Chameleon: Operating System Support for Dynamic Processors

Sankaralingam Panneerselvam and Michael M. Swift
Computer Sciences Department, University of Wisconsin–Madison
{sankarp, swift}@cs.wisc.edu

Abstract

The rise of multi-core processors has shifted performance efforts towards parallel programs. However, single-threaded code, whether from legacy programs or ones difficult to parallelize, remains important. Proposed asymmetric multicore processors statically dedicate hardware to improve sequential performance, but at the cost of reduced parallel performance.

However, several proposed mechanisms provide the best-of-both-worlds by combining multiple cores into a single, more powerful processor for sequential code. For example, Core Fusion merges multiple cores to pool caches and functional units, and Intel's Turbo Boost raises the clock speed of a core if the other cores on a chip are powered down.

These reconfiguration mechanisms have two important properties. First the set of available cores and their capabilities can vary over short time scales. Current operating systems are not designed for rapidly changing hardware: the existing hotplug mechanisms for reconfiguring processors require global operations and hundreds of milliseconds to complete. Second, configurations may be mutually exclusive: using power to speed one core means it cannot be used to speed another. Current schedulers cannot manage this requirement.

We present Chameleon, an extension to Linux to support *dynamic processors* that can reconfigure their cores at runtime. Chameleon provides *processor proxies* to enable rapid reconfiguration, *execution objects* to abstract the processing capabilities of physical CPUs, and a *cluster scheduler* to balance the needs of sequential and parallel programs. In experiments that emulate a dynamic processor, we find that Chameleon can reconfigure processors 100,000 times faster than Linux and allows applications full access to hardware capabilities: sequential code runs at full speed on a powerful execution context, while parallel code runs on as many cores as possible.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design, Performance

Keywords Dynamic Processors, Hotplug, Processor Proxy, Reconfiguration, Scheduling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS '12 March 3–7, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03. . . $10.00

1. Introduction

Multicore processors require parallel code to achieve high performance. However, parallel programming is hard, and legacy code may never be rewritten to take advantage of extra processing cores. Asymmetric chip multi-processors (ACMP) provision a chip with a small number of powerful processors for sequential code and simpler processors for parallel code [18, 32]. These processors also improve power efficiency for code that sees little benefit from powerful processors. Commercial examples of the ACMP architecture include combined CPU/GPU chips such as AMD's Fusion processor [8], and the TI OMAP 5 [56]. However, high performance on sequential code sacrifices parallel performance: the chip area and power dedicated to powerful cores could provide many more simple cores for parallel execution.

Hardware techniques that combine multiple cores or hardware threads into a more powerful execution engine have the potential to provide high performance on both parallel and sequential code. On parallel code, the processor can be configured into a large set of less-powerful cores, providing performance through parallelism. On sequential code, such a processor can combine several of the cores to improve the performance of a single thread. Such a *dynamic processor* can provide high performance over a wide range of workloads by switching between parallel and sequential modes [22]. For example, Core Fusion [26] pools execution resources such as caches and functional units to improve performance. Speculative multi-threading [20, 31, 37] executes different portions of a single thread in parallel on different cores, and Intel's Turbo boost increases the speed of one core when others are disabled [9]. Such processors may be able to reconfigure between sequential and parallel modes in microseconds.

Current operating systems are ill equipped for processors that can be reconfigured at runtime. They must know at all times which processors are available for cross-processor communication and global operations, which occur tens to hundreds of times per second. Furthermore, they require expensive global operations to reconfigure when changes occur. For example, the hotplug mechanism in Linux can take more than 200ms to add a new processor, far longer than the projected reconfiguration time for dynamic processors. In addition, processors may have mutually exclusive configurations, such as borrowing resources from one core to improve performance of another, that existing schedulers do not support.

Chameleon is an operating system extension to Linux that supports dynamic processors with three new capabilities. First, *processor proxies* enable rapid reconfiguration by removing global operations. Instead, another processor takes the place of an offline processor in any communication or global operation. Second, *execution objects* abstract physical cores and hardware threads into logical objects against which threads are scheduled, so the scheduler need not be aware of physical hardware details. Third, Chameleon's *cluster scheduler* decides when and what to reconfigure and provides a *taxation* mechanism that allows a program or administrator

to balance the benefit of faster sequential execution against reduced performance for other threads.

Chameleon focuses on the mechanisms needed to support reconfiguration but does not predict or measure the speedup a thread achieves in different configurations. Recent work on asymmetric processors addresses many issues that are also applicable to dynamic processors, such as identifying code that benefits from more powerful processors [48] or sequential bottlenecks in parallel programs [55]. These mechanisms are orthogonal to Chameleon's purpose, and could be incorporated to notify the system of when faster sequential performance is desirable or possible.

We evaluate Chameleon by emulating dynamic processors on conventional hardware and show that: (i) processor proxies reduce the latency of reconfiguration from 150ms to 2.5μs, (ii) Chameleon can leverage idle cores to achieve maximum performance for either parallel or sequential tasks via reconfiguration, (iii) fast reconfiguration allows productive use of a configuration for even a single scheduling quantum, and (iv) under contention, Chameleon's taxation allows flexible control over whether parallel or sequential code is favored. This can either allow high-priority sequential code to preempt other processors or prevent important parallel programs from having processors borrowed for sequential programs.

We begin by reviewing dynamic processor technology in Section 2. We follow with the design of Chameleon in Section 3 and implementation details in Section 4. We evaluate Chameleon in Section 5, and finish with related work and conclusions.

2. Dynamic Processors

While most computers have a static set of processors, hardware trends indicate that future computers may support a dynamically variable set of processors, either for performance, reliability, or power efficiency.

2.1 Hardware Mechanisms

We see at least four reasons why the number of execution contexts exposed to an operating system may vary at runtime.

Performance techniques. Many researchers have demonstrated single-thread performance increases by combining several cores into a single more powerful processing element. Core Fusion and TRIPS increase performance by combining resources, such as functional units, into a larger execution engine that can achieve higher ILP [26, 50]. Core Fusion, for example, claims nearly 80% speedup for some programs and requires only 400 cycles to reconfigure. Speculative multi-threading executes loop iterations or function calls in parallel [20, 31, 37, 44]. Sun's canceled Rock processor had SMT contexts that could switch to automatically prefetch data into the cache [10] for improved sequential performance.

Figure 1 shows an example of these architectures. Part (a) shows the native cores on a system, and part (b) shows how cores can be fused together to act as more powerful cores. This example is representative of Core Fusion and speculative multithreading.

A common feature of all these mechanisms is that the performance gain is less than linear in the number of cores: executing two threads on two cores accomplishes more work than executing a single thread on a fused core. Thus, a system must balance the need for sequential performance against other uses of the cores.

Power management. Processors may disable cores to save power or to transfer power to the remaining cores, as in Intel's Nehalem Turbo Boost feature [9]. In addition, processors may be *over provisioned*, in that they contain more processing elements than can be used simultaneously [15]. As a result, a system may switch between a single large, fast core and a smaller number of more efficient and slower cores when parallelism is available or better cores provide

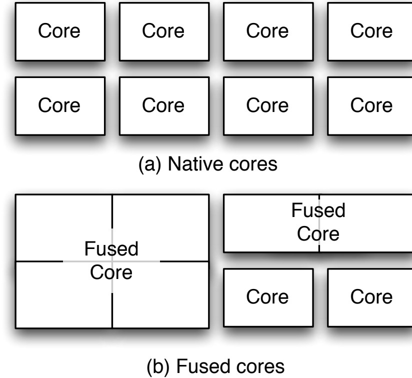

Figure 1. The native cores shown in (a) can be reconfigured, for example into a 4-core unit and a 2-core unit shown in (b).

little benefit [42]. This also enables a variety of specialized processors for specific tasks, such as encryption.

Reliability techniques. Processing cores may be combined to improve reliability. For example, redundant execution techniques run a thread simultaneously on multiple cores and automatically recover from failures when outputs differ [1, 47, 57]. When surplus cores are available, these techniques promise inexpensive error detection and fault tolerance.

Virtualization. When hosting a web site at a cloud provider, a VM can drop to a single processor when workloads are light and use more virtual processors when workloads are heavy.

2.2 Operating System Impact of Reconfiguration

These mechanisms all vary the set of processors available to the operating system. However, most OSes must know exactly which CPUs are available at all times. For example, interprocessor communication, whether through inter-processor interrupts or lightweight RPC [5], requires that the OS know if the destination processor is available. In addition, operating systems maintain *per-CPU data structures*, such as run-queues or packet receive queues, to avoid lock and memory contention. Finally, operating systems perform *all-processor operations* that require the cooperation of all processors, such as read-copy-update (RCU) operations on Linux [39]. Thus, the OS must reconfigure internal data structures whenever the set of available processors change.

We examined the Linux 2.6.31-4 kernel to discover the extent and frequency of these operations. We measure inter-processor communication running `pmake` to build the Linux kernel on a 24-CPU machine with 24 `pmake` processes.

- *Inter-processor interrupts.* Running `pmake` resulted in delivery of 40 IPIs/second/CPU for rescheduling and TLB shootdowns.
- *All-processor operations.* Global RCU callbacks were invoked 140 times per second per CPU when running `pmake`.
- *Per-CPU structures.* We found 446 separate variables in Linux that are defined as per-CPU. The *arch* subdirectory defines 294 variables, which mostly refer to hardware structures. If the set of processors change, these per-CPU data structures must be updated or initialized to reflect the change: 15 subsystems register 35 callbacks to update their per-CPU structures when the set of CPUs change.

Based on these observations, we find that the Linux kernel is intimately aware of the set of available processors. If this set were to change rapidly, numerous subsystems would have to be notified and many per-CPU structures updated. Furthermore, operations that require all processors can only execute when the set of CPUs is stable,

either delaying these operations or blocking frequent reconfiguration. While virtual machines can implement reconfiguration, frequent cross-processor communication demonstrates that operating systems must know the set of available processors.

3. Design

Chameleon is an operating system extension to support dynamic processors. We have three design goals for Chameleon:

1. *Rapid adaptation* allows the use of a processor configuration for short periods with low overhead.
2. *Abstracted hardware* provides the operating system with a set of hardware configurations for scheduling threads.
3. *Flexible, intuitive scheduling* allows existing scheduling paradigms/controls, such as fairness and priority, to apply to dynamic processors.

We seek maximum flexibility in the use of dynamic processors, and thus want to minimize the overhead introduced by the OS when reconfiguring hardware. Low cost allows use of fine-grain reconfiguration, as often as every scheduling quantum. In addition, we want to maintain the operating system's abstract view of hardware so it need not be aware of the details of how a dynamic processor reconfigures, just that other configurations are available. Finally, we seek to extend existing thread schedulers to use dynamic processors, so that existing scheduling policies are naturally extended to cover reconfiguration.

We target Chameleon at dynamic processors with a single instruction set that offer increased performance by disabling some execution contexts and reusing the hardware or power from those contexts. The rapid adaptation targets processors that cannot receive interrupts at all cores in some configurations, such as Core Fusion. In contrast, with Intel's Turbo Boost disabled cores can still receive interrupts, and hence do not benefit from this mechanism.

Our design follows a best-effort approach: the system makes no real-time guarantees, but strives to execute programs as fast as possible. Furthermore, we designed Chameleon to improve performance rather than manage power: the same mechanisms should apply, but must be driven by different policies because fusing CPUs is likely to be less efficient than running on a native CPU. Finally, Chameleon does not address uses of dynamic processors for reliability [1, 57]. Such systems place mandatory requirements on the OS like certain threads must always run in a reliable configuration.

3.1 Rapid Adaptation

Future dynamic processors may be able to change the set of available processors rapidly. Operating systems currently use a *hotplug mechanism* or *power management* to adapt to the changes in the set of processors.

Hotplug. Processor hotplug mechanisms [33, 43, 53] allow an operating system to accommodate the addition, removal, or replacement of a processor. They are designed for two uses: maintenance, to remove a failing processor or dynamically add capacity; and virtualization, to change the allocation of processors to a virtual machine. These are both infrequent events, so hotplug implementations optimize for low overhead in the common (no reconfiguration) case, rather than for frequent changes. Reconfigurations that leave a processor able to receive interrupts, such as low-power states [2], can be done with power management rather than hotplug as we describe below.

We measured the performance of hotplug in Linux, and found that it takes 150ms to take a processor offline and 220ms to bring it online. In comparison, the hardware latency of starting a processor is only 10ms. Much of the software overhead in hotplug comes from constructing and distributing per-CPU data structures and quiescing the system with a global barrier so that the mask of available processors can change. The extra delay when bringing a processor online is largely due to initializing architecture-related registers.

Power management. Operating systems also support power management, which can take a processor offline for short periods to conserve energy. The latency of entering a sleep state is on the order of microseconds, of which very little is spent in software. However, in a sleep state the processor can still receive interrupts, so it is still available to the operating system when needed and global state updates are not necessary.

Thus, power management can only be used for reconfiguration if a processor can still receive interrupts. While this is possible for current architectures, such as Turbo Boost, it may not be possible for architectures that reconfigure hardware, such as Core Fusion.

Thus, we find that both hotplug and power management are inadequate for dynamic processors: hotplug is too slow for frequent reconfiguration, and power management places requirements on the hardware such as receiving interrupts.

Processor proxies. Chameleon provides rapid reconfiguration through *processor proxies*, which are agents running on a separate processor that act on behalf of an offline processor. The proxy can access the private per-CPU data structures of the unavailable CPU. When a physical CPU is temporarily offline due to reconfiguration, Chameleon creates a processor proxy for it on another CPU. The kernel moves its communication endpoints, such as interrupts, to that CPU. Operations that require the presence of a core, such as a TLB shootdown or a read-copy-update (RCU) operation, invoke the proxy and therefore can continue without waiting for the unavailable CPU.

Processor proxies are similar to multiplexing virtual CPUs on a single processor with a hypervisor. However, processor proxies only virtualize the external interface to a processor, such as interrupts and RCU operations. Thus, a processor proxy does not schedule or run threads. In contrast, a VCPU may run any code and forces the hypervisor to schedule or timeslice multiple CPUs on a single physical CPU.

3.2 Abstracted Hardware

Increasingly, operating systems must know the topology of the processor on which they run: whether two execution contexts are hyperthreads on the same core, whether they are cores that share a cache, and whether they share an interface to memory as in a NUMA configuration. The topology allows the operating system to make intelligent scheduling and memory-management decisions, such as to schedule threads from the same process on hyperthreads that share a core, and to allocate memory from a region attached to the thread's core. Asymmetric multiprocessors similarly require informing the OS of a core's increased (or decreased) capabilities.

With dynamic processors, this need for hardware information increases: in addition to the static configuration of hardware, the OS must know about possible dynamic configurations. In Figure 1, for example, the OS must know that the four cores on the left and the pair on the right can be fused into more powerful processors.

Execution objects. Chameleon abstracts the physical capabilities of the machine with an indirection layer called *execution objects*. We call the hardware state needed to run a thread (registers, program counter) an *execution context*, which may be a core, a hyperthread, or a fused set of cores. An execution object is a kernel structure that represents a possible execution context, and contains information about the capabilities of the context along with `activate` and `deactivate` methods for reconfiguring the hardware. The smallest native context (*i.e.*, no combining of processors) is a

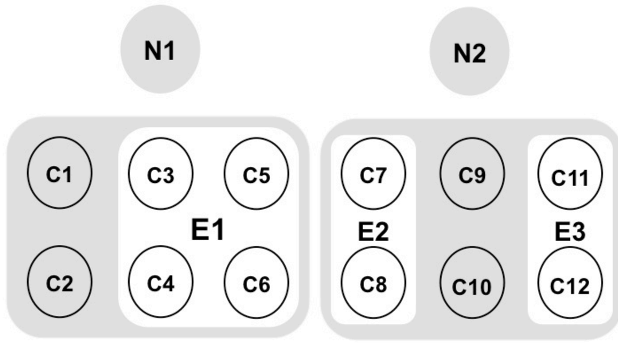

Figure 2. **System with 12 CPUs (C1-12) managed through nodes.. E2 and E3 are execution objects with two CPUs and E1 with four CPUs.**

CPU (also referred to as a thread context). A *fused execution object* (or fused object) represents an object that requires more than one native CPU. For example, in Figure 1(b), Chameleon creates a fused object for each fused core.

At any time, the current hardware configuration can be represented as a set of *active* execution objects. Chameleon creates an object when needed by a thread, and the OS invokes `activate` when dispatching a thread to the hardware configuration represented by the object. Each execution object identifies a *representative CPU*, which is the CPU to which interrupts must be delivered when the object is in use.

Execution objects expose *properties* to aid in scheduling decisions. A property is a characteristic of the underlying hardware, such as the relative performance of the configuration or whether additional features/instructions are available (*e.g.*, if only some configurations support SIMD or floating-point operations).

Nodes. Chameleon separates processors into *nodes*, which are groups of CPUs managed together. Each node has a *node manager*, which is responsible for selecting native CPUs within the node to merge into an execution object. The node manager knows the constraints of the hardware, such as which CPUs can be fused or which CPUs can shift power. In addition, it knows the properties of every execution object it can create. Figure 2 shows an example of 2 nodes.

When a thread requests an execution object with a property (described in Section 3.3), the scheduler selects a node and invokes its manager to request an execution object. The manager makes its best effort to assign the requested resources to the application from the CPUs it manages. As long as a thread is scheduled on an execution object, the node manager keeps the execution object alive. When the thread terminates or requests a different object, the resulting free CPUs can now be assigned to another execution object or left available. The manager prevents fragmentation, which can occur when idle cores are available but hardware constraints prevent them from being fused, by reshuffling the assignment of physical resources to execution objects.

The node manager assigns the CPUs comprising an execution object when creating the object, and does not choose from available CPUs when dispatching a thread. However, a rebalancing mechanism, described in Section 4 can relocate threads and create new execution objects if needed.

Node managers are similar to the system knowledge base (SKB) in Barrelfish [5] and Linux and Windows scheduling domains used for NUMA processors [40, 51]. Unlike the SKB, node managers have a restricted focus on processing. Compared to scheduling domains, nodes add constraints on which execution objects can be used simultaneously.

3.3 Scheduling

Chameleon extends the OS to schedule threads on execution objects in addition to physical CPUs. The major challenges Chameleon addresses are:

- *Which:* Finding the possible execution objects upon which to schedule a thread.
- *When:* Prioritizing threads relative to each other.
- *Where:* Moving threads to avoid conflicts and minimize reconfiguration overhead.

The Chameleon scheduler leaves scheduling queues attached to native CPUs. When a thread is ready to run, it decides whether to activate an execution object.

Property matching. A key challenge for any asymmetric processor is determining whether and how much a thread benefits from an enhanced execution context. Chameleon does not address this problem, but instead provides a general mechanism to match threads to execution objects. As previously noted, node managers associate properties with a fused object that describe its capabilities, and implement logic to match a request for specific processing features against the properties of different configurations. For example, an execution object providing high instruction-level parallelism for fast sequential execution could have the property `sequential`.

Currently, Chameleon requires that threads specify their desired properties. This could be done explicitly, through a system call, or implicitly by a separate profiling mechanism as in ACMP schedulers (*e.g.*, CAMP [48]). However, Chameleon should work with any mechanism that assigns properties to threads. In the case of a single threaded program, the property might be `sequential`, indicating that the program wants fast execution and does not depend on other threads.

When placing threads in a run queue, the Chameleon scheduler selects a node and invokes its manager to match the desired properties of a thread with the properties offered by the node's execution objects. If there is a match, the node manager creates the object, and Chameleon adds the thread to the run queue of the object's representative CPU and attaches the execution object to the thread; otherwise, it relies on the native OS scheduler to place the thread. This is similar to the matchmaking process of the Condor cluster system [46], but with a restricted set of properties.

Cluster scheduling. Chameleon schedules a thread on an execution object as if it is gang scheduling a group of threads on the constituent native CPUs: when all the CPUs are available, the thread will activate the execution object to fuse them. However, Chameleon's scheduler can also elect not to use all the CPUs and instead execute the thread on a single native CPU. Thus, when a thread becomes the next to run, the scheduler determines whether to activate an execution object. We term this *cluster scheduling*.

Chameleon decides when to fuse CPUs together and when to let them run their own threads. Each native CPU in an execution object has its own run queue with threads to execute. When a thread using an execution object becomes ready (*i.e.*, runnable), Chameleon adds it to the run queue for the execution object's representative CPU, and creates *virtual threads* that represent it in the run queues of the CPUs it will borrow. For example, in Figure 2, a thread desiring to run on fused context E2 would be placed in its representative, CPU C7, and would have a virtual thread representing it on CPU C8.

Virtual threads represent a thread's ability to borrow CPUs to form a fused object: if a virtual thread has the priority to be dispatched, then its thread can borrow the CPU. Until then, the CPU is available for its own threads. When the scheduler dispatches a thread using an execution object, it checks whether it can activate

Component	Lines
Processor Proxies	600
Execution Objects	850
Cluster Scheduling	550

Table 1. Implementation Complexity

the execution object. For example, when the scheduler dispatches a sequential thread on C7, it checks to see whether the virtual thread would have been dispatched on the other CPU C8. If so, it preempts any thread on C8, activates the execution object E2, and then dispatches the thread. If not, the thread is directly dispatched on C7.

Cross-CPU contention. Standard CPU scheduling assumes a many-to-one relationship between threads and CPUs, in that many threads share a single CPU and no thread uses more than one CPU. As a result, traditional notions of priority and fair share for threads apply only to a single CPU. However, dynamic processors introduce a new possibility: a thread can preempt multiple CPUs. Thus, threads must have a separate priority or a share for CPUs they borrow. Simply assigning a thread a global share across several CPUs, as in Linux group scheduling [12] is insufficient because it does not reflect the inefficiency of fused objects: under contention, a thread does better on a single CPU than it does running half as long on two CPUs, because speedups are less than linear. With a Linux process group, both configurations would be treated as equivalent.

Chameleon extends the existing notion of priority and share by allowing a thread to have different priorities on its representative CPU and on the CPUs it wants to borrow. We term this mechanism *taxation*: a thread is charged for its use of other CPUs adjusted by a tax rate. Effectively, this means that virtual threads may have different priorities than the thread on the representative CPU. If the tax rate is high, then a thread is charged more for borrowing a CPU than the threads that live on that CPU; thus, its priority is effectively lower and it will not be able to preempt those threads. If the tax rate is low, then it will be able to use other CPUs more cheaply than the threads on those CPUs, and it will be able to preempt them.

4. Implementation

Chameleon is implemented as an extension to the Linux 2.6.31-4 kernel. The code changes required by Chameleon were largely concentrated in two Linux subsystems: inter-processor interrupts, to implement proxies; and scheduling, to call into Chameleon during a context switch when activating execution objects. Table 1 shows the amount of code comprising Chameleon's major components.

4.1 Processor Proxies

Processor proxies speed reconfiguration because they remove much of the work to change the set of processors. Proxies consist of two elements: (i) methods to create and destroy proxies, (ii) a new execution context for executing interrupts and bottom halves on behalf of the disabled context. The CPU going offline is the *proxied CPU*, and the one that will act as its proxy is the *proxying CPU*.

Proxy creation. The activation of an execution object launches the creation of a proxy by sending a notification to the proxied CPU. The CPU that sends this request will be the proxy. The receiving CPU prepares to go offline by switching to the idle thread, which removes the need to participate in RCU operations, and by ensuring interrupts for the CPU will be delivered to its proxy. These interrupts fall into two categories: device interrupts, which can be redistributed to any online CPUs, and IPIs, which must be sent to the proxying CPU. Each is handled by a separate mechanism.

For device interrupts, the IOAPIC maintains a redirection table indicating the core to which external interrupts should be sent [24]. However, reprogramming the IOAPIC is slow, as we show in Section 5. Instead, Chameleon leverages APIC *logical addressing* when possible: device interrupts are broadcast to a logical address and each CPU ANDs a local mask against the interrupt's address and delivers the interrupt if any common bits are set. Chameleon therefore does logical-address renaming by adding the proxied CPU's identifier to the mask for the proxying CPU and resetting the identifier for the proxied CPU. This causes interrupts, both external and inter-processor, destined for the proxied CPU to be delivered to the proxying CPU automatically.

However, logical address renaming may not be available on all systems because the number of bits representing different CPUs is limited. Thus, we implemented a separate software mechanism to redirect IPIs to the proxying CPU. When creating a proxy, Chameleon records in a *redirection table* that the proxied CPU is being proxied. When a CPU sends an IPI, it consults the redirection table to learn where the IPI should be sent. In this case, Chameleon reprograms the IOAPIC to redirect device interrupts.

When using logical address renaming, redirecting interrupts can cause an IPI to be delivered to two CPUs or not delivered at all, based on the order in which the mask of proxying CPU and proxied CPU are changed. Since IPI handlers in Linux are idempotent and may be called multiple times without harm, Chameleon always updates the local mask of proxying CPU to include proxied CPU's identifier before resetting the mask of proxied CPU. Reversing this order could result in loss of IPIs. Furthermore, when tearing down a proxy, Chameleon invokes IPI handlers on the proxied CPU before invoking the scheduler in case an IPI was lost. These issues do not arise with device interrupts because the hardware ensures they are delivered to only one CPU even if the logical address matches multiple CPUs.

Proxy context. We add a new execution context to the OS, in addition to process context and interrupt context, termed a *proxy context*. A separate proxy context exists on a CPU for each of the processors it proxies and executes only when the proxying CPU receives inter-processor interrupts (IPIs) on behalf of the proxied CPU. We augmented the per-CPU structures with two variables to track proxies: `proxied_by` for a CPU that is being proxied, and `proxying_for` on a CPU that is acting as a proxy.

On receiving any IPIs, the proxying CPU invokes the corresponding IPI handler natively for the proxying CPU in the proxied CPU's context. Handling IPIs requires access to per-CPU variables, which are normally accessed through the x86 segment registers. When entering proxy context, Chameleon sets the FS register to point to the per-CPU data of the proxied CPU, and resets the register when leaving proxy context. In addition, we modified the `thread_info` macro, which normally uses the stack pointer to find the CPU state of the running task. In proxy context, the macro directs accesses to the data for the proxied CPU.

With proxies, Chameleon ensures that kernel operations requiring the involvement of an offline CPU can proceed, as the role of that CPU is handled by its proxy. This includes inter-processor interrupts for scheduling, read-copy-update operations, and TLB shootdowns, which are dropped because the proxy code flushes the TLB when resuming normal operation.

In some architectures processor proxies may not always be needed: if CPUs can continue to receive interrupts when disabled, the object does not create a proxy but instead halts the CPUs. This occurs when using Chameleon with hyperthreads: the execution object schedules the idle thread on the other hyperthread. Similarly, with Intel's Turbo Boost feature other cores enter a sleep state but can still receive interrupts.

4.2 Execution Objects and Node Managers

Chameleon assigns all the CPUs on a socket to a node and instantiates a node manager for each node. The node manager is a kernel

component that tracks the execution objects and CPUs on a node. For now, Chameleon uses the existing topology information provided by platform drivers (*e.g.*, ACPI). We emulate a dynamic processor by informing node managers that they may construct fused execution objects for pairs of hyperthreads on a core and pairs of adjacent cores.

The only property we have implemented is `sequential` with a level, which is the $\log_2$ of the number of hyperthreads in the object. Threads can request a sequential object of a specified level. Property matching compares the desired level of a thread against available execution objects to find one with the same level. The `activate` method on an execution object configures the hardware to create the desired execution context, and must be called from the object's representative CPU (the lowest numbered CPU in the set). It also creates proxies for the native CPUs borrowed by the object, and then directs the hardware to reconfigure. The `deactivate` method does the reverse of `activate`; it directs the hardware to enable native CPUs and removes proxies for all the CPUs involved.

A thread invokes the node manager to request an execution object. The node manager creates an execution object and assigns unallocated CPUs that fit the object's constraints (contiguous IDs for our emulation) to the execution object. A CPU can only belong to one non-nested execution objects. Thus, in Figure 2, CPU C10 could not be part of objects E2 and E3. This constraint may be mandatory for processors that share physical resources, such as Core Fusion, but could be relaxed for systems with more flexible resource sharing.

Any request for change in the type of execution object for a thread also invokes the node manager. If the set of available native CPUs changes, such as when a thread terminates or changes its request for an execution object, the node manager can reassign CPUs between objects to avoid fragmentation that arises when enough idle CPUs are available to create a fused object, but hardware constraints prevent its creation.

4.3 Cluster Scheduling

Chameleon's cluster scheduler is built on top of the native Linux scheduler, the Completely Fair Scheduler (CFS) [28]. CFS is similar to Borrowed Virtual Time scheduling [14], and schedules tasks according to the CPU time they have used recently rather than priority. The tasks within a runqueue are ordered by a *virtual runtime* value (`vruntime` in the task structure), which is a measure of how long the task has run. The task with the lowest virtual runtime value (*i.e.*, is the furthest behind) is selected to run next. The virtual runtime value increases in proportion to the time spent executing.

Chameleon schedules all tasks in native Linux run queues. For threads requesting an execution object, Chameleon adds the thread to the run queue of the object's representative CPU. However, virtual threads, used to track a thread's priority on the borrowed CPUs, are *not* added to run queues. Instead, Chameleon leverages the CFS scheduler design: it records the virtual runtime value a thread *would receive* on another CPU had it been scheduled. This value is set when a thread is added to the queue and does not change, making it fast to record. When seeking to activate an object for a thread, the cluster scheduler is allowed to borrow a CPU, say C, if the thread's recorded virtual runtime value for CPU C is below the virtual runtime value of the thread currently running on CPU C. This indicates that, had it been a real thread, it would have been dispatched already. We describe below how this comparison is performed efficiently.

4.3.1 Activating execution objects

When a thread reaches the head of the run queue, Chameleon consults the thread's task structure to see whether the thread requested an execution object. If so, it checks whether it can activate the object. To make this efficient, Chameleon extends the Linux task structure with a `vruntimes[]` array containing an element for each CPU comprising an execution object. When adding a thread to a run queue, the cluster scheduler updates `vruntimes[]` for the CPUs in the execution object. In addition, each CFS runqueue stores the `vruntime` value a new thread would receive in its `min_vruntime` variable, and Chameleon copies this into the thread when initializing `vruntimes[]`.

When a thread finally becomes the next to run, the cluster scheduler compares the thread's `vruntimes[CPU]` with the virtual runtime value of the active thread for each CPU it needs to borrow. It only preempts the neighboring CPUs if the thread's `vruntime` is lower on *all* the needed CPUs. This requires preempting other threads, so we enable kernel preemption.

If the scheduler cannot form the execution object desired by a thread, it will try to instantiate an object with just the available CPUs, if one has the thread's requested properties (*e.g.*, sequential). For example, in Figure 2, if a thread wants object E1 but either CPU C5 or C6 is unavailable, the scheduler will see if it can form an object from CPUs C3 and C4. If so, it activates the execution object with just the available CPUs.

4.3.2 Preempting Execution Objects

While a thread executes on a fused object, threads scheduled on its borrowed CPUs may wake up and become runnable at higher priority. In addition, as a thread on a fused object executes, its `vruntime` increases so that it may no longer be the highest priority thread on all the borrowed CPUs.

Chameleon tracks how long a thread on a fused object can run by calculating when the next thread on every borrowed CPU is allowed to run, based on `vruntime` values of the threads it preempted and the running thread's `vruntimes[]` values. During every timer tick, the cluster scheduler checks whether any of the threads on a fused object's constituent CPUs are allowed to run, and if so deactivates the object and reschedules the running thread.

Reconfiguration takes time, so care must be taken to avoid reconfiguring too often. As we show in Section 5, fusing an execution object can take up to 8 μs. Chameleon relies on the existing `sysctl_sched_min_granularity` variable from CFS to set how long a thread can run before being preempted when it is no longer the highest priority thread.

4.3.3 Taxation

Taxation controls the relative priority of threads executing natively and threads that want to borrow a CPU for an execution object. CFS already adjusts the rate at which virtual runtime accrues based on priority: lower-priority tasks accumulate virtual runtime faster, so they must wait longer to execute again, while high priority tasks accumulate it more slowly, letting them run sooner. Taxation further adjusts that rate. Much as priority adjusts both scheduling latency and CPU share, taxation can act as a share, so that unimportant tasks only use idle CPUs for fused objects, or as a mechanism to reward threads that achieve high speedups with a fused object.

Chameleon adds an `eo_tax_percent` field to every task structure. When a thread runs on an execution object, the scheduler calculates the delta to its `vruntime` for the representative CPU and all borrowed CPUs, weighted by its priority. For borrowed CPUs, the scheduler multiplies the delta by the tax rate before adding to the `vruntimes[]` array.

Figure 3 shows the impact of changing the tax rate when running two CPU-bound tasks on neighboring CPUs. Task A requests an execution object, while B runs on one of the CPUs assigned to the execution object. If the tax rate is high (>> 1), then task A is charged more than task B for using its CPU and thus it sticks with native execution. If the tax rate is 1, then task A receives all of its

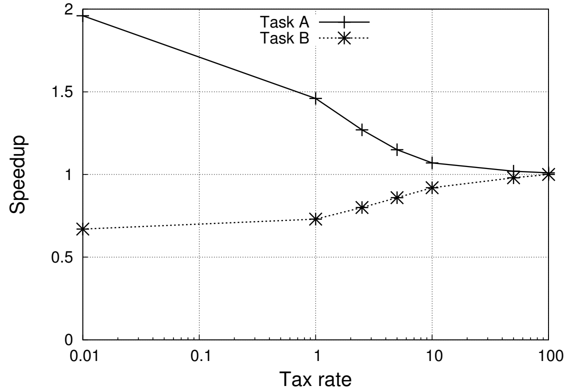

Figure 3. The effect of taxation on two identical tasks. Execution time is relative to the thread executing alone on a single native CPU.

CPU plus half of task B's CPU. If the tax rate is low (<< 1), then task A can preempt task B much of the time because it is charged less than B for using the CPU. Thus, taxation implements a cross-CPU priority mechanism.

Tax rates can be set automatically from the speedup a thread receives on a fused context. We observe that a thread that shows a good speedup deserves more access to a fused object. If progress rates metrics, such as hardware performance counters or application-specific heartbeats [23] are available, a thread's taxation rate can be set according to its speedup. Threads with good speedups deserve lower tax rates, allowing them to borrow neighboring CPUs aggressively, while threads with low speedups should only borrow idle CPUs and deserve high tax rates.

4.3.4 Property Matching

A thread declares its properties of interest by providing the identity of the property (an integer identifier), and a weight indicating the importance of the property. The node manager matches properties by comparing a thread's requested properties against the properties of all possible fused objects in the node. The matching process produces a list of possible execution objects. Currently, all properties are optional, so a thread will still be scheduled on any available CPU if an object with its requested properties is not available.

The current Chameleon implementation relies on a programmer or external agent to specify the properties a thread desires. To prioritize threads that benefit more from a fused object, the node manager uses the weight provided by a thread to identify which thread receives more benefit from a fused object when there is contention. Existing ACMP scheduling mechanisms or static profiling could provide information about the speedup of fused objects [49].

4.3.5 Rebalancing

The node manager spreads out threads requiring a fused object within a node so they can run concurrently, and to shift resources to threads that benefit from them more. When an execution object is freed, the CPUs it used are given to the active execution objects if they need them, such as an execution object that requested 4 CPUs but received only 2 CPUs. This assignment is prioritized based on the weight assigned to threads' properties.

When the node manager is not able to assign the requested number of native CPUs to an execution object, it tries to reclaim CPUs from low-priority threads and reassign these CPUs to a new fused object. If this is not possible, the node manager breaks the largest execution object and assign half the CPUs to a new execution object. This ensures that native CPUs are not reserved for a single thread when multiple threads request fused objects.

In addition, the node manager defragments execution objects. If the requested number of CPUs for an execution object are available within a node, but physical constraints prevent them from being fused (*e.g.*, they are not physically contiguous), the node manager detects fragmentation. It will then migrate threads to create a contiguous block of CPUs that can be used to satisfy future requests. Whenever the node manager makes any change to the physical resource assignment to the execution object, it notifies the execution object, which also updates the `vruntimes[]` for any threads scheduled on the object.

5. Evaluation

We evaluate Chameleon through emulation to answer these questions:

1. *Cost:* What is the latency of reconfiguring with processor proxies, and the added costs of scheduling with execution objects?
2. *Benefit:* Does Chameleon enable threads needing higher sequential performance to receive it, while allowing parallel programs full use of the CPUs in a system?
3. *Contention:* Does Chameleon behave reasonably and predictably when there are many threads contending for resources?

As dynamic processing hardware is not yet available, we evaluate Chameleon through emulation.

5.1 Experimental Platform

We emulate dynamic processors and ACMPs on a standard multicore system by varying the performance of the different CPUs. We performed our experiments on 32-bit Linux kernel version 2.6.31-4 running on a system with 12 GB RAM and two Intel Xeon X5650 chips, each chip containing six cores and each core with two hyperthreads. We refer hyperthreads as CPUs in this section. We instructed Chameleon to create two nodes of 6 cores (12 CPUs) each, and allow each node to create execution objects with 2 or 4 CPUs. We disabled TurboBoost because it varied the frequency when the system entered the P0 state, leading to widely fluctuating results. We leave the minimum scheduling granularity at the default value, 16ms.

We could not use DVFS to emulate asymmetric performance, as on our platform it applies to an entire socket rather than a single CPU. Instead, we use Intel's *clock-modulation* feature [25, 58] similar to past research on dynamic processors [3]. This mechanism is used for thermal throttling and controls the processor duty cycle by stopping the clock for short periods (less than 3μs) at regular intervals. There are eight levels available through the `IA32_CLOCK_MODULATION` model specific register (MSR). These levels reduce performance from 100% down to 12.5% of full performance in steps of 12.5%. For emulation, the `activate` method on an execution object creates the processor proxy and raises the duty cycle. Unlike real ACMPs or dynamic processors, the performance impact of clock modulation is independent of the code execution. Thus, a program sees a performance drop of 50% if the duty cycle is cut in half.

Table 2 lists the configurations we use in experiments. In all cases, we assume the baseline performance is 50% of maximum, and we emulate a faster CPU or fused object by increasing the duty cycle. For a symmetric CMP, we set all 24 CPUs to the baseline speed. For other models, we try to keep the approximate chip complexity similar. The emulated asymmetric CMP systems have either 3 or 6 fast CPUs with 75% or 100% of native performance. We use the native Linux scheduler in ACMP configuration but pin specific threads to the more powerful CPUs. As different programs see different speedups on asymmetric processors, we assign sequen-

Architecture Model	CPUs slow, fast	Speedup in Duty Cycle percent low	med	high
CMP	24, -	50	50	50
ACMP-3	12, 3	75	87.5	100
ACMP-6	12, 6	62.5	75	75
Dynamic	Fuse 2 threads	62.5	75	75
	Fuse 4 threads	75	87.5	100

Table 2. CMP, ACMP, and Dynamic configurations

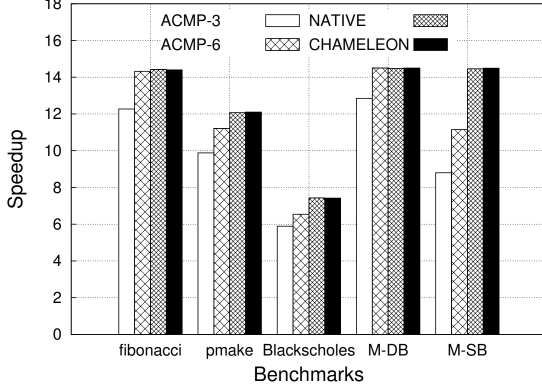

Figure 4. Performance of parallel programs.

tial programs to three performance categories (low, medium, high), with different speedups on the fast CPUs.

For dynamic processors, we follow past work [22] and set the maximum performance of fusing 2 threads to 75% (a 50% improvement over the base CPU's performance), and of 4 threads to 100%, doubling the performance of a base CPU. Again, we set different speedups for each sequential performance category to mirror the ACMP speedups. Achieving the same speedups with ACMP and dynamic processors may be optimistic, but it helps illustrate the benefit of being able to reconfigure.

We do not add additional delay to emulate the hardware cost of reconfiguring. Core Fusion estimates the delay at 400 cycles plus a pipeline flush [26], and other systems do not give any latencies. However, processor proxies flush the TLB when they are torn down, so the flush plus cost of subsequent TLB misses is included in our results.

5.2 Workloads

Past work has shown that I/O and memory-bound workloads receive little benefit from faster cores [4], so we largely evaluate with CPU-bound programs. Table 3 lists the workloads we use to evaluate Chameleon. As we do not have real hardware, variations in how a program performs on specific hardware cannot be evaluated. Thus, we instead evaluate on programs with different styles of a parallelism: task parallel threads on Intel's Thread Building Blocks (Fibonacci), task parallel processes (pmake); data-parallel threads (Blackscholes), OpenMP with a static binding of tasks to threads (Mandelbrot-SB); and OpenMP with a dynamic binding of tasks to threads (Mandelbrot-DB), which can execute more tasks given a more powerful CPU. We ran a variety of SpecCPU benchmarks on our emulator, and found they performed similarly to simple kernel benchmarks, so we also use a simple N-Queen program for sequential programs. We repeat experiments at least three times and report the average results. As there is little variance in the measurements, we do not include error bars.

5.3 Baseline Results

We evaluate Chameleon on single workloads to validate the emulator and to evaluate how close Chameleon gets to ideal performance.

Program	Description
Fibonacci	Task-parallel threads
pmake	Task-parallel processes
Blackscholes [7]	Data-parallel threads
Mandelbrot-DB (M-DB)	OpenMP kernel, dyn. binding
Mandelbrot-SB (M-SB)	OpenMP kernel, static binding
gcc [21]	Low-CPU single thread
astar [21]	Low-CPU single thread
dealII [21]	Medium-CPU single thread
lbm [21]	Medium-CPU single thread
sjeng [21]	High-CPU single thread
N-Queen [34]	High-CPU single thread
h264ref [21]	High-CPU single thread

Table 3. Workloads

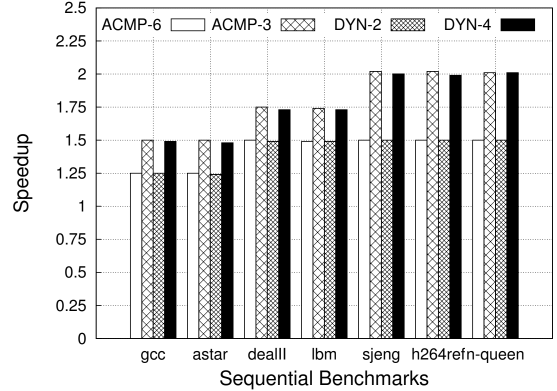

Figure 5. Performance of sequential programs.

Parallel performance. Figure 4 shows the performance of parallel workloads on four configurations: the CMP, the two ACMP models, and Chameleon. For all the parallel workloads, the best performance comes from the CMP configuration, which provides 24 CPUs. In the ACMP models, the lost parallelism outweighs the powerful CPUs present in the system except for Fibonacci and Mandelbrot with dynamic scheduling. Programs with static thread scheduling, such as Blackscholes, suffer from load imbalance when the threads on the fast CPUs wait for the threads on slow ones to finish. Chameleon uses all CPUs to achieve performance similar identical to the CMP.

Sequential performance. Figure 5 shows the performance of single-threaded workloads with varying benefit from faster CPUs. We execute each program on a single hyperthread, with the rest of the system idle. On ACMP systems we pin the program to a powerful core. For Chameleon, we evaluate both fusing two and four CPUs (DYN-2 and DYN-4). For these workloads, Chameleon is able to achieve the same performance as the asymmetric systems, despite using processor proxies. These results demonstrate that proxying introduces very little overhead and allow programs full access to the hardware's performance. In addition, they demonstrate that our emulation mechanism provides identical speedups across different programs.

Overhead. The overhead of Chameleon arises in two places: reconfiguration and scheduling. The latency of fusing and splitting CPUs is shown in Table 4. We present two versions of Chameleon's split and fuse operations: **Proxy - APIC** reprograms the IOAPIC to deliver interrupts to the proxying CPU during a fuse, and **Proxy - Logical** uses the logical addressing technique described in Section 4.1. When reprogramming the IOAPIC, proxy creation takes between 250-375μs. Logical addressing does not communicate with the slow IOAPIC and is 50-100 times faster. Compared to the native Linux hotplug mechanism, Chameleon is up to 75,000 times

Operation	Hotplug	Proxy - APIC	Proxy - Logical
Fuse 2 CPUs	150ms	250μs	2μs
Fuse 4 CPUs	430ms	375μs	8μs
Split 2 CPUs	220ms	20μs	1.5μs
Split 4 CPUs	640ms	60μs	4.2μs

Table 4. Latency of reconfiguration

faster at fusing (hot *un*plugging a CPU) and 160,000 times faster at splitting an execution object of two CPUs (hot plugging a CPU).

The number of processors that can be addressed with logical addresses may be limited in some systems, and in such cases IOAPIC reprogramming is needed. The fuse case for **Proxy - APIC** is more expensive than splitting because Chameleon reprograms the IOAPIC to remove the proxied CPU. When splitting a proxy, it does not reprogram the IOAPIC to include the proxied CPU (this is the same behavior as Linux hotplug). Fusing and splitting four CPUs is costlier since the proxy creation and destruction phases are carried out sequentially. The major savings compared to hotplug come from avoiding the notification of subsystems that the set of CPUs has changed. The remainder of our experiments uses the IOAPIC reprogramming method, as logical addressing does not currently support 24 CPUs.

We measured the added scheduling work during context switches, and there was no difference between context switching native threads on Chameleon and native Linux: both took between 2.5μs - 3μs. These results demonstrate that the added latency of Chameleon's scheduling techniques is low.

IPIs handled by proxy. We ran the pmake workload alongside a sequential application making use of an execution object with 4 CPUs. The proxy created during the activation of the execution object is responsible for handling the IPIs destined to the CPUs of the execution object and in this case was proxying for 3 CPUs. The proxy handled 10 IPIs per second, mostly TLB shootdowns. This is lower than the 40 IPIs/second reported in Section 2 because the offline CPUs switch to the idle thread, which avoids rescheduling IPIs. However, the proxy still receives TLB shootdowns because the idle thread leaves the previous user-mode address space from `pmake` on the processor.

5.4 Scheduling Mixed Workloads

We evaluate Chameleon with a mix of workloads under two situations: *over-provisioned*, when there are more CPUs than threads; and *under-provisioned*, where there are not enough CPUs. In each case, we start a mix of parallel and sequential programs at the same time and measure their completion time. If one program finishes early, the other program can make use of its CPUs. We use the Mandelbrot and N-Queen kernels because they are purely CPU bound and their performance reflects only the added effect of scheduling decisions by Chameleon, as shown in Figure 5.

Ideally, when there are idle CPUs, Chameleon will opportunistically use them to execute sequential tasks. When there is contention, Chameleon should only use them if the tax rate allows sequential threads to preempt threads on neighboring CPUs.

Over-provisioned performance. We measure the performance of executing a parallel program requiring 16 CPUs and 2-4 sequential programs simultaneously. The results in Figure 6 show that for two sequential programs, Chameleon creates a 4-CPU fused object for each task that achieves almost double the baseline performance. The four sequential threads are scheduled on 2-CPU fused objects, and achieve 50% speedup over baseline. The case of 3 sequential threads is explained more in the following paragraph. Thus, Chameleon is able to effectively place sequential threads when there are idle CPUs, and can balance them to achieve maximum performance.

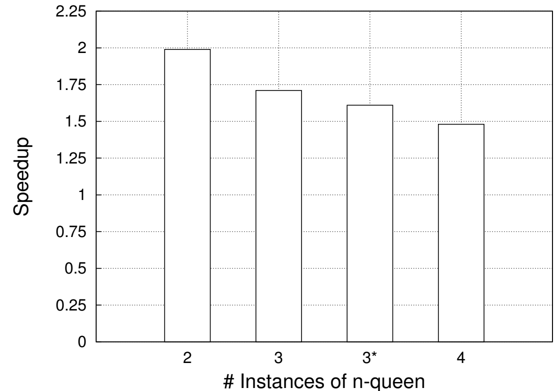

Figure 6. Over-provisioned performance of 2-4 sequential programs and a parallel program. The column marked by 3* was run without the node manager notification mechanism.

This experiment also demonstrates Chameleon's load balancing capability. With three sequential threads and eight available CPUs, one of the threads can run on a 4-CPU fused object, while the others run on 2-CPU fused objects. When the faster thread completes, the node manager distributes the physical resource used by the just finished thread to currently active threads, allowing both remaining threads to use 4-CPU fused objects. The column labeled 3* shows the benefit of notifying the node manager when a task completes. When we disable notification, which redistributes idle CPUs, performance suffers because only one thread uses a 4-CPU fused object even when idle CPUs are available.

Under-provisioned performance. We perform a similar test using three sequential programs and a parallel program with 24 threads. As there are no idle CPUs, Chameleon must decide whether to timeshare CPUs between the two programs. For these experiments, we test with both static and dynamic binding of the parallel threads, and vary the tax rate between 0.01, 1, and 100. We also present the result for the ACMP-3 configuration. In the ACMP configuration, the sequential programs were pinned to the fast CPUs while letting Linux schedule 15 parallel threads.

Figure 7 shows the speedup for the sequential and parallel programs relative to running on a single baseline CPU. As this is a single hyperthread, the speedup from using more CPUs is less than linear in the number of CPUs available. With a low tax rate, the sequential program is able to borrow the neighboring CPUs most of the time for a speedup of 97% over baseline (the speedup is less than 100% because the parallel thread occasionally uses the CPU). With a tax rate of 1, the sequential program borrows CPUs approximately half the time. But since the parallel threads also use the CPU the observed speedup is less than 50%. With a high tax rate, the sequential workloads were not able to preempt the parallel threads. As a result they complete at baseline speedup.

The parallel program shows similar variation: for dynamic scheduling where threads can have varying amounts of work, performance drops in direct proportion to the time N-Queen borrows a CPU: with the high tax rate, it gets a 11.4x speedup over baseline, while with a low tax that drops to 10x. With a low tax, N-Queen finishes quickly and the parallel program then uses all 24 CPUs. With static scheduling, where every thread must perform the same amount of work, performance varies from 11.4x speedup with the high tax rate to 8.8x speedup with a low tax rate, because of load imbalance while the N-Queen is running. Thus, the use of tax must consider both the benefit to sequential programs and the cost to the parallel programs that are preempted. The N-Queen program's use of the CPU is similar when run with both the static and dynamic parallel programs, so its performance does not change.

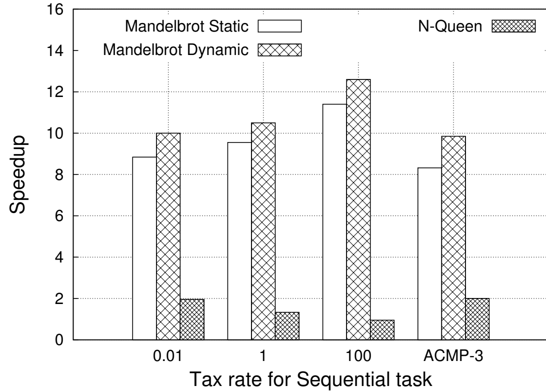

Figure 7. Under-provisioned performance of 3 instances of N-Queen program and a parallel program with varying taxation rates.

Compared to the ACMP, Chameleon achieves the same sequential performance and parallel performance on the dynamic parallel program with a low tax rate because it has more CPUs. The higher tax rates trade lower sequential performance for parallel performance exceeding the ACMP. These results show how taxation adjusts the priority of sequential and parallel programs. Its benefit may depend on how well parallel programs react to losing a CPU.

Mixed sequential workloads. This experiment demonstrates Chameleon's ability to prioritize the use of CPUs based on the speedup a thread achieves. In this experiment we ran a mix of sequential workloads of different classes: one high CPU (sjeng), one medium CPU (lbm) and two low-CPU workloads (astar) with staggered start times. We annotated each program's thread with its speedup as the weight on its sequential property, similar to what might be provided by an ACMP scheduler. We constrained all the programs to a single node with a total of 10 CPUs.

Ideally, the rebalancer will spread the available CPUs across the sequential programs and prioritize remaining CPUs to the programs with the best speedup. Figure 8 shows the results of this experiment. The two low CPU (astar) workloads were started first and were each given a 4-CPU fused object by the node manager. When lbm – a medium speedup application – was started subsequently, the manager borrowed 2 CPUs from one of the low-speedup astar applications and gave them to lbm at event ⓐ. Similarly, when sjeng – a high speedup application – started, it borrowed 2 CPUs, one each from lbm the other astar, at event ⓑ.

In short, when the remaining two workloads started, the node manager noted that the demand for sequential performance (a total of 16 CPUs forming 4 fused objects) exceeded the supply (10 CPUs), and split the objects in use by the astar instances so the two remaining programs could each receive a fused object. As described in Section 4.3.5, the node manager uses the speedups of the programs to allocate the remaining CPUs to the program receiving the most benefit, sjeng, which then runs on a 4-CPU fused object. The other three programs all used a 2-CPU fused object. Upon exit of any workload, the node manager distributes the released CPUs to the active workloads based on their speedup. In this case lbm completes first at event ⓒ and its resources are given to one astar instance. After sjeng exits at event ⓓ, its resources are given to the second astar.

Thus, Chameleon's node manager and rebalancer are able to allocate CPUs across a set of workloads with different speedups, when notified of those speedups.

5.5 Chameleon on Real Hardware

The previous results show that Chameleon can exploit the flexibility of dynamic processors to achieve improved performance.

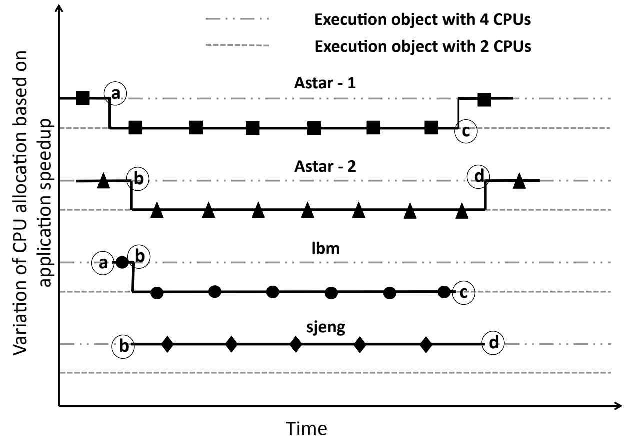

Figure 8. CPU allocation to execution object in a workload mix of sequential programs with different speedups.

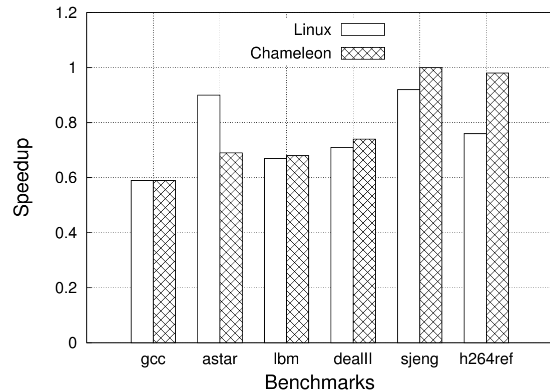

Figure 9. Chameleon on SMT.

Chameleon can also prioritize threads on existing hyperthreaded processors. Since hyperthreads share the cache and processor pipeline, running two hyperthreads on a core may reduce the performance when compared to running a single thread. As a result, the Linux scheduler tries to schedule threads on separate cores before using two hyperthreads on a single core. However, when there are more threads than cores, it is possible that a thread that benefits from running alone on a core may get scheduled on a shared CPU.

We apply Chameleon's cluster scheduler to this problem by treating each core as a fused execution object, which when activated forces one of the hyperthreads to idle. We ran one instance of six SPEC application annotated by their corresponding speedup values as listed in Table 3 over 4 physical CPUs (8 hyperthreads) with the native Linux scheduler and Chameleon. The baseline is the application running on a non-shared CPU. Figure 9 compares the speedup achieved on the two systems. Linux has no knowledge each application's speedup and thus any application may be scheduled on a non-shared core. In this case, astar and sjeng achieve a good speedup. Chameleon therefore prioritizes these two programs and schedules them on non-shared cores. Thus, Chameleon is able to prioritize resources for threads that benefit more from running alone, while Linux treats all threads equally.

6. Related Work

Asymmetric Scheduling. Several recent works investigate scheduling parallel and multiprogrammed workloads on asym-

metric or heterogeneous processors [19, 30, 35, 48, 49]. However, many of these systems focus on power efficiency, so they seek to identify which threads gain the most efficiency. Chameleon could use these techniques to identify which threads would perform best on a powerful execution object. Mogul et al. investigate the use of ACMPs for operating systems [42], but focus on static assignment of OS threads to simple cores; Chameleon could implement this policy using its property mechanism. Most similar to Chameleon is Luo et al.'s work on the use of helper threads and cache resizing on an ACMP [38]. Similar to Chameleon, this work determines when to allocate resources to speed sequential threads, but focuses again on identifying which threads gain the most benefit. ACS accelerates critical sections on an ACMP [55]. However, the latency of Chameleon's reconfiguration is still too long to help individual critical sections.

Gang scheduling. Chameleon's cluster scheduling is similar to gang scheduling in that it will try to schedule multiple CPUs simultaneously. Past work on gang scheduling [45, 27, 16, 6] focuses primarily on time-sharing gangs of threads. While Chameleon can do this, there is little benefit because performance improves by running time-sliced threads in parallel on separate cores. Thus, Chameleon is most effective when there are idle threads to be used opportunistically, or when a sequential thread has higher priority than competing workloads.

Tesselation uses a form of gang scheduling to provide a *cell* of processors to an application, which is similar to Chameleon's execution objects [11]. However, it provides cores for software process to use, while Chameleon provides cores to hardware for an enhanced CPU.

Support for reconfigurable hardware. Recent work on scheduling for reconfigurable hardware has largely focused on embedded and real-time systems [52, 29, 17]. In these environments, precise models of the transition costs and the execution time of code on different hardware are needed. These systems also place mandatory requirements on scheduling, so flexible tradeoffs like Chameleon's taxation are not used. In contrast, Chameleon focuses largely on best-effort workloads and must rely on admission control to meet performance goals.

Several projects discuss OS support for introducing reconfigurable logic onto a processor [13, 36, 54]. However, OS support for these systems focuses on efficiently allocating the reconfigurable logic to specific functions rather than on thread scheduling.

Windows 7's support for core parking [41], which coalesces threads onto a single core to disable the remaining cores, is similar to Chameleon's scheduling of threads on the execution object. It is also used to balance threads between hyperthreads. However, core parking targets all threads at a specific subset of CPUs, rather than context switching between configurations.

7. Conclusions

Dynamic processors will lead to new opportunities for improving performance, reliability, and power consumption by reconfiguring the set of running processors. Existing operating systems cannot react to changes fast enough to fully utilize reconfiguration, and do not have scheduling mechanisms to take advantage of them. Chameleon extends Linux to enable rapid reconfiguration through processors proxies, allowing use of reconfiguration even for short periods. It abstracts the reconfiguration abilities of the hardware with execution objects and nodes, which expose the new capabilities of the hardware to programmers and the scheduler. Chameleon's cluster scheduling with taxation allows sequential code to use idle cores and provides a flexible tradeoff between single-thread performance and parallel performance.

We plan to extend Chameleon for other uses of dynamic processors. Chameleon focuses on performance benefits of dynamic processors, but they should also promise power efficiency and reliability, which demand different scheduling policies. In addition, we plan to investigate other forms of dynamic processors that may not fit Chameleon's model. For example, processors with dark silicon accelerators cannot be used unless other cores are powered off.

Acknowledgements

This work is supported in part by National Science Foundation (NSF) grant CNS-0834473. We would like to thank our shepherd, Angela Demke Brown, and the anonymous reviewers for their invaluable feedback. Swift has a significant financial interest in Microsoft.

References

[1] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: Building high availability systems with commodity multi-core processors. In *Proc. of the 34th ISCA*, June 2007.

[2] J. Allarey, V. George, and S. Jahagirdar. Power management enhancements in the 45nm intel core microarchitecture. *Intel Technical Journal*, 12(3):169–178, oct 2008.

[3] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Proc. of the 32nd ISCA*, pages 298 – 309, June 2005.

[4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proc. of the 27th ISCA*, pages 282–293, June 2000.

[5] A. Baumann, P. barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proc. 22nd SOSP*, Oct 2009.

[6] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for cmps. In *Proc. of the 24th ICS*, pages 189–199, 2010.

[7] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proc. 5th Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[8] N. Brookwood. Amd fusion. family of apus: Enabling a superior, immersive pc experience. `http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf`, Mar 2010.

[9] J. Charles, P. Jassi, A. N. S, A. Sadat, and A. Fedorova. Evaluation of the Intel Core i7 Turbo Boost feature. In *Proc. International Symposium on Workload Characterization*, Oct. 2009.

[10] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous speculative threading: A novel pipeline architecture implemented in sun's rock processor. In *Proc. of the 36th ISCA*, June 2009.

[11] J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Resource management in the tessellation manycore os. In *HotPAR*, 2010.

[12] J. Corbet. CFS group scheduling. `http://lwn.net/Articles/240474/`, 2007.

[13] M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. In *Proc. Design, Automation and Test in Europe*, 2003.

[14] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proc. 17th SOSP*, 1999.

[15] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proc. of the 38th ISCA*, June 2011.

[16] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *JPDC*, 16(4):306 – 318, 1992.

[17] W. Fu and K. Compton. Scheduling intervals for reconfigurable computing. In *Proc. 16th FCCM*, Apr. 2008.

[18] R. Grant and A. Afsahi. Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications. In *Proc. 20th IPDPS*, Apr. 2006.

[19] B. Hamidzadeh, Y. Atif, and D. J. Lilja. Dynamic scheduling techniques for heterogeneous computing systems. *Concurrency: Practice and Experience*, 7(7):633–652, 1995.

[20] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The stanford hydra cmp. *IEEE Micro*, pages 71–84, March-April 2000.

[21] J. L. Henning. Spec cpu2006 benchmark descriptions. *Computer Architecture News*, 34(4):1–17, 2006.

[22] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, pages 33–38, July 2008.

[23] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E., Miller, and A. Agarwal. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proc. 7th ICAC*, 2010.

[24] Intel Corp. 82093AA I/O ADVANCED PROGRAMMABLE INTERRUPT CONTROLLER (IOAPIC). `http://www.intel.com/design/chipsets/datashts/29056601.pdf`, 1996.

[25] Intel Corp. Thermal protection and monitoring features: A software perspective. `http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/54118.htm`, 2005.

[26] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: Accomodating software diversity in chip multiprocessors. In *Proc. of the 34th ISCA*, June 2007.

[27] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proc. of the 17th PACT*, 2008.

[28] M. T. Jones. Inside the linux 2.6 completely fair scheduler, Dec 2009. http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/.

[29] H. Kooti, E. Bozorgzadeh, S. Liao, and L. Bao. Transition-aware real-time task scheduling for reconfigurable embedded systems. In *Proc. Design, Automation and Test in Europe*, Mar. 2010.

[30] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proc. EuroSys*, 2010.

[31] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Proc. ICS*, pages 85–92, July 1998.

[32] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. of the 36th MICRO*, Dec. 2003.

[33] M. Laux. Solaris processor sets made easy. `http://developers.sun.com/solaris/articles/solaris_processor.html`, 2001.

[34] C. Letavec and J. Ruggiero. The n-queens problem. *INFORMS Transactions on Education*, 2(3), 2002.

[35] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proc. of SC2007*, Nov. 2007.

[36] E. Lübbers and M. Platzner. Reconos: Multithreaded programming for reconfigurable computers. *ACM Trans. Embed. Comput. Syst.*, 9, October 2009.

[37] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *Proc. of the 1998 ICS*, pages 77–84, July 1998.

[38] Y. Luo, V. Packirisamy, W.-C. Hsu, and A. Zhai. Energy efficient speculative threads: dynamic thread allocation in same-isa heterogeneous multicore systems. In *Proc. 19th PACT*, pages 453–464, 2010.

[39] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, , R. Russell, D. Sarma, and M. Soni. Read-copy update. In *Proc. of the Ottawa Linux Symposium*, July 2001.

[40] Microsoft Corp. New NUMA Support with Windows Server 2008 R2 and Windows 7. `http://archive.msdn.microsoft.com/64plusLP`, 2008.

[41] Microsoft Corp. Processor power management in windows 7 windows server 2008 r2. `http://download.microsoft.com/download/3/0/2/3027D574-C433-412A-A8B6-5E0A75D5B237/ProcPowerMgmtWin7.docx`, Jan. 2010.

[42] J. C. Mogul, J. Mudigonda, N. L. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-isa cmps to save energy on operating systems. *IEEE Micro*, 28(3):26–41, 2008.

[43] Z. Mwaikambo, R. Russell, A. Raj, and J. Schopp. Linux kernel hotplug CPU support. In *Proc. of the Ottawa Linux Symposium*, pages 181–194, 2004.

[44] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proc. 8th PACT*, Oct. 1999.

[45] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. of the 3rd ICDCS*, pages 22–30, 1982.

[46] R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughput computing. In *Proc. HPDC*, pages 140 –146, July 1998.

[47] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. of the 27th ISCA*, pages 25–36, June 2000.

[48] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proc. EuroSys*, 2010.

[49] J. C. Saez, D. Shelepov, A. Fedorova, and M. Prieto. Leveraging workload diversity through os scheduling to maximize performance on single-isa heterogeneous multicore systems. *J. Parallel Distrib. Comput.*, 71:114–131, January 2011.

[50] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proc. of the 30th ISCA*, pages 422–433, June 2003.

[51] S. Siddha and V. Pallipadi. Chip multi processing aware Linux kernel scheduler. In *Proc. of the Ottawa Linux Symposium*, pages 337–348, 2006.

[52] S. P. Smith. Dynamic scheduling and resource management in heterogeneous computing environments with reconfigurable hardware. In *International Conference on Computer Design*, 2006.

[53] W. Stanek. Windows Server 2008 R2: A primer. `http://technet.microsoft.com/en-us/magazine/ee677582.aspx`, Nov. 2009.

[54] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Trans. Computers*, 53(11), Nov. 2004.

[55] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proc. 14th ASPLOS*, 2009.

[56] Texas Instruments. OMAP 5 platform. `http://www.ti.com/ww/en/omap/omap5/omap5-OMAP5430.html`, 2011.

[57] P. Wells, K. Chakraborty, and G. Sohi. Mixed-mode multicore reliability. In *Proc. of the 14th ASPLOS*, Mar. 2009.

[58] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *Proc. USENIX ATC*, 2009.