WARP: Enabling Fast CPU Scheduler Development and Evaluation

Haoqiang Zheng[1,2] and Jason Nieh[1]
[1]Columbia University and [2]VMware, Inc.

***Abstract*—Developing CPU scheduling algorithms and understanding their impact in practice can be difficult and time consuming due to the need to modify and test operating system kernel code and measure the resulting performance on a consistent workload of real applications. To address this problem, we have developed WARP, a trace-driven virtualized scheduler execution environment that can dramatically simplify and speed the development of CPU schedulers. WARP is easy to use as it can run unmodified kernel scheduling code and can be used with standard user-space debugging and performance monitoring tools. It accomplishes this by virtualizing operating system and hardware events to decouple kernel scheduling code from its native operating system and hardware environment. A simple kernel tracing toolkit can be used with WARP to capture traces of all CPU scheduling related events from a real system. WARP can then replay these traces in its virtualized environment with the same timing characteristics as in the real system. Traces can be used with different schedulers to provide accurate comparisons of scheduling performance for a given application workload. We have implemented a WARP Linux prototype. Our results show that WARP can use application traces captured from its toolkit to accurately reflect the scheduling behavior of the real Linux operating system. Furthermore, testing scheduler behavior using WARP with application traces can be two orders of magnitude faster than running the applications using Linux.**

I. INTRODUCTION

CPU schedulers are an important part of every operating system and the choice of scheduling algorithm can have a substantial impact on the overall performance of the system. However, developing CPU scheduling algorithms and understanding their impact in practice can be a difficult and time consuming process. This problem is exacerbated by current trends toward multi-core systems that require better CPU scheduling to make effective use of CPU resources. Because CPU schedulers are part of the operating system, most of their development is done in the operating system kernel, where progress can be slow, debugging is difficult, and simple mistakes can crash the system.

Progress is slow in large part due to the inherently long testing cycle with kernel scheduler development. Any small change to the scheduler requires recompiling a complex operating system kernel and rebooting the system with the updated kernel. Because the scheduler is such a crucial part of the operating system, developers are often initially faced with the daunting task of just getting the operating system to boot up correctly with a new scheduler implementation. Even once the kernel is up and running, developers must often run a given application workload many times to identify unexpected errors that prevent the system from providing correct scheduling behavior. Repeatability of experiments is a primary goal, so that errors can be quickly corrected and the impact of scheduler changes can be determined for a given workload. However, many realistic applications are difficult to run in an easily repeatable manner to measure fine-grain scheduling behavior. Furthermore, running realistic application workloads on various schedulers is a very time consuming process that also requires root access to dedicated hardware resources to obtain accurate results that quantify the behavior of a real system.

Two approaches are commonly used in practice to reduce the difficulties inherent in CPU scheduler development and evaluation. One approach is to use a virtual machine monitor (VMM) such as VMware [1] to run test schedulers and their associated operating systems in virtual machines. While this can aid in the general process of debugging kernel code, it does not help isolate problems due to scheduler implementation errors since the developer is still faced with the need to work with an entire operating system kernel in the virtual machine. Furthermore, it does not reduce the time required for running extensive scheduler tests with application workloads, but in fact may increase the time due to added virtualization overhead.

Another approach is to use scheduler simulation studies[2], [3] instead of a real kernel implementation. While this can reduce the time required for testing scheduling algorithms using simulated workloads, it requires developing a scheduler simulator and scheduling code for the simulator. This code is almost certainly not reusable in a real kernel scheduler implementation, which means that the developer must incur the added costs of developing two scheduler implementation, one for the simulator and another for the operating system kernel. The simulator may also not provide a process model that matches what is typically used in the kernel, which can result in erroneous assumptions when trying to develop a scheduler that works well in practice. Since simulators typically cannot run real application workloads, they also do not provide any opportunity to consider the impact of scheduler design on real application workloads, which can differ substantially from simplistic simulation workloads.

To simplify and speed the development of CPU scheduling techniques, we have developed WARP. WARP is a user space CPU scheduler development and evaluation platform designed to combine the realism of kernel-based development and the speed of simulator-based approaches while avoiding the weaknesses of either approach. WARP is designed based on a key observation: for most benchmarks used to evaluate a CPU scheduler, the CPU scheduler is only involved a fraction of the time. For example, when running a CPU-bound application like gzip, the time spent making scheduling decisions is 0.01% of the time. The remaining execution is generally of no interest to the CPU scheduler and could be skipped as long as the execution timing characteristics are preserved. WARP speeds CPU scheduling testing and evaluation by warping system time such that all CPU scheduler related executions are performed exactly the same as real systems while all executions unrelated to the CPU scheduler are skipped.

WARP consists of two components: a trace-driven virtual scheduler execution environment and a kernel scheduling event tracing toolkit. WARP encapsulates kernel scheduling code in an environment that provides a simple process model and virtualizes operating system and hardware components that can affect CPU scheduling decisions. This virtualization decouples kernel scheduling code execution from the rest of the operating system so that scheduling code can be run without running other parts of the operating system. WARP thereby isolates scheduler execution and behavior from the rest of the operating system, reducing the difficulty of testing and debugging scheduler code.

WARP virtualization uses a scheduler para-virtualization technique to work with existing unmodified kernel scheduling code with only minimal changes to a few header files. New schedulers that are developed using WARP can be easily run in the context of a real operating system. Furthermore, WARP runs kernel scheduling code entirely in user-space and can be used with standard user-space debugging and performance monitoring tools, avoiding the difficulties of developing schedulers in the kernel and the need for privileged access to a machine during development. Kernel schedulers can then be developed and debugged in user-space and then directly inserted into an operating system kernel to run on production systems.

WARP combines its virtual execution environment with a tracing toolkit that gathers process traces by logging timestamped process state transitions and other CPU scheduling related events on real systems. WARP can then replay these traces in its virtual environment with the same timing characteristics as in the real system. Once the traces have been captured, they can be replayed over and over again to provide repeatable, realistic application workloads. The traces can be used with different schedulers to compare their performance and behavior. They can also be used to help tune a given scheduling algorithm without the complexity of having to run the actual applications repeatedly in a precise manner. Furthermore, WARP processes its traces at a rate proportional to the number of scheduling events and completely independent of application execution time. As a result, WARP can provide repeatable measurements of scheduling behavior that can be done in a fraction of the time of executing the same application workloads in a full-fledged operating system environment.

We have implemented a WARP Linux prototype to demonstrate its effectiveness. We show that WARP can be used across different major versions of the Linux kernel with multiple kernel scheduler implementations. Using a range of application workloads running on a Linux scheduler, we show that WARP can be used to capture scheduling event traces on a real system and replay those events in its virtual execution environment with the same timing characteristics. Our results show that WARP can replay those traces with different scheduler implementations in a manner that accurately reflects the behavior of the real Linux operating system. Furthermore, testing scheduler behavior using WARP with application traces can be two orders of magnitude faster than testing with by running the applications using Linux. Our experimental results demonstrate that WARP provides a useful tool for enabling the study and development of CPU scheduling algorithms, and understanding the impact of scheduling techniques on real applications in practice.

This paper describes the design, implementation, and evaluation of WARP. Section II discusses related work. Section III describes the WARP design, including its scheduler-independent virtual process model, kernel tracing toolkit, CPU scheduler para-virtualization technique, and virtual execution environment. Section IV discusses the implementation of WARP for use with the Linux operating system. Section V presents experimental results demonstrating the speed and accuracy of the WARP virtual execution environment for reflecting real kernel scheduling behavior. Finally, we present some concluding remarks and directions for future work.

II. Related Work

Virtual machines [1], [4], [5], [6] allow users to run guest operating systems and real applications in user mode. Virtual machines can simplify kernel development by enabling testing to be done in a virtual machine without compromising the hardware. However, they still require working with the complexity of an entire operating system and impose additional performance overhead on application execution. WARP takes a different approach by only running operating system components related to CPU schedulers. It does so entirely in user space. The result is that WARP can provide simpler, faster CPU scheduler development and evaluation.

Various simulators have been developed. General-purpose system simulation was first introduced in the Gordon Simulator [7]. General-purpose system simulation requires a description of the environment to be simulated, which is difficult to provide for today's complex computer systems. Whole machine simulation [8] is useful for providing detailed architectural measures of whole machine performance, but shares all the disadvantages of using virtual machines for CPU scheduler development with the additional drawback of being very slow.

General-purpose discrete event task simulators have been developed [3] which can be used for simulating schedulers with different workloads, including using different task arrival rates, execution times, I/O distributions, and available parallelism. Such simulators require developing a scheduler specifically for the simulator. Developing the real CPU kernel scheduler requires another step, and that kernel scheduler cannot be tested using the simulator. Moreover, the simulator cannot be effectively used to evaluate the impact of scheduler design on realistic application workloads.

Trace-driven simulation has been widely used [9], [2], [10] to study resource management. Advantages and disadvantages of trace-driven models have been previous discussed [11]. Previous

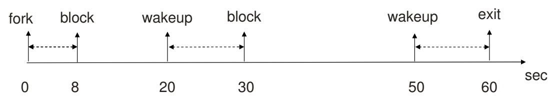

Fig. 1: Life Cycle of a Simple Workload

trace-driven approaches have not focused on low-level CPU scheduler behavior. As a result, they are limited in completeness and detail with respect to being used for modeling detailed kernel-level CPU scheduler behavior. They also cannot be used with unmodified CPU schedulers from real operating systems.

Much work has been done in studying process models and the characteristics of various workload distributions [12], [13], [14], [15], [16], [17]. However, researchers disagree on which model is best for describing real workloads. For example, one study, based on measurements of the lifetimes of 9.5 million UNIX processes, concluded the probability of a process's lifetime exceeding T seconds is rT^k, where $-1.25 < k < -1.05$ and r normalizes the distribution. In contrast, another study claims that long processes have exponential service times [13]. Since WARP uses a generic scheduling event based workload model, it supports workloads of any distribution since pseudo traces following any characteristics can be manually created and thus simulated.

III. Design

WARP is designed with the goal of being able to run kernel scheduling code to determine scheduler behavior as quickly as possible. However, running a kernel scheduler today entails also running applications that need to be scheduled and running other operating system code necessary to support application and scheduler execution. Most of the time then gets spent on executing application code and non-scheduler kernel code. This makes sense for running applications since a system would be very inefficient if it spent most of the time running scheduling code. However, this means running real applications in a real operating system is a very inefficient way of testing and measuring a scheduler since most of the time is spent on activities unrelated to scheduling.

Consider the simple example shown in Figure 1, with the process execution being described using UNIX-style semantics. The process is created via fork at time 0 s and finishes executing at time 60 s. The process blocks at time 8 s and 30 s, and wakes up at time 20 s and 50 s. Process creation, blocking, wake up, and exit are all events that can require the scheduler to take action to decide what to run. However, while the process takes 60 s to complete, the scheduler related code ends up only being called six times, ignoring timer interrupts for now. On a modern computer, six calls to CPU scheduler functions typically take only a few microseconds to run. As a result, during almost the entire 60 s lifetime of the process, the system is either running the process in user space, serving its non-scheduler related systems calls, or sitting in the idle loop. While real workloads used to evaluate a scheduler may be much more complex, the CPU time spent on executing the scheduler code is usually only a small fraction of the total execution time.

WARP speeds scheduler execution by recognizing that in general process execution between scheduling events only affects CPU scheduling decisions based on the actual execution time the process runs. Many schedulers, such as those that allocate CPU cycles fairly among processes, use the amount of time that processes have run to determine which process should be scheduled to run next. WARP introduces a process model in which a process that is scheduled to run can tell WARP immediately how much CPU time it will use until the next scheduling event. A process that blocks can also tell WARP immediately how long it will remain unrunnable. As a result, WARP is able to always know what the next scheduling event will be and when it will occur. After processing a scheduling event, WARP advances the system clock to the time at which the next scheduling event takes places and processes the next scheduling event.

Using this process model, WARP does not need to execute application code associated with processes since the execution has no effect on the scheduler behavior once WARP knows when and what scheduling events are going to happen. In a sense, WARP works like a time machine for schedulers, warping the system time to the next time a scheduling event occurs and thus skipping all process execution unrelated to scheduling. WARP therefore can effectively test and measure scheduling behavior for a process workload much faster than a real operating system since the time spent on scheduler code is usually a tiny fraction of the total real workload execution time.

A. Tracing Toolkit and Process Model

To determine how long a set of processes will run and block for a given application workload, WARP provides a kernel scheduling event tracing toolkit to log all scheduling events that occur when running the real application workload on the real operating system. To minimizing tracing overhead, events are stored in a memory log while an application workload is being traced. The event trace log is then stored to a file for future use. After the application workload has been run once, WARP has a complete log of when and what scheduling events occur. Since the trace provides complete process timing information, WARP can then replay the event trace time warping among scheduling events to model the application workload to the scheduler. WARP only needs to process the scheduling events and can ignore the time spent in application execution. WARP uses the event trace rather than having to repeatedly run the application workload over and over again to analyze scheduling behavior across different schedulers for the given workload.

WARP uses a simple process state model which reduces a process's lifetime down to a series of time intervals in which the respective process is either runnable or not runnable. To capture the necessary information for this model, the WARP tracing toolkit can be used to instrument an operating system kernel to log three categories of scheduling events: (1) when a running process voluntarily gives up the CPU, such as when a process blocks, yields, or exits, (2) when a process becomes runnable, such as when a process is created via fork or a process wakes up, and (3) when the scheduling parameters of a process

```
eid FORK   pid  length cpid
eid INIT   pid  0      cpid init_state
eid PRIO   pid  length sched_param
eid YIELD  pid  length
eid BLOCK  pid  length
eid WAKEUP pid  length deid dpid distance
eid EXIT   pid  length
```

Fig. 2: Format of the Events

change, such as when the UNIX `nice` system call is used to adjust the priority of a process or when the scheduling policy used for a process changes. These three categories of events account for all events external to a scheduler that can result in process state changes that a scheduler would need to be aware of in making scheduling decisions. Note that WARP does not track when a running process involuntarily gives up the CPU as in the case of a process being preempted by a timer interrupt when its time quantum expires. This event depends on the internal scheduling algorithm used and will change depending on the scheduler used. WARP creates event traces that are independent of the scheduling algorithm used so that they can be used with different schedulers.

Figure 2 shows the seven specific events logged by the tracing toolkit and the format of each event. All events have four common fields: `eid`, `type`, `pid`, and `length`. `eid` specifies the event identifier; each event has a unique identifer. `type` specifies the type of the event. Some types of events may have additional event-specific fields. `pid` specifies the process identifier of the process that generated the event. `length` specifies the timing of the event.

`type` specifies one of seven event types: BLOCK, YIELD, EXIT, FORK, INIT, WAKEUP, and PRIO. BLOCK, YIELD, and EXIT are self-explanatory events that correspond to a running process voluntarily giving up the CPU. FORK, INIT, and WAKEUP are events that correspond to a process becoming runnable. In particular, FORK indicates when a parent process creates a child process, with the child process inheriting its scheduling parameters from the parent. INIT indicates a special form of process creation which unlike FORK explicitly specifies the initial scheduling state of the created process. INIT events are usually used to represent processes that already exist before event tracing is started. PRIO is the event that corresponds to a process having its scheduling parameters change.

FORK, INIT, WAKEUP, and PRIO have additional event-specific fields. FORK and INIT have an additional `cpid` field which specifies the pid of the newly created child process. The `pid` and `cpid` fields are used by WARP to re-establish the child-parent relationship of processes in an event trace. INIT also has an additional `init_state` field which specifies the initial scheduling state of the created process. WAKEUP has three additional fields: an event identifier `deid`, a process identifier `dpid`, and a time `distance`. If `deid` is 0, then this event is an external WAKEUP event and the process is supposed to sleep for `length` ms before it wakes up. Otherwise, this event depends on the event `deid` of process `dpid` and should wake up `distance` ms before event `deid` is handled. PRIO has a `sched_param` field which specifies the scheduling parameters that are being changed, such as the nice value that would be specified in a `nice` system call.

The `length` field specifies the timing of different events, but WARP must be careful in determining how event time should be logged in the trace. For example, suppose during a certain execution, a process P_1 wakes up at time 0 ms and blocks at time 10 ms. To log the event times, one approach would be to use the relative wall clock time since the last scheduling event to time an event. In this case, the time of the BLOCK event will be specified as 10 ms relative to the previous event, which is the WAKEUP event. However, this relative time is an artifact of the scheduler. Using a different scheduler might result in a different ordering of events such that another process runs between when P_1 wakes up and blocks, resulting in different relative timing of the two events. Another approach would be to use the absolute wall clock time 0 ms and 10 ms as the event times for the two scheduling events generated by P_1. However, the wall clock time still does not represent the true behavior of the process but is an artifact of the scheduler used. If the workload is run under a different system with a different scheduler, the wall clock time the two scheduling events will also be different.

WARP uses a hybrid approach to properly specify event times depending on the type of event. Events such as FORK, BLOCK, PRIORITY, YIELD, and EXIT occur when the given process is running. For these events, WARP uses the total CPU time received by the process since the last WAKEUP event of the process. Since the CPU time needed between two scheduling events is decided by the code executed by the process and is relatively independent of the system load the scheduler used, using the relative CPU time as the event time more accurately represents the actual behavior of a process. A WAKEUP event is the one event that occurs when the given process is not running. Since WAKEUP therefore does not depend on the CPU time accumulated by a process, WARP uses the wall clock time since the last time the process blocked.

WARP implicitly assumes that the wall clock time between when a process blocks and wakes up is independent of when it is scheduled. While this assumption often holds, there are circumstances in which the time depends on the order in which processes are scheduled. For example, if multiple processes block performing disk I/O, the order of disk I/O requests may change depending on when processes are scheduled to run, resulting in the disk scheduler processing I/O requests in different ways. The amount of time a process blocks may then change due to these other resource dependencies. Since WARP focuses exclusively on CPU scheduling, it does not account for such other resource dependencies which may affect process behavior.

While the WARP kernel tracing toolkit is designed to produce event trace files directly from running application workloads, event trace files can also be created manually and do not need to be based on any real workload since WARP does not care about how the trace files are created as long as the file format is correct. Manually created trace files can be quite useful. One problem that often arises when evaluating CPU schedulers is that it can be hard to find or develop a real application benchmark with the desired workload characteristics. As a result, CPU schedulers are usually only evaluated using a small set of benchmarks. Since these benchmarks only represent a very limited set of all possible workloads, unidentified problems can remain in a scheduler even if the scheduler provided good performance as measured by these benchmarks. Using simple scripts, scheduling event traces can be easily created with varying process arrival rates, execution times, and interactiveness following a broader range of well-studied workload distributions [14], [12], [13]. These resulting event traces can be used by WARP to study scheduling behaviors more comprehensively.

WARP event traces are designed so that they can be gathered from different real workload executions and yet mixed together. WARP requires all event traces to be executed to be identified in a standard format workload specification file (WSF). The WSF format is similar to the `inittab` of a Linux system in that each entry in a WSF specifies an event trace. WARP creates a trace object for each event trace in the WSF. WARP identifies an event by both its event identifier and the associated trace object. An event from one trace will not interfere with an event from another trace even if they have the same event identifier. The ability to mix workloads by executing their respective event traces together adds more flexibility to the scheduler evaluation process. Instead of having to gather the traces of all the possible mixes of the applications of interest, traces can be gathered for each application separately and then mixed in any desired fashion in WARP.

B. WARP Virtualization

WARP runs and executes virtual processes using event traces rather than running real processes. As a result, WARP does not need to provide the full functionality of an operating system to support real application execution. While WARP does not need to support direct application code execution given its trace-driven execution model, it does need to run kernel scheduler code. However, a real CPU scheduler is not a standalone program. It is part of the operating system kernel and is designed to run in kernel mode as part of an entire operating system. The process model used by typical operating systems also does not permit the kind of time warping that WARP uses. Nevertheless, in most operating systems, there is a clear delineation between kernel code related to scheduling and other operating system code, with scheduling code typically being localized to a few source files. Other operating system functions used by schedulers are often localized to a set of header files separate from core scheduler code to provide appropriate levels of abstraction. This functional separation reflects clean modular kernel design.

WARP takes advantage of its virtual process model and the functional separation of scheduling code to provide a virtualized execution environment that decouples the scheduler from the rest of the kernel and enables it to be executed in user space as a separate entity from the operating system. WARP only needs to provide a virtualized environment sufficient for running the scheduler. Many parts of the operating system such as file system, network, and device functionality that are needed to execute real processes can simply be ignored. The remaining parts of the operating system that schedulers do depend on are virtualized by WARP in a manner transparent to the CPU scheduler so that the scheduler does not know that it is running in WARP instead of a real operating system. The result is that WARP does not need to run non-scheduler kernel code. WARP virtualizes both operating system functions and hardware events that can affect CPU scheduling decisions. We discuss WARP virtualization in the context of system clocks, timer interrupts, CPUs, virtual memory, and I/O virtualization.

WARP virtualizes the system clock and its related functions to provide a different notion of time from the underlying system and allow warping of the perceived time seen by the scheduler to time travel instantaneously between scheduling events. Hardware clocks are often used to provide timing information for the CPU scheduler and other parts of the kernel. For example, in a Pentium-based system, timing information can be read either from the system Real Time Clock (RTC) or from the CPU Time Stamp Counter (TSC) [18], the latter being a 64 bit register that counts CPU cycles and provides much higher resolution. While it is possible for a scheduler to access such timing hardware directly, operating systems typically provide higher-level functions for accessing such timing information for portability. To support its own notion of time, WARP virtualizes these timing-related functions that read the hardware clocks to instead read a virtual clock maintained by WARP. This allows WARP to control how time passes by how it chooses to update its virtual clock and decouples the scheduler's notion of time from the underlying system. WARP initializes its virtual clock to zero and updates it during time warp execution.

WARP also virtualizes the timer interrupt and its underlying APIC timers to enable time warp execution. A real system typically has a set of configurable APIC timers which can generate periodic timer interrupts. The timer interrupt causes the CPU scheduler timer event handler to be executed. This handler is an important part of any CPU scheduler since it usually causes decisions to be made regarding how much longer a process can execute before it is preempted. Since real system timer interrupts are not delivered to a scheduler running in WARP, WARP simulates the behavior of timer interrupts by calling the scheduler timer event handler periodically. WARP determines when to call the timer interrupt based on its own virtualized notion of time. By ensuring that the timer interrupt handler is called in the same fashion, WARP ensures that the scheduler's state is updated in a proper manner and that the resulting scheduling decisions are performed in the same way as a real system. Furthermore, since the scheduler does not know whether a real timer interrupt or WARP caused the timer interrupt handler to be invoked, WARP's timer virtualization is also transparent to the CPU scheduler.

WARP provides virtualized CPUs to the scheduler to enable WARP to choose the system configuration that is used to run the scheduler and to provide the necessary CPU abstraction in the absence of the physical hardware of a real system. In particular, a real operating system has an initialization phase in which CPUs are probed and the kernel variables representing the status of the CPUs are properly initialized. These initialization functions are not necessary for WARP since there is no real underlying hardware, only the virtualized hardware abstraction provided by WARP. WARP modifies the hardware probing code so that all the kernel objects and variables representing hardware configurations are initialized directly based on user specification. For example, WARP users can specify the number of CPUs to be simulated with the scheduler. In a Linux kernel, each bit of the variable `cpu_online_map` represents the online status of the corresponding CPU. WARP instead sets `cpu_online_map` based on the options provided by a user. Linux also uses `sched_domain` to represent the topology relationship of all the CPUs. WARP can initialize `sched_domain` based on user specification. For example, users can use WARP to simulate a system with 16 Intel Nehalem sockets (each socket has 4 cores sharing L3) just by initializing sched domains accordingly. Linux CPU schedulers typically read `cpu_online_map` and `sched_domain` when making load balancing decisions. The schedulers cannot differentiate whether these kernel objects are initialized by hardware probing or user specification. Thus, WARP can be used to simulate systems with any number of CPUs and any CPU topology. CPU scheduler developers can even do CPU topology related performance tunings using WARP by sanity checking each migration decision. For example, it is usually preferable to do migrations within the same last level cache (LLC), unless no CPU on the local LLC is idle. If any migration violates this policy, a WARP user can single step the execution that results in this migration decision to figure out what caused the problem. Since WARP runs in user space and WARP execution is always repeatable, finding such problems is relatively easy.

WARP also virtualizes other functions that directly access and manipulate CPU state. For example, the function used to context switch from one process to another typically involves low-level CPU state manipulation. WARP instead replaces such functionality with a nop function since no real hardware context switching is necessary with WARP virtual CPU hardware.

WARP provides a simple virtual CPU abstraction to handle basic scheduling functionality. By design, this abstraction does not fully represent all the complexities inherent in modern CPU architectures. This allows WARP to be simple to implement, easy to use, and fast. However, it also means that WARP currently does not account for more complex CPU behavior. For example, in a real system with multiple CPUs, minimizing how often a process is switched from one CPU to another can improve performance by taking advantage of cache affinity. WARP's design using simple virtual CPUs and is trace-driven virtual process model would not account for cache affinity effects in measuring scheduler performance.

WARP provides its own virtualized functions for memory management as it affects CPU schedulers. Most of the memory management parts of the operating system are not needed by WARP since CPU schedulers do not deal with memory management. However, CPU schedulers typically require basic memory management functionality, most notably kernel memory allocation and deallocation routines. Since WARP runs kernel scheduling code in user space, such kernel memory allocation/deallocation functions do not exist in the WARP environment. Instead, WARP virtualizes those functions by replacing them with their user space counterparts. For example, WARP redefines `kmalloc` as `malloc`, and `kfree` as as `free`.

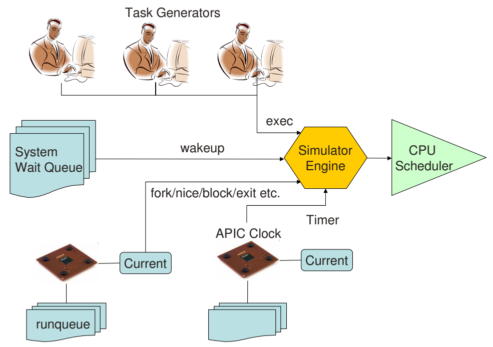

Fig. 3: Structure of WARP

WARP does not need to provide any real operating system I/O subsystem functionality since that part of the operating system is not directly used by CPU schedulers. However, WARP does need to deal with the fact that processes block waiting on I/O requests to be processed since those processes are often placed on various wait queues that CPU schedulers may also manipulate. In a real system, a process blocks when the resource requested is not available. When a process blocks, it is removed from the runqueue and placed on a wait queue of the requested resource. Once the resource becomes available, the kernel will typically be notified via an interrupt and the process will be woken from the wait queue and moved back to the runqueue. Even though WARP does not run application code that causes real processes to block, WARP still needs to account for when its virtual processes block. WARP implements a system-wide virtual wait queue to represent all the I/O subsystems. Whenever a process in WARP blocks, it is added to this virtual wait queue regardless of why it blocked. The virtual wait queue is simply a queue ordered based on the wakeup times of the processes. Since a CPU scheduler does not differentiate whether a process is woken from a real I/O wait queue or from WARP's virtualized wait queue, the I/O virtualization is also transparent.

C. Time Warp Execution

Given an event trace and a virtual execution environment for running kernel scheduler code, WARP can now proceed with the actual trace-driven execution of an application workload using a given scheduler. WARP traces scheduling events in the order in which they originally occur, but once they are written to trace files they are reordered in a format more amenable to scheduler execution. All INIT events, which correspond to the initial set of processes to be scheduled when the trace began, are grouped together at the beginning of the file. All remain events are ordered first by process identifier, then by time, thereby grouping together all events generated by a given process.

Figure 3 illustrates WARP execution model. WARP starts with an event trace file and creates virtual processes corresponding to all INIT events. WARP has a component called a task generator which is responsible for creating virtual processes corresponding to these INIT events. Once the initial processes are created, WARP then starts running the scheduler code to determine how the scheduling behavior of the systems should proceed. The primary task of WARP at any given point in time is to determine the next scheduling event that will occur. This event is not simply the next event in the original time ordered event trace since WARP can use a different scheduling algorithm than what was running on the system that generated the event trace. WARP determines which event is the next event by consider three possibilities.

First, the next event could be caused by a process currently running on a CPU. WARP can determine the time of this event by going to the part of the event trace file corresponding to the process identifier of the running process and finding the next scheduling event for that process. Since it is running, the next event for the process will be its scheduling parameters have changed, it forks another process, or it voluntarily gives up the CPU because it blocks, yields, or exits. This time will be specified in CPU time relative to when the process was created or last woken up. For comparison purposes, WARP converts this time to an absolute wall clock time under the assumption that the process will continue to run. When WARP is running with a multiprocessor hardware configuration, WARP determines the time of the next event for each process currently running on a CPU.

Second, the next event could be caused by a virtual timer interrupt occurring. WARP knows the virtual hardware configuration and can easily determine the time of the next timer interrupt based on when the last timer interrupt occurred and the periodicity of the timer interrupt as specified in the hardware configuration. When using a multiprocessor hardware configuration, WARP provides a separate timer for each CPU, resulting in each CPU having its own timer interrupts. These timer interrupts are not aligned as is the case in real systems.

Third, the next event could be caused by the first process in the system wait queue being woken up. The wake up time of a process is specified in the event trace in wall clock time relative to when the process blocked. WARP converts this time into an absolute wall clock time based on when the process was inserted in the system wait queue. Since the system wait queue is ordered by wake up time, WARP can easily determine the earliest wake up time of any process. WARP currently uses a single system wait queue across all CPUs regardless of the number of CPUs configured.

WARP compares the event times corresponding to running processes, timer interrupts, and blocked processes. It determines the event corresponding to the earliest of those event times as being the next scheduling event. WARP advances its system time to the time of the next scheduling event. Time warping the system time does not cause any problems in making scheduling decisions since any execution of application or operating system code that happened between scheduling events is irrelevant to the CPU scheduler. WARP then processes that event and the repeats this basic execution loop to continue to process subsequent scheduling events. WARP repeats event execution until all processes specified in the event trace have completed execution.

When a process is running, the maximum amount of time that WARP can advance the system clock per scheduling event is the time between timer interrupts. The execution efficiency WARP gains in this case is the difference between how long it takes WARP to perform its scheduling event processing versus the timer interrupt frequency. When no processes are running and CPUs are idle, WARP does not need to process each timer interrupt since there are no processes to schedule. In this case, WARP only needs to determine the earliest time that a process will wake up off the system wait queue. All timer interrupts until that time will result in no scheduling decisions. As a result, WARP can advance the system clock until the earliest time that a process will wake up off the system wait queue. The execution efficiency WARP gains in this case is proportional to the amount of idle time.

IV. WARP Linux Implementation and Usage

We have implemented a WARP prototype based on Linux. There are two separate components, a Linux kernel scheduling event tracing toolkit and the virtual execution environment. WARP's kernel scheduling event tracing toolkit is mostly implemented in a Linux loadable kernel module. The module provides simple event tracepoint functions that are inserted in various places in the Linux kernel scheduler to obtain the necessary information for WARP's process model. Once loaded, the toolkit captures all calls to scheduler-related functions, including fork, exec, exit, block, wakeup, change priority and yield. For each such scheduling event, the toolkit writes the necessary information about this event to a preallocated kernel buffer. The toolkit creates a kernel thread which is responsible for writing the contents of the buffer to a user space file. The kernel thread runs at a low priority and wakes up periodically. Its execution will not affect higher priority processes in the system. The tracing toolkit has very low overhead since logging a scheduling event is only a matter of a memory copy of a few dozens bytes. The toolkit also provides a set of scripts to further process the log file generated by the kernel loadable module. After processing, all the scheduling events belonging to the same task are stored together. This way, the trace file can be parsed by WARP more efficiently.

WARP's virtual environment is implemented to work with existing unmodified Linux scheduler code. It includes a patch to a small number of header files in the existing Linux kernel source code. Once applied, the patch will update some Linux header files and add a few new files necessary for executing WARP. Among the thousands files in the Linux source code tree, the WARP implementation made only minor modifications to about twenty files and all the other files, including the core scheduler functionality in `sched.h` and `sched.c`, are used directly without any modification. Setting up the WARP virtual execution environment for use with a Linux kernel is very simple. WARP provides a shell script that can do the patching automatically on the specified Linus source code tree. Once the setup script finishes patching, a WARP executable program can be created by a simple “make”. WARP has been tested across both Linux 2.4 and 2.6 kernel versions and works seamlessly with these systems without any problems.

Once WARP is built and the event traces are collected, it can be used to develop, debug and evaluate CPU schedulers. Our WARP prototype takes three mandatory command line parameters: the number of CPUs of the system to be simulated, how long in WARP virtual time should it run, and a workload description file that contains the event traces.

Consider a sample scenario to further illustrate how WARP can be used. Suppose a developer wants to develop a new Linux scheduler based on Linux 2.6.7. The developer can start with building WARP by running the setup script on the Linux 2.6.7 kernel source code tree. WARP provides a set of sample trace files. So once WARP is built, the developer can immediately use the trace files to experiment with the existing Linux 2.6.7 scheduler by running WARP with a debugger such as GDB. With GDB, the developer can set break points at any scheduler function and track how each scheduling decision is made. The developer can also examine the states of all the runqueues and processes and check how the process states change. After getting some first hand experience with the existing Linux scheduler, the developer can start to code his own scheduler by modifying the necessary parts of `sched.h` and `sched.c`, then recompile WARP to build the new scheduling kernel. Compiling WARP is done the same way as compiling the Linux kernel, providing a familiar development environment. However, since WARP includes much fewer files, it compiles substantially faster.

Once WARP is rebuilt with the new scheduler code, the developer can run the code using the sample event trace files. The developer can create his own event trace files with the help of our tracing toolkit. Since WARP is a user space program, a system reboot is no longer necessary to test the new scheduler code. If the code has bugs, WARP might crash when executed. But the crashing of a user space program is far less intimidating than a kernel failure. With the help of GDB, the developer can step through the execution to quickly identify the problems. He can check each scheduling decision to see if the scheduler actually works as expected. If necessary, the developer can make changes to the scheduler and recompile. The developer can also use standard application performance analysis tools like the GNU profiler `gprof` to study the overhead of the scheduler being developed. Figure 4 shows a snapshot of `gprof` output gathered during a WARP execution. The snapshot is very informative in that it tells how many times each scheduler function is called and how much time is spent on each one. While gathering this kind of information from the real kernel can require substantial kernel instrumentation, `gprof` output like that shown in Figure 4 can be obtained in just a few seconds.

Once the developer is satisfied with the behavior of the scheduler, he can build the real kernel with the new scheduler

```
Each sample counts as 0.01 seconds.
   cumulative self          self total
time seconds seconds calls call call name
13.83 2.77  1.08  170190 0.00 0.00 switch_to
11.91 3.70  0.93  171036 0.00 0.00 schedule
3.33  5.47  0.26  300039 0.00 0.00 scheduler_tick
3.07  5.71  0.24  300039 0.00 0.00 rebalance_tick
1.92  6.44  0.15  180996 0.00 0.00 recalc_task_prio
0.64  7.49  0.05  74623  0.00 0.00 load_balance
0.51  7.61  0.04  90499  0.00 0.00 try_to_wake_up
```

Fig. 4: `gprof` Output for Scheduler Code

by simply copying `sched.h` and `sched.c` back to a fresh kernel source code tree and recompile it. The scheduler code developed using WARP can now be used without modification directly in the Linux kernel.

V. Experiments

We have evaluated our Linux WARP prototype in terms of development and evaluation speed and scheduling accuracy compared to developing and running in the Linux kernel. We use both real application workloads and microbenchmarks to demonstrate that WARP is an effective tool for CPU scheduler development and evaluation. We provide a measure of development time using WARP, measure the overhead of WARP’s kernel scheduling event trace toolkit, measure the scheduler execution speed of WARP, and measure the scheduling accuracy of WARP. All of these measurements were performed on a PC with a 2.8 GHZ Intel P4 CPU and 512 MB RAM. The CPU hyperthreading feature was not used for these measurements. All measurements were performed using a Linux 2.6.7 kernel unless otherwise indicated.

A. *WARP as a Development Tool*

To study the effectiveness of WARP in CPU scheduler development, we asked students in an advanced undergraduate operating systems course at Columbia University to try out WARP. Students are required work in groups of three to implement a weighted round robin CPU scheduler for Linux as one of their course projects. Nine groups of students chose to use WARP. Among the nine groups, seven groups compiled the scheduler successfully and fixed at least some bugs using WARP. Among these seven groups, one group fixed most of the bugs using WARP and another group fixed all bugs using WARP and GDB. The group which solved all the bugs using WARP left us feedback saying: “WARP is absolutely useful. We might not have been able to submit a working scheduler if it had not been for WARP.” Among the nine groups, two groups had problems with getting WARP working because they did not follow the WARP setup instructions correctly. Despite the fact that almost all of the students were new to Linux kernel programming, seven out of nine groups of students used our WARP prototype successfully to develop and debug some parts of their kernel scheduler implementations. This result with inexperienced kernel developers provides evidence from a real user study of the benefits of WARP.

WARP has also been extensively used by several experienced Linux kernel developers. We have moved to doing most of our

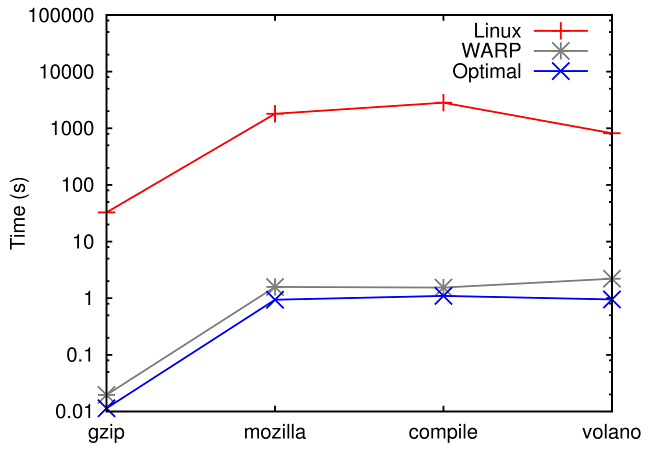

Fig. 5: Comparing Execution Time

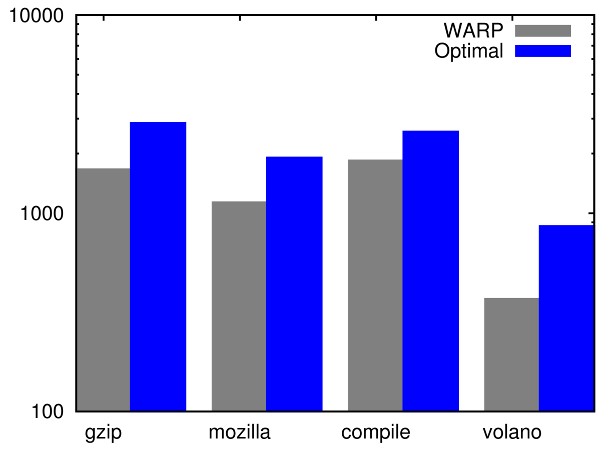

Fig. 6: Comparing Speedup

CPU scheduler debugging and evaluation using WARP. WARP has helped us to find numerous bugs, including those that would be hard to find by just working with real systems. For example, WARP helped us to find a data overflow bug. One of the users of the CKRM CPU scheduler [19], developed by one of the authors, reported that when running four Linux kernel compile sessions concurrently, one of the sessions can stall for a long time. On receiving this report, we created the virtual workload using the Linux kernel compile traces we gathered. We specified the number of CPUs and the share setting for each Linux kernel compile session the same way as reported in the bug. The stall also happened when running the virtual workload in WARP. Using WARP, we can use gdb to single step the execution without affecting the virtual execution behavior. We singled stepped the execution and found a data overflow problem which caused an accounting value to become negative. We solved this bug within one hour after receiving the bug report. Without WARP, solving such bugs can take days.

B. *WARP Simulation Speed*

We measured the speed of WARP using the event traces gathered from a set of representative real workloads, including Gzip file compression, Mozilla web browsing, Linux kernel compilation, and VolanoMark [20]. The Gzip workload consists of running gzip to compress a 53 MB data file. Gzip represents a typical CPU bound application. The Mozilla workload contains a trace that we gathered during a 30 minute web surfing session. Mozilla represents a common interactive desktop application. The Linux kernel compilation workload contains many short running processes with both CPU and I/O demands. It represents a typical batch application. VolanoMark is an industry standard Java chat server benchmark. It represents a typical multithreaded server application. The four workloads together represent commonly used workloads in desktop and server environments.

We executed these workloads on Linux and ran the event traces of these workloads using WARP. We also measured the time spent on scheduler functions when running these workloads on a real Linux system. The time spent on the scheduler functions is the minimum amount of time needed to evaluate a scheduler since skipping the execution of any scheduler function may result in incorrect scheduling decisions. For this reason, we refer to the time a real Linux system needs to execute the scheduler functions as the optimal execution time. Figure 5 compares the amount of time needed to run these workload using Linux and WARP. It also shows the optimal scheduler execution time for each workload. Figure 6 shows the same results in terms of speedup compared to Linux, with WARP speedup being the ratio of Linux over WARP execution time for the workloads, and optimal speedup being the ratio of Linux over optimal execution time.

Figure 6 shows that scheduler evaluation using WARP is 1666, 1136, 1845 and 370 times faster than using Linux for the Gzip, Mozilla, Linux compilation, and Volano workloads, respectively. The optimal speedup for the same four workloads is 2856, 1909, 2581 and 862, respectively. The speedup of WARP varies according to the workload used. This is not surprising since WARP's speed advantage comes primary from skipping the execution of all the application and operating system code unrelated to the scheduler. WARP achieves a more modest speedup of 370 times versus Linux on CPU scheduler intensive applications like Volano since it spends more time on the scheduler functions than other workloads. For all of the other workloads, CPU scheduler evaluation using WARP is more than three orders of magnitude faster than Linux. Figure 6 also shows that WARP execution time is roughly twice the optimal execution time. The additional processing needed by WARP includes reading the trace from the event trace file, dynamically creating and destroying scheduling events, sorting the scheduling events, and updating system timing information. While this overhead can be further reduced by code optimization, the difference between WARP and optimal execution is relative minor compared to the orders of magnitude performance advantage of WARP over Linux.

C. *Event Tracing Toolkit Overhead*

The overhead of WARP's kernel event tracing toolkit comes from two sources: (1) capturing each scheduling event and logging the necessary event information to the pre-allocated event buffer, and (2) writing the contents of the event buffer to a user space trace file. We measured the event logging overhead for various scheduling events. Our results showed that FORK, BLOCK, WAKEUP, PRIO, and EXIT required .74 μs, .18 μs, .19 μs, .19 μs, and .33 μs, respectively. Processing the fork and exit events take longer because the tracing toolkit creates an object to store the necessary timing information of a process when a process forks, and frees the object when the process exits. However, the submicrosecond overhead for all events is sufficient low that it is negligible for normal process execution.

We have also measured the overhead caused by writing the buffer to the event trace file. Since the buffer writing overhead depends on the number of events created by the workload, we measure the overhead using three of the previously discussed workloads: Mozilla web browsing, Gzip file compression, and VolanoMark. The buffer writing overhead per event for the three workloads was .46 μs, .55 μs, and .30 μs, respectively. Again, the submicrosecond per event overhead is sufficient low that it is negligible for normal process execution.

D. Simulation Accuracy

In addition to providing substantial improvements in scheduler evaluation speed, we demonstrate that WARP is also an effective scheduler evaluation platform. We compare the scheduling behavior of three Linux schedulers running under WARP versus a real system: the Linux 2.4 scheduler, the Linux 2.6.7 scheduler, and the EBS scheduler [21]. The latter was extensively discussed on the Linux kernel Mailing List as an improvement to the Linux 2.6 scheduler. Given the popularity of Linux, these schedulers represent some of the most widely used schedulers in real systems. Furthermore, given the strong focus in the Linux community on improving the kernel scheduler, it is certainly of great interest to understand how well the EBS scheduler works compared to the stock Linux 2.4 and Linux 2.6 schedulers. EBS is claimed to have the same scalability and efficiency of the current Linux 2.6 scheduler but with improved allocation fairness and system response time. To compare these schedulers, we created two simple micro benchmarks, `fairbench` and `responsebench`, to evaluate the allocation fairness and system responsiveness of the three schedulers.

In a Linux system, users specify the relative priority of a task by setting its nice value. A user would expect a high priority task to receive better service than the lower priority ones. `Fairbench` is designed to evaluate how fair the schedulers are when scheduling CPU bound tasks of different nice values. CPU bound tasks are used because their behaviors are predictable and thus fairness can be more precisely evaluated. `Fairbench` creates twenty-five CPU bound tasks when it is executed. These tasks are assigned nice values of -20, -10, 0, 10 and 19, respectively, with five tasks for each of the five different nice values. We gathered a trace of real `fairbench` execution and executed this trace on all three schedulers using WARP. Figure 7 shows the average CPU time received by tasks of different priorities in the first three minutes. `24`, `26` and `EBS` represent the Linux 2.4, Linux 2.6, and EBS schedulers, respectively. The data labeled with subscription "warp" represents the results gathered from WARP execution, while the data with subscription "real" represents the results gathered from real execution. There are five vertical bars in each group, each corresponding to a different nice value, with the ordering of bars from left to right corresponding to nice values -20, -10, 0, 10 and 19, respectively. Figure 7 shows that WARP and real system results are almost identical across all the three different schedulers. This confirms that WARP execution is accurate for the `fairbench` workload.

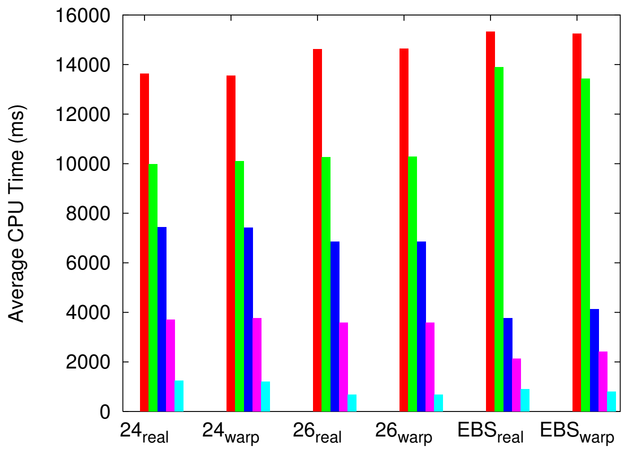

Fig. 7: Allocation Accuracy (WARP vs. Real)

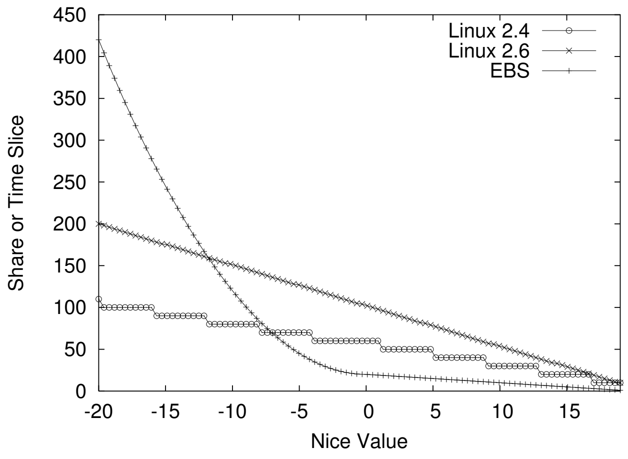

Fig. 8: Nice to Share Mapping

Figure 7 also shows that Linux 2.4 and Linux 2.6 do a reasonable job differentiating among the different task priorities, with a clear "staircase" shown for CPU time received by tasks of different priorities. In both schedulers, a difference of 10 in the nice value resulted in more than a 30% difference in CPU time received. However, using the EBS scheduler, the CPU time received by tasks with nice values of -20 and -10 were quite similar, with only a 10% difference. This is certainly a problem since it limits the effectiveness of performance differentiation.

To understand what caused the difference, we need to know more about how the three schedulers deal with tasks of different

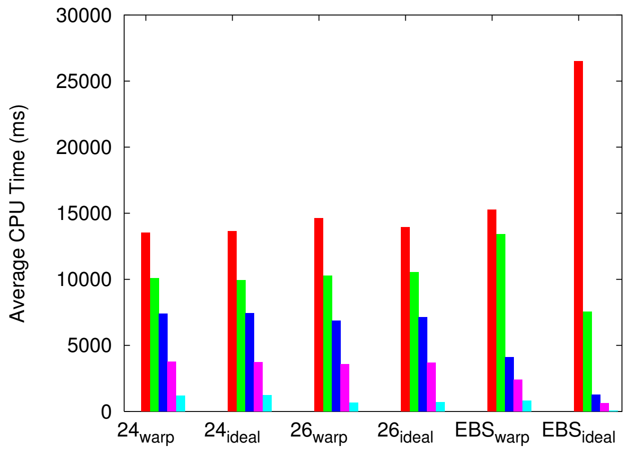

Fig. 9: Allocation Accuracy (WARP vs. Ideal)

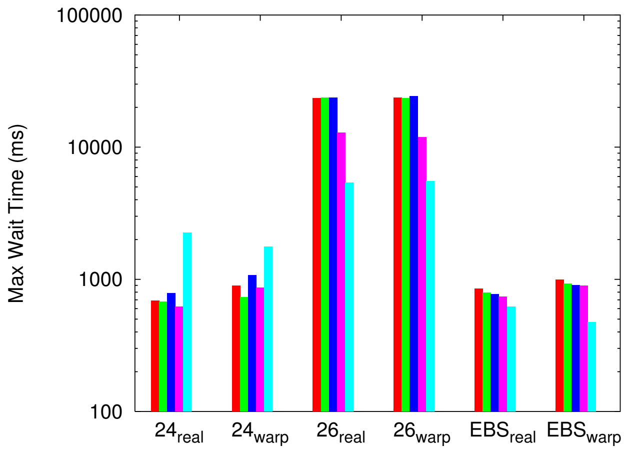

Fig. 10: System Responsiveness (WARP vs. Real)

priorities. There are basically two ways to differentiate CPU bound tasks of different priorities: either run them with the same frequency but give them different `time_slice` values or use a fixed `time_slice` value but run the tasks in different frequency. Linux 2.4 and Linux 2.6 schedulers take the first approach. Figure 8 shows how the nice values are mapped to the `time_slice` values in the different schedulers. Both Linux 2.4 and 2.6 use a linear nice value to `time_slice` mapping but are slightly different in the dynamic range used. CPU bound tasks with nice values -20, -10, 0, 10, and 19 will receive time slices 110, 80, 60, 30, and 10 ms, respectively, in Linux 2.4 and time slices 200, 151, 102, 53, and 10 ms, respectively, in Linux 2.6 for each round of allocation. The EBS scheduler takes the second approach. It uses a fixed `time_slice` value of 100 ms, but runs tasks with different frequencies. Figure 8 shows tasks with nice values -20, -10, 0, 10, and 19 are mapped non-linearly to share values 420, 120, 20, 10, and 1, respectively. Each time a scheduling decision needs to be made, EBS picks the task that has received the lowest `usage_per_share` to run. The usage of a task is estimated using Kalman filter techniques [22]. The reason why different schedulers choose to have different nice to share or `time_slice` mapping is beyond the scope of this paper. Instead, we focus our discussion on if these schedulers precisely maintain their designed allocation policy.

While it is usually hard to maintain the dynamic CPU usage for interactive tasks, one would expect that when all tasks are CPU bound, CPU time received by each task would be proportional to its share value in all the three schedulers. To verify this, we compute the amount of CPU time each task in `fairbench` should receive during a 3 minutes run for each of the schedulers based on the nice to share mapping provided above. Figure 9 compares this computed CPU time (shown as "ideal") with WARP. Again, the five vertical bars in each group are ordered left to right corresponding to nice values -20, -10, 0, 10 and 19, respectively. The results show that while Linux 2.4 and Linux 2.6 schedulers precisely maintained the designed scheduling policy, there is a huge difference between the design share and actual time received by each task when the EBS scheduler is used. Clearly, the usage estimation technique used by EBS is not effective for this certain workload. From this measurement, we can conclude that EBS does not show good allocation fairness for the `fairbench` workload.

Our second measurement is based on `responsebench`. Similar to `fairbench`, `responsebench` also starts twenty-five tasks when it is executed. Instead of running CPU bound tasks, it runs tasks of different interactiveness. Each task running `responsebench` repeats a loop in which it runs for 5 ms then sleeps for `sleep_length` ms. `sleep_length` is provided as a parameter to control the interactiveness of this task. In this measurement, the tasks were run in five different interactiveness levels, with `sleep_length` set to be 5, 10, 20, 50, and 100 ms for these levels. Again, five tasks were created for each interactiveness level.

Since `responsebench` was designed to measure the system responsiveness, we report the maximum wait time received by tasks of different interactive level instead of the CPU time for this benchmark. The wait time represents how much time is spent on waiting on runqueue for each of the 5 ms run during the task execution. Figure 10 compares the system responsiveness of the three schedulers: Linux 2.4, Linux 2.6 and ESP, with both WARP and real execution results provided. The five vertical bars in each group represent the worst wait time received by tasks of the following five different nice values: -20, -10, 0, 10 and 19 respectively. Figure 10 shows the WARP and real execution results for Linux 2.6 are almost identical. This is because the WARP execution results for Linux 2.6 scheduler were based on the trace of the real Linux 2.6 execution shown in Figure 10. So in this case, the WARP execution is basically an off-line replay of the same execution. Since a trace gathered using WARP tracing toolkit can be used to evaluate different schedulers, the WARP execution results for Linux 2.4 and EBS schedulers are also generated based on this trace. While the real and WARP execution results for Linux 2.6 are almost identical, small differences can be seen for results of the other two schedulers. This is largely because we use the function `usleep()` to control how long a task sleeps for each round. Since `usleep()` is known to be not exactly accurate, the behavior of `responsebench` varies on different executions and thus the execution results are also slight different. But in general, this kind of measurement annoyance is small and does not affect the evaluation of the three schedulers.

Figure 10 shows that EBS did a good job for the system response benchmark. Its worst case response time for each task is generally at par with Linux 2.4 while an order of magnitude better than Linux 2.6. It is a surprise to us that Linux 2.6 is the worst for `responsebench` among all the three schedulers. By tracing Linux 2.6 scheduler execution using WARP, we found that this is all because of a certain "optimization" Linux 2.6 does for tasks that are determined to be interactive. As we mentioned, Linux 2.6 does CPU time allocation based on rounds and in each round a task can receive a maximum of `time_slice` ms. However, if a task is determined to be interactive, the Linux 2.6 scheduler makes an exception and allows this task continue to run for a configurable amount of time. This "optimization" certainly helps improve the system response time when the interactive task consumes less than its allocated share. However, when the system is busy and the interactive task consumes more than its allocated share, having a task run faster in advance will only result in it being punished later and thus receive worse overall system response time.

While more measurements are needed to fully understand the relative strength and weakness of the three schedulers, our measurements already demonstrate that WARP is an accurate and effective tool for scheduler evaluation. Since WARP execution is always repeatable and can be traced step by step using GDB, we found it very helpful in analyzing the differences in the three schedulers. In this sense, it is a more effective CPU scheduler evaluation environment than a real system.

VI. Conclusions and Future Work

WARP is a trace-driven virtualized CPU scheduler execution environment that can dramatically simplify and speed the development and evaluation of CPU schedulers. WARP virtulizes the components needed by a CPU scheduler. It simplifies CPU scheduler development by running unmodified Linux CPU schedulers at user-space. WARP's kernel tracing toolkit can be used to capture traces of all CPU scheudling related events from a real system. WARP can replay these event trace based virtual workloads with the same timing charactersitics as the real workloads. By using the time-warping technology, WARP executes these virtual workloads more than two orders of magnitude faster than running real applications.

Resource contention on shared resources such as cache, execution unit, and memory bus can significantly impact application performance. Much research has been done to understand the impact of such resource contentions and to develop CPU schedulers that try to mitigate their performance impact. These algorithms may use performance counters provided by modern CPUs to guide scheduling decisions. WARP currently does not model these performance counters and the associated hardware components, and thus cannot be used directly to analyze the effectiveness of such algorithms. However, WARP's CPU and process model can be extended to simulate the behavior of shared hardware resources. For example, WARP's process model can be extended to include the memory reference per instruction information in the trace its CPU model can be extended to include a cache simulator. WARP can then be used to simulate resource contention on a shared cache. Future work on WARP can explore what extensions to WARP can support schedulers that leverage hardware-specific functionality while continuing to provide fast CPU scheduler development and evaluation.

Acknowledgment

Steve Blackburn provided helpful comments on earlier drafts of this paper. This work was supported in part by NSF ITR grant CNS-0426623 and NSF grant CNS-0717544.

References

[1] M. Rosenblum, "VMware's Virtual Platform: A Virtual Machine Monitor for Commodity PCs," in *Hot Chips 11*, Aug. 1999.
[2] S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," *IEEE Transactions on Software Engineering*, vol. SE-14, no. 9, Sep. 1988.
[3] H. Bodhanwala and L. M. Campos, "A General Purpose Discrete Event Simulator," *Symposium on Performance Evaluation of Computer and Telecommunication Systems*, July 2001.
[4] J. Dike, "User-mode Linux," in *Proceedings of the 5th Annual Linux Showcase and Conference*, Nov. 2001.
[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Oct. 2003.
[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Libuori, "KVM: the Linux Virtual Machine Monitor," in *Ottawa Linux Symposium*, June 2007.
[7] G. Gordon, "A General Purpose Systems Simulation Program," in *Proceedings of the Eastern Joint Computer Conference*, Dec. 1961.
[8] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete Computer System Simulation: The SimOS Approach," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 3, no. 4, 1995.
[9] J. Casmira, D. Kaeli, and D. Hunter, "Tracing and Characterization of NT-based System Workloads," *Digital Technical Journal, Special Issue on Tools and Languages*, vol. 10, no. 1, 1998.
[10] M. Harchol-Balter and A. B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing," *ACM Transactions on Computer Systems*, vol. 15, no. 3, 1997.
[11] S. W. Sherman and J. C. Browne, "Trace Driven Modeling: Review and Overview," in *Proceedings of the 1st Symposium on Simulation of Computer Systems*, 1973.
[12] W. Leland and T. J. Ott, "Load-balancing Heuristics and Process Behavior," *SIGMETRICS Performance Evaluation Review*, vol. 14, no. 1, 1986.
[13] C. G. Rommel, "The Probability of Load Balancing Success in a Homogeneous Network," *IEEE Transactions on Software Engineering*, vol. SE-17, 1991.
[14] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riodan, "Modeling of Workload in MPPs," in *Job Scheduling Strategies for Parallel Processing*, 1997.
[15] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante, "The Impact of I/O on Program Behavior and Parallel Scheduling," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, June 1998.
[16] M. Calzarossa and G. Serazzi, "Workload Characterization: A Survey," *Proceedings of the IEEE*, vol. 81, no. 8, 1993.
[17] D. G. Feitelson, "Workload Modeling for Performance Evaluation," in *Performance Evaluation of Complex Systems: Techniques and Tools*, 2002.
[18] D. D. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., 2000.
[19] S. Nagar, R. V. Riel, H. Franke, C. Seetharaman, V. Kashyap, and H. Zheng, "Improve Linux Resource Control Using CKRM," in *Ottawa Linux Symposium*, July 2004.
[20] Volano LLC, "Volanomark Benchmark," *http://www.volano.com/benchmarks.html*.
[21] Aurema Inc, "Linux EBS Scheduler," *http://ebs.aurema.com*.
[22] P. S. Maybeck, *Stochastic Models, Estimating, and Control*. New York: Academic Press, 1979.