Dynamic Processors Demand Dynamic Operating Systems

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

Abstract

The rise of multicore processors has lead to techniques that dynamically vary the set and characteristics of cores or threads available to the operating system. For example, Core Fusion merges multiple cores for faster processing. While the mechanics of the change, such as merging two cores into a more powerful core, can be handled by a virtualization layer, operating systems still have an interest in the exact set of cores available. For example, the read-copy-update mechanism in Linux must contact all cores to complete an update. Thus, the OS must be informed when the set of CPUs changes. We demonstrate through an analysis of a recent Linux kernel that (i) there are over 15 subsystems - each subsystem can have multiple callbacks registered – that depend on knowing the set of cores, and (ii) the existing *hotplug* mechanism is poorly suited to handling dynamic processors due to its poor performance and scalability. Based on this analysis, we propose two mechanisms, *processor proxies* and *parallel and deferred hotplug* to provide low-latency reconfiguration. In initial experiments, we show that we can reduce the latency of reconfiguration in Linux by 95 percent.

1 Introduction

The rise of multicore processors has lead to many proposals that dynamically vary the set of processors available. For example, with Core Fusion [11], speculative multi-threading [7, 6, 19, 12, 14], and speculative subordinate microthreading [3], two or more processors or thread contexts combine to improve performance. Similar techniques combine multiple contexts dynamically to improve reliability through redundant execution [1, 16].

Furthermore, other trends may also lead to a varying set of processors. With virtualization, the hypervisor may dynamically change the number of virtual processors available to an operating system as workloads change [24, 23]. Current Intel Nehalem processors can boost the performance of one core by disabling or *parking* other cores [4, 10].

The critical feature of these architectures is that *the number of physical execution contexts may change dynamically*. As a result, operating systems built with the assumption that the set of processors is static or changes only slowly will not be able to leverage these hardware features.

We argue that to support these future processors, operating systems must become much more dynamic in how they treat the set of processors. Existing hotplug mechanisms assume reconfigurations are rare, and that when they occur, they are fairly permanent. In a dynamic system, though, processors may disappear temporarily when running a single task, and return soon after. Rather than assuming that the set of processors is static most of the time, operating systems should accept that it might change.

We present a case study of the Linux kernel to demonstrate how a modern operating system depends on the set of available processors. We find that there are at least 15 subsystems with 35 callbacks that must be notified every time the available processors change. Furthermore, we analyze the existing hotplug mechanism and find that it takes over 100 ms to add and 25 ms remove a CPU, and that latency increases as more processors are added or removed. Thus, the existing mechanism is not suitable to support dynamically reconfigurable processors.

Based on this analysis, we propose two changes to how operating systems manage physical processors. First, we propose a new abstraction, the *processor proxy*, which is a virtual stand-in for a processor. When a processor that has been disabled is needed for a global operation, a virtual stand-in takes over its role, responding to messages on its behalf. From the perspective of other processors, the disabled processor is thus still available. Second, we propose modifications to the hotplug mechanism to *parallelize* hotplug, allowing multiple cores to be added or removed at a time, and to *defer* notification of processor-set changes, to avoid reconfiguring when the change is only transient. With these changes, processors can be stopped or started on the timescale of a single scheduling quantum, allowing for more reliable, higher performing, and efficient systems.

In the remainder of this paper, we first discuss motivating hardware features and analyze the existing Linux

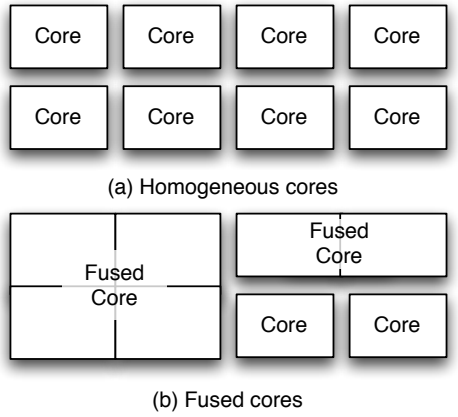

Figure 1: **Example dynamically reconfiguration. The homogeneous cores shown in (a) can be reconfigured, for example into a 4-core unit and a 2-core unit shown in (b).**

mechanisms for adding and removing processors. We then discuss the design of our proposed changes, followed by a brief discussion of our initial implementation efforts.

2 Motivation

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 Dynamically Reconfigurable Processors

Most current processors are packaged as multicore chips, with multiple processor cores on a single package and one or more threads per core. Despite the fixed number of cores in a processor socket, we see at least four reasons why the number of CPUs exposed to an operating system may vary.

First, many researchers have demonstrated single-thread performance increases from combining several processing cores into a single more powerful processing element. Core Fusion increases performance by combining core resources, such as functional units, into a larger core that can achieve higher ILP [11]. Similarly, speculative multi-threading executes loop iterations or function calls in parallel [8, 22]. Slipstream processing [9] improves performance through prefetching. These techniques aid the mountains of single-threaded code left behind by multicore processors.

Figure 1 shows an example of these architectures. Part (a) shows a homogeneous multicore system, and part (b) shows how cores can be fused together to act as more powerful cores. This is representative of core fusion and speculative multithreading.

Second, 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, 20, 25]. When surplus cores are available these techniques promise inexpensive error detection and fault tolerance.

Third, future processors may be *over provisioned*, in that they may contain more processing elements than can be simultaneously powered on [2]. Furthermore, processors may disable cores to save power or to transfer power to the remaining cores [4]. 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.

Finally, virtualization may lead to varying numbers of virtual processors. When competing workloads are light, a virtual machine could use more virtual processors to get more work done, and when workloads are heavy, drop down to a single processor.

These mechanisms all vary the set of processors available to an operating system over time on short time scales. Thus, operating systems must support changing the number of available processing cores. While virtual machines can implement reconfiguration [1, 27], we next show that operating systems are dependent on knowing the set of available processors.

2.2 OS Dependence on Processors

The challenge arising when the set of available processors changes is that many parts of modern operating systems depend on either the identity of a particular processor or require cooperation of all or most processors.

We examined the Linux 2.6.31-4 kernel to discover the extent to which it depends on knowing the current set of processors. We examine three facets of the Linux: (i) what portions of the OS have per-CPU data structures, (ii) what portions of the OS must be notified when the set of processors change, and (iii) how frequent are operations that require all processors? These three factors determine the cost and complexity of dynamically changing the set of available processors. We answer these questions by examining the Linux kernel source code and by dynamically measuring its behavior on a 2.5 GHz Intel Core2-Quad system running a mix of Unix command line utilities.

Per-CPU Data Structures. A subsystem may declare a variable to be *per-CPU*, meaning that it has a different address and value on every CPU [5]. Current uses of per-CPU variables include counter variables, buffers, callback queues and run queues. We found 446 separate variables in Linux that are defined as per-CPU. The *arch* subdirectory defines 294 variables, which mostly refer to hardware

Subsystem	Callbacks
arch	7
kernel	5
memory manager	5
scheduler	3
lib	2
net	2
rcu	2
timer	2
fs	1
ring buffer	1
workqueue	1
cpuset	1
block	1
acpi	1
topology	1
Total	**35**

Table 1: **Callbacks registered by each subsystem.**

structures. Of the remaining 149 variables, 70 are used in the core kernel routines, 32 in drivers, 16 in networking, 12 in memory management and 4 in file systems. If the number or set of processors change, these per-CPU data structures must be updated or initialized to reflect the change.

Processor Change Notification. A subsystem that wants to maintain state about different processors can request notifications through the *hotplug* mechanism when the set of processors change. When a processor is added or removed, the kernel calls all registered callback functions with the identity of the processor affected.

We profiled our system and recorded the set of functions the kernel invokes when a processor is added or removed. We found 35 callbacks split across 15 subsystems, shown in Table 1. Most of these callbacks are to the subsystems described above that store per-CPU data, although not all. These callbacks delay reconfiguration because the kernel must invoke many functions when reconfiguring.

All-processor Operations. Several operations within the Linux kernel require communication with all processors. For example, the read-copy update (RCU) mechanism [15] is used to update read-mostly global data structures. It ensures quiescence of data structure update by making the update only after all the CPUs have undergone at least one context switch. TLB shootdowns similarly require communication with all processors on which an address space has been mapped. To gauge the frequency of global operations, we measured the frequency of RCU operations. They occur approximately 90 times per second per processor. Thus, if the set of CPUs changes, these RCU operations could be delayed.

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 has quiesced, either delaying these operations or blocking frequent reconfiguration.

3 Hotplug

Operating systems currently support the dynamic change of processors through a *hotplug* mechanism. On Windows, it is only possible to add processor or replace processor, but not remove them [21]. Solaris and Linux support both adding and removing of processors dynamically [13, 17].

Hotplug targets 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.

Hotplug Implementation. The Linux hotplug implementation provides two operations accessible from the command line: offlining a processor and onlining a processor. To offline a processor, the kernel follows the steps in Table 2. When a CPU is brought online, the process is reversed. The key features of this implementation are that: (i) it is a lengthy, time-consuming process, (ii) it requires quiescing the system globally to update the mask of available CPUs, and (iii) it assumes that processors, once offlined, never come online again by unnecessarily removing all references to them.

However, in the case of dynamically reconfigurable processors, these three features make hotplug inappropriate. First, reconfiguration could occur frequently, for example every time a process is scheduled. Thus, a slow hotplug process could prevent performance gains from reconfiguration. Second, hotplug requires quiescing the system, which becomes expensive as the number of processors grows. Furthermore, it prohibits separate portions of the chip reconfiguring simultaneously, again limiting scalability. Finally, a reconfigurable processor may only temporarily disable a core, so hotplug performs unnecessary cleanup and initialization.

Hotplug Performance. We measure the performance of Linux hotplug by adding or removing one or more processors on a four-core system while no applications are run-

1.	Grab a global lock to serialize hotplug events.
2.	Check correctness (at least one CPU remains, etc.).
3.	Notify interested subsystems a CPU is going down.
4.	Migrate processes, interrupts, timers, bottom halves, and tasklets away from the outgoing CPU.
5.	Schedule a separate thread on every processor in the system to disable interrupts and freeze the system. While the system is frozen, take the CPU out of `cpu_online_mask`.
6.	Disables interrupts for the CPU and cleanup processor-related state.
7.	Put outgoing CPU in idle to prevent other tasks from being scheduled on it.
8.	Disable/halt the outgoing CPU.
9.	Notify interested subsystems that the CPU is offline.
10.	Release the global lock.

Table 2: **Steps to take a CPU offline.**

Hotplug Operation	Cores	Native (msec)	Par (msec)	Proxy (msec)
OFFLINE	1	25	25	1.7
	2	60	60	4
	3	137	130	6.5
ONLINE	1	106	106	1.2
	2	214	111	2.8
	3	331	131	6

Table 3: **Native hotplug, parallel hotplug, and proxy latency to take 1-3 CPUs offline and online.**

ning (a best case scenario). Table 3 shows the latency of offline and online operations in column 3. Overall, taking a processor offline is much faster than bringing one online, because it need not probe for processor characteristics or initialize hardware structures. In addition to these latencies, we measured that all other processors in the system are halted for 2.6 ms when offlining a processor. Thus, as systems get larger, a significant amount of time is lost on unrelated cores.

While the performance is very fast compared to the frequency of maintenance or virtualization events, it is too slow to be used when reconfiguring hardware for specific processes on a system.

4 Design

Operating systems gain great scalability benefits by their awareness of processors and per-CPU data structures. We seek to retain those benefits yet improve reconfiguration speed with two mechanisms:

1. *Processor proxies* stand in for processors when they are offline.
2. *Parallel and deferred* allow multiple simultaneous hotplug operations and to defer hotplug until a configuration is stable.

Processor proxies address short-term reconfiguration, while deferred and parallel hotplug reduces the frequency and latency of long-term reconfiguration operations. These are initial steps at fully supporting dynamic reconfiguration, which may take many additional steps.

4.1 Processor Proxies

We observe that much of overhead of hotplug operations is notifying other processors and initializing/cleaning up software data structures. However, if other processors are not notified, they may block waiting for global operations to complete.

We therefore create a container, called a *processor proxy*, for the OS data structures referring to a processor. When a physical CPU is unavailable, a processor proxy is created for it on another CPU and takes its place when necessary. The kernel moves its communication endpoints, such as interrupts, to that CPU. Any operations that require the presence of a core, such as a TLB shootdown or an RCU operation, can continue without waiting for the unavailable CPU.

Processor proxies are similar to multiplexing virtual CPUs (VCPUs) 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 general code. In contrast, a VCPU may run any code and forces the hypervisor to schedule or timeslice multiple CPUs on a single physical CPU. Furthermore, because the OS controls processor proxies, it can prevent preemption while holding spinlocks [26].

We add a new execution context to the OS, beyond process context and interrupt context, termed a *proxy context*. A 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 and performs the required operation. No threads are scheduled on a processor proxy.

The key challenge is ensuring that all the per-CPU state is available to code executing in a proxy context. Our design leverages the Linux implementation of per-CPU variables, which are accessed through segments on x86 processors. A segment in the GDT (Global Descriptor Table) points to the memory segment holding the per-CPU structures. Since each CPU has its own GDT, each processor contains a different GDT entry that points to its private segment. Linux loads this GDT entry into the FS register when in kernel mode, and access to per-CPU variable use segment-based addressing with this register. When switching to a processor proxy, we load the proxy's per-CPU segment into the proxying processor's GDT, and set

the FS register to the proxy's segment.

An additional complication arises in code making use of the `thread_info` variable, which refers to the CPU state of the task currently running on the processor. The kernel provides access to this structure through the stack pointer, which is not changed when executing in a proxy context. We modify the macro for accessing `thread_info` to redirect accesses from proxy context to the right data.

Processor proxies speed reconfiguration because they remove much of the work to change the set of processors. Instead, the major task is to reprogram the interrupt controller. To set up a processor proxy, interrupts must be routed from the existing processor to its proxy. These fall into two categories: device interrupts, which must be rerouted or redistributed to other online CPUs, and IPIs, which must be sent to the proxying CPU.

4.2 Parallel and Deferred Hotplug

Our second goal is to make hotplug more scalable when it is used. It is likely that reconfigurable processors will be reconfigured en masse, for example when four cores are fused into a single more powerful core.

We parallelize the implementation of hotplug by allowing multiple CPUs to change state simultaneously. The current implementation of the hotplug grabs a global lock while notifying subsystems of the change in processors, thereby serializing changes to the processor set. We parallelize this implementation by adding a new interface for adding removing a set of processors at once. This allows the local operations to initialize or cleanup a processor to proceed in parallel, and ensures that notification to subsystems occur once with a mask or processors, rather than once per CPU. When bringing a CPU online, this greatly improves performance because multiple CPUs can be initialized in parallel.

Deferred hotplug leverages processor proxies to only execute hotplug operations when a reconfiguration is long lasting. We modify the hotplug subsystem to allow remotely offlining a CPU, by executing on a separate physical CPU from the one taken offline. We accomplish this by changing the `take_cpu_down` function to take an additional parameter identifying the CPU to take offline. This is safe because `take_cpu_down`, which normally shuts down the CPU by disabling interrupts, now disables the proxy for the CPU instead, so it similarly does not receive interrupts.

The benefit of parallel and deferred offline is that transitions between configurations occur with lower latency and lower total overhead. In conjunction with processor proxies, these mechanisms support rapid reconfiguration that allows different processes to use unique processor configurations.

4.3 Preliminary Evaluation

We have begun the implementation of processor proxies and deferred/parallel hotplug in Linux. We have verified that processor proxies correctly delivery inter-processor interrupts, execute bottom halves, and allow RCU operations to complete while proxying a CPU. Furthermore, the system can run arbitrarily long while proxying.

We evaluate our changes with same configuration reported previously in Section 2. The results are shown in Table 3, columns 4 and 5. For parallel hotplug, we find a small benefit when taking multiple CPUs offline. For bringing CPUs online, though, we find there is little overhead to bringing multiple CPUs online at once, leading to a 60 percent reduction in time for the reasons noted earlier. Processor proxies provide much larger benefit, reducing time for reconfiguring a single processor by 95%. We find these results encouraging, but do not yet reach our goal of reconfiguring faster than a scheduling quantum. We continue to look at how other parts of the hotplug mechanism can be optimized for dynamically reconfigurable processors.

5 Conclusions

The rise of multicore processors leads 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 these mechanisms. We propose two mechanisms to make operating systems more dynamic: processor proxies to temporarily stand in for an unavailable processor, and parallel and deferred hotplug, to reduce the latency of reconfiguration operations. Furthermore, new affinity mechanisms, such as pair-wise affinity of threads [18] are also needed. With these mechanisms, the bulk of the operating system can remain unchanged while the set of physical CPUs available changes rapidly.

Acknowledgements

This work was supported by NSF Award CNS-0834473. Swift has a financial interest in Microsoft Corp.

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 Annual Intnl. Symp. on Computer Architecture*, June 2007.

[2] K. Chakraborty, P. Wells, and G. Sohi. A case for over-provisioned multicore system. Technical Re-

port UWCS TR1607, University of Wisconsin Technical Report, 2007.

[3] R. S. Chappel, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th ACM International Symposium on Computer Architecture*, pages 186–195, May 1999.

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

[5] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Associates, Feb. 2005.

[6] P. K. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single program speculative multithreading (SPSM) architecture: Compiler-assisted finegrained multithreading. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architec Compilation Techniques, PACT '95*, pages 109–121, June 1995.

[7] 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.

[8] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, Sept. 1997.

[9] K. Z. Ibrahim, G. T. Byrd, and E. Rotenberg. Slipstream execution mode for CMP-based multiprocessors. In *Proc. of the 9th IEEE Symp. on High-Performance Computer Architecture*, pages 179–190, Feb. 2003.

[10] Intel Corp. High-powered business results with lower power costs. `http://ipip.intel.com/go/wp-content/uploads/2009/05/21419_energy_efficient_wp_final.pdf`, 2009.

[11] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: Accomodating software diversity in chip multiprocessors. In *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture*, June 2007.

[12] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 85–92, July 1998.

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

[14] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *Proceedings of the 1998 International Conference on Supercomputing*, pages 77–84, July 1998.

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

[16] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and implementation of redundant multhreading alternatives. In *Proc. of the 29th Annual Intnl. Symp. on Computer Architecture*, pages 99–110, May 2002.

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

[18] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proc. of the 22nd ACM Symp. on Operating System Principles*, Oct. 2009.

[19] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, Oct. 1999.

[20] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. of the 27th Annual Intnl. Symp. on Computer Architecture*, pages 25–36, June 2000.

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

[22] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proc. of the 27th Annual Intnl. Symp. on Computer Architecture*, June 2000.

[23] VMware Inc. VMware ESX server 2: Architecture and performance implications. `http://www.vmware.com/pdf/esx2_performance_implications.pdf`, 2005.

[24] C. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[25] P. Wells, K. Chakraborty, and G. Sohi. Mixed-mode multicore reliability. In *Proc. of the 14th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

[26] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proc. of the Intnl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2006.

[27] P. M. Wells, K. Chakraborty, and G. S. Sohi. Dynamic heterogeneity and the need for multicore virtualization. *ACM SIGOPS Operating Systems Review*, 43(2), Apr. 2009.