REEact: A Customizable Virtual Execution Manager for Multicore Platforms

Wei Wang
University of Virginia
wwang@virginia.edu

Tanima Dey
University of Virginia
td8h@virginia.edu

Ryan W. Moore
University of Pittsburgh
rmoore@cs.pitt.edu

Mahmut Aktasoglu
Pennsylvania State University
msa203@cse.psu.edu

Bruce R. Childers
University of Pittsburgh
childers@cs.pitt.edu

Jack W. Davidson
University of Virginia
jwd@virginia.edu

Mary Jane Irwin
Pennsylvania State University
mji@cse.psu.edu

Mahmut Kandemir
Pennsylvania State University
kandemir@cse.psu.edu

Mary Lou Soffa
University of Virginia
soffa@virginia.edu

Abstract

With the shift to many-core chip multiprocessors (CMPs), a critical issue is how to effectively coordinate and manage the execution of applications and hardware resources to overcome performance, power consumption, and reliability challenges stemming from hardware and application variations inherent in this new computing environment. Effective resource and application management on CMPs requires consideration of user/application/hardware-specific requirements and dynamic adaption of management decisions based on the actual run-time environment. However, designing an algorithm to manage resources and applications that can dynamically adapt based on the run-time environment is difficult because most resource and application management and monitoring facilities are only available at the operating system level. This paper presents REEact, an infrastructure that provides the capability to specify user-level management policies with dynamic adaptation. REEact is a virtual execution environment that provides a framework and core services to quickly enable the design of custom management policies for dynamically managing resources and applications. To demonstrate the capabilities and usefulness of REEact, this paper describes three case studies—each illustrating the use of REEact to apply a specific dynamic management policy on a real CMP. Through these case studies, we demonstrate that REEact can effectively and efficiently implement policies to dynamically manage resources and adapt application execution.

Categories and Subject Descriptors D.4.1 [*Process Management*]: Multiprocessing/multiprogramming/multitasking

General Terms Performance, Management, Design

Keywords Chip multiprocessor, Resource management, Runtime adaptation, Virtual Execution Environment

VEE'12, March 3–4, 2012, London, England, UK.

1. Introduction

As the core counts and sophistication of modern chip multiprocessors (CMPs) increases, adaptive resource management techniques are needed to realize the full potential of these microarchitectural advances. There have been a flurry of such techniques proposed by the research community, such as cache contention management [7, 17, 34], processor temperature management [31, 32], and process variation management [29, 30]. While these and other adaptive policies have shown significant promise, *Users*, such as system administrators, application developers, and other technical experts, are in need of a platform to create, customize, and deploy adaptive resource management policies that address a myriad of design goals and requirements. These policies must also be *plug and play* as user-specific, application-specific and hardware-specific goals change.

For example, consider two users. The users are executing the same application on two identical machines (i.e., the same CMP architecture). However, the users have two distinct requirements. $UserA$ has a tight power budget and prefers a management policy that focuses on minimizing power consumption. $UserB$ desires a management policy that prioritizes performance over power consumption. These goals are contradictory and require distinct management policies. Given the wide range of distinct requirements, it is desirable that management frameworks support flexibility as user requirements, or application requirements, or even hardware characteristics, change.

Currently, most resource management policies are implemented in the operating system (OS) kernel. However, the OS is not well-suited for implementing and incorporating custom policies for two reasons:

1. The OS is not designed to take into account application-specific information when making management decisions. However, with application-specific information, user-level management policies can adjust the execution of applications in a way that is not possible with current OSes. For example, consider an application that uses work-stealing. Work-stealing allows the number of worker threads spawned by the application to be dynamically adjusted. If details of the work-stealing design are known, the management policy can dynamically increase the number of the application's worker threads when the system

is underutilized, and reduce the number of threads for better fairness when the system is over-utilized (see Section 4.2).

2. The complexity of modifying OS policies or adding new ones to the OS is high, which can prevent users from designing their own policies. Even after a custom policy is implemented and carefully tested, much effort has to be made when the same policy is ported to another OS or the user-goal is changed. Furthermore, custom policies may introduce security issues if they are not carefully designed and tested.

Implementing resource management policies at the user-level is easier, and it allows utilization of application-specific information. Previous work has proposed different techniques for user-level resource management [2, 8, 12, 35]. However, these techniques do not necessarily provide easy customization by the user, and some techniques only target a subset of resource management problems of CMPs. Moreover, some techniques do not provide the means to utilize online performance monitoring technology available in modern CMPs. This monitoring capability is widely used by many policies to monitor the system and dynamically adjust management decisions [7, 17, 29–32, 34].

In this paper, we advocate using a user-level virtual execution environment (VEE) to provide a framework for easy integration and development of custom CMP resource management policies. To design a user-level VEE framework, there are several challenges to overcome. The first challenge is to provide the necessary resource management facilities to allow easy development of customized resource management policies. These facilities include allocating hardware resources, adjusting application execution, and collecting run-time information about the resource landscape and application status. Furthermore, the VEE-implemented policies should dynamically adapt based on the actual run-time environment. Additionally, the VEE should be carefully designed so that management overhead does not outweigh the benefit of a custom management policy.

This paper presents a *Customizable Virtual Execution Manager* (REEact) which provides the flexibility to implement dynamic custom resource management policies. Situated between applications and the OS and hardware, REEact is active at run time, and can use both application and hardware run-time information to manage and coordinate the applications running in the system.

REEact provides the capability to specify custom management policies that need dynamic adaptation. It offers basic services for resource and application management to permit the incorporation of different management and coordination policies and mechanisms, including those that are customized to a workload, computing environment and/or system goals. These services are exposed through easy-to-use application programming interfaces (API), allowing quick development and testing without the burden or difficulty of modifying global OS policies.

With REEact, custom policies can be easily ported across platforms, so long as a few basic facilities are available on the target platform, such as thread pinning and access to hardware performance counters. Currently, REEact supports two operating systems, Linux and Solaris, and two ISAs, x86 and SPARC. Moreover, although we introduce a new layer into the system, REEact is very lightweight. Its overhead is typically less than 3% (see Section 3).

To demonstrate REEact's capabilities and usefulness for run-time management, we present three case studies. The first case study examines how REEact can implement a custom thermal management policy for a malfunctioning machine. The second case study illustrates REEact in the context of an application-specific policy that aims to maximize utilization while avoiding unfairly starving other programs from execution. The last case study describes using REEact to dynamically control hardware prefetchers to reduce power consumption without sacrificing performance.

The contributions of this paper include:

1. The REEact framework that provides the capability to easily write user-specific, application-specific and hardware-specific management policies with dynamic adaptation. We describe REEact's software architecture, which is designed to be easy-to-use, extensible, configurable and portable. REEact also permits the implementation of policies that consider application semantics.
2. A thorough evaluation of the overhead and scalability of REEact on a 32-core CMP platform. We demonstrate that, by careful design and implementation, a user-level virtual execution environment, like REEact, can perform aggressive and fine-grained on-line monitoring, dynamic adaptation and multi-application/thread coordination with very low overhead ($<$3%) and high scalability (64 thread contexts).
3. Presentation of three case studies that demonstrate REEact's flexibility for providing custom dynamic resource management. Evaluation of the three custom policies show the flexibility, effectiveness and low overhead of REEact. The results also highlight the benefits of customization. Over conventional systems, case study 1 improves performance by up to 16%; case study 2 improves the fairness of processor time allocation significantly without sacrificing overall processor utilization; and case study 3 improves performance by up to 69%, energy consumption by up to 43%, and energy-delay-product by up to 142%.

This paper is organized as follows. Section 2 presents the high-level structure and operation of REEact. Section 3 evaluates the overhead of REEact. Section 4 illustrates the operation and utility of REEact by presenting three case studies where REEact manages the use of CMP resources as by specified policies. Section 5 discusses related work and Section 6 concludes the paper.

2. REEact Framework

This section provides an overview of the REEact software architecture, and it describes the design of the framework and implementation choices.

2.1 REEact Software Architecture Overview

Figure 1 sketches the flow of using REEact to implement the specified policies. In REEact, a specified policy requires the implementation of two procedures using the API provided by REEact: a global procedure that manages the execution of multiple applications, and a local procedure that manages the execution of an application (and its threads).

The modification to an existing application to use REEact is straightforward—the main program of the application is modified to include a call to the REEact execution manager. This call essentially places the control of the main thread of execution (as well as subsequent threads the application may create) under the control of REEact. No further modification to the application is required.

During execution, REEact is initialized and it invokes the two procedures of the custom policy, and carries out the specified operations. By coordinating with the OS and the hardware, these operations manage applications and hardware resources.

To illustrate REEact's structure and operation, we first present a simple example. In this example, REEact manages the execution of three multi-threaded applications on a 16-core CMP. For ease of explanation, each core has a single execution context (i.e., simultaneous multi-threading or hyper-threading is not supported). For this example, REEact manages the use of the computation resources (i.e., the cores) and dynamically allocates or deallocates

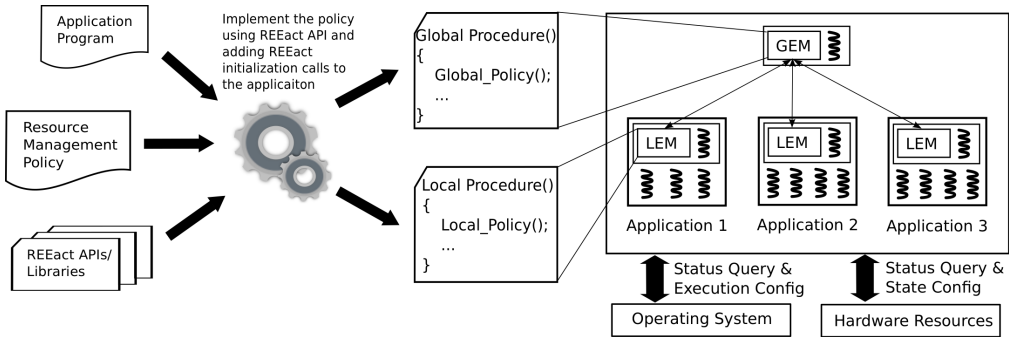
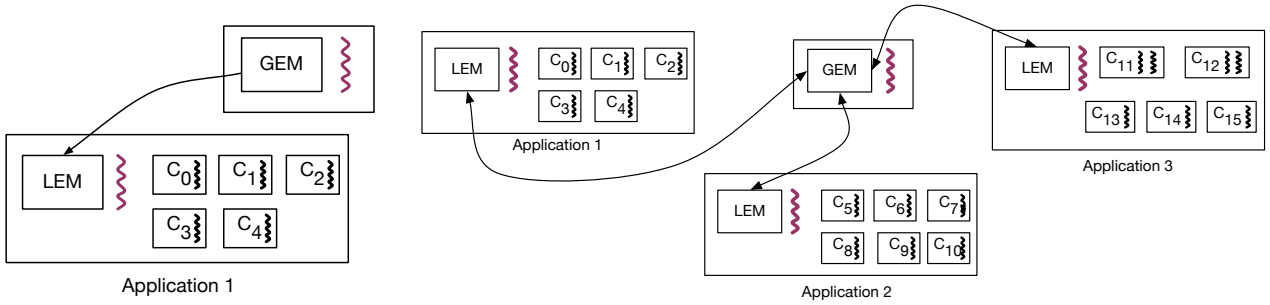


Figure 1. A custom policy requires implementing two procedures using the REEact API, and adding REEact's initialization calls to the application.



(a) REEact management of a single, multi-threaded application.

(b) REEact management of three, multi-threaded applications.

Figure 2. REEact dynamically allocated/deallocated resources (i.e., cores) using a custom FCFS policy.

them during thread creation and termination following a specified first-come, first-served (FCFS) policy.

In this example, the three applications, App_1, App_2, App_3, create five, six, and seven threads, respectively. We assume that App_1 acquires all its resources before App_2 begins execution, and App_2 acquires all its resources before App_3 begins execution.

Initially, REEact initiates a global execution manager (GEM), which invokes the global procedure to manage multiple applications. In this example, all 16 cores are managed by the GEM. Since GEM's execution has very low overhead, it is allowed to execute on any core (even a core that is allocated to an application thread). When App_1 begins execution, it calls the REEact initialization API routine. This call creates a local execution manager (LEM), which invokes the local procedure to manage the application. The first action of the LEM is to communicate with the GEM and request a core. From this point on, the LEM executes on any core that has been allocated to it (at this point it has one core).

As App_1 executes, it creates other threads. REEact intercepts thread creation calls, and notifies the LEM. The LEM communicates to the GEM requesting another core (recall the policy is FCFS). At this point, the GEM has available cores and one is allocated to this LEM. As application execution continues, additional threads are created and cores are allocated in a similar manner.

Figure 2(a) illustrates the structure of REEact at this point. In the figure, five cores (C_0–C_4) have been allocated to App_1—one for each thread of the application. The LEM thread, like the GEM thread, has low overhead and it is permitted to run on any of the cores allocated to the application (i.e., C_0–C_4).

When App_2 starts, the same process occurs. Here, App_2 creates six threads and is therefore allocated six cores (C_5–C_{10}). Then App_3 begins execution and requests cores (there are now five unallocated cores remaining). The first five thread creations result in a core being allocated for each thread. However, the sixth thread creation results in the GEM informing App_3's LEM that no core is available. The LEM then must map this thread to a core already allocated to it. The process is similar for the seventh and final thread. The final state of execution is illustrated in Figure 2(b). The last two threads of App_3 are mapped to cores C_{11} and C_{12}, respectively.

Note that during the execution of an application, a thread may terminate. REEact intercepts thread termination and notifies the applications' LEMs. If the termination of a thread frees a core, the core may be dynamically reallocated to other threads. The LEM may map an existing thread to the core, or return the core to the GEM for global reallocation. In this example, if a thread in App_1 terminates, the core is returned to the GEM which makes a global decision to offer it to App_3. App_3 can then map one of the threads that is sharing a core to its own core.

REEact supports managing applications with multiple policies. Currently, individual policies are combined manually. REEact also permits the co-existence of multiple GEMs, where each GEM controls some applications and resources using different policies.

2.2 REEact Design

2.2.1 REEact Components

REEact provides the framework and services to enable the implementation of management policies (essentially the global and local procedure). These services, such as GEM/LEM communication and thread mapping, are provided through various REEact components.

API Component	API Methods	Notes
Communicator	sendMessage	
	getMessage	blocking
	timeoutGetMessage	non-blocking with timeout
Monitor (HW Status)	readPMUperThread	
	readPMUperCore	
	getCoreTemperature	
	getTemperatureThreshold1	threshold temperature of DVFS
	getTemperatureThreshold2	threshold temperature of core shutdown
	getCoreFrequency	
	getHWComponentState	e.g. whether prefetcher or L2 cache is disabled?
Monitor (Sys. Util.)	getCoreUtilization	
	getSystemLoad	
SW Actuator	pinAppsToCores	
	pinThreadsToCores	
HW Actuator	enableHWComponent	e.g. enable or disable hardware prefetchers
	adjustFrequency	adjust processor/core frequency or duty cycle
App. State Tbl. (GEM)	getUnallocCores	
	getAllocCores	
	allocOrDeallocCore	
	getTotalCoreCount	get the total number of cores
	getCoresofCache	get the cores that share a cache
	getAllL2Caches	get a list of available L2 caches
App. State Tbl. (LEM)	getCurrentlyUsingCores	get the cores allocated to this LEM
	getAppThreads	

Table 1. API component and their associated methods currently provided by REEact

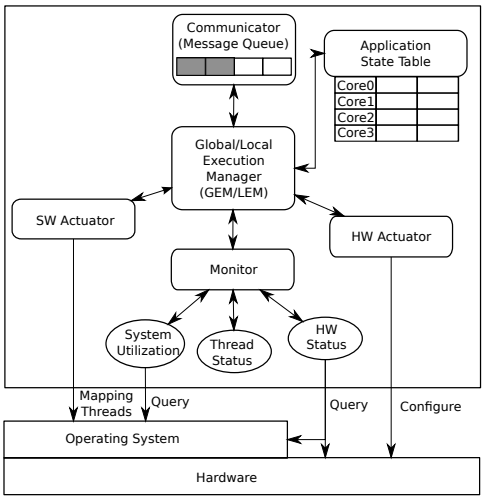


Figure 3. Essential components of REEact.

Figure 3 shows the essential components of REEact. Because proper resource management decisions have to be made based on the actual states of applications and hardware resources, REEact has monitors to collect their run-time status. Hardware and software actuators are provided to adjust resource allocation and application execution. A GEM makes global management decisions based on the run-time information collected by the LEMs, and the communication comment provides facilities for communication between GEMs and LEMs. REEact also provides application status tables so that GEM/LEM can keep track of resource allocation. The following paragraphs briefly describe each component.

The **Global/Local Execution Managers (GEM/LEMs)** have three major duties. First, they initialize and release REEact component objects during application start-up and termination. Second, they maintain the tree structure introduced in Section 2.1 (Figure 2(b)). Third, they execute customized policy procedures.

The GEM and LEMs are instantiated during application start-up. When several applications execute simultaneously, the application that starts first becomes the "master" which creates the GEM and the first LEM. Applications that start later only create LEMs. The GEM and LEM perform all necessary operations to create other REEact components. A LEM also identifies the GEM and creates a two-way communication link between them. And lastly, the GEM/LEM invokes the specified policy procedures (see Figure 1).

The **Communicator** transfers data among the GEM and LEMs asynchronously. It is designed as a message queue attached to a GEM/LEM. Each message is composed of three parts: a sender ID, a message type and the message body. The actual meaning of the message is policy dependent.

The **Monitor** provides capabilities to monitor the status of both the hardware and the applications. It collects information about the hardware, system utilization, and the status of threads.

Hardware information that can be collected includes the output of performance monitoring units (PMU), core frequency, core temperature, and whether a particular hardware component (e.g., prefetchers, L2 caches, etc.) is enabled.

The system utilization information includes the load (utilization) of each core, and the overall load. REEact also monitors the status of application threads, such as thread creation, termination and suspension.

The **SW Actuator** configures the execution of applications. The SW actuators map both applications and threads to cores. How cores are allocated depends on the policy used.

The **HW Actuator** configures a hardware (HW) component. A HW actuator can enable or disable a hardware component of a core (e.g., prefetchers), or adjust processor frequency by setting special bits of model specific registers (MSR).

The **Application Status Table (AST)** contains the current states of hardware resources and applications. Each GEM and LEM has

its own AST, which contains the information about the resources and threads that it controls. ASTs can be extended to include policy-required information.

2.2.2 REEact API

REEact components are represented as objects and their services are exposed through methods that operate on these objects. The core of REEact is the GEM and LEM classes and associated methods.

Table 1 lists the methods currently provided. For communication among the multiple applications, we provide methods to send and receive messages as part of the communicator component. There are two methods for receiving messages—a blocking method and method with timeout. Currently, the methods for monitors and actuators are designed to provide access to typical resource controls [18–20, 28, 32–34]. We provide the methods for the monitor component to collect the profiling information either per-core or per-application-thread basis. There are additional methods to obtain current temperature, temperature threshold, frequency and other hardware status of the individual physical cores. We also provide methods to gather information about core utilization. The methods for the software actuator component enable the control of the application using software, including on which core an application's threads run. The methods for a hardware actuator enable the control of the state of a particular hardware device, for example, to enable or disable the hardware prefetchers. The methods for the application state table enable the gathering of information about all the applications running in the system for the GEM and one application for LEMs. Additional services and features will be added as REEact expands to support more operating systems and architectures.

2.3 REEact Implementation

The next paragraphs describe the actual implementation of REEact components on Linux-x86 and Solaris-SPARC.

Global/Local Execution Manager (GEM/LEM): A GEM/LEM is implemented as a helper thread, which is created during REEact initialization at the beginning of application execution (recall that to use REEact, a REEact initialization call is added to the application's main program).

Communicator: The communicator is implemented using shared memory and semaphores. The message queue of a GEM or LEM is essentially a portion of memory that is shared by all GEM/LEMs. Therefore, posting a message or reading a message is an access to this shared memory. Each queue is associated with a semaphore, which notifies the GEM/LEM on arrival of a message.

Monitor: On Linux-x86, the monitor uses Perfmon2 to read PMU data [13]. On Solaris-SPARC, the monitor uses "Libcpc" to read PMU data.

On x86 architectures, the core frequency, temperature and other hardware component states are acquired by reading MSRs. REEact reads the MSRs through a special driver that we designed and implemented. On SPARC, these hardware states are acquired from the OS.

System utilization information (e.g., processor utilization) is acquired from the OS. For example on Linux, this information can be read through the "stat" file (per-core) or "loadavg" file (all-cores) under "/proc". On Solaris-SPARC, this information can be acquired from library "Libkstat".

To intercept thread status changes (e.g., thread creation, termination, suspension), we link application thread functions (e.g., pthread_create, pthread_join) to REEact's thread functions. Once a thread status change is detected, the monitor notifies the GEM and LEM by sending messages to them.

SW Actuator: For thread mapping, the SW actuator uses the core-affinity system call. Core/processor affinity is readily available in today's commodity operating systems, including Linux, Solaris and Windows. Once a thread is mapped to a set of cores, the OS does not move the thread to use any other cores. Less rigid mechanisms could also be used; for example, a mechanism could be provided to convey scheduling and allocation hints about thread co-location to the operating system. These mechanisms could be targeted by a user policy in REEact to guide OS management decisions.

HW Actuator: The HW actuators enable/disable hardware components, and adjust core frequency by setting special bits of MSRs. We implement a driver that allows reading and writing MSRs at the user-level.

Application Status Table (AST): In the default implementation, the ASTs are lists that store resource allocation information. Policies can define their own lists (or other data structures) to store any data they need.

3. REEact Overhead Evaluation

This section evaluates the overhead of the REEact framework implementation. REEact incurs overhead when reading hardware performance counters, by monitoring and managing thread creation, and through communication between the LEMs and GEM.

To determine the run-time overhead of REEact, we chose several multithreaded applications from PARSEC [5], and we measured REEact's overhead as we scaled the total number of threads and applications. The experiments were run on a CMP machine that has four Intel Xeon X7550 processors each of which has eight cores. As the processors are hyper-threaded, each core has two thread contexts and consequently the machine supports 64 thread contexts. Each physical core has a private 32KB L1-cache, a 256KB L2-cache and one 16MB L3-cache shared by eight physical cores. This machine is running Linux 2.6.32.

We conducted a series of experiments with workloads consisting of different numbers of randomly chosen PARSEC benchmarks. The benchmarks used in the experiments are: *blackscholes (BS)*, *bodytrack (BT)*, *canneal (CN)*, *dedup (DD)*, *facesim (FA)*, *ferret (FE)*, *fluidanimate (FL)*, *streamcluster (SC)* and *swaptions (SW)*. To measure the overhead of reading performance counters, we varied the period of reading counters from 10 milliseconds to 32 seconds for a fixed number of applications and threads per application. As we increased the period, REEact's overhead decreased because of less frequent access to the performance counters.

To measure REEact's overhead, we varied the number of applications from 1 to 16 and threads per application from 1 to 8, for a fixed counter-reading period. We set the counter-reading period to 10 milliseconds. We experimentally checked PARSEC and SPEC, and discovered that no benchmark has a phase shorter than 10ms (similar results are also reported by previous research [9]). Therefore, 10ms is small enough in practice to capture phase changes in the application threads. If an application does have phases less than 10ms, then phase changes may go undetected, and there could be some penalty. However, as the phases are short ($<$ 10ms), any penalty should not be high/significant. We ran experiments with four one-application, four two-application, three four-application, one eight-application and one sixteen-application workload with different number of threads under REEact control. The benchmarks used in the workloads are described in Table 2. For each workload, the overhead was measured by calculating the average percentage difference in each application's execution time normalized with respect to the native execution (i.e., no REEact).

Table 3 shows the overhead results. We did not measure the overhead for 16 applications with 8 threads per application as the total number of threads exceeds the number of thread contexts for

Workload size	Benchmark Initials used in each workload
$\|WL\|$=1	{BS}, {BT}, {CN}, {FA}
$\|WL\|$=2	{BS, BT}, {BS, CN}, {BS, FA}, {BS, SW}
$\|WL\|$=4	{BS, BT, CN, FA}, {BS, CN, FA, SW}, {BS, BT, CN, FA}
$\|WL\|$=8	{BS, BT, CN, FA, FE, FL, SC, SW}
$\|WL\|$=16	{BS, BT, CN, FA, FE, FL, SC, SW, BS, BT, CN, FA, FE, FL, SC, SW}

Table 2. The benchmarks used in each workload for measuring REEact overhead. $|WL|$ denotes the number of applications in a workload, e.g., $|WL| = 2$ means two applications in a workload. For each workload size, each set in the next column represents one workload composed by $|WL|$ number of PARSEC benchmarks (indicated by the acronym).

	Number of threads			
	1	2	4	8
$\|WL\|$=1	0.77	1.36	1.46	2.74
$\|WL\|$=2	0.79	1.82	1.45	0.63
$\|WL\|$=4	0.6	0.84	0.94	2.49
$\|WL\|$=8	1.43	1.07	1.18	1.37
$\|WL\|$=16	1.27	1.47	1.32	—

Table 3. Total overhead (in percentage) as the number of applications and threads per application is varied, for counter-reading period=10 milliseconds. $|WL|$ denotes the number of applications in a workload, e.g., $|WL| = 2$ means two applications in a workload.

the experimental machine. From the table, we observe that for a particular workload size, as the number of threads increases (reading across a column), the overhead slightly increases. The increase is due to the increased number of messages. For a fixed number of threads, as we increase the number of applications, the overhead varies slightly. This slight variation is caused by the differences in the workloads. For all 19 experiments, the average overhead of the REEact framework is at most 3%, which is acceptably small. Note that, the overall overhead of REEact given in Table 3 includes the overhead of sending messages across processors. The low overall overhead (less than 3%) implies that the use of cross-processor messaging has low overhead and is not a bottleneck for the machines we examined.

4. Case Studies

This section describes three case study policies that are implemented in REEact to demonstrate its flexibility and usefulness. These policies tackle diverse problems: thermal management, fairness among parallel applications, and performance/energy-consumption management.

4.1 Case Study 1: Fighting the Broken Screw

In the first case study, we demonstrate how to use REEact to implement a policy that reacts to thermal emergencies. We have a CMP computer that frequently experiences overheating. This computer has an Intel Q6600 quad-core processor. The processor has four cores. Each core has 32KB L1 I-cache and 32 KB L1 D-cache. Every two cores of this processor shares one 4MB L2 cache. The machine is running Linux 2.6.25.

In this machine, one of the four screws that fasten the heat sink and fan to the chip is broken, causing some cores (especially *core0*) to easily reach a very high temperature. Most CMP processors have hardware mechanisms to prevent overheating. For this processor, it uses dynamic voltage/frequency scaling (DVFS) and voluntary core shutdown. By reading the on-die digital thermal sensors (DTS), this processor monitors its cores' temperatures. If any core's temperature exceeds a factory predefined threshold T_1, the core's frequency and/or voltage are reduced. If a core's temperature keeps climbing and reaches the critical temperature T_2, the core is shutdown temporarily. Although these mechanisms prevent

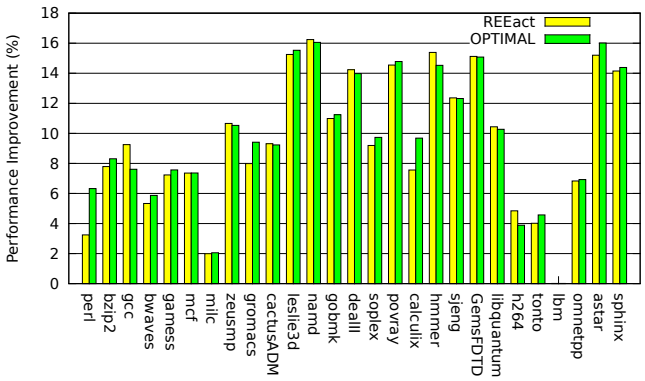


Figure 4. Performance (execution time) improvement of SPEC with REEact broken-screw policy and manually tuned optimal execution, compared to the Linux default scheduler.

catastrophic overheating, they have negative impact on system performance. If we can detect cores that are vulnerable to overheating, and schedule the threads to use these cores less frequently, then we can achieve better performance as well as prevent unnecessary overheating of the cores.

REEact's monitoring methods support reading core temperatures (using MSR "IA32_THERM_STATUS"), as well as dynamically detecting threshold temperatures T_1 and T_2. To solve the broken-screw thermal problem, we developed a custom policy using REEact. In this policy, the GEM and LEMs work together to detect overheating and migrate application threads appropriately. Policy 1.a and 1.b give the pseudocode for the global (GEM) and local (LEM) procedures respectively. When an application first requests a core for its newly created thread (through its LEM), the GEM randomly allocates a free core to it. During execution, the LEM periodically (every 10 seconds) checks if any of its cores are overheating. Depending on the level of overheating, the LEM requests different cores:

1. If the overheating core's temperature is between T_1 and T_2, the LEM asks the GEM for a core with temperature below T_1 (Policy 1.b line 16). If the GEM finds such a core (Policy 1.a lines 11-19), it responds with the new core (Policy 1.a lines 35-37), to which LEM maps its threads (Policy 1.b line 21).
2. If the overheating core's temperature is equal to or higher than T_2, the LEM marks this core as a hot core, and asks the GEM for the coolest non-marked core (Policy 1.b line 13-14). Once the GEM responses with a new core (Policy 1.a lines 21-31), the LEM re-maps its threads to it (Policy 1.b line 21). If the GEM fails to find a core, the LEM and its threads stay on the hot core.

We evaluated this policy using the SPEC2006 benchmarks. Figure 4 shows the performance (execution time) improvement of SPEC benchmarks controlled by REEact with the overheating pre-

vention policy, as well as manually tuned optimal execution, compared to the Linux default scheduler, which does not consider core temperature and can freely use any core. The optimal execution time is determined by trying all possible thread-to-core mappings, and choosing the one with the minimum execution time.

The maximum speedup of REEact over the Linux default scheduler is 16%, and the average speedup is 9.5%. The results show the benefit of using REEact framework by implementing this adaptive policy customized for this specific machine. The results also show that REEact and this policy has very low overhead, with at most 3% slowdown compared to optimal results. This slowdown is partially caused by the reactive nature of the policy: it only migrates threads when overheating actually occurs. The policy can be further refined to be proactive instead of being reactive [32].

Policy 1.a Fighting the broken screw: GEM policy procedure (boldfaced functions are REEact methods)

```
1: /* input parameters: ref to all LEM objs, ref to GEM obj, ref to monitor and ref to
   GEM's app state table */
2: INPUT: List lemList, Gem gem, Monitor m, AppStatTbl ast
3: List freeCores ← ast.getUnallocCores();
4: Int T1 ← m.getTempThreshold1();
5: Int T2 ← m.getTempThreshold2();
6: LOOP
7:    Message msg ← gem.getMessage();
8:    Lem lem ← msg.sender;
9:    Core oldCore ← msg.value;
10:   Core newCore ← oldCore;
11:   IF msg.type = TemperatureAboveT1 THEN
12:      /* search for a core that has a temperature below T1*/
13:      FOR ALL c IN freeCores DO
14:         Int t ← m.getCoreTemperature(c);
15:         IF t < T1 THEN
16:            newCore← c;
17:            BREAK;
18:         END IF
19:      END FOR
20:   ELSE IF msg.type = TemperatureAboveT2 THEN
21:      Int lowestT ← T2;
22:      /* List of cores that "lem" has used and had thermal issue (core temperature
         above T2) */
23:      List hotCores ← lem.ast.getCoresAboveT2();
24:      /* search for an unallocated core that has lowest temperature and no thermal
         issue for "lem" yet */
25:      FOR ALL c IN freeCores DO
26:         t ← m.getCoreTemperature(c);
27:         IF hotCores.doNotHave(c) AND (t < lowestT) THEN
28:            newCore ← c;
29:            lowestT ← t;
30:         END IF
31:      END FOR
32:   END IF
33:   /* ask LEM "lem" to run on newCore */
34:   IF newCore ≠ oldCore THEN
35:      lem.sendMessage(runOnCore, newCore);
36:      freeCores.add(oldCore);
37:      freeCores.remove(newCore);
38:   END IF
39: END LOOP
```

4.2 Case Study 2: Maximizing Utilization without Penalty

We next demonstrate a REEact policy that maximizes the utilization of a multicore system without unnecessarily starving other applications from execution.

Consider a shared, heavily utilized multicore server, where diverse CPU-intensive workloads arrive without notice. The run times may be unknown and multi-threaded workloads likely can run with a user-specified number of threads.

Recognizing that new workloads are likely to arrive, a user may restrict the number of threads a multi-threaded program uses, leaving CPU resources for future workloads. However, it may be difficult or impossible to know when future workloads will arrive

Policy 1.b Fighting the broken screw: LEM policy procedure (boldfaced functions are REEact methods)

```
1: /* input parameters: ref to this LEM obj, ref to GEM obj, ref to monitor and ref to
   this LEM's app state table */
2: INPUT: Lem lem, Gem gem, Monitor m, AppStatTbl ast
3: Int T1 ← m.getTempThreshold1();
4: Int T2 ← m.getTempThreshold2();
5: /* extend ast with a new list for the cores on which this LEM has thermal issue
   (core temperature above T2) */
6: List ast.CoresAboveT2 ← new List();
7: LOOP
8:    /* single-threaded app has only one thread and one core*/
9:    Core curCore ← ast.getCurrentlyUsingCores();
10:   Thread appThread ← ast.getAppThreads();
11:   Int coreT ← m.getCoreTemperature(curCore);
12:   IF coreT ≥ T2 THEN
13:      ast.CoresAboveT2.add(curCore);
14:      gem.sendMessage(TemperatureAboveT2, curCore);
15:   ELSE IF coreT ≥ T1 THEN
16:      gem.sendMessage(TemperatureAboveT1, curCore);
17:   END IF
18:   /* Get message with timeout "timeout" (non-blocking)*/
19:   Message msg ← LEM.timeoutGetMessage(timeout);
20:   IF (msg ≠ NULL) AND (msg.type = runOnCore) THEN
21:      lem.pinThreadstoCores(appThread, msg.value);
22:      sleep(timeout);
23:   END IF
24: END LOOP
```

or what their properties will be. Consequently, at times the system may be underutilized.

Alternatively, the user may assume that no new workloads will arrive. The initial workload can then use as many threads as there are CPUs (assuming CPU bound threads). Future workloads, if any, may be starved for CPU time if they generate fewer number of threads than the first application. A better policy is to dynamically adapt each program in response to system utilization.

A multi-threaded program on an otherwise idle system should use all available cores. If workloads arrive, current workloads should reduce their CPU usage (e.g., by shutting down worker threads). If workloads exit, existing workloads should expand to use the newly freed CPU resources. In this way, the system is consistently at 100% utilization, yet no program is unnecessarily starved of CPU time.

We developed a policy, called AutoMax, to dynamically expand and shrink running programs in REEact. Policy 2.a shows the global policy procedure as implemented in the GEM. Policy 2.b shows the local policy procedure as implemented in the LEM.

The GEM determines a minimum and maximum amount of utilization that the system should try to achieve (lines 4-5). It then monitors overall load, as given by the OS (line 7). The load is a measure of how many cores the system would need to properly run the current workload.

Each application is launched under the GEM's control with a LEM as described in Section 2. AutoMax obtains and permutes the list of current LEMs for the current workload (line 8). The LEM list is permuted to avoid favoring one application over another. A permuted list on average will balance the number of shrink and expand requests sent across applications and is also simple and fast to implement.

If the utilization is below a minimum threshold (line 10), a LEM is randomly selected to expand (line 11). Similarly, if the utilization is above the maximum threshold (line 12), a LEM is chosen randomly to shrink. The LEM may elect not to expand or shrink, in which case the next LEM in the list, if any, is given a chance to expand or shrink (lines 19-26).

The GEM then pauses for a few seconds (line 27). This pause (5 seconds) allows applications to shrink or expand before rebalancing is again considered. How a LEM responds to a request to expand or

shrink is application specific (Policy 2.b). Applications may require a minimum number of threads or have a maximum number of threads to run (i.e., due to a limited amount of parallelism). It is up to the LEM to reject expand or shrink messages as appropriate.

We conducted experiments to compare the performance of applications using static thread allocation versus the AutoMax policy. We modified *blackscholes* from PARSEC to support AutoMax. The modification required having worker threads perform work-stealing from a global queue of uncompleted work units. An expand request creates a new worker thread. Once a thread has received a work unit, it must complete the work unit before obtaining more work, or shutting down due to a shrink request. The modification of *blackscholes* was easy since it has a global thread number counter that explicitly tells the benchmark how to distribute their work. As developers embrace libraries that abstract away explicit thread creation and work allocation (e.g., OpenMP, Intel Thread Building Blocks), AutoMax will become more easily applicable.

Policy 2.a AutoMax: GEM Policy Procedure (boldfaced functions are REEact methods)

```
1:  /* input parameters: ref to LEM objs, ref to GEM obj, ref to monitor, ref to app
    state table */
2:  INPUT: List lemList, Gem gem, Monitor m, AppStatTbl ast
3:
4:  Double minUtilization ← ast.getTotalCoreCount() - ε1
5:  Double maxUtilization ← ast.getTotalCoreCount() + ε2
6:  LOOP
7:     Double curUtilization ← m.getSystemLoad()
8:     List permutedLL ← permute(lemList)
9:     /* Depending on system load, ask apps to expand or shrink their threads */
10:    IF curUtilization < minUtilization THEN
11:       MSGTYPE msgType = expand
12:    ELSE IF curUtilization > maxUtilization THEN
13:       MSGTYPE msgType = shrink
14:    ELSE
15:       permutedLL.removeAll() /* No app must change */
16:    END IF
17:    /* loop until an app has expanded or shrunk, or until no app can be changed */
18:    Bool didExpandOrShrink ← FALSE
19:    REPEAT
20:       Lem lem ← permutedLL.getFirst&Remove()
21:       lem.sendMessage(msgType, NULL)
22:       Message msg ← gem.getMessage()
23:       IF msg.type = changeResponse AND
          msg.sender = lem  THEN
24:          didExpandOrShrink ← msg.value
25:       END IF
26:    UNTIL didExpandOrShrink OR permutedLL.isEmpty()
27:    sleep(time);
28: END LOOP
```

Policy 2.b AutoMax: LEM Policy Procedure (boldfaced functions are REEact methods)

```
1:  /* input parameters: ref to this LEM obj, ref to GEM obj, ref to monitor and ref to
    this LEM's app state table */
2:  INPUT: Lem lem, Gem gem, Monitor m, AppStatTbl ast
3:
4:  LOOP
5:     Message msg ← lem.getMessage();
6:     IF msg.type = expand OR msg.type = shrink THEN
7:        /* the application decides its response */
8:        Bool canChange ← canExpandOrShrink(msg.type);
9:        IF canChange THEN
10:          /* the application decides how to expand or shrink */
11:          expandOrShrink(msg.type);
12:       END IF
13:       gem.sendMessage(changeResponse, canChange)
14:    END IF
15: END LOOP
```

The experiments were conducted on a 4-core CMP machine. This machine has one Intel Xeon E5335 processor. The processor has four cores. Each core has 64KB L1 I-cache and 64KB L1 D-cache, and every two cores share one 4MB L2 cache. The machine uses Linux 2.6.29.

To evaluate our policy, we ran *blackscholes* on this machine and injected synthetic single-threaded programs. Each synthetic program instance performs a fixed amount of CPU-intensive work, taking 50s on an idle system. After the synthetic programs finish, we waited 30s before starting the next period. In total, 3 waves of injections occured. The waves represent users periodically running one or more short running tasks. Figures 5(a), 5(b), and 5(c) show these results. We give the arithmetic average execution time of the synthetic programs.

As *blackscholes* is statically assigned more threads, its execution time decreases, eventually starving the synthetic programs for CPU time. In contrast, AutoMax avoids harming the performance of the synthetic programs.

With one single-threaded synthetic program active in a wave (Figure 5(a)), AutoMax *blackscholes* has an execution time closest to that of *blackscholes* when statically allocated 4 threads. When no synthetic programs are running, *blackscholes* can create new threads and use all cores on the system. The synthetic programs have an execution time closest to that of when *blackscholes* is statically allocated 3 threads (leaving a core for their use). Thus, AutoMax is able to maximize the performance of the synthetic programs without penalizing the execution time of *blackscholes*.

With two synthetic program active in a wave (Figure 5(b)), AutoMax *blackscholes* has a run time closest to that of *blackscholes* when statically allocated 3 threads. The synthetic programs have an execution time closest to that of when *blackscholes* is statically allocated 3 threads. An ideal policy would have the synthetic programs' execution time be as if *blackscholes* was statically allocated 2 threads, leaving 2 cores available for each synthetic program.

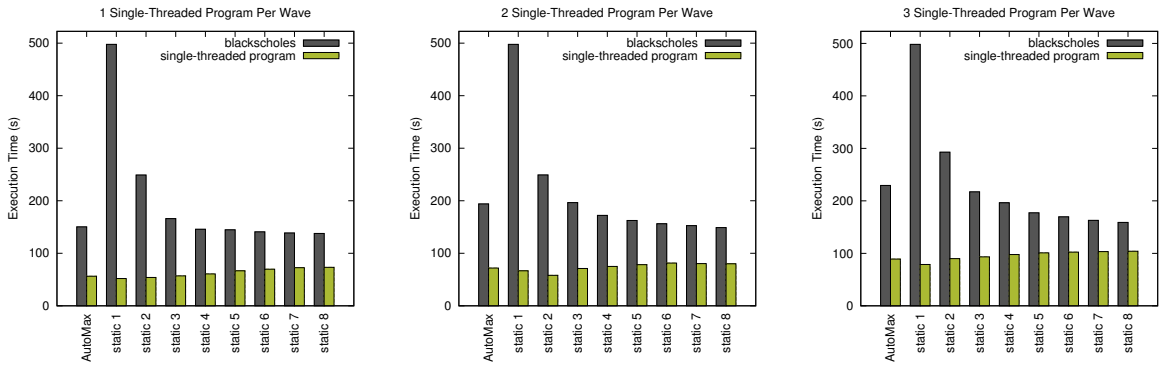
With three synthetic program instances active in a wave (Figure 5(c)), AutoMax *blackscholes* has a run time closest to that of *blackscholes* when statically allocated 3 threads. The synthetic programs have an execution time closest to that of when *blackscholes* is statically allocated 2 threads. A policy which does not impact the performance of the synthetic benchmarks would cause the synthetic programs to have the same execution time as if run alongside an instance of *blackscholes* that is statically allocated 1 core.

Based on the results shown in Figures 5(a), 5(b), and 5(c), the AutoMax version of *blackscholes* is adapting itself to other workloads on the system (i.e., the synthetic programs).

4.3 Case Study 3: Reducing Energy Usage without Performance Penalty

In the third case study, we present a REEact policy for reducing system energy consumption. This policy is based on the following observations: hardware prefetchers may have no or negative performance impact if many cache lines prefetched are not used by the application [26]. However, fetching these useless cache lines requires extra energy. Therefore, we can disable the prefetchers when they do not improve performance to save energy. With REEact, we can dynamically examine how the applications use prefetchers, and control the prefetchers accordingly.

Policy 3.a and 3.b give the pseudo-code for the global (GEM) and local (LEM) procedures. When a new thread is created or unblocked (from synchronization wait), REEact's thread status monitor sends a new message to the GEM (Policy 3.a line 6). Upon receiving this message, the GEM starts to test two configurations: one with prefetchers enabled and one with prefetchers disabled. The GEM also notifies the LEMs to collect the number of instructions retired for these two configurations (Policy 3.a lines 7-27). Each configuration is executed for 0.5 seconds and each LEM reports the number of instructions retired to GEM (Policy 3.b lines



(a) One program per wave (b) Two programs per wave (c) Three programs per wave

Figure 5. AutoMax compared to static policies. The number after "static" means the number of statically created blackscholes threads (not controlled by REEact).

Policy 3.a REEact Power management: GEM policy procedure (boldfaced functions are REEact methods)

```
1:  /* input parameters: ref to all LEM objs, ref to GEM obj, ref to monitor, ref to
    GEM's app state table, ref to HW actuators */
2:  INPUT: List lemList, Gem gem, Monitor m, AppStatTbl ast, HWActuator hwAct
3:  Int coreCnt ← m.getTotalCoreCount();
4:  LOOP
5:    Message msg ← gem.getMessage();
6:    IF msg.type = ThreadActivated THEN
7:      /* start sampling configuration with prefetchers on*/
8:      /* enable prefetchers on all cores */
9:      hwAct.enableHWComponent(c, prefetcher, on);
10:     FOR ALL lem IN lemList DO
11:       lem.sendMessage(readInsnRetired, NULL);
12:     END FOR
13:     Int insnPfOn ← 0;
14:     FOR ALL lem IN lemList DO
15:       Message msg ← gem.getMessage();
16:       insnPfOn ← insnPfOn + msg.value;
17:     END FOR
18:     /* start sampling configuration with prefetchers off*/
19:     hwAct.enableHWComponent(c, prefetcher, off);
20:     FOR ALL lem IN lemList DO
21:       lem.sendMessage(readInsnRetired, NULL);
22:     END FOR
23:     Int insnPfOff ← 0;
24:     FOR ALL lem IN lemList DO
25:       Message msg ← gem.getMessage();
26:       insnPfOff ← insnPfOff + msg.value;
27:     END FOR
28:     /* Comparing two configuration */
29:     BOOL switch ← off;
30:     IF insnPfOn > insnPfOff × 1.02 THEN
31:       switch ← on;
32:     END IF
33:     hwAct.enableHWComponent(c, prefetcher, switch);
34:   END IF
35: END LOOP
```

8-13). The GEM compares the results, and disables the prefetchers if the configuration with enabled prefetchers does not retire more instructions (Policy 3.a lines 28-33). To avoid measurement errors from PMUs, the configuration with enabled prefetchers is considered superior only if it has at least 2% more instructions retired.

We evaluated this policy on a computer with an Intel quad-core processor Q9550. Each core has 32KB L1 I-cache and 32KB L1 D-cache. There are two 6MB L2 caches each shared by two cores. For each L1 cache, there are two prefetchers (DCU and IP) that prefetch data into it. For each L2 cache, there are also two prefetchers (hardware prefetcher and adjacent cache line prefetcher) that prefetch data into it. The DCU prefetcher and the adjacent cache line prefetcher prefetch the next cache line of the missed cache line into the cache. The IP prefetcher and the hardware prefetcher look for a stride in the memory access pattern and prefetch the next expected data [16]. We tested eleven PARSEC benchmarks individually. Each benchmark was configured to run with four threads using native input sets. We ran each benchmark for three iterations and computed the average energy consumption and execution time of PARSEC's "region of interest" (parallel region). We collected the energy consumption of the processor using the methodology from Esmaeilzadeh et al. [14].

Figure 6 shows the results of seven benchmarks running under REEact, as well as two other static configurations where prefetchers are always enabled or disabled. Four benchmarks, *bodytrack*, *raytrace*, *vips* and *blackscholes*, are omitted because configuring prefetchers does not impact their performance or energy consumption. The results show that REEact always provides the best performance and lowest energy consumption. Compared to always enabling prefetchers, REEact improves execution time by up to 8%. It improves energy consumption by 12% and the energy-delay-product (EDP) by up to 19%. Compared to always disabling prefetchers, REEact improves execution time by up to 69%, energy consumption by 43% and EDP by up to 142%. The results show that REEact has little (at most 1%) overhead over the best static prefetcher configuration.

Policy 3.b REEact power management: LEM policy procedure (boldfaced functions are REEact methods)

```
1:  /* input parameters: ref to this LEM obj, ref to GEM obj, ref to monitor and ref to
    this LEM's app state table */
2:  INPUT: Lem lem, Gem gem, Monitor m, AppStatTbl ast
3:  List appThreads ← ast.getAppThreads();
4:  LOOP
5:    Message msg ← lem.getMessage();
6:    Double time = 0.5; /* sample for 0.5 seconds */
7:    IF msg.type = readInsnRetired THEN
8:      /* read InsnRetired for all the threads of this LEM */
9:      List insnCnts ← insnCnt +
        m.readPMUperThread(appThreads, InsnRetired, time);
10:     gem.sendMessage(InsnRetiredValue, insnCnts.sum());
11:   END IF
12: END LOOP
```

5. Related Work

There has been much work in designing and implementing virtual execution environments (VEEs) for various purposes. Nesbit

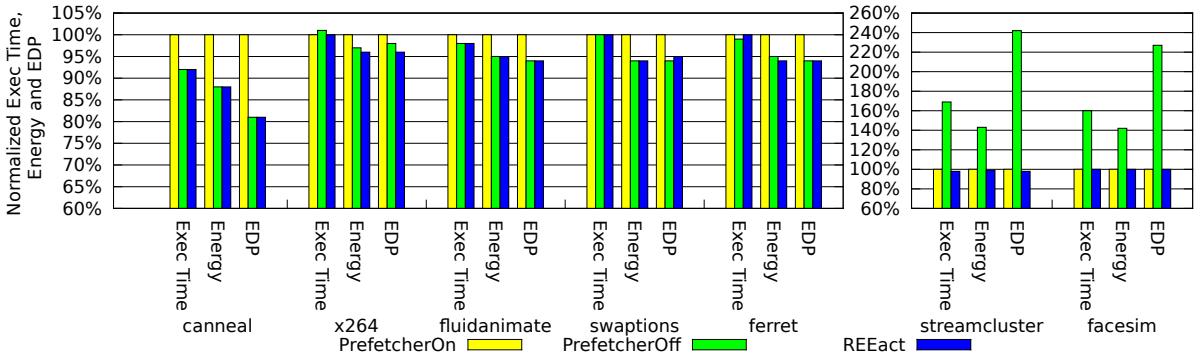


Figure 6. Execution time, energy consumption and EDP for PARSEC benchmarks running under REEact and two static configurations where prefetchers are always enabled or disabled (normalized to the configuration where prefetchers are always enabled; the lower bar is better).

et al. proposes virtual private machine (VPM) as an abstraction for managing spatial and temporal resources in multicore systems [22]. These VPMs consist of several software policies for managing resources which translate system-level requirements into different hardware mechanisms. In other work, the Xen hypervisor is extended to create a framework for supporting application-specific resource management in many core systems [23]. Cuvillo describes a Thread Virtual Machine (TVM) in the form of a thread library to allow applications to achieve full resource utilization [8]. AKULA is a tool-set for experimenting and testing scheduling algorithms that mitigate shared cache contention among single-threaded applications [35]. Noll et al. describes a virtual machine to allow programmers to use higher level programming constructs and mimic the behavior of a homogeneous shared memory multiprocessor, hiding the heterogeneity in the underlying Cell processors [24]. Similar to these researchers, we advocate the use of VEEs to dynamically manage applications and hardware resources on CMPs. However, these VEEs are usually specialized for a particular purpose (mostly for shared resource management) or an architecture, while our REEact is designed to provide a generic framework and a wide variety of services that can be used for a range of diverse purposes. Multikernel shares the same application and architecture-specific management spirit with us, but it focuses on improving the OS kernel design [4]. Log-based architecture (LBA) is similar to REEact in that LBA also constantly monitors on-line applications [6]. The difference is that LBA focuses more on application security and correctness. REEact shares the spirit of user-level application management with scheduler activation [2] and shares the spirit of application-specific resources management with exokernel [12]. However, for CMP resource/application management, extensive user-level/kernel-level thread interactions and application-specific resource abstractions are not always necessary.

There has been prior work on virtualization and hypervisors. Xen is a hypervisor that allows the existance of large number of guest OSes on the same machine with low overhead and safe resource isolation [3]. This work describes a virtualization architecture consisting of a microhypervisor and environment that provides operating system functionality including virtual-machine monitors (VMM) in user-level [27]. This user-level VMM allows execution of unmodified guest OSes in the virtual machine. On the other hand, REEact provides a VEE that supports the design and implementation of customizable user-level resource management policies for the execution of applications.

There also has been prior work addressing the problems described in the three case studies. Kadin et al. describes centralized and distributed dynamic thermal management for CMPs [18]. A low overhead thermal manager using core swapping has been proposed to address thermal emergencies [20]. A proactive dynamic temperature management technique is described by Yeo et al. [32]. Most of these software techniques can be easily implemented with REEact. Prior work has also been done in having multi-threaded programs automatically adapt their behavior. Libraries like Intel's Threading Building Blocks [25] abstract away the work of threads, and use techniques such as work stealing and/or a task scheduler to ensure that all threads (and cores) in a system are active. Our AutoMax policy differs from [25] in that applications have their own policy (in the LEM) and are able to choose how to respond, instead of being limited to the Intel task scheduler's available policy/policies. Works such as [21] are orthogonal to our approach, and might allow for the automatic creation of LEM policies. Java's thread pool preallocates a pool of worker threads [1]. An application can then use these pre-created threads to process its tasks. It can also return the threads to the pool if there are no more tasks to do. Thread pool helps minimizes the overhead due to thread creations and terminations. Thread pool can be included in REEact to help design management policies like case study 2. Several recent studies address the over-aggressiveness of current prefetchers. Zhang et al. proposed throttling processor's duty cycle and prefetchers to improve resource sharing fairness, which can be implemented with REEact [33]. A similar hardware solution was proposed by Ebrahimi et al. [11]. New hardware prefetcher designs are also proposed to reduce the negative performance impact [10, 15, 26].

6. Summary

Various user requirements, application behaviors, and hardware configurations, have greatly complicated the management of CMP hardware resources and applications. Management policies addressing such variations depends on user/application/hardware-specific requirements and require dynamic adaptation. This paper tackles these management problems by enabling the design and implementation of custom resource management policies with REEact, a user-level VEE framework. This framework provides several services for dynamic management and coordination of hardware resources and applications. REEact allows easy development of custom policies to meet different user requirements. It also facilitates the development of policies that consider application-specific information and hardware characteristics. We describe the design and implementation of REEact, and use it with three case studies, each focusing on distinct issues on CMPs—thermal management, resource efficiency and fairness, and power management. These

case studies highlight the benefits brought by REEact, such as flexibility, effectiveness, ease-of-use, and separation of concerns. Through these case studies, we also demonstrate that a carefully designed user-level VEE, like REEact, can effectively manage resources and applications with little run-time overhead.

Acknowledgments

This research is supported by National Science Foundation grants CCF-0811689, CNS-1012070, CCF-0811295, CCF-0811352, CCF-1147388, CNS-1017882, CCF-0963839 and CCF-0811687. We appreciate the insightful comments and constructive suggestions from the anonymous reviewers. We would also like to thank Jason Mars and Lingjia Tang for their valuable input.

References

[1] Java(TM) 2 Platform Standard Edition 5.0 API Specification. `http://java.sun.com/j2se/1.5.0/docs/api/`.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, 1991.

[3] 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 Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 29–44, 2009.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[6] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 63–65, 2006.

[7] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115, 2007.

[8] J. del Cuvillo. *Breaking away from the OS Shadow: A Program Execution Model Aware Thread Virtual Machine for Multicore Architectures*. PhD thesis, University of Delaware, Newark, DE, USA, 2008.

[9] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 220–231, 2003.

[10] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 316–326, 2009.

[11] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 335–346, 2010.

[12] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, 1995.

[13] S. Eranian. Perfmon2: A flexible performance monitoring interface for Linux. In *Ottawa Linux Symposium*, pages 269–288, 2006.

[14] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: measured power, performance, and scaling. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–332, 2011.

[15] I. Hur and C. Lin. Memory Prefetching Using Adaptive Stream Detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 397–408, 2006.

[16] Intel. Intel 64 and IA-32 architecture software developer's manual, 2011.

[17] L. Jin and S. Cho. SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 361–371, 2009.

[18] M. Kadin, S. Reda, and A. Uht. Central vs. distributed dynamic thermal management for multi-core processors: which one is better? In *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*, pages 137–140, 2009.

[19] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[20] E. Kursun, G. Reinman, S. Sair, A. Shayesteh, and T. Sherwood. Low-Overhead Core Swapping for Thermal Management. In *Workshop on Power-Aware Computer Systems*, 2004.

[21] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 270–279, 2010.

[22] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private machines: A resource abstraction. Technical report, In University of Wisconsin - Madison, ECE TR, 2007.

[23] D. Nikolopoulos, G. Back, J. Tripathi, and M. Curtis-Maury. VT-ASOS: Holistic system software customization for many cores. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–5, 2008.

[24] A. Noll, A. Gal, and M. Franz. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. In *Workshop on Cell Systems and Applications*, 2008.

[25] C. Pheatt. Intel®threading building blocks. *J. Comput. Small Coll.*, 23(4):298–298, 2008. ISSN 1937-4771.

[26] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, 2007.

[27] U. Steinberg and B. Kauer. Towards a scalable multiprocessor user-level environment. In *Workshop on Isolation and Integration for Dependable Systems*, 2010.

[28] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 121–132, 2009.

[29] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 363–374, 2008.

[30] J. Winter and D. Albonesi. Scheduling algorithms for unpredictably heterogeneous CMP architectures. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC*, pages 42 –51, 2008.

[31] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, and L. Jin. Dynamic Thermal Management through Task Scheduling. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 191–201, 2008.

[32] I. Yeo, C. C. Liu, and E. J. Kim. Predictive dynamic thermal management for multicore systems. In *Proceedings of the 45th Annual Design Automation Conference*, pages 734–739, 2008.

[33] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, pages 23–23, 2009.

[34] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Pro-*

ceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, pages 129–142, 2010.

[35] S. Zhuravlev, S. Blagodurov, and A. Fedorova. AKULA: a toolset for experimenting and developing thread placement algorithms on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 249–260, 2010.