EOS: Automatic, In-vivo Evolution of Kernel Policies for Better Performance

Yan Cui, Quan Chen, Junfeng Yang
Columbia University

Abstract

Today's monolithic kernels often implement a small, fixed set of policies such as disk I/O scheduling policies, while exposing many parameters to let users select a policy or adjust the specific setting of the policy. Ideally, the parameters exposed should be flexible enough for users to tune for good performance, but in practice, users lack domain knowledge of the parameters and are often stuck with bad, default parameter settings.

We present EOS, a system that bridges the knowledge gap between kernel developers and users by automatically evolving the policies and parameters *in vivo* on users' real, production workloads. It provides a simple *policy specification API* for kernel developers to programmatically describe how the policies and parameters should be tuned, a *policy cache* to make in-vivo tuning easy and fast by memozing good parameter settings for past workloads, and a *hierarchical search engine* to effectively search the parameter space. Evaluation of EOS on four main Linux subsystems shows that it is easy to use and effectively improves each subsystem's performance.

1 Introduction

A classic principle in OS kernel design is to separate mechanisms and policies [19]. Specifically, kernel developers build a small yet expressive set of mechanisms, on top of which users can implement flexible policies optimal for their workloads without the need to re-implement the mechanisms. In today's monolithic kernels such as Linux, this principle manifests in a different form. Specifically, these kernels often implement a small, fixed set of policies, while exposing many parameters for users to a policy or adjust the specific settings of a policy. For example, Linux provides three I/O scheduling policies named deadline, cfq, and noop (see §7.1 for details), and users can write a policy's name to ˇ/sys/block/¡disk name¿/queue/scheduler to select the policy. These policies each have between 0 to 12 parameters and users can write to ˇ/sys/block/¡disk name¿/queue/iosched/¡parameter name¿ to change the values of the parameters.

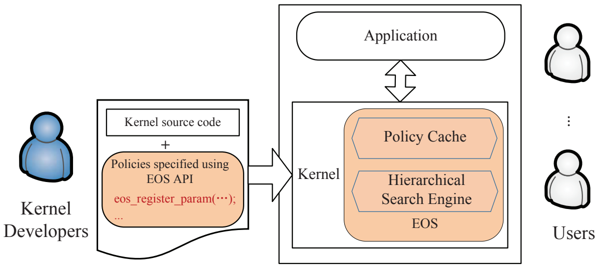


Figure 1: EOS's architecture.

Ideally, the parameters exposed by the kernels should be flexible enough so that users can tune the parameters to get good performance. Unfortunately, this ideal breaks down in practice because of a knowledge gap between kernel developers and users. The developers implement the policies so they know the effects of the parameters well, but they do not know what workloads users will run. They typically use workloads that matter to themselves (e.g., m̌ake -j of the Linux kernel [3]) to set the default policies and parameters. Users know their workloads well but often lack deep understanding of the kernel internals. For example, the Linux completely fair scheduler (CFS) has over 10 parameters with obscure names tightly tied to the CFS algorithm and implementation. A "brief" Linux performance tuning guide alone has several hundred pages [25], most of which are just rules of thumb. Even performance-tuning experts consider the tuning of OS performance as a *black art* [34, 45, 42]. Users thus rarely tune the parameters or tune them correctly, and get stuck with the bad, default policies and parameters set by kernel developers. Our experiments show that these default settings sometimes degrade performance by over 10 times (§7.1).

This paper presents EOS, a system that bridges the knowledge gap between kernel developers and users by automatically evolving the policies and parameters *in vivo* on users' real, production workloads. Key in EOS are three new ideas, illustrated in Figure 1 and described below.

First, EOS provides a simple *policy specification API* for kernel developers to programmatically describe how the policies and parameters should be tuned while implementing the policies. Specifically, they can use the API to describe: (1) *metadata* of the parameters that annotate the parameters with additional information, such as where the parameters are stored in memory (so that EOS can modify them) and the value ranges of the parameters; (2) *sensors* that capture the characteristics of the workloads, such as the average I/O request size, which EOS uses to identify workloads (see next paragraph); and (3) optimization *targets* that developers intent to measure the system's performance, such as the number of I/O requests per unit of time, so EOS knows what to optimize for. This API helps developers pass their domain knowledge to EOS so that it can automatically tune the policies and parameters in vivo.

Second, EOS provides a *policy cache* to make in-vivo tuning easy and fast. It is often very time-consuming to search a large policy and parameter space to find a good setting. Fortunately, workloads are often stable or repetitive over time. For instance, Wikipedia HTTP traces show highly repetitive patterns everyday [13]; a web proxy I/O trace at MSR cambridge shows highly stable behaviors [8]. Thus, once EOS finds a good policy and parameter setting for a workload, it stores this setting into the policy cache. Next time a similar workload comes, EOS simply reuses the cached setting. A second use of the policy cache is to store the intermediate result before the search of a good setting for a workload is done. Since there are many settings to search, EOS may find a good setting only after the workload has repeated many times. EOS stores the intermediate result into the policy cache so that next time it does not have to restart the search from the very beginning.

Third, EOS provides a *hierarchical search engine* to effectively search the policy and parameter space. This framework works as follows. At the top level, it uses a simple, threshold-based algorithm to detect which kernel component (e.g., I/O scheduling or page replacement) is the bottleneck. Then, at the component level, it enumerates through the policies and, for each policy, it searches through different values of the parameters. For each policy and parameter setting, it measures the system's performance, and picks the best performing setting. EOS currently uses a greedy descendant algorithm [18] by default at this level, and allows developers to plug in their favorite search algorithms.

We explicitly designed our EOS system to bridge the knowledge gap between kernel developers and users; it is orthogonal to the massive bodies of work on creating search algorithms that find good or optimal settings out of a huge parameter search space. EOS aims to find a good parameter setting that significantly improves performance. The setting does not have to be the optimal because finding an optimal setting often requires sophisticated tuning algorithms and is extremely time-consuming. We explicitly designed EOS to handle stable or repeatable workloads, not flash workloads because flash workloads occur rarely.

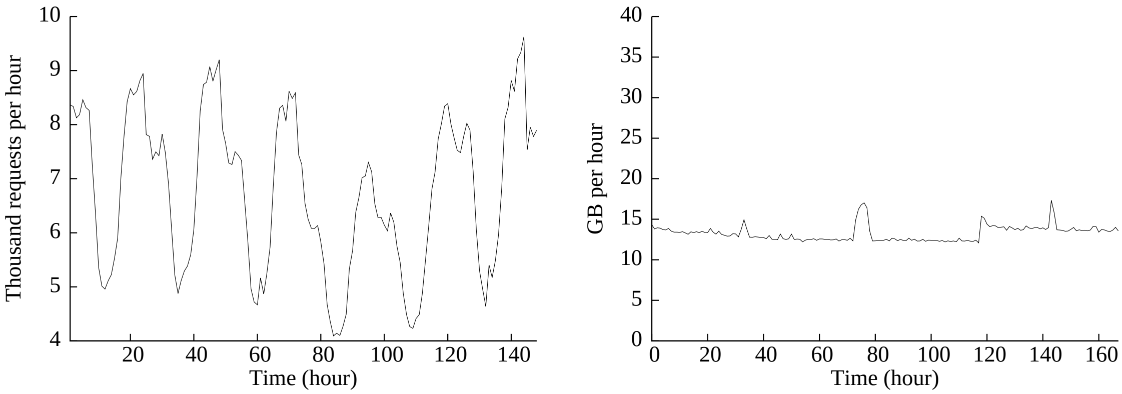
We implemented EOS in Linux. The policy specification API consists a set of C macros and functions. To enable this API, developers simply include a header file. The policy cache and the hierarchical search framework are implemented as a Linux kernel module, dynamically loadable for flexibility of use. Its current search algorithms are tailored for local policies that do not tightly depend on external environments such as networks because of the unpredictability of these environments. For instance policies in the network stack are out of the scope of EOS.

We evaluated EOS on the policies in all four main non-networking subsystems in Linux 3.8.8: disk I/O scheduling, CPU scheduling, synchronization, and page replacement. We augmented synchronization and page replacement with state-of-the-art policies to provide a more thorough evaluation of EOS. Results show that (1) EOS is easy to use, requiring a couple of hours and 16 to 50 lines of code to specify the policies in each subsystem and (2) it effectively improves each subsystem's performance by an average of 1.24 to 2.58 times and sometimes over 13 times.

The rest of this paper is organized as follows. §2 presents EOS's design goals. §3 describes EOS's policy specification API, §4 EOS's policy cache, and §5 EOS's hierarchical search engine. §6 discusses implementation issues. §7 evaluates EOS's performance. §8 discusses related work, and §9 concludes.

2 Design Goals

A key design goal in EOS is to find a parameter settings with good performance; it does not aim to find the absolute optimal setting. Massive bodies of work have been devoted to find optimal settings to maximize system performance. However, current methods for finding optimal settings are still too complex and too slow. We believe this design goal makes EOS ready for practical use by kernel developers and users. In addition, once advances are made to the algorithms for finding optimal settings, EOS can simply adopt them.



(a) One-week Wikipedia HTTP trace

(b) One-week MSR Web proxy I/O trace

Figure 2: Traces of two example real-world workloads. The Wikipedia trace is periodical, and the MSR trace is stable.

A second key design goal in EOS is to focus on relatively steady, repeatable workloads, not flash workloads. This design goal benefits EOS in two ways. First, steady, repeatable workloads give EOS enough time to search through many policy and parameter settings to find a good setting. This search process may take some time, making it difficult to catch up flash workloads that last for very short periods of time. Second, steady, repeatable workloads increase the hit ratio of EOS's policy cache, improving its speed. Recall that the policy cache maps workloads to settings that yield good performance on the workloads. The more steady or repeatable a workload is, the more likely EOS finds a setting in this cache for the workload.

In practice, many workloads match this design goal. Figure 2 shows traces of two such workloads, both collected from real-world server environments. The first is a HTTP trace to Wikipedia [14], one of the world's most popular websites. It covers a one-week period, randomly picked from all traces in WikiBench [13], a realistic web hosting benchmark. The trace is very periodical, repeating a similar spiky pattern every 24 hours. The second trace is an I/O throughput (measured in GB/hour) trace to a Web proxy server at MSR Cambridge [8], obtained from the widely used IOTTA trace repository [15]. The trace is rather steady with few spikes. It is unsurprising that many workloads, especially those in server environments, are steady or repeatable because events that cause flash workloads are rare.

```
struct eos_param {
    const char *name; // unique name of the parameter
    const char *subsys; // the subsystem this parameter belongs
    unsigned long min_value, max_value; // value range
    int linear_or_exponential_search;
    unsigned long (*getter)(void *param); // optional
    void (*setter)(void *param, unsigned long value); // optional
    void *param; // where the parameter is stored in memory
};
struct eos_guard {
    struct eos_param *param;
    unsigned long value;
};
void eos_register_param(struct eos_param *param,
                   struct eos_guard *guard)

struct eos_subsys {
    const char *name; // unique name of the subsystem
    void (*get_sensors)(int size, unsigned long *sensors,
                             int *similarity_threshold);
    unsigned long (*get_optimization_target)();
    int (*is_bottlenecked)(); // returns zero unless bottlenecked
};
void eos_register_subsys(struct eos_subsys *subsys);
```

Figure 3: EOS's policy specification API.

3 Policy Specification API

Figure 3 shows EOS's policy specification API. A key data structure in the API is struct ěos_param which describes the metadata of a parameter. It includes the unique name of the parameter, the kernel subsystem this parameter belongs to (for hierarchical search), the value range of the parameter (for simplicity, EOS requires that parameters have only unsigned long values), how to adjust the value of the parameter in search (linearly by adding or subtracting 1 or exponentially by doubling or halving). In addition, this struct provides a setter and a getter method for accessing the value of the parameter, and field p̌aram points to where the parameter is in memory. As a shortcut, developers can leave either or both of the methods NULL, and EOS simply accesses the p̌aram as a pointer to an un-

```
// called to detect and register a block device
int blk_probe(void) {
    . . .
    // register disk into system
    add_disk(disk);

    // [EOS] I/O scheduling policy for this disk
    struct eos_param *param_sched;
    param_sched = kmalloc(sizeof(struct eos_param), GFP_KENREL);
    param_sched->name = . . . // name based on disk->disk_name
    param_sched->subsys = "disk";
    param_sched->min_value = 0; // three I/O scheduling policies:
    param_sched->max_value = 2; // 0: deadline; 1: cfq; 2: noop
    . . .
    eos_register_param(param_sched, NULL);
    . . .
}
```

Figure 4: Annotating a disk parameter in EOS.

signed long. Once a kernel developer fill this struct for a parameter, she can register this parameter with EOS with method ěos_register_param.

Figure 4 shows an example annotating a parameter in the disk I/O scheduling subsystem. b̌lk_probe is a Linux function for detecting and registering disks. After a disk is registered via ǎdd_disk(disk), developers use EOS's API to describe the parameters that control the I/O scheduling policy for this disk. Since there are three disk I/O scheduling policies, the minimum value and the maximum value of this parameter are 0 and 2 respectively.

EOS provides a second struct ěos_guard to capture the dependencies between parameters. A parameter may be active only when another parameter has a certain value. For instance, the quantum parameter in disk I/O subsystem is only active when the scheduler parameter is set to CFQ. Thus, it is useless to tune the quantum parameter unless scheduler is set to CFQ. To express parameter dependency, a kernel developer passes an ěos_guard when registering a parameter. While in theory a parameter may depend on multiple other parameters with constraints other than equalities, in practice we never observed such cases for all parameters in all four Linux kernel subsystem we evaluated. Thus, EOS's API supports specifying dependency only on one parameter with an equality constraint, though augmenting it is easy.

EOS requires a third struct ěos_subsys for supporting a kernel subsystem. This struct includes three important methods. First, method ǧet_sensors collects the characteristics of a workload, which the policy cache uses to distinguish different workloads. The argument šize specifies the size of šensors and šimilarity_threshold, two arrays to hold outputs from this method. šensors holds the concrete values of the workload characteristics, such as the average size of all I/O requests. šimilarity_threshold specifies a percentage variance such that if two sensor values are within the similarity threshold, they are considered the same. (See §4 for more details.) Second, method ǧet_optimization_target returns an unsigned long value representing the subsystem performance on the workload, which EOS seeks to maximize when it searches through different parameter settings. Kernel developers can choose to let users provide workload-specific optimization targets by providing a system call for setting the optimization target. Lastly, method ǐs_bottleneck checks whether the subsystem is the bottleneck right now. A typical implementation of this method is to compare the percentage of time the subsystem is busy with a threshold (e.g., 80%).

Besides the data structures and methods described so far, EOS's policy specification API also provides syntactic sugar to further ease the use of the API. The most useful syntactic sugar is the macro ěos_register_param_static: it lets kernel developers register a ěos_param struct for a file- or global-scope parameter at compile time, as opposed to calling ěos_register_param at runtime, so that developers can put this macro right next to the declaration of a parameter. To implement this macro, EOS puts compile-time registered ěos_param structs in a special ELF section in the Linux kernel or in a kernel module using linker script tricks.

4 Policy Cache

The policy cache brings two benefits. First, it memoizes good parameter settings and reuses them on similar workloads, greatly reducing the time spent in searching for good parameter settings. Second, it helps make the search incremental. Specifically, when a workload runs for a time shorter than what it takes for EOS to find a good parameter setting, EOS stores the intermediate search result in the policy cache, so that the next time a similar workload runs, EOS can resume the search.

EOS maintains a sub-cache for each kernel subsystem because a subsystem's parameters are typically independent of another subsystem's. A sub-cache is organized as a list of <`workload signature`, `parameter setting`> pairs, where the workload signature captures the characteristics of a workload and the parameter setting makes the subsystem perform well on the workload. EOS computes the workload signature by invoking the subsystem's ǧet_sensors method.

To search a sub-cache to see if a workload exists, EOS invokes ǧet_sensors to compute the signature of the workload, denoted s_1. It then scans its list and compares the signature with each signature s_2 on the list to see if s_1 is within the similarity threshold of s_2. For instance, if s_1 has a field with value 8, s_2's corresponding field has a value 10 and similarity threshold 20, then EOS considers that these two fields have similar values because 8 is within 20% of 10. If EOS determines that all fields of the two signatures are similar, it considers the two workloads similar and reuses the setting associated with workload s_2 for workload s_1. (There may be multiple entries matching a given signature; EOS always returns the first match.)

To store intermediate search results, EOS stores additional information used by the search engine (§5) in addition to the current parameter setting. When a similar workload runs, EOS uses the additional information to resume the search.

Implementation-wise, EOS limits the cache size to be 1000 and replaces entries that are least recently used. In our evaluation, cache replacement never occurred for any of the workloads. A 12-hour Wikipedia trace used only 130 entries, the maximum of all evaluated workloads. EOS persists the policy cache across reboots to ˇ/var/eos/cache to save warm-up time.

5 Hierarchical Search Engine

EOS's hierarchical search engine operates at two levels: it first detects which subsystem is bottlenecked and then searches for a good parameter setting within the subsystem. Once the subsystem is no longer a bottleneck, a second subsystem may become bottlenecked and EOS moves on to tune the second subsystem. The rational to focus on one subsystem at a time matches the general performance tuning experience: performance problems typically occur when only one subsystem is bottlenecked. To detect which subsystem is bottlenecked, EOS invokes the subsystem's ˇis_bottleneck method.

To search for a good parameter setting, EOS can in principle leverage the algorithms proposed by the massive bodies of prior work on performance tuning [32]. We opted for an algorithm called orthogonal search [40] because this algorithm is simple, finishes quickly, and finds parameter settings with good performance. Operationally, this algorithm iterates through the list of parameters and finds the best value for each parameter. It then combines these best values into the resultant parameter setting.xs The intuitions are that (1) parameters are largely independent so they can be searched separately and (2) the effects of the parameters on performance are largely monotonic, e.g., if increasing the value of a parameter improves performance, then we should keep increasing the value.

We made two modifications to this algorithm to make it more efficient within the context of EOS. First, our modified algorithm prioritizes toward the more limited parameters – those with smaller number of possible values. The intuition is that, since the number of possible values is small, the difference in the values often has a large impact on performance. Thus, once EOS finds a good value for a limited parameter at the beginning of the search process, it enjoys a good application performance for the rest of the search, improving the average application performance. Second, our modified algorithm respects the dependencies between parameters. Recall that kernel developers can express when a parameter is active depending on another parameter using EOS's API (§3). Our algorithm does not search a parameter if it is not active.

EOS periodically checks whether it should initiate a new search process. In our current implementation, this period is every 15 minutes. When searching within a subsystem, EOS checks the subsystem's performance by calling ğet_optimization_target 5 seconds after it activates a setting to allow the setting to stabilize.

6 Implementation

We implemented EOS in Linux 3.8.8. The policy specification API consists of a set of C macros and functions. To enable this API, kernel developers simply include a header file. The policy cache and the hierarchical search engine are implemented as a dynamically loadable kernel module, making it flexible for users.

To provide a more thorough evaluation of EOS, we further modified Linux to add several additional policies to two subsystems. We describe these modifications in the next two subsections.

6.1 Synchronization Subsystem

Linux uses ticket spin lock as the basic low-level synchronization policy. However, its performance is not good when a lock is seriously contended or seldom contended [22]. To solve this problem, we made two modifications to the spin lock implementation, described below.

The first modification adds a back-off to the ticket spin lock. The pseudo code is shown in Figure 5. Instead polling the `current` variable in the lock constantly which causes cache line bounces in high contention, a new lock requester pauses for $C \times N$ before each cache polling, where C represents the polling weight for each requester and N is the number of the current lock requesters. The default value of C is set to zero.

```
struct spinlock_t {
    int current;
    int next;
};

void spin_lock(spinlock_t *lock) {
    int t=fetch_and_inc(&lock->next);
    while (t != lock->current)
        pause(C*(t − lock->current));
}
void spin_unlock(spinlock_t *lock) {
    lock->current++;
}
```

Figure 5: Pseudocode for backoff-based ticket lock.

The second modification adds two new locking policies to handle low-level and high-level contention. These two policies are: (1) Test-and-Test-And-Set (TTAS) lock, which implements double-checked locking [41]; and (2) MCS lock, which lets each lock requester spins on its own cache line [36].

We call this new lock implementation the *mixed lock*. Based on a global variable in the kernel (`method_tuner`), the mixed lock dynamically switches to a lock policy to cope with different contention levels. This design is motivated by reactive lock [31].

The data structure of each mixed lock contains four components, including the data structures of three locks and a m̌ode field, indicating which lock policy this mixed lock currently is using. Note that we cannot use m̌ethod_tuner to figure out the current lock policy of a mixed lock because the lock may be already held under a different policy. All the lock-policy-specific data structures are stored in the same cache line for high performance. The m̌ode variable is mostly read but seldom written, so it is stored in a different cache line to reduce false sharing.

The mixed lock interface is as follows. The functions `enum release_mode acquire_mixed_lock (struct lock* lock, struct qnode *node)`, `enum release_mode acquire_mixed_trylock (struct lock* lock, struct qnode *node)`, `void release_mixed_lock(struct lock* lock, struct qnode* node, enum release_mode mode)` and `void init_lock(struct lock *lock)` are used to acquire, try to acquire, release and initialize a mixed lock, respectively. The function `acquire_mixed_lock(...)` is used to route the lock request to a specific lock protocol (TTAS, MCS or back-off based ticket) based on the `mode` variable of the mixed lock. For each lock protocol, if the lock is acquired successfully, the `method_tuner` variable is checked to determine whether to switch to another protocol, or else, the `mode` variable of the mixed lock is examined to select the correct protocol to wait. The return value of `acquire_mixed_lock` indicates whether to switch to a different lock protocol and acts as a parameter of the lock releasing function.

When releasing a mixed lock, if the `mode` variable indicates that we do not need to change lock protocol, EOS simply releases the trivial lock by calling the releasing function of the current lock protocol. protocol should be switched to another protocol, EOS acquires the target protocol, modifies the `mode` variable, invalidates current protocol and finally releases the target protocol. In this way, we can ensure that only one valid lock protocol exists and requesters at invalid protocols will fail the request and retry the valid protocol.

For the three lock protocols in the mixed lock, we give priority to the TTAS lock by setting it as the default protocol, because most locks in the kernel are seldom heavily contended. Thus, in function `init_lock`, the `mode` variable is TTAS and other locks are invalid. The mixed lock needs a total 300 lines of C code to implement.

6.2 Page Replacement Subsystem

Linux uses LRU2Q as the page replacement policy to determine which pages should be reclaimed if the free pages are not enough for future use. Specifically, it maintains two LRU lists, one with pages accessed only once, and the other with pages accesses more than once. When a page needs to be evicted, LRU2Q considers pages on the accessed-once LRU list first.

We implemented another two page replacement policies: (1) Adaptive Replacement Cache (`ARC`) [35] and (2) Clock with Adaptive Replacement and Temporal filtering (`CART`) [20].

Besides the two LRU lists used in LRU2Q, ARC or CART uses two more lists (called *nonresident lists*) to memorize the pages that are reclaimed from the memory recently. When a new page is read into the memory, ARC or CART check whether the page is memorized in the nonresident lists and puts the page into appropriate LRU lists. For instance, if the nonresident lists show that the page was on the accessed-more-than-once LRU list, ARC or CART adds the page directly to this LRU list, bypassing the accessed-once list.

According to previous researches [20, 35], if we do not consider the overhead of maintaining the nonresident lists, both ARC and CART always perform better than the traditional LRU page replacement policy theoretically. How-

```
const struct pr_policy arc_pr_policy = {
/* Initialize the structures. */
  .init_lruvec = init_lruvec_arc,

/* Decide which pages to reclaim and do the reclaiming. */
  .get_scan_count = get_scan_count_arc,
  .shrink_lruvec = shrink_lruvec_arc,
  .balance_lruvec = balance_lruvec_arc,

  .should_continue_reclaim = should_continue_reclaim_arc,
  .too_many_isolated_compaction=too_many_isolated_compaction_arc,

/* Capture activity and statistics */
  .page_accessed = page_accessed_arc,
  .activate_page = activate_page_arc,
  .deactivate_page = deactivate_page_arc,
  .update_pr_statistics = update_pr_statistics_arc,

/* Add/release pages from the LRU lists. */
  .add_page = add_page_arc,
  .release_page = release_page_arc,

/* Helpers used for specific scenarios. */
  .rotate_inactive_page = rotate_inactive_page_arc,
  .reset_zone_vmstat = reset_zone_vmstat_arc,

/*For nonresident lists*/
  .nonres_remember = nonres_remember_arc,
  .nonres_forget = nonres_forget_arc,
  .isolate = isolate_arc,
  .putback_page = putback_page_arc,
};
```

Figure 6: Functions implemented in ARC page replacement policy.

ever, in practice, because the extra overhead of maintaining the nonresident lists is not negligible, ARC and CART can perform worse than LRU2Q (see §7.4 for details).

In order to support dynamic policy switch, we isolated the implementation of page replacement policies with the page replacement mechanism. Specifically, we modified and added a set of interfaces into the Linux kernel so that EOS can switch the page replacement policies at runtime. Figure 6 gives the functions we implemented in ARC page replacement policy. Kernel developers can easily develop their own page replacement policies following the same pattern in Figure 6.

To switch the page replacement policy, we implement `void pr_change(struct zone *zone, enum pr_policy_list old_p, enum pr_policy_list new_p)` that moves all the pages in the LRU lists of `old_p` to the LRU lists of `new_p`. After that, Linux kernel will use the new policy to guard the page replacement in memory.

In EOS, we set CART instead of LRU2Q as the default policy because we use the nonresident list hit ratio, which can only be collected in ARC and CART, as the workload sensor (see §7.4 for details). One limitation of this implementation is that once we switch the page replacement policy to LRU2Q, we cannot switch to other policies.

Subsystem	Spec LOC	Average Speedup
Disk IO scheduling	50	2.58 ×
CPU scheduling	36	3.20 ×
Synchronization	30	1.81 ×
Page replacement	16	1.24 ×

Table 1: *Summary of Results*. The policies in each subsystem require 16 – 50 lines of code to specify. EOS's performance improvements range from 1.24 to 2.58 times.

7 Evaluation

We focus our evaluation on the following three questions:

1. Is EOS easy to use?
2. Can EOS find parameter settings that significantly outperform than default?
3. Can EOS consistently benefit different kernel subsystems?

Each of the next four subsections presents a case study of applying EOS to one of the four main non-networking subsystems in the Linux kernel: disk I/O scheduling, CPU scheduling, synchronization, and page replacement subsystems. Before we diving into the detailed results for each subsystem, Table 1 summarizes EOS's performance. For every kernel subsystem evaluated, EOS improved the subsystem's performance by 1.24× to 3.20 × on average. In addition, to specify the policies, we only needed to write 16 to 50 lines of code within a few hours; developers of these subsystems can most likely spend less time and annotate the parameters better. These results show that EOS is easy to use and can effectively improve performance.

7.1 I/O Scheduling

Specifying policies. In the Linux I/O scheduling system, we annotated four parameters, previously reported to have large performance impact [16, 28]. The first parameter specifies which I/O scheduling policy to use for a disk. The version of Linux we used supports three policies: (1) completely fair queuing ("čfq") [1], (2) deadline-based scheduling("ďeadline") [2], and (3) first-come first-served("ňoop") [4]. To change this parameter (method šetter in struct p̌aram), we used Linux's function ělevator_change. The second parameter specifies the maximum number of requests on a disk queue. The third parameter specifies how many pages to read ahead. The last parameter specifies the time slice allocated to each process when the I/O scheduling policy is set to čfq. To identify workloads, we used five sensors: the number of the concurrent processes, the read-write ratio of the requests, the average I/O request size, the average seek distance between two consecutive disk requests, and the average time between two consecutive requests. These sensors capture typical workload characteristics that significantly affect performance. We set the optimization target to be the number of sectors read or written per second. In total, it took us roughly 2 hours and 50 lines of code to specify the policies in the I/O subsystem.

Experimental setup. We used an evaluation machine with Ubuntu 13.04 server edition, a quad-core 3.6 GHZ Xeon E5-1620 processor, 32 GB memory, and two 2 TB hard disks. We used seven popular I/O benchmarks and one 12-hour Wikipedia [14] I/O trace as the evaluation workloads. Specifically, the škip benchmark [39] is a micro-benchmark that starts multiple processes, each of which repeatedly reads 4 KB from a 2 GB file and skips the following 4KB. The v̌skip benchmark executes škip in a KVM virtual machine to evaluate EOS's performance in a virtual environment. The žmIO [33] benchmark is another micro-benchmark that starts multiple processes, each of which sequentially reads 64 KB from a raw disk using direct I/O. The TPC-B benchmark [10] measures the number of executed transactions per second for a database. We used PostgreSQL [5] as the database and runs TPC-B on another machine. The table under test was much larger than the total memory size on our evaluation machine. The TPC-H benchmark [12] is a decision support benchmark; we ran four processes issuing Q2 requests to a PostgreSQL database simultaneously. The TPC-C benchmark [11] evaluates OLTP applications by simulating an environment where multiple users submit transactions simultaneously into a database. The vTPC-C executes TPC-C in a KVM virtual machine. Parallel grep [27] is a macro-benchmarks that uses the GNU `grep` utility to look for a non-existent string in the Linux kernel 3.8.8. Eight processes are started simultaneously, but each process calls `grep` in its own Linux source code copy to reduce the possible kernel lock contentions. (We only show results of running škip and TPC-C in a virtual environment because the other workloads had no difference executing in a virtual environment or not.)

Results. Figure 7 shows the performance improvements of the benchmarks compared to the default setting

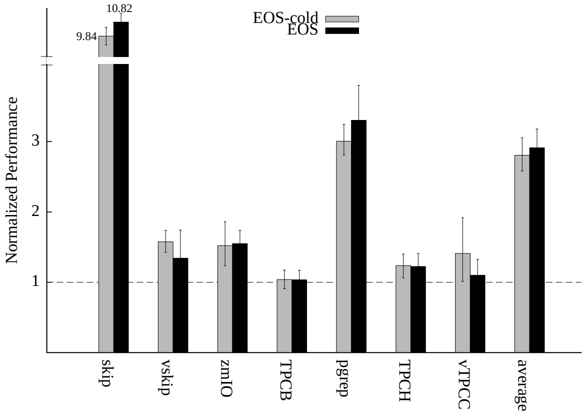


Figure 7: EOS's speedup of disk I/O scheduling subsystem over default setting. EOS-cold represents results with a cold policy cache, and EOS a warm cache. The last two columns show average speedup of all workloads.

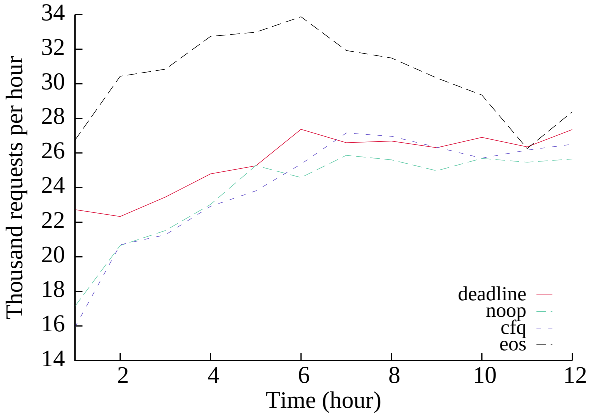


Figure 8: EOS's speedup of disk I/O scheduling subsystem on a 12-hour Wikipedia trace.

(deadline in the version of Ubuntu we used, three other parameters). With a cold cache, EOS achieves over 11× speedup on škip and 3× speedup on parallel ğrep. Its improvements on other applications are also quite significant, up to 41%. With a warm cache, EOS gains another 10.0% on average. Since we designed EOS to handle relatively steady, repeatable workloads, we expect that the warm-cache case is more often than cold-cache.

Figure 8 presents EOS's performance on a 12-hour Wikipedia trace. It outperforms the default policy by over 19.8% for almost the entire workload.

Figure 9 shows the parameter settings EOS selected for each workload. Note that EOS may select more than one settings over repeated executions when the settings have

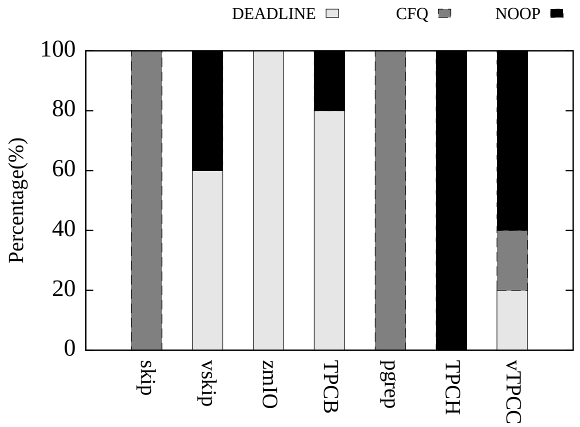


Figure 9: Percentage of time a disk I/O scheduling policy is selected by EOS.

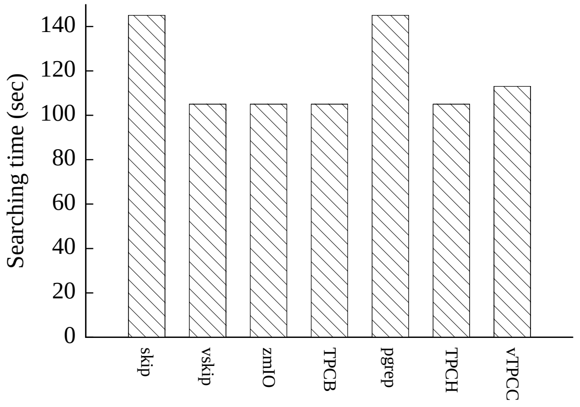


Figure 10: Search time for each disk I/O scheduling workload.

roughly the same performance. The figure also shows that there is no single scheduling policy that fits all workload.

Figure 10 shows the time it takes for EOS to find a good setting for each workload. This time ranges from 105 seconds to 145 seconds.

EOS uses workload signatures to search for settings in the policy cache, so we also studied the differences and similarities of the signatures of the evaluated workloads. Figure 11 plots the average seek distance vs the average time between two consecutive disk requests for the workloads over repeated executions. These two sensors suffice to place each workload in its own singleton cluster.

7.2 CPU Scheduling

Specifying Policies. In the CPU scheduling subsystem, we annotated three parameters. Specifically, šysctl_sched_latency_ns specifies the length of time for scheduling each runnable process once. šysctl_sched_min_granularity_ns specifies the minimum time a process is guaranteed to run when scheduled. šysctl_sched_wakeup_granularity_ns, it describes the ability of processes being waken up to preempt the current process. A larger value makes it more difficult preempt the current process [30]. To identify workloads, we used one sensor: the number of retired instructions executed in user space. We set the optimization target to be the same. Overall, it took us 2 hours and 36 lines of code to specify the policy.

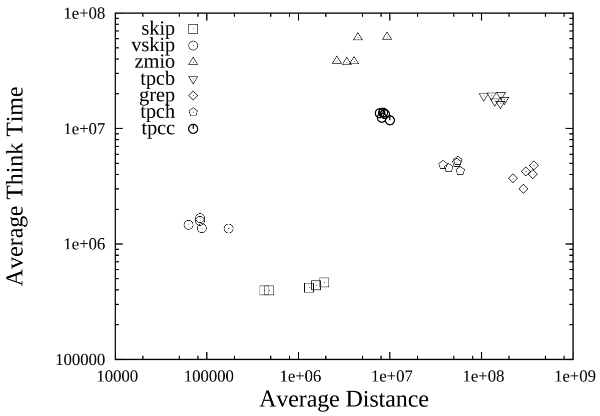


Figure 11: Workload clusters based on two sensors: average distance and average time between two consecutive I/O requests.

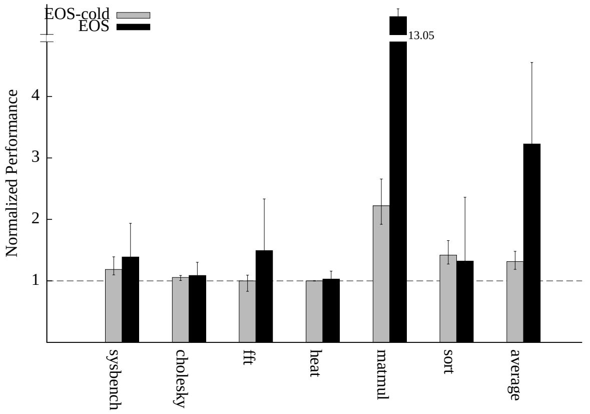


Figure 12: EOS's speedup of CPU scheduling subsystem over default setting.

Experimental Setup. The evaluation machine is the same as the one used to evaluate the disk I/O subsystem. To demonstrate the performance improvements of our system, we used the CPU scheduling benchmark in SysBench, a widely used systems benchmark [17] and five parallel computing benchmarks in the popular Cilk [9] benchmark suite. SysBench starts multiple threads, each of which repeatedly locks a mutex, yields the CPU, and unlocks the mutex. The Cilk benchmarks run basic algorithms such as merge sort and fast Fourier transformation in parallel.

Results. Figure 12 shows EOS's performance improvements compared to the default. With a cold cache, EOS worked well for most workloads and achieved more than 2× speedup on matmul. With a warm cache, it performed even better, achieving a 13× speedup on matmul. To better understand this huge speedup, we analyzed the executions of matmul. It turns out that, during the execution of matmul, sometimes many threads have no work to do and are busy trying to steal work from other threads without yielding the CPU. These futile work-stealing requests waste many CPU cycles. In this scenario, EOS correctly adjusted the CPU scheduling parameters to preempt threads more often so that the few threads with work to do can make good progress, significantly improving performance.

It took EOS 200 seconds to finish the search for each workload. (There is only one CPU scheduling policy in the version Linux we used, so EOS adjusted the parameters of this policy only.) We also studied the signatures of the workload. Based on the signatures, the workloads fall into three clusters: (1) SysBench, (2) fft, (3) heat, and (4) the rest of the workloads.

7.3 Synchronization

Specifying policies. In the synchronization subsystem, we annotated two parameters. `method_tuner` specifies which locking policy to use. `val_tuner` specifies the polling weight in the back-off ticket spin lock (C in Figure 5). v̌al_tuner is active only when m̌ethod_tuner is set to be back-off based ticket lock. To identify workloads, we used one sensor: the average lock acquisition time (the time between a lock is requested and the lock is granted). We used the number of lock acquisitions per second as the optimization target. In total it tooks us 2 hours and 30 lines of code to specify the policies in the lock subsystem.

Experimental Setup. We use an evaluation machine with Ubuntu 13.04 server edition, eight 48-core 1.9GHZ AMD Opteron processors. We used four benchmarks as workloads. Specifically, fops [23] is a micro-benchmark that measures the locking performance of the Linux directory entry cache. It starts multiple processes, each of which repeatedly opens and closes a private file. The mmapbench benchmark is a micro-benchmark written by us to measure the locking performance of the Linux memory map operation. It spawns multiple processes, each of which repeatedly maps the same continuous 500 MB in shared, read-only mode, reads the first byte, and destroys the mapping. Parallel postmark [27] is a macro-benchmark that simulates file server workloads hosting email and news services. It starts multiple threads, each of which runs file system operations repeatedly on an independent set of files (between 0.5 and 10K bytes in size). To measure locking instead I/O performance, we used the ťmpfs. dbench [7] is a popular macro-benchmark for benchmarking file systems. It starts multiple processes, each of which does many file operations such as read, write, link, and unlink. We also used ťmpfs for dbench.

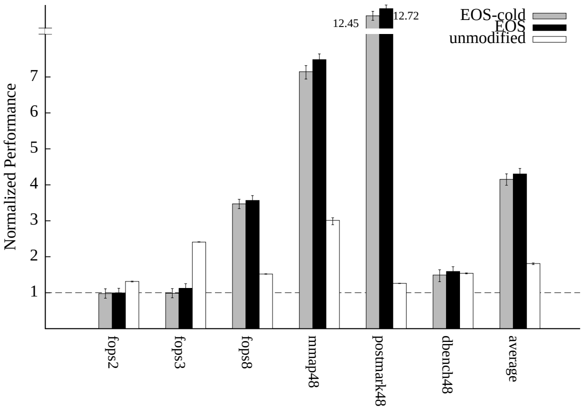


Figure 13: EOS's speedup of synchronization subsystem over default setting.EOS-cold represents results with a cold policy cache, EOS a warm cache, and unmodified the results of the unmodified Linux spin lock implementation.

Results. Figure 13 presents EOS's performance improvements over the default. To create different lock contention levels, we varied the number of processes or threads used by each benchmark. For example, fops8 means running fops with eight processes. With a cold cache, fps8, mmapbench, and parallel postmark performed much better with EOS than the default (up to 13×). With a warm cache, EOS performed even better. (Since we modified Linux's spin lock to support multiple locking policies, our code adds some overhead. Thus, Figure 13 shows the performance of Linux's unmodified spin lock for reference.)

Figure 14 shows the locking policy EOS selected for each workload. For every workload, EOS consistently selected the same policy for the workload over repeated executions. When the contention level is low (fops2), it selected TTAS. When the contention level is medium (fops3

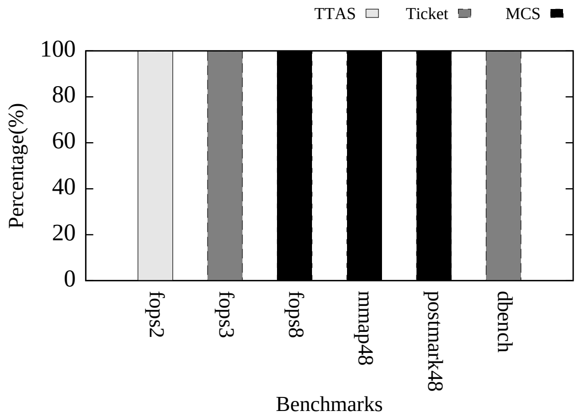


Figure 14: Percentage of time a lock policy is selected by EOS.

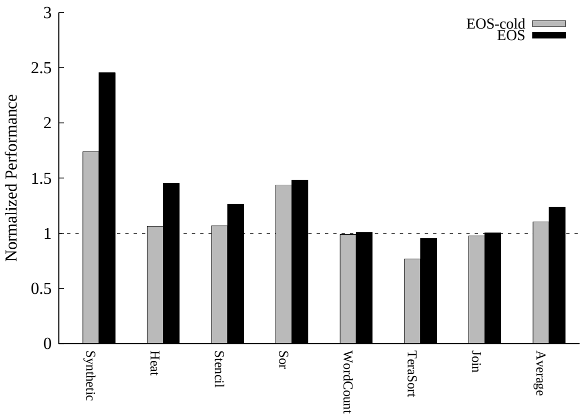


Figure 15: EOS's speedup of the page replacement subsystem over default setting.

and dbench), it selected the back-off based ticket lock. When the contention level is high (fops8, mmapbench, and parallel postmark), it selected MCS. The figure also shows that there is no single locking policy that fits all workload.

It took EOS 15 seconds to finish the search for fops2, fops8, mmapbench, and parallel postmark; and 75 seconds for fops3 and dbench. We also studied the signatures of the workloads. Based on the signatures, each workload falls into its own singleton cluster.

7.4 Page replacement

Specifying policies. In the Linux memory management system, we annotated one parameter that specifies which page replacement policy to use for memory. To identify workloads, we used one sensor: the hit ratio of the nonresident lists. This sensor characterizes a performance-critical memory access pattern of a workload. Specifically, when a hit occurs on a page on a nonresident list, the kernel can use access history stored in the list to effectively improve performance. We set the optimization target to be the number of page-in or page-out operations per second. In total, it took us roughly 2 hours and 16 lines of code to specify the policies in the memory management subsystem.

Experimental setup. We used an evaluation machine with Ubuntu 13.10 desktop edition, a quad-core 3.4 GHZ i7-2600 processor, 4 GB memory, and one 1 TB hard disk. We used one synthetic micro-benchmark and six memory-intensive benchmarks for big data area and scientific computing area. Specifically, WordCount and TeraSort are example programs in Hadoop package [6]. WordCount counts the number of different words appear in a 10GB file. TeraSort sorts a large number of numbers (overall size is 10GB) using merge sort. Join [48] is one of the popular big data benchmarks developed based on hive [46]. It does join operation on two large tables in a mysql database. Heat [24] simulates the heat distribution over time on a metal plate. Stencil [24] does the 9-point iterative stencil computing. Sor [24] is the successive over-relaxation algorithm. šynthetic is a micro-benchmark we developed. it requests 3.8GB memory space and writes random data to the space round-and-robin for 10 times. The data set of all the above benchmarks are larger than the memory size so that the page replacement happens frequently.

Results. Figure 15 shows the performance improvements of the benchmarks compared to the default setting. With a cold cache, EOS achieves 1.74× speedup on `synthetic`. With a warm cache, EOS gains another 72% on `synthetic`. On real world application `Sor`, EOS achieves 1.43× speedup with a cold cache and achieves 1.48× speedup with a warm cache. On average, EOS can speed up the applications by 1.24 times.

For every workload, EOS consistently selected the same policy for the workload over repeated executions. It selected CART for Heat, Stencil, Sor, and šynthetic, and the default LRU2Q for WordCount, Terasort, and Join. (ARC is not selected for any workload.) The figure also shows that there is no single page replacement policy that fits all workload.

It took EOS 15 seconds to finish the search for each workload. We also studied the signatures of the workload. Based on the signatures, the workloads fall into three clusters: (1) Heat, (2) Stencil and Sor, and (3) WordCount, TeraSort and Join.

8 Related Work

Massive bodies of work has been devoted to improving performance. However, to our knowledge, no prior work proposed the idea of a programming API for kernel developers to describe the policy parameters; nor did any prior work proposed the idea of a general policy cache for memoizing good parameter settings and reusing them on similar workloads. Below we compare EOS to the closely related work.

Performance auto-tuning. Self-adapting operating systems [44] aim to collect traces from real workloads and run simulations with the traces to adapt the operating system implementation, not just the parameters. While this goal is exciting, we are not aware of any implementation of such a system.

Many performance tuning algorithms have been proposed, some of which may be incorporated to EOS's search engine for searching a good parameter setting within a kernel subsystem. For example, IBM researchers use generic algorithms to search for the optimal parameter setting for the Anticipatory I/O scheduler and the Zaphod CPU scheduler [38]. Using several synthetic benchmarks, they improved performance by 9% on average. EOS improved performance much more potentially because it can (1) cache prior tuning results and (2) tune not only the specific settings of a policy, but also which policy to use.

Borrowing from control theory, feedback-based tuning algorithms iteratively adjusts parameter settings based on some form of feedback. Reactive lock [31] dynamically selects the correct locking protocol based on the lock contention level; our lock implementation is motivated by reactive lock and further generalizes it by adding a back-off based ticket lock for mid-level contention. Application heart beat [29] is one mechanism for application to provide custom feedback. It is shown to make Linux's CFS scheduler to deliver predictable performance and temperature [21]. Berkeley Tessellation kernel uses feedback control to allocate an optimal number of cores for an application [26]. Feedback-based algorithms tend to take many iterations, so EOS's policy cache can help them avoid costly tuning when an optimal setting already exists in the cache.

Another approach is model-based performance tuning where researchers construct a performance model for a real system, trains the model parameters with offline simulation, and applies the model online to select optimal parameters. However, model construct typically requires deep expertise, and is time consuming to build. One case study took over a year to optimize four parameters in Berkeley DB [45]. EOS's policy cache can be viewed as a weak, automatically constructed model that captures a partial mapping from workloads to good parameter settings.

Ad hoc parameter caching. Three prior systems touch upon the idea of caching good parameter settings for certain workloads, but in an ad hoc way. One system caches good intermediate "genes" to speed up genetic algorithms [37]. Linux caches congestion control window sizes based on IP addresses [43]. Another system caches the optimal number of virtual machines allocated for an online service based on hardware performance counters [47]. The caching in these systems is limited to a particular algorithm or parameter, whereas EOS supports generalized policy caching.

9 Conclusion

We have presented EOS, a system that bridges the knowledge gap between kernel developers and users by automatically evolving the policies and parameters *in vivo* on users' real, production workloads. It provides a simple *policy specification API* for kernel developers to programmatically describe how the policies and parameters should be tuned, a *policy cache* to make in-vivo tuning easy and fast by memozing good parameter settings for past workloads, and a *hierarchical search framework* to effectively search the parameter space. Evaluation of EOS on four main Linux subsystems shows that (1) it is easy to use, requiring 16 to 50 lines of code to specify the policies in each subsystem, and (2) it effectively improves each subsystem's performance by an average of 1.24 to 2.58 times and sometimes over 13 times.

References

[1] CFQ IO scheduler. access date: 1 May, 2014. http://en.wikipedia.org/wiki/CFQ.

[2] DEADLINE IO scheduler. access date: 1 May, 2014. http://en.wikipedia.org/wiki/Deadline_scheduler

[3] kernbench. access date: 1 May, 2014. http://ck.kolivas.org/apps/kernbench/kernbench-

[4] NOOP IO scheduler. access date: 1 May, 2014. http://en.wikipedia.org/wiki/Noop_scheduler.

[5] PostgreSQL. access date: 1 May, 2014. http://www.postgresql.org/.

[6] Hadoop project. access date: 10 Apr, 2014. http://hadoop.apache.org.

[7] DBENCH. access date: 12 Dec, 2013. http://dbench.samba.org/.

[8] MSR Cambridge Traces. access date: 13 Dec, 2013. http://iotta.snia.org/tracetypes/3.

[9] The Cilk Project. access date: 14 Mar, 2014. http://supertech.csail.mit.edu/cilk/.

[10] TPC-B. access date: 15 Jan, 2014. http://www.tpc.org/tpcb/.

[11] TPC-C. access date: 15 Jan, 2014. http://www.tpc.org/tpcc/.

[12] TPC-H. access date: 15 Jan, 2014. http://http://www.tpc.org/tpch/.

[13] WikiBench. access date: 21 Apr, 2014. http://www.wikibench.eu/.

[14] Wikipedia. access date: 30 Apr, 2014. https://www.wikipedia.org/.

[15] IOTTA repository. access date: 30 Dec, 2013. http://iotta.snia.org/.

[16] Best Practice: If You Cannot Use the Deadline I/O Scheduler, Configure the CFQ I/O Scheduler. access date: 31 Oct, 2013. http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=%2Fliaat%2Fliaatbpbloc

[17] sysbench-0.4.8. access date: 31 Oct, 2013. http://sourceforge.net/projects/sysbench/.

[18] Automatically Tuned Linear Algebra Software (ATLAS). In *http://math-atlas.sourceforge.net/*, Access Date: October 14, 2013.

[19] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of SOSP*, pages 90–105. ACM, 2003.

[20] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *USENIX Conference on File and Storage Technologies*, volume 4, pages 187–200, 2004.

[21] D. B. Bartolini, R. Cattaneo, G. C. Durelli, M. Maggio, M. D. Santambrogio, and F. Sironi. The autonomic operating system research project: achievements and future directions. In *Proceedings of 50th ACM DAC*, 2013.

[22] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[23] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. 2012.

[24] Q. Chen, M. Guo, and H. Guan. LAWS: Locality-aware work-stealing for multi-socket multi-core architectures. In *the 28th International Conference on Supercomputing (to appear)*. ACM, 2014.

[25] E. Ciliendo, T. Kunimasa, and B. Braswell. Linux performance and tuning guidelines. In *IBM white paper*.

[26] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moret, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanovic, and J. D. Kubiatowicz. Tessellation: Refactoring the os around explicit resource containers with continuous adaptation. In *Proceedings of 50th ACM DAC*, 2013.

[27] Y. Cui, Y. Wang, Y. Chen, and Y. Shi. Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems. In *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[28] D. A. Heger and S. L. Pratt. Workload dependent performance evaluation of the linux 2.6 i/o schedulers. In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2004.

[29] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *The 7th IEEE/ACM International Conference on Autonomic Computing and Communications (ICAC)*, pages 79–88, 2010.

[30] J. Kobus and R. Szklarski. Completely fair scheduler and its tuning. In *draft on Internet*.

[31] B.-H. Lim. Reactive synchronization algorithms for multiprocessors. In *Ph.D. thesis of MIT*, February 1995.

[32] M. Loukides and G.-P. D. Musumeci. *System performance tuning*. O'Reilly, 1990.

[33] S. A. M. Marazkis and A. Bilas. Spamd: Stressing i/o in modern servers. In *Proceedings of MASCOTS 2012*. IEEE, 2012.

[34] R. McDougall, J. Mauro, and B. Gregg. Solaris performance and tools: Dtrace and mdb techniques for solaris 10 and opensolaris.

[35] N. Megiddo and D. S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[36] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 269–278. ACM, 1991.

[37] J. Moilanen. I/o workload fingerprinting in the genetic-library. In *Proceedings of the Linux Symposium*, pages 1–7, Ottawa, Canada.

[38] J. Moilanen and P. Williams. Using genetic algorithms to autonomically tune the kernel. In *Proceedings of the Linux Symposium*, pages 327–338, Ottawa, Canada, 2005.

[39] T. C. R. Nou and J. Giralt. Automatic i/o of the io scheduler scheduler selection through online workload analysis. In *9th International Conference on UIC/ATC*, pages 431–438. IEEE, 2012.

[40] A. Puri, H.-M. Hang, and D. Schilling. An efficient block-matching algorithm for motion-compensated coding. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'87.*, volume 12, pages 1063–1066. IEEE, 1987.

[41] L. Rudolph and Z. Segall. *Dynamic decentralized cache schemes for MIMD parallel processors*, volume 12. ACM, 1984.

[42] B. samadi. Tunex: A knowledge-based system for performance tuning of the unix operating system. In *IEEE Transactions on Software Engineering*, 1989.

[43] P. Sarolahti and A. Kuznetsov. Congestion control in linux tcp. In *USENIX Annual Technical Conference, FREENIX Track*, pages 49–62, 2002.

[44] M. I. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the Sixth Workshop on Hot Topics on Operating Systems (HotOS)*, 1997.

[45] D. G. Sullivan. Using probabilistic reasoning to automate software tuning. In *Ph.D. thesis in Harvard University*, 2003.

[46] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[47] N. Vasi, D. Novakovi, S. Miuin, D. Kosti, and R. Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 423–436, 2012.

[48] T. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zhen, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: a big data benchmark suite from internet services. In *The 20th IEEE International Symposium On High Performance Computer Architecture (HPCA-2014)*. ACM, 2014.