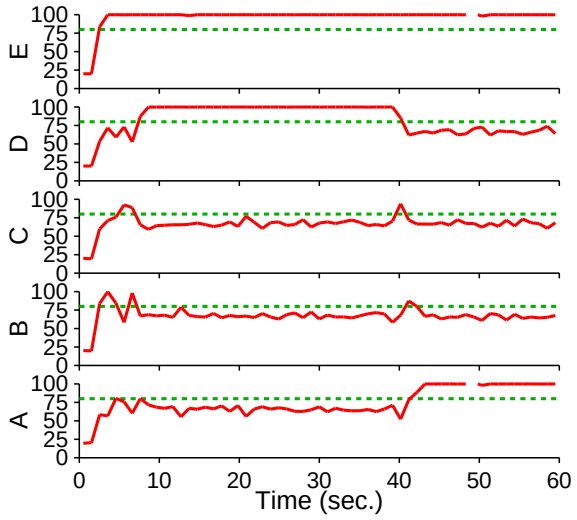


Figure 12: **Allocation Timeline for CFS with Intrinsic Imbalances.** *The five graphs report the amount of CPU given over time to each of 5 processes, lettered A-E, running on 4 CPUs with CFS. The y-axis is the percent of a CPU each process is allocated. The dashed line is the expected allocation for a perfect fair balance.*

4.5 Resolution of Mixed CPU Workloads?

All of our previous experiments have examined homogeneous workloads in which every process had identical characteristics and properties. For the remainder of this paper, we turn our attention to understanding how O(1), CFS, and BFS balance heterogeneous workloads. Load balancing is more difficult with heterogeneous processes because processes are no longer interchangeable. For example, placing two CPU-bound processes on the same processor is not the same as assigning two IO-bound processes.

In this section, we first use Harmony to extract a scheduler's policy for balancing processes with different CPU requirements. We then determine how this policy is implemented by each scheduler. Finally, we examine the effect of these policies on performance.

Given a workload with a mix of heavy and light CPU processes, our first goal is to determine how each scheduler balances those heavy and light CPU processes across processors. In an ideal *weighted balance*, the aggregate CPU demand is the same on each processor. To simplify the task of identifying the ideal weighted balance, we construct workloads such that a balance can only be created by placing a single heavy and a single light process on each processor. We use workloads of four heavy processes (100% CPU-bound) and four light processes (CPU requirements varying from 5 to 100%).

To determine how closely the dynamic balance chosen by each scheduler matches the ideal weighted balance,

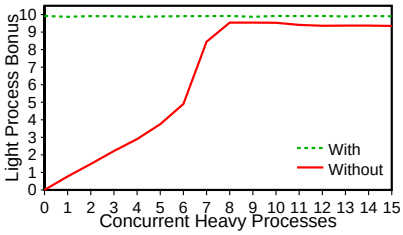


Figure 14: **Sticky Priority Bonuses in O(1).** *A single light process (85% CPU) is run against a variable number of heavy processes; the y-axis show the magnitude of the bonus given to the light process. The line marked "Without" starts the experiment on a cold system. The line marked "With" starts the experiment after warming the system by temporarily running a heavy lead of 16 processes.*

we compare the run queues lengths for the two cases. The first case is represented by the ideal: a run with the processes statically balanced such that there is one heavy and one light process per CPU. Even with the ideal balance, there exists variation in the run queue lengths at each CPU over time. This variation is due both to the light process sleeping at random intervals and how each scheduler decides to allocate the CPU between the light and heavy processes; capturing these non-subtle variations in run queue length is the point of constructing this ideal static balance. For intuition, the top graph in Figure 11 shows the run queue lengths over 100ms for a statically balanced heavy/light workload; each run queue length varies between one and two.

The second case is the behavior of the scheduler when it performs dynamic balancing. The bottom graph in Figure 11 shows an example of the run queue lengths when the loads are dynamically balanced; in this case, each run queue length varies between 0 and 4. To measure how close the dynamic balance is to the ideal static balance, we compare the variance across the run queues. The difference in the variance recorded during the static and dynamic balanced experiments is normalized using symmetric absolute percent error such that the worst possible match is represented by 100% and a perfect match is assigned 0%.

The two graphs in Figure 13 show how well the O(1) and CFS schedulers match the ideal weighted balance for a variety of heterogeneous workloads. We begin by focusing on the first graph, which shows the balance achieved in the long term; in this case, each workload is first run for 30 s to give the scheduler time to distribute processes and then the run queue variance is measured and reported for the next 30s.

The results in the first graph indicate that in the long term both CFS and O(1) usually place processes such that run queue variance is within 25% of the ideal placement. We now discuss these two schedulers in detail.

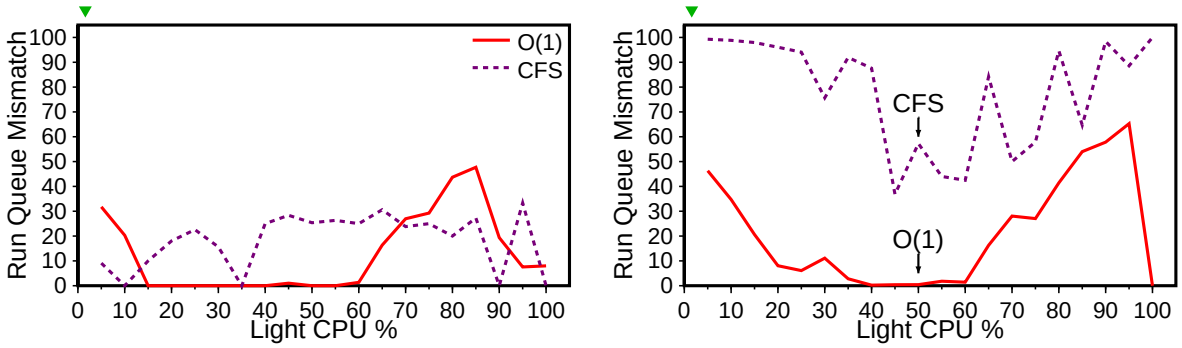


Figure 13: **Run Queue Match.** *Both graphs report the symmetric mean absolute percent error of the variance of the four run queues using a dynamic balance performed by the O(1) or CFS scheduler, as compared to an ideal static balance. The first graph examines the long-term results (30 seconds after a 30 second warm-up); the second graph examines the short-term results (one second after a one second warm-up). In all cases, four heavy and four light processes are started on a single CPU; the amount of CPU used by the light process is varied along the x-axis.*

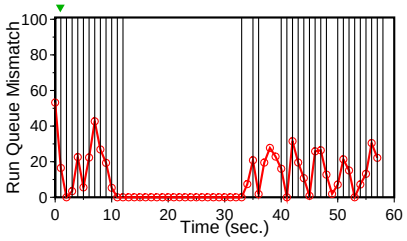


Figure 15: **Losing Balance in CFS.** *This timeline illustrates that CFS finds the ideal weighted balance (between time 11 and 33), but then migrates processes and losses the balance. The vertical lines indicate process migrations.*

While the O(1) scheduler places some heterogeneous workloads very fairly, it does not match the ideal placement well for two regimes of workloads: for a light process using 5-10% of the CPU or one using 65-90% of the CPU. We have examined these cases in more detail and found the following. In the low range(5-10%), we observe that heavy processes are divided across processors evenly, but light processes are clustered in pairs. An extra light process per CPU results in an extra CPU demand of 10% in the worst case and the O(1) scheduler appears to view this an acceptable imbalance.

In the high range with a light process using 65 to 90% of the CPU, we discovered that the light processes receive a much larger allocation than expected, once it has been assigned to a particular CPU. To improve interactivity, the O(1) scheduler gives priority-bonuses to processes that are not CPU-bound; this causes light processes to wait less in the run queue and alters the run queue variance. We discovered that the priority-bonus given to jobs that have been recently migrated is higher due to a phenomena we refer to as *sticky bonuses*. Specifically, because the light process received too little of the CPU in the past when the experiment was being initialized, O(1) gives it more of the CPU in the present. These sticky bonuses, and not the load balancing policy, cause the mismatch between the run queues. Further analysis confirms that the O(1) scheduler achieves weighted balances in the 65-90% light CPU range.

To better illuminate the behavior of sticky bonuses, Figure 14 illustrates two different experiments: one in which bonuses are sticky and one in which they are not. In both experiments, a single light process (85% CPU) is run against a variable number of heavy processes and the magnitude of the bonus given to the light process is reported. The line marked “Without” shows the base case in which the experiment is started on a cold system; in this case, the magnitude of the bonus increases as the light process competes against more heavy processes. The line marked “With” shows what occurs when processes have a past history: in this experiment, the system is warmed by having the light process compete with a constant heavy lead of 16 processes; after these 16 heavy processes are stopped, the previous experiment is repeated. The “With” line illustrates that priority bonuses remain even after the load is reduced; thus, processes maintain bonuses even after the conditions that created the bonus cease to exist. Further experiments (not shown) indicate that O(1) maintains bonuses after migration as well.

In contrast to O(1), CFS consistently misses a weighted balance by a more constant amount. Further analysis reveals that CFS actively searches for a weighted balance by continuously migrating processes at a rate of 4.5 per second on average. When CFS finds a weighed balance, it stops migrating processes for several seconds. After this brief pause, it resumes migrating processes again. This effect is illustrated in Figure 15, which shows that the run queue variance exactly matches that of the ideal case for periods of time (e.g., between 11

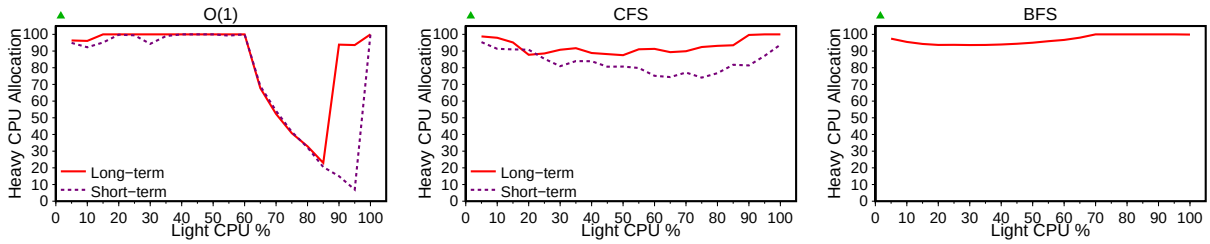


Figure 16: **CPU Allocations for Heavy Processes with O(1), CFS, and BFS.** *Each graph shows the percentage of CPU given to the heavy processes; the allocations are normalized to that received in the ideal static balance. Results for both the short and long-term balances are shown.*

and 33 seconds in this run) and then differs significantly. Therefore, CFS's run queues alternate between being in a weighted balance and being in flux, causing a roughly 25% mismatch on average.

We infer from these results that O(1) and CFS strive for a weighted balance. O(1) allows some minor imbalances for light processes. CFS also continues to search for better balances even when it has found the best one.

4.5.1 Which Process to Migrate?

We next examine how O(1) and CFS find weighted balances. Specifically, we are interested in how these schedulers pick a particular process to migrate.

Using the same experiment from the previous section, we analyze the initial balance instead of the long-term balance achieved. This analysis gives the scheduler one second to find a balance, and then analyzes the run queue variance of the following second. We then compare the dynamically-balanced run queue variance with its ideal static counterpart, as in the previous section.

We expect to see two possible implementations of a weighted balance policy. In the first, the scheduler uses its knowledge of the past behavior of each process to select one for migration. We call this implementation *informed selection*. For example in our mixed CPU experiment, informed selection would enable each target processor to select a single heavy and a single light process for migration. Informed selection should result in a scheduler quickly finding a weighted balance and therefore the short and long-term balances should be roughly the same.

A *blind selection* implementation ignores process characteristics when selecting processes to migrate. Blind selection schedulers are likely to perform several rounds of trial-and-error migration before finding their desired balance. The initial and long-term balances of these schedulers would often be very different; this results in run queue graphs that are not similar.

The two graphs side-by-side in Figure 13 enable us to compare the long-term and short-term results for the two schedulers. For the O(1) scheduler, the short-term results match closely with the long-term results; therefore, we infer that O(1) uses informed selection. However, CFS's short-term and long-term balances do not match at all. Performing further analysis, we discovered that CFS does not select the correct processes for migration initially. Target processors often take two heavy or two light processes instead of one of each. These processors occasionally take too many processes as well. From these results we hypothesis that CFS uses a blind selection implementation.

4.5.2 Impact on CPU Performance?

Finally, we examine the performance implications of the O(1), CFS, and BFS policies for handling mixed CPU workloads. Using the previous workloads of four heavy and four light processes, we report the relative CPU allocation that the heavy processes receive with each scheduler relative to the ideal static layout; we focus on the heavy processes because they suffer the most from load imbalances. The three graphs in Figure 16 report the relative slowdowns given the three different schedulers.

The first graph in Figure 16 reports the slowdown for heavy processes in both the short and long term with the O(1) scheduler. This graph illustrates that when a heavy process competes against a light process consuming less than 60% of the CPU, the O(1) scheduler delivers nearly identical performance to the heavy process as the ideal static layout; however, heavy processes incur a significant slowdown when competing against a process using between 60 and 90% of the CPU. This degradation is directly a result of the sticky bonuses described earlier: even though the heavy and light processes are balanced correctly, the O(1) scheduler gives a boost to the light processes to account for the time in which they were competing with many other processes on a single processor. As expected, the impact of the sticky bonuses wears off over the longer time period for some of the workloads.

The second graph in Figure 16 reports results for CFS; in the long term, CFS's continuous migration policy causes approximately a 10% reduction in performance for the heavy processes. In the short term, CFS performs slightly worse: its blind selection policy causes a 20%

performance degradation for heavy processes.

The third graph shows the relative slowdown for heavy processes using BFS compared to an ideal static balance. Because BFS does not have per-processor run queues, this is the first metric we have shown for how BFS handles heterogeneous workloads. These results show that BFS balances processes such that they receive allocations very similar to those they would achieve with an ideal static balance: within 4%. This balance is achieved by performing an average of 375 migrations every second; this disregard for processor affinity may have serious performance implications for some workloads.

To summarize, all three schedulers have a weighted balance policy. O(1) uses informed selection to find a weighted balance or a close proximity, but O(1)'s per CPU policy of providing sticky bonuses results in severe performance degradation for CPU-bound processes even after migration. CFS continually searches for better balances even after it has found the most appropriate allocation; because weighted balances are discarded, it is unsurprising that CFS uses blind selection when picking a process to migrate. The performance cost of CFS's continuous migration on heavy processes is relatively low ($<$ 10%) since this policy ensures that CFS never spends too long in the best or worst balance. Finally, BFS achieves a near perfect weighted balance (within 4%) by aggressively migrating processes.

4.6 Resolution of Priority Classes?

In our final set of experiments, we examine policies for scheduling heterogeneous workloads with mixed priority classes. Like the previous heterogeneous workload, these workloads are difficult to balance because processes are no longer interchangeable. We are again interested in discovering how these processes are distributed amongst processors, how this distribution takes place, and the performance cost of these policies.

The experiments we use are similar to the mixed CPU requirements experiments except we replace the heavy and light processes with high and low priority processes, varying the differences in priority from 2-38. Due to space constraints, we include only a summary of the results from these experiments. We find that O(1), CFS, and BFS all divide the four high priority processes evenly across the four processors. However, each scheduler handles the low priority processes differently.

The O(1) scheduler clusters low priority processes together on a few processors. When a large priority difference exists between processes, the O(1) scheduler continuously migrates groups of low priority processes (1.5 migrations per second). The performance impact of the O(1) policy is most evident for small differences in priority, in which case the performance of the high priority process may be degraded by up to 20%.

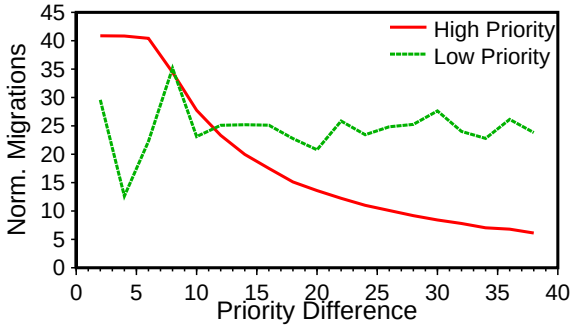


Figure 17: **Migrations for Mixed Priorities with BFS.** *The graph shows the number of normalized migrations per second for the four high and four low priority processes in the workload. The difference in priority between the two classes is varied along the x-axis. To fairly compare high and low priority processes, migrations are normalized by dividing the raw count by their CPU allocation (in seconds).*

CFS divides low priority processes evenly amongst processors, provided the priority difference is small. As priority differences increase, the low priority processes tend to be clustered together on a few processors. Similar to its policy for handling processes with mixed CPU requirements, CFS continuously migrates processes and pauses migration briefly when it finds an acceptable balance. CFS's blind selection causes up to a 75% performance drop for high priority processes in the short term, but less than 4% in the long term.

In contrast to previous experiments, BFS provides some targeted processor affinity for mixed priority workloads. When the priority difference between processes is small (2 to 6), BFS compensates for the small allocations given to low priority processes by migrating them less and providing more processor affinity (Figure 17). In this range, low priority processes are about 1.9 times more likely to execute on the same processor than the high priority processes. In contrast, when the priority difference is large (16 to 38), low priority processes are roughly 2.3 times more likely to run on a different processor when compared to high priority processes. These results strongly suggest that BFS provides differentiated processor affinity based on process priorities. BFS's policy of clustering low priority processes can result in periodic reductions of CPU allocations for high priority process of up to 12%.

5 Related Work

Several studies have applied policy extraction techniques to CPU schedulers [12, 38, 39]. Hourglass is a tool that runs synthetic, instrumented workloads to detect context switch times, timer resolutions, and low-level kernel noise [38]. This tool deals strictly with per-processor scheduling policy, whereas Harmony specifi-

Does the scheduler perform load balancing across processors? (§4.1) For all three, yes.
Does it contain mechanisms for maintaining affinity? (§4.1) O(1) pays the strongest attention to affinity; BFS is the weakest; CFS is in-between.
How does the scheduler determine how many processes to migrate? (§4.2) O(1) uses global information and performs a minimal number of migrations; CFS uses a randomized pairwise strategy, hence performing more migrations. BFS has a centralized queue and constantly migrates processes.
How long does the scheduler take to get to a stable balance? (§4.3) O(1) is relatively quick (due to its minimal migrations); CFS takes an order of magnitude longer.
How long before the scheduler detects an imbalance? (§4.3) If idle, immediately; all schedulers are work-conserving and thus steal work when idle. If non-idle, O(1) and CFS use a periodic check to detect imbalances, which increases in frequency when some imbalance has been detected.
When there is an intrinsic imbalance, how does the scheduler react? (§4.4) O(1) is most unfair, and thus can lead to notable imbalances across processes while maintaining affinity; CFS moves processes somewhat frequently and is more fair, at the cost of affinity. BFS is most fair, constantly moving processes across all CPUs, also at the cost of affinity.
With heterogeneous workload (heavy vs. light CPU), how are processes migrated? (§4.5) O(1) does a good job of balancing heavy and light processes, but some scheduling state is maintained across migrations (perhaps inadvertently). CFS continually tries new placements, and thus will migrate out of good situations (even though unnecessary). BFS and its central queue once again is fair and does well.
With heterogenous workloads (high vs. low priorities), how are processes migrated? (§4.6) All schedulers do well with high-priority processes, dividing them evenly amongst processors. BFS seems to provide targeted processor affinity to mixed-priority workloads.

Table 1: **The Load-balancing Policies Extracted by Harmony.**

cally addresses multiprocessor scheduling.

During the development of FreeBSD's ULE CPU scheduler, the developers also created a synthetic workload simulation tool called Late [39]. Developers used Late's synthetic workloads to measure timer resolutions, fairness, interactivity, and basic performance. Late does not include measurements of run queue lengths or processor selection, limiting its scope of analysis.

The LinSched tool runs the CFS scheduler in a userspace simulator [12]. Researchers and kernel developers can use this tool to observe the behavior of CFS and evaluate new scheduling policies. The goals of Harmony are quite similar to those of LinSched; only the approach differs. Harmony is designed to be generally applicable to a variety of operating systems, whereas LinSched is primarily focused on CFS.

Other systems have also been the focus of policy extraction. Semantic block-level analysis is a technique designed to analyze the behavior of journaling file systems [37]. Shear is a tool that measures the characteristics of RAIDs [18]; by generating controlled I/O request patterns and measuring the latency, Shear can detect a broad range of storage properties. Similar microbenchmarking techniques have been applied to SCSI disks [45], memory hierarchies [46], and TCP stacks [35].

Application and microbenchmark-driven workloads have been used to analyze system-call behavior [25, 33, 42]. These analyses are used to enable accurate simulations, debug problems, and optimize performance.

6 Conclusion

Multicore systems are now commonplace, but multiprocessor scheduling is still under active development. In this paper, we presented Harmony, a system that enables detailed analysis of scheduling behavior. Our specific results are summarized in Table 1; our more general result is that a tool such as Harmony is a necessary and important piece in the scheduling developer's toolkit.

References

[1] PCLinuxOS 2010 Edition is now available for download. http://www.pclinuxos.com/?p=579.

[2] ZenWalk 6.4 is Ready. http://www.zenwalk.org/-modules/news/article.php?storyid=107.

[3] CyanogenMod Android Rom. http://www.cyanogenmod.com/-home/4-1-6-is-here-with-100-more-jet-fuel/comment-page-1, 2009.

[4] Operating system share by groups for sites in all locations. https://ssl.netcraft.com/ssl-sample-report//CMatch/osdv_all, January 2009.

[5] Top500 operating system family share. http://top500.org/stats/list/36/osfam, November 2010.

[6] ASANOVIC, K., BODIK, R., CATANZARO, B., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., PATTERSON, D. A., PLISHKER, W. L., SHALF, J., WILLIAMS, S. W., AND YELICK, K. A. The Landscape of Parallel Computing Research: A View from Berkeley. Tech. Rep. UCB/EECS-2006-183, University of California, Berkeley, Dec 2006.

[7] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHUPBACH, A., AND SINGHANIA, A. The Multikernel: A New OS Architecture for

Scalable Multicore Systems. In *SOSP '09* (Big Sky, MT, October 2009).

[8] BLAGODUROV, S., ZHURAVLEV, S., AND FEDOROVA, A. Contention Aware Scheduling on Multicore Systems. *ACM Transactions on Computer Systems 28*, 4 (December 2010).

[9] BOVET, D., AND CESATI, M. *Understanding the Linux Kernel, Third Edition*, 3rd ed. O'Reilly Media, Inc., 2005.

[10] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *OSDI '10* (Vancouver, BC, December 2010).

[11] BRUNING, M. A Comparison of Solaris, Linux, and FreeBSD Schedulers. http://www.opensolaris.org/os/article/2005-10-14_a_comparison_of_solaris__linux__and_freebsd_kernels/, or just use Google to search for the title, October 2005.

[12] CALANDRINO, J., BAUMBERGER, D., TONG LI, J. Y., , AND HAHN, S. Linsched: The linux scheduler simulator. In *PDCCS '08* (Sept 2008), pp. 171–176.

[13] CANTRILL, B., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic Instrumentation of Production Systems. In *USENIX '04* (Boston, MA, June 2004), pp. 15–28.

[14] CAPRITA, B., CHAN, W. C., NIEH, J., STEIN, C., AND ZHENG, H. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *USENIX '05* (2005), pp. 337–352.

[15] CAPRITA, B., NIEH, J., AND STEIN, C. Grouped distributed queues: distributed queue, proportional share multiprocessor scheduling. In *PODC '06* (2006), pp. 72–81.

[16] CHANDRA, A., ADLER, M., GOYAL, P., AND SHENOY, P. Surplus fair scheduling: a proportional-share cpu scheduling algorithm for symmetric multiprocessors. In *OSDI'00* (2000).

[17] CORBET, J. Ks2009: How google uses linux. *LWN.net* (Oct 2009).

[18] DENEHY, T. E., BENT, J., POPOVICI, F. I., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Deconstructing Storage Arrays. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)* (Boston, MA, October 2004), pp. 59–71.

[19] DUDA, K. J., AND CHERITON, D. R. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *SOSP '99* (1999), pp. 261–276.

[20] EIGLER, F. C., PRASAD, V., COHEN, W., NGUYEN, H., HUNT, M., KENISTON, J., AND CHEN, B. Architecture of systemtap: a Linux trace/probe tool. http://sourceware.org/systemtap/archpaper.pdf, July 2005.

[21] FEDOROVA, A., SELTZER, M., SMALL, C., AND NUSSBAUM, D. Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design. In *USENIX '05* (Anaheim, CA, April 2005).

[22] GOUGH, C., SIDDHA, S., AND CHEN, K. Kernel Scalability – Expanding the horizon beyond fine grain locks. In *Linux Symposium* (2007), vol. 1, pp. 153–166.

[23] GOYAL, P., GUO, X., AND VIN, H. M. A hierarchial cpu scheduler for multimedia operating systems. In *OSDI '96* (1996), pp. 107–121.

[24] HOFMEYR, S., IANCU, C., AND BLAGOJEVIĆ, F. Load balancing on speed. In *PPoPP '10* (2010), pp. 147–158.

[25] JOUKOV, N., TRAEGER, A., IYER, R., WRIGHT, C. P., AND ZADOK, E. Operating system profiling via latency analysis. In *OSDI '06* (2006), pp. 89–102.

[26] KAZEMPOUR, V., FEDOROVA, A., AND ALAGHEBAND, P. Performance implications of cache affinity on multicore processors. In *Euro-Par '08* (2008), pp. 151–161.

[27] KOLIVAS, C. BFS – The Brain F*** Scheduler. http://ck.kolivas.org/patches/bfs/sched-BFS.txt.

[28] KUMAR, A. Multiprocessing with the completely fair scheduler. *IBM developerWorks* (Jan 2008).

[29] LI, T., BAUMBERGER, D., AND HAHN, S. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP '09* (2009), pp. 65–74.

[30] MCDOUGALL, R., AND MAURO, J. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*, 2nd ed. Sun Microsystems Press, 2007.

[31] MOLINAR, I. CFS Scheduler. Linux_2.6.36/Documentation/scheduler/sched-design-CFS.txt.

[32] MOLINAR, I. Goals, Design and Implementation of the new ultra-scalable O(1) scheduler. Linux_2.6.18/Documentation/sched-design.txt.

[33] NARAYANASAMY, S., PEREIRA, C., PATIL, H., COHN, R., AND CALDER, B. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems* (2006), SIGMETRICS '06/Performance '06, pp. 216–227.

[34] PABLA, C. S. Completely fair scheduler. *Linux Journal* (Aug 2009).

[35] PADHYE, J., AND FLOYD, S. Identifying the TCP Behavior of Web Servers. In *SIGCOMM '01* (San Diego, CA, August 2001).

[36] PIGGIN, N. Less Affine Wakeups. http://lwn.net/Articles/124982/, Feb. 2005.

[37] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and Evolution of Journaling File Systems. In *USENIX '05* (Anaheim, CA, April 2005), pp. 105–120.

[38] REGEHR, J. Inferring Scheduling Behavior with Hourglass. In *FREENIX '02* (Monterey, CA, June 2002).

[39] ROBERSON, J. Ule: a modern scheduler for freebsd. In *2nd USENIX Conference on BSD* (2003).

[40] SOLOMON, D. A. *Inside Windows NT*, 2nd ed. Microsoft Programming Series. Microsoft Press, May 1998.

[41] TORRELLAS, J., TUCKER, A., AND GUPTA, A. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing 24* (1995), 139–151.

[42] TRAEGER, A., DERAS, I., AND ZADOK, E. Darc: dynamic analysis of root causes of latency distributions. In *SIGMETRICS '08* (2008), pp. 277–288.

[43] TUCKER, A., GUPTA, A., AND URUSHIBARA, S. The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications. In *SIGMETRICS '91* (San Diego, CA, May 1991).

[44] VASWANI, R., AND ZAHORJAN, J. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *SOSP '91* (Pacific Grove, CA, October 1991).

[45] WORTHINGTON, B. L., GANGER, G. R., PATT, Y. N., AND WILKES, J. On-line extraction of scsi disk drive parameters. In *SIGMETRICS '95/PERFORMANCE '95* (1995), pp. 146–156.

[46] YOTOV, K., PINGALI, K., AND STODGHILL, P. Automatic measurement of memory hierarchy parameters. In *SIGMETRICS '05* (2005), pp. 181–192.

[47] ZAHORJAN, J., LAZOWSKA, E., AND EAGER, D. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Processors. *IEEE Transactions on Parallel and Distributed System 2*, 2 (April 1991), 180–198.