# *Configurable fine-grain protection for multicore processor virtualization*

# Configurable Fine-Grain Protection for Multicore Processor Virtualization

David Wentzlaff[1], Christopher J. Jackson[2], Patrick Griffin[3], and Anant Agarwal[2]
wentzlaf@princeton.edu, cjj@tilera.com, griffinp@google.com, agarwal@tilera.com
[1]Princeton University [2]Tilera Corp. [3]Google Inc.

## Abstract

*Multicore architectures, with their abundant on-chip resources, are effectively collections of systems-on-a-chip. The protection system for these architectures must support multiple concurrently executing operating systems (OSes) with different needs, and manage and protect the hardware's novel communication mechanisms and hardware features. Traditional protection systems are insufficient; they protect supervisor from user code, but typically do not protect one system from another, and only support fixed assignment of resources to protection levels. In this paper, we propose an alternative to traditional protection systems which we call configurable fine-grain protection (CFP). CFP enables the dynamic assignment of in-core resources to protection levels. We investigate how CFP enables different system software stacks to utilize the same configurable protection hardware, and how differing OSes can execute at the same time on a multicore processor with CFP. As illustration, we describe an implementation of CFP in a commercial multicore, the TILE64 processor.*

## 1. Introduction

Multicore computing systems have changed the face of what is achievable on a single chip. Instead of a chip being a unit in a larger system, or a single system being realized on a chip, one chip can now contain a collection of systems. Each of these systems can execute an independent operating system (OS) concurrently within a single multicore processor. One example of this is the integration of a server-farm on a chip. Another is a multicore cell phone where the GUI processor executes a full featured OS, while the baseband processor executes a very thin real-time OS. The ability to integrate multiple systems on a single chip levies new requirements on protection systems that manage and restrict access to multicore resources.

When dealing with collections of systems integrated on one chip, traditional protection systems break down. This is because traditional protection systems are concerned with protecting supervisory code from user code, but do not address peer-to-peer protection, protecting one system from another system. Traditional protection systems are concerned with temporally protecting resources, while in multicore systems, a protection system must both temporally protect and spatially isolate access to resources. Spatial isolation is the need to isolate different system software stacks concurrently executing on spatially disparate cores in a multicore system. Spatial isolation is especially important now that multicore systems have directly accessible networks connecting cores to other cores and cores to I/O devices.

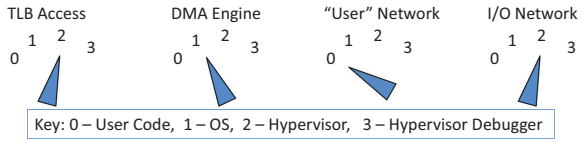


**Figure 1. With CFP, system software can dynamically set the privilege level needed to access each fine-grain processor resource.**

The spatial aspect of multicore systems mixed with the increase of silicon area afforded to multicore designers has fueled a resurgence of architectural feature innovation. Examples include in-core DMA engines, directly accessible networks, and co-processor accelerators. It is desirable that these new features not be available to arbitrary code, but rather that they can be restricted, virtualized, and multiplexed between multiple OSes executing on a single chip.

Multicore computing chips are a fabric for desktop OSes, embedded OSes, real-time OSes, and very thin runtimes. While full-featured OSes share similar protection requirements, the diversity of multicore OSes share fewer commonalities. Therefore, multicore processors require configurability in their protection systems to be able to execute a diverse set of OSes. This configurability becomes even more important when executing a real-time system where emulation can detrimentally affect real-time performance.

We introduce Configurable Fine-Grain Protection (CFP) as a mechanism to control access to different on-chip resources. *CFP is a unified hardware mechanism which enables dynamic modification of the protection level required to access a particular hardware resource. In CFP, the protection level needed to access each protectable hardware resource is individually configurable per resource.* Resources can be grouped by similar function into protection groups. CFP is configured dynamically by software, which lies in stark contrast to the traditional fixed assignment of features

to protection levels. *The ability to change a hardware resource's protection level allows differing protection models to be created within a single chip.* CFP is simple to implement in hardware as it only requires a few bits per resource group being protected. The protection level needed to access a particular hardware resource or set of resources can be set in CFP much in the same way that a knob can be turned. Figure 1 demonstrates how the protection level needed to access four different hardware resources, TLB access, DMA Engine access, User Network access, and I/O Network access can be each individually set. In this example, user code can access the user network, the operating system can access the DMA engine, and the TLB and I/O network is reserved for use by the hypervisor.

We illustrate these concepts though the implementation of CFP in the Tilera TILE64 processor, a homogeneous general-purpose multicore with 64 cores. This processor contains all of the protection features to protect cores from each other, and to run spatially and/or temporally multiplexed operating systems in a safe protected environment; it protects 48 groups of resources using CFP. We discuss the implementation overhead of CFP and the unique system software opportunities that it affords.

# 2. Protection Challenges

In this section, we begin with an overview of the requirements that multicore puts on protection systems. We discuss different resources that need to be protected and challenges which are introduced specifically by integrating many cores onto one piece of silicon. This section then describes previous protection systems and places Configurable Fine-Grain Protection (CFP) in the space of possible protection systems. Finally we categorize how each of the different protection systems address the challenges placed by multicore.

## 2.1. Requirements on Protection Systems

The requirements in building a fully self-virtualizable [19] multicore processor architecture extend the requirements placed onto protection systems by uniprocessor system software. First, the system needs to have differing classes of software executing on it and some manner to restrict access to resources to certain classes of software. There also needs to be a way for these different classes of software to safely communicate. Safe communication means passing information between different classes of software without allowing the privileges of one class to be passed to another.

Uniprocessor systems are primarily concerned with protecting access to memory, I/O devices, in-processor state, and special instructions which modify processor state. Multiprocessor systems have extended these uniprocessor requirements with the need to restrict communication between any two processors in a single system. Architectures have also become more sophisticated in the types of resources that need to be protected. For instance, multicore architectures have seen the integration of in-core performance counters, debug registers, cryptography units (e.g., Niagara 2 [16]), and DMA engines (e.g., TILE64 [27]).

### 2.1.1. Spatial Isolation

Multicore and multiprocessor systems have extended uniprocessor protection requirements even further due to the need to integrate multiple system software stacks on a single system. One example of this can be seen in the server consolidation space where multiple machines are aggregated and put into one physical computer or chip. Software virtualization [26] and hybrid hardware-software schemes [9] have been used to address this challenge.

Multicores and multiprocessors such as TRIPS [20], Raw [24], the Transputer [28], the J machine [18], and Alewife [2] have introduced architecturally-exposed, direct communication channels. Several of these architectures go so far as to map communication networks into the processor register space thereby increasing performance of network communication. This introduces the challenge of *spatial isolation*. Spatial isolation is the need to isolate different system software stacks executing on different portions of a multiprocessor or multicore system. What makes this unique is that memory protection alone is not sufficient to restrict different cores communicating via direct messaging networks. For many of these systems, the spatial nature of the communication is important; for example, if cores are connected in a 2D grid, then the location of communicating processes is critical. Directly accessible networks may also be used to connect to I/O devices. Finally, the destination for messages on these networks are not always known at message insertion time; Transputer channels and the Raw static network have this property. For these reasons, spatial isolation needs to be able to cut a multicore chip into multiple isolatable domains or regions.

### 2.1.2. Diverse Fine-Grain Resource Control

Multicore systems are opening up diverse new fields which are taxing traditional protection systems. This is because multicore systems enable the integration of not only systems on a single chip, but *systems of systems* which were never envisioned to share one piece of silicon. For example, on a multicore cell phone, different OSes may run concurrently on a single chip: an application processor may run Android Linux while the baseband processor is executing an embedded OS. The Linux instance wants all of the protections afforded by modern protection systems. However, the baseband processor may not; it may want its memory to be protected from the Linux instance, while still receiving direct access to on-chip networks and I/O devices. A protection system that allows fine-grain control over resources can present each system integrated onto the chip with the hardware view optimal for its OS.

| | Binary Translation | Rings (Machine Global) | Rings (Per Core) | Rings (Per Core with fine-grain MMIO) | Capabilities | CFP |
|---|---|---|---|---|---|---|
| Hardware Cost (Rank) | None (0) | Inexpensive (1) | Inexpensive (2) | Moderate (3) | Expensive (4) | Extra Hardware vs. Rings (3) |
| Protect Memory | Yes; slow | Yes; with MMU | Yes; with MMU | Yes; with MMU | Yes; Capability/word. Expensive HW | Yes; With MMU |
| Protect I/O | Yes; slow | Yes; with MMU | Yes; with MMU | Yes; with MMU | Yes; Capability protects | Yes; Memory Mapped and network connected I/O |
| Protect In-core Widgets | Yes; slow | Yes; at fixed Ring | Yes; at fixed Ring | Only memory mapped | Yes; Capability protects | Yes |
| Spatial Isolation | Yes; slow | No | No | Only memory mapped | Yes | Yes; memory and register mapped networks |
| Multiple OSes | Yes; slow | No | Yes | Yes | Yes | Yes |
| Multiple OSes different protection requirements | Yes; slow | No | No | Yes; but only memory mapped resources | Yes; Capability protects | Yes |

**Table 1. Requirements of Multicore protection and how different protection systems address them.**

## 2.2. Protection Mechanisms

In this section we provide an overview of different protection systems and finish by building Table 1 which shows how each protection system addresses or fails to address the protection requirements set out by multicore systems. Further comparison with specific implementations of protection systems can be found in Section 7.

### 2.2.1. Binary Translation

Dynamic binary translation systems such as VMWare [26] can provide protection solely through software rewriting and sandboxing. The downside of these systems is that this rewriting introduces extra complexity, extra resources in terms of memory usage, and the performance of rewriting all software can be slow, especially if protection checks are costly or happen frequently.

### 2.2.2. Rings

The notion of Rings was introduced in Multics [21] as strict hierarchical sets of privilege. Every resource that a less privileged class of software can access, a more privileged class of software can access. Early implementations had 8 hardware rings and we see Rings today in our desktop processors with original x86 processors having 4 rings while x86 virtualization hardware [3, 13] has added additional rings. Hardware resources are statically designated by the machine architect at machine fabrication and design time to reside in a certain ring. Rings can be efficiently implemented in hardware with only a few bits of storage and simple compare logic. If a resource has been placed in an incorrect ring or a system requires more rings than are provided by hardware (virtualization), a *trap-and-emulate* approach can be taken [19]. In trap-and-emulate, a trap is taken to a more privileged ring if a less privileged ring attempts to access a resource that the less privileged ring is not allowed access to. The more privileged code can then decide to emulate the functionality or refuse access depending on software policies. Trap-and-emulate can become a performance impediment if the emulation occurs often. An example of where this can effect performance is if two resources are statically assigned to a particular ring but these two resources should have been assigned to different rings. Using the traditional trap-and-emulate model, software that needs to access one, but not the other resource, must be executed in a less privileged ring. In this case, it is not only access to the restricted resource which must be trap-and-emulated, but also access to the unrestricted resource thereby penalizing its performance.

The base model of Rings is that a ring is machine global. This can cause problems when multiple system stacks share a single machine. An example is server aggregation where two copies of Linux are executing both at a fixed ring, but the two Linux operating systems do not trust each other. To solve this problem, Rings with per core privilege levels were introduced for multiprocessor systems. Rings with per core privileges are still inflexible with respect to assigning a resource to a ring level and are not able to address the spatial isolation requirements for directly connected networks or the fine-grain resource control requirements.

Rings can be extended to have some more fine-grain control by using memory mapped I/O to protect different resources. Rings with MMIO can only be as fine-grain as a page size, utilizes a TLB entry per resource being controlled, cannot protect resources which are not accessed via memory (such as register mapped resources or resources which need to be accessed quicker than a memory access), and by definition cannot be used to restrict access to memory structures such as TLBs and memory control registers. Examples of fine grain MMIO control that allow for some degree of spatial isolation include IBM's Logical Partitions (LPARs) [12] and SPARC's Logical Domains (LDoms) [29].

### 2.2.3. Capabilities

A capability is a communicable token which authorizes the holder of the token access to a resource. When a piece of code attempts to access a resource, it either explicitly or implicitly presents the token. Hardware or firmware validates the access. Capabilities can provide very fine-grain access and have been used to restrict access down to a particular byte in memory or hardware resource. Capabilities [10] were introduced as a software concept in the Multics project and later transitioned into hardware realizations in the System/38 [5], AS400 [22], and i432 [7]. Capabilities in the context of hardware protection systems are typically

non-modifiable and are managed by hardware or firmware. While capabilities are very flexible and provide fine-grain access, the storage overhead of maintaining capabilities can be high. Also, the added hardware complexity of capability checking can be expensive.

#### 2.2.4. Configurable Fine-Grain Protection

CFP provides the fine-grain protection feature of capabilities with the low hardware overhead of rings. It is effectively a ring architecture where resources can be moved between different rings. CFP also allows for configurable access to resources which are not memory mapped. This ability enables spatial isolation by protecting access to register mapped networks. The fine-grain control of CFP enables specific network links to be disabled thereby spatially isolating cores from other cores and cores from I/O devices. Finally, CFP allows the protection level needed to access different fine-grain resources to change, thereby enabling different OS models to execute on one multicore processor.

In Table 1 we compare these different protection systems and weigh the relative performance, hardware cost, and problems they solve.

# 3. Configurable Fine Grain Protection

The key characteristic of a CFP system is that it enables efficient and dynamic management of fine-grain groups of hardware resources by differing software components. The assignment of management for different hardware features does not need to be determined at silicon fabrication time, nor does the assignment need to be uniform across a single multicore processor. In this section, we describe how software components can be divided into classes, and how permission to access various resources is given to those classes.

## 3.1. Protection Levels

A CFP system supports multiple Protection Levels, or PLs. At any given time, each independent execution entity is associated with one of those PLs; this is termed its Current Protection Level, or CPL. The CPL controls what the entity is currently allowed to do. PLs form an ordered hierarchy or rings[21] of access rights. An entity whose CPL is $x$ has all of the powers of one whose CPL is $x - 1$, and may have others that the latter does not have.

Each processor in a multicore system is an independent entity and thus has its own CPL. However, there may be other entities within the system. For instance, in TILE64, each core contains a DMA engine, which is controlled by but runs asynchronously from that core's main processor. It would be undesirable for the DMA engine to gain enhanced privileges when the CPL of the associated main processor was elevated; each DMA engine therefore has its own CPL, separate from that of the main processor.

An entity's CPL changes over time. For the main processor in a core, a PL is typically associated with each level of the system software stack, and the PL changes depending upon where in the stack the processor is currently executing. A processor's CPL may be increased when an interrupt occurs, and may be decreased when interrupt processing is complete. An example mapping of 4 PLs to software components follows:

PL 0 Application program
PL 1 Operating system (Supervisor)
PL 2 Hypervisor
PL 3 Hypervisor debugger

| Resource | MPL | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| DMA | ✓ | | | |
| Network 1 | | | | ✓ |
| Network 2 | | | ✓ | |
| HW Accelerator | | ✓ | | |
| TLB | | | | ✓ |

**Figure 2. Resources with configurable MPLs.**

## 3.2. Minimum Protection Levels

A key characteristic of CFP is providing fine-grain access control to a large number of resources. In order to provide this fine-grain control, each hardware resource is associated with a separate Minimum Protection Level, or MPL. Figure 2 shows example hardware resources and how they can be individually configured to require a different Minimum Protection Level. This table is a compact representation of how MPLs can be set like knobs in Figure 1. Depending upon the resource, the per-resource MPL has some or all of the following functions:

1. It may control access to a resource. Any entity with a CPL greater than or equal to a resource's MPL is allowed to use that resource. Examples include accessing a special purpose register or accessing a register-mapped network.
2. It influences the PL at which the interrupt handler associated with the resource runs. When an interrupt occurs, the core's main processor vectors to the interrupt specific handler at the maximum of the MPL of the associated resource and the CPL of the executing core. Therefore the PL may or may not be elevated.
3. It may control the level of access given to an entity which is not a processor (for example, the MPL for the DMA engine corresponds to the PL that the DMA engine is running at).

One major issue with managing such a flexible system is configuring the MPL registers themselves. The design of MPLs must prevent privilege escalation. To prevent privilege escalation, a MPL can only be modified if the CPL of the processor is greater or equal to the MPL. Also, to prevent delegating to a more privileged PL which may not be expecting the particular interrupt connected with a MPL, MPLs cannot be raised higher than the CPL of the processor. Therefore, the processor may delegate control of a particular resource to a lower protection level, and may later

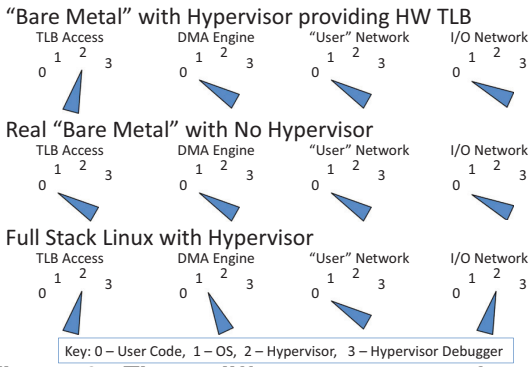


**Figure 3. Three different system software stacks require MPLs to be set differently for different resources.**

revoke that control, but it cannot relinquish access to the resource by raising its MPL above the CPL. Relinquishing control can only be done via an interrupt/system call to a higher PL. Also, this system prevents the PL of a non-main-processor entity, like a DMA engine, from having its MPL set higher than the CPL of the main processor.

The final challenge with MPL configuration is how to manage access to the registers which themselves configure the MPLs. This can be accomplished through the use of one-hot control registers which are associated with both a resource, and a target MPL for that resource; these are protected at a level which is the maximum of the resource's current MPL and that target MPL. For instance in a 4 level CFP system, there are 4 registers per MPL per resource. When a '1' is written to the second MPL configuration register, this sets the resource's MPL to 2. In order for software to write this register, the CPL must be at least the maximum of 2 and the resource's current MPL setting. This prevents unauthorized changing of the MPL, and prevents an MPL to be set to a level higher than the currently running code. By such a methodology, a privilege fixed-point can be reached.

Of course, not all MPL configurations make sense. It would be silly to give a PL access to the DMA engine TLB but not to the DMA engine itself, while the reverse might be quite plausible. CFP leaves it up to the system software to configure MPLs in a sensible manner.

### 3.3. Examples

Figure 3 shows examples of how different software stacks may require MPLs to be set to differing levels. The top example shows a bare metal use model where access to the TLB is still mediated by the hypervisor. This model can allow the hypervisor to provide a hardware page table abstraction to an OS on a machine with a software managed TLB; the hypervisor can also validate memory references in this configuration. In the second configuration, a full bare metal OS can access all of the resources of the machine. Finally, we see the configuration of a full stack Linux running on top of a hypervisor. Because MPL controls are per core, these three configurations can be concurrently executing on one multicore chip.

## 4. Interrupt Processing

In this section, we describe how the mechanisms which govern resource permissions in CFP also control operation of the interrupt facility. We describe how the interrupt facility allows transfer of control between software components while maintaining proper resource protection and how downcalls and upcalls can be made in a safe manner.

### 4.1. Interrupts

In a CFP system, each interrupt has an associated resource, and thus an associated MPL. An interrupt's MPL influences the PL at which the service routine executes; we term this the Target Protection Level, or TPL. It also controls the code which comprises that service routine.

#### 4.1.1. Exception Context and Interrupt Critical Section

When an interrupt occurs, the processor modifies certain state in order to execute the interrupt service routine. In order to eventually resume execution of the interrupted code, the original value of modified state must be saved somewhere. The Exception Context register is used for this purpose. Each PL has one exception context which is used to save the processor's interrupt state. The contents of the exception context are accessible via special-purpose registers (SPRs), which are protected by a fixed MPL equal to the PL for that context; this means that code running at a certain CPL can access its and lower PLs' exception contexts.

This leads to a minor problem, however. Since there is only one context per PL, and since an interrupt does not necessarily increase the CPL, it would be possible for another interrupt, targeted for a particular PL, to occur during an interrupt service routine's execution at that same PL. This second interrupt would cause the exception context for the first interrupt to be overwritten, thus preventing return to the originally interrupted code. One might set the interrupt mask in the first service routine to prevent further interrupts; however, there would be no way to guarantee that the instructions to mask the interrupt would execute before the second interrupt could happen.

To prevent this problem, a processor state bit called the Interrupt Critical Section (ICS) bit is used. When the processor is in an ICS, maskable interrupts are deferred. ICS is automatically entered at the start of every interrupt service routine, and may also be explicitly entered or exited by the processor via access to a special-purpose register. Since interrupt service always modifies ICS state, the ICS state is saved as part of the exception context state thereby enabling interrupt nesting. Short interrupt routines can run entirely within an ICS. For long interrupt routines, it is advisable to save the exception context onto the stack and re-enable interrupts by lowering the ICS bit. Before a return from interrupt, the ICS bit should be re-enabled to prevent an in-

terrupt from occurring during exception context refill. If a non-maskable interrupt is fired while in an ICS, the Double Fault handler is entered; this can safely crash the system or dump debugging information.

#### 4.1.2. Detailed Interrupt Processing

Conceptually, before completing the execution of every instruction, and committing its results, the processor performs the following algorithm; if an interrupt is actually chosen to be serviced, the results of the instruction are not committed.

```
for interrupt in highest_priority to
   lowest_priority:
 if interrupt is asserted:
   tpl = max(cpl, mpl[interrupt])
   if interrupt is maskable and
       interrupt_mask[tpl] contains interrupt:
     continue
   if tpl == cpl and processor is in ics:
     if interrupt is implicitly masked by ics:
       continue
     else:
       interrupt = DOUBLE_FAULT
       tpl = max(cpl, mpl[DOUBLE_FAULT])
   if interrupt != DOUBLE_FAULT or tpl != cpl:
     exception_context[tpl] = (pc, cpl, ics)
   pc = vector_address[tpl, interrupt]
   cpl = tpl
   ics = true
   break
```

Upon execution of an `iret` instruction, the processor does the following:

```
(pc, cpl, ics) = exception_context[cpl]
```

### 4.2. Upcalls

Software components request services from other software components. Depending upon the components, these requests are often hierarchical; an application may request services from an operating system or supervisor, the supervisor may request services from a hypervisor, etc. In a CFP environment this would mean that those requests are flowing from lower to higher protection levels.

In some cases a request is implicit; for instance, an application might access a virtual address which had not been mapped into its address space. This would cause a TLB Miss interrupt, and the supervisor might, in response, allocate an empty page of physical memory and then map it into the address space of the user process. In many cases, though, the request needs to be made explicitly.

This is done via dedicated software interrupt (`swint`) instructions, which, when executed, trigger a corresponding software interrupt. While the number of `swint` instructions matches the number of PLs, there is no fixed association between the two; like all resources, each software interrupt has a separate, changeable MPL. Often, a supervisor or hypervisor assigns a `swint` to its PL, and uses it to provide a system call service to lower PLs.

Unlike other resources, access to software interrupt instructions is not protected; their assigned MPLs serve only to determine the target PL for the interrupt service routine. If a software component wishes to restrict access to a software interrupt which it provides, it may do so by checking the CPL saved in the exception context and disallowing requests from specific lower PLs.

### 4.3. Downcalls

While increasing the CPL is the most common way to request a service, there are situations where you might want to instead decrease the CPL to accomplish a task. In effect, this delegates processing of an interrupt to code running at a lower protection level. For instance, a hypervisor might want to handle the Double Fault interrupt, to detect a faulty supervisor; however, if that interrupt were generated by an application program, it might want to allow the supervisor to handle it instead.

In many cases, this is quite easy to do. If an interrupt which occurs at CPL 0 is handled at CPL 2, and the interrupt service routine then decides that it should be handled at CPL 1 instead, it would perform the following algorithm, which we will term a downcall:

- Read the contents of exception context 2, and write it to exception context 1; this will be the PC of the interrupted code, a CPL of 0, and whatever the ICS state was at the time of the interrupt.
- Write exception context 2 with a PC of the desired interrupt handler in PL 1, a CPL of 1, and enable ICS.
- Execute an `iret` instruction.

After the return from interrupt, the PL 1 interrupt service routine is executed, and it begins with the exact same state it would have had if the interrupt had gone to it originally. When it is done, it returns from the interrupt and the originally executing code is resumed.

The tricky case comes when you want to delegate handling of an interrupt to the PL at which the interrupted code was running. For instance, a hypervisor might receive an I/O interrupt which it would like to delegate to a supervisor. If the interrupted code (supervisor) was not in an interrupt critical section, and the interrupt of interest is not masked at the delegatee PL, returning to the lower-level interrupt routine can be done as above. However, if the interrupted code was in an ICS, that procedure would destroy the state of the lower PL's exception context. Alternatively if the to-be-delegated interrupt is masked at the supervisor PL, then the interrupt should not be delivered until the interrupt is unmasked. The downcall mechanism was the most challenging part of the CFP design. In the end we came up with two solutions. One has potentially poor performance, but is transparent to the delegated-to code; the other provides good performance, but requires coordination from the delegated-to code.

The first solution is to temporarily virtualize the ICS state bit and the interrupt masks for the lower PL, by raising

the MPL which controls access to their associated special-purpose registers. This is akin to a more traditional trap-and-emulate approach. When the delegated-to code accesses the ICS or interrupt masks, an interrupt to the delegator occurs, which can emulate the instruction and determine if it is safe to deliver the pending interrupt. After the interrupt is delivered, the MPL of the interrupt masks and ICS can be lowered. This solution has the downside of possibly emulating all interrupt instructions, interrupt mask modification, and ICS modification for as long as any delegated interrupt is pending. Since a delegated interrupt may be pending forever, there is no guarantee that the system will ever leave this partial emulation state.

The second method requires the cooperation of the delegated-to code, but is somewhat easier to implement. With this method, the delegator arranges for the delegatee to receive a notification, via interrupt, that it should make a special system call to the delegator. Since the notification comes via interrupt, it will not be delivered until the delegatee exits the critical section. The delegatee's interrupt routine is then executed, and it makes the special service request to the higher PL. The higher delegator PL now knows that the delegatee is ready to receive an interrupt and it performs the original downcall without masking concerns.

To enable this second method, software-triggerable interrupts can be provided, each of which can be associated with a PL by setting an associated MPL register. Each interrupt is asserted when a corresponding special-purpose register is set to 1. This solution provides high performance and does not require emulation flows, but does require the cooperation of the delegatee PL. This cooperation is not a security hole because not requesting a delayed interrupt is the same as simply leaving the interrupt masked.

# 5. Applications Enabled by CFP

CFP enables flexible system-level software stacks. As described in the following sections, CFP resource assignment allows system software to optimize for fast resource access at a particular privilege level, or to optimize for debuggability, or for any other criteria. These resource management decisions can be changed dynamically and made independently by each core in the system.

## 5.1. Spatial Partitioning of Multiple OSes

The rapidly increasing level of integration found in multicore systems-on-a-chip has allowed designers to pull more and more hardware components into a single piece of silicon. Modern SoCs include generic processor engines, specialized accelerators, and highly-integrated I/O systems. Just as previously separate hardware components are being integrated on a single chip, we expect that multicore SoCs will pull together multiple types of system software into a single environment. In effect, multicores will have to support collections of systems, and not simply systems.

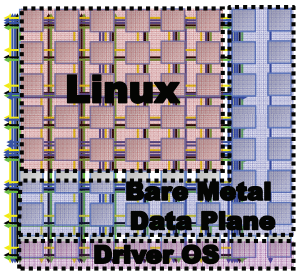


**Figure 4. Three OSes concurrently executing on a tiled multicore processor.**

For example, many networking applications are composed of data plane and control plane software. The data plane software typically runs on an embedded real-time OS, characterized by a minimal supervisor layer and direct application access to the hardware. The control plane, by contrast, is often implemented with a complete multi-tasking OS (e.g., Linux) that provides extensive system services and protected hardware access. In very high speed networking applications, additional cores may be dedicated to low-level network processing (driver cores). Figure 4 shows three OS domains concurrently executing on a tiled multicore.

Per-core resource protection allows system integrators to build interesting hybrid system software stacks. One of the key resources available in many multicores is a set of on-chip hardware networks that allow direct core-to-core communication without synchronizing through shared memory. In a CFP system, these networks can be used to implement communication between the data plane and the control plane. Interestingly, the communication resources can be protected differently in each environment. The control plane, which often implements preemptive multi-tasking, might choose to set the on-chip network MPL to the supervisor level, avoiding problems that would result from swapping out a user-level process while it is in the midst of issuing a packet to the on-chip network. Conversely, the data plane might allow direct hardware access by the application code in order to maximize performance. This ability to have differing protection models executing at the same time on a chip comes directly out of CFP. What makes CFP unique is that this level of protection system configurability can be done without the emulation cost associated with previous solutions. The emulation costs become particularly relevant when executing real-time OSes and applications which cannot tolerate timing jitter.

Interestingly, in our example, the spatial division of MPLs allows both pieces of the system to communicate with each other via an agreed protocol on the same hardware network, even though the network protection models on each side are completely different. *This sort of flexibility would not be possible in a system that provided only a single, global protection level for the on-chip network.*

## 5.2. Selective Virtualization

The protection of hardware resources in a CFP system is divided into many independent protection sets, each guarding a particular hardware resource. For example, in the TILE64 family of processors, each core has independent MPLs for guarding access to the TLBs, the DMA engine, and each inter-core communication network. This allows more privileged software to choose on a per-resource basis whether to: give less privileged software direct access to a hardware resource, allow virtualized access to that resource, or reserve access to that resource for itself.

As an example, selective virtualization can be used to optimize context-swap time. A common example of this is management of floating point units. Many OSes do not save the state of the floating point unit on each context-swap, but rather wait for the new process to access the unit before saving the previous context's state and loading in the new state. This requires that the hardware implementation allow the OS to temporarily disable the floating point unit, so that a fault will be generated when the user space program accesses the floating point unit, allowing the OS to complete the context-swap.

Previous architectures have provided special-purpose mechanisms for performing the above optimization on a limited set of resources. We expect that future multicore SoC implementations will have many more special-purpose hardware engines. CFP provides a fine-grain and uniform interface for independently protecting each of the many special-purpose hardware engines in multicore SoCs. This allows the system software to dynamically enable access to each resource as necessary, reducing context swap overhead while still allowing direct hardware access if desired.

## 5.3. Delegation of MPLs

The configurable aspect of CFP allows system software to conveniently pass resource protection decisions to lower layers of the software stack. For example, consider a three-layered system consisting of application, OS, and hypervisor. At system boot time, all MPLs are set to the hypervisor's protection level. As the hypervisor starts up the OS, it can choose to shift control of some resources to the OS by setting each resource's MPL to the supervisor privilege level. Similarly, the OS itself can choose to re-delegate certain resources to userspace by setting the resource's MPL to the user protection level. There is no need for the supervisor to ask the hypervisor to switch a resource's MPL down to user level; once the hypervisor gives the supervisor control of a resource the supervisor can re-delegate that control as it chooses.

## 5.4. Dynamic Modification of MPLs

Just as access to each resource can be delegated to lower-privilege system software, it can also be promoted back up to a higher level. This allows any more-privileged layer to regain control of a resource when needed. In CFP, each resource's MPL can be moved up or down dynamically.

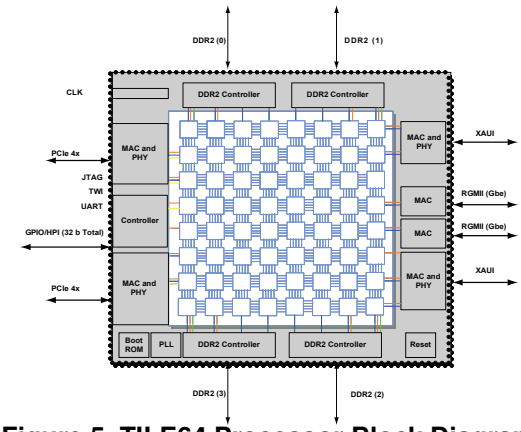


**Figure 5. TILE64 Processor Block Diagram.**

An interesting use of dynamic MPL modification is debugging direct hardware access by less-privileged system software. For example, many embedded applications choose to give direct hardware access to user-level programs in order to improve performance. User-level access allows the application code to control the hardware without having to make requests to more privileged software via syscalls or a virtualization interface. However, user-level access necessarily means that application writers are responsible for writing and debugging hardware drivers. Dynamic MPL modification can be used to selectively and temporarily fine-grain virtualize a resource, and therefore allow debugging or tracing the user-level driver's access to hardware without changes to the driver itself.

# 6. Hardware Realization and Results

## 6.1. Implementation Overview

As a vehicle to discuss multicore protection and as a first implementation of many of the ideas in this paper, we describe the multicore protection features of the TILE64 processor. The Tilera TILE64 processor consists of a two-dimensional grid of 64 identical processor cores. Each core is a full-featured computing system that can independently run an entire OS. Likewise, multiple cores can be used together to run a multi-processor OS such as SMP Linux. Also, different OSes can concurrently execute on TILE64 with different subsets of cores all in a protected and isolated manner. For instance, SMP Linux, VxWorks, and various customer-specific OSes have been ported to TILE64 and CFP allows these different OSes to execute simultaneously.

Figure 5 shows the 64-core TILE64 processor. The mesh networks in the TILE64 Processor connect to I/O and DDR-2 memory controllers. Each core combines a processor, cache, and switch. The switch implements five independent networks used for memory traffic (2 networks), I/O, user-mode messaging, and compiler controlled inter-core communication [27]. Each processor is a three-way VLIW pro-

cessor with independent program counter (PC). Inside each core, there is a two-level cache hierarchy, a DMA engine, and complete support for interrupts, protection and virtual memory. The TILE64 processor is the first implementation of CFP and contains 4 protection levels; there are 48 different categories of resources per core which are individually fine-grain protected under CFP.

## 6.2. Multiple Protection Levels

One of the key assumptions made with CFP is that it is desirable to have more than two protection levels. While CFP can be used to assign resources between two protection levels, its true power comes when there are even more levels of protection. TILE64 contains 4 protection levels.

The hardware cost of having additional protection levels lies not in the registers holding the current protection level (going from two to four costs one extra bit), but rather in the per-level additional state. The TILE64 processor duplicates the interrupt mask registers, with one per protection level. The TILE64 has 25 maskable interrupts. For each maskable interrupt, there is a 4-to-1 multiplexer based on the CPL to determine whether the interrupt is masked at a given protection level as shown in Figure 6a. Conveniently the timing cost of this multiplexing step can be pipelined as changing interrupt masks is a long latency operation, thus no effect on clock speed is realized.

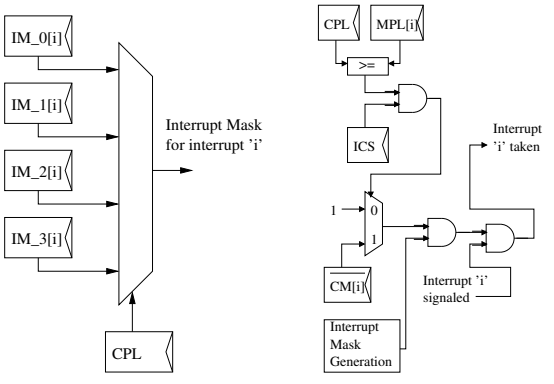


**Figure 6. a) Selecting which MPL's interrupt mask (IM) to utilize for an interrupt 'i'. b) Interrupt signaling logic for a particular maskable interrupt 'i'. CM[i] denotes whether an interrupt 'i' is critically masked inside of an interrupt critical section (ICS).**

The other state that needs to be duplicated on a per PL basis is the exception context state. This is the state used to save off the PC, CPL, and ICS bit. A multiplexer is also needed to choose which PC to return to on a return from interrupt instruction. This operation is uncommon, so its timing is non-critical.

## 6.3. CFP Implementation

CFP enables much flexibility in system software design and enables new ways to build protected collections of systems-on-a-chip for a very small hardware cost when compared to statically configured protection systems. The primary additional cost of CFP is the cost of all of the MPL bits. In TILE64, there are 48 differing MPLs, each of which can select between four protection levels which are base-two encoded. Thus, there are an additional 96 state holding elements in this design. In addition, there are comparators for each of the maskable interrupts which in parallel determine whether a particular interrupt will fire based on the MPL, CPL, ICS bit, and interrupt masks. Figure 6b shows the logic behind the interrupt masking decision.

When an interrupt occurs, the appropriate Target PL and appropriate interrupt must be determined. Multiple interrupts can be fired concurrently, and TILE64 contains a static ordering of interrupts, thus a priority encoder is required. This priority encoder is not used to determine whether an interrupt has occurred (only an OR tree is needed); it instead is used to determine the destination interrupt vector and target PL to transfer control flow to. A large priority encoder can take a long time to resolve, but it can be pipelined at the expense of increasing interrupt latency.

A multicore processor can have many special purpose registers (SPRs) controlling processor behavior. With CFP, each SPR is assigned to one MPL. In order to access the SPR without faulting, the CPL of the processor must be greater than or equal to the MPL. While it may seem challenging to validate access to hundreds of SPRs, judicious choice of the SPR namespace alleviates much of this problem. The top bits of the SPR number can determine which MPL protects a given SPR. Thus the protection check reduces to a multiplexer that chooses an MPL to compare against the CPL. The select line on the multiplexer is the top bits of the SPR number. From a timing perspective, the multiplexer needs to complete between the time that the instruction is decoded and the time for which interrupts are signaled, which can be several pipeline stages in a modern processor.

## 6.4. Network Protection

A key component of spatial isolation is network protection. The hardware cost of network protection is composed of two main components, limiting access to the networks and hardwalling [27] individual outbound network links. Each network contains an MPL which describes the minimum level needed for the processor to access the respective network. If the CPL is less than the MPL and a network access occurs, then a network access interrupt is taken. Unlike SPR protection checking, in TILE64, networks are register mapped. Because of this, a single instruction can access multiple networks; thus protection comparators are needed for each directly accessible network (3 in the case of the

TILE64). The cost of these comparators is small as it is three two-bit comparators, and the timing is the same as the SPR protection check timing.

## 6.5. Overall Area Impact

In this section, we describe the mechanisms and costs of implementing spatial protection and CFP. The overall cost is on the order of a hundred flip-flops, and comparators. In the TILE64 design, this protection logic is less than 0.5% of a core's area. We feel that this is small enough to justify adding CFP to more processors and in particular embedded multicore processors in the future. Also, the gains in term of debuggability and the ability to execute full OSes, multiple OSes, and OSes with differing protection needs far outweigh the area cost.

## 6.6. Performance Advantage of CFP

While much of the benefit of CFP is of a qualitative nature, it is worth examining the performance advantage of CFP compared to a trap-and-emulate approach. We investigate a parallelized 1D Jacobi relaxation problem which uses user-accessible direct networks to communicate values on three architectures. The first architecture is the TILE64 architecture utilizing CFP. The second architecture has dynamic networks, but they are protected at the supervisor level (ArchSup), and the third architecture has networks which are protected at the hypervisor level (ArchHyp).

We break down the cycles needed to accomplish a single user-mode network read or write on ArchSup assuming the interrupt and trap latencies of the TILE64. In the architecture where the dynamic networks are statically protected at the supervisor level (ArchSup), when a user process attempts to access the dynamic network, an interrupt is taken to the supervisor (7 cycles for interrupt plus 30 cycles to save register state). The supervisor then determines whether the running process has the right to access the dynamic network by checking a field in the process structure (two memory references, assuming one L1 hit and one L2 hit, 10 cycles, and a branch to test the access control 1 cycle). After validating the access, the OS reads the instruction from memory that caused the fault to determine the register being written to the network (8 cycles assuming instruction is in L2 cache, plus a compare with a constant, 2 cycles, bit extract, 1 cycle, and a register access, 1 cycle). It then writes/reads the value to the network and returns from interrupt to the subsequent instruction (1 cycle to write/read value from network, 30 cycles to restore saved registers, 3 cycles to increment the return program counter, and 7 cycles to return from interrupt). This total comes out to an aggressive 101 cycles. In all likelihood this will take longer either due to the trap handling code not being hot in the cache or a needed data value not being in the cache.

On the ArchHyp architecture, where access to an on-chip network is statically protected by the hypervisor, the performance overhead is a superset of ArchSup. The additions being that the hypervisor will do its own protection checks (7 cycles for interrupt plus 30 cycles to save away register state, two memory references, assuming one L1 hit and one L2 hit, 10 cycles, and a branch to test the access control, 1 cycle) and then downcall into the supervisor (30 cycles to restore saved registers, and 7 cycles to return from interrupt). In addition to the ArchSup case, after validation, the supervisor must call into the hypervisor to actually send/receive the message via a hypercall (7 cycles for interrupt plus 30 cycles to save away register state, 30 cycles to restore saved registers, and 7 cycles to return from interrupt to the supervisor). We very aggressively assume that the hypervisor and supervisor code is in the cache and the whole operation takes 260 cycles.

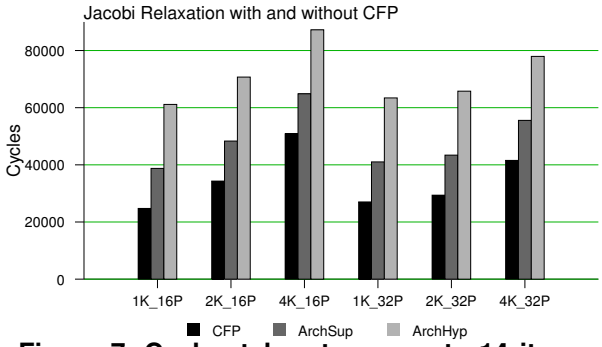


**Figure 7. Cycles taken to compute 14 iterations of a 1D Jacobi relaxation for 1024, 2048, and 4096 entries on 16 or 32 processors. Test run on TILE*Pro*64 with CFP, ArchSup with the dynamic network protected at the supervisor level, and ArchHyp with the dynamic network protected at the hypervisor level.**

We ran a one-dimensional Jacobi relaxation for 14 iterations for 1024 (1K), 2048 (2K), and 4096 (4K) entries on 16 (16P) or 32 (32P) processors on real TILE*Pro*64 hardware. This test used the user dynamic network to pass values between cores at the boundaries of a split Jacobi matrix. We ran on a 64 core TILE*Pro*64 development system running at 700MHz with 4GB of DDR2 RAM. The TILE*Pro*64 contains CFP support and does not trap when accessing the user dynamic network. To emulate ArchSup and ArchHyp, we ran on the same TILE*Pro*64 system, but introduced stalls when reading or writing of the network as described above. We used this conservative approximation approach because the TILE*Pro*64 already contains CFP hardware used by the production system software and there was no business need to create a hypervisor which would be slower and not use the CFP hardware. As can be seen on Figure 7, ArchSup is 28% to 57% worse performance than CFP and ArchHyp is 71% to 147% worse than CFP.

This study favored ArchSup and ArchHyp, but it is very possible that the performance of these nominal implementations could be significantly worse. One thing that could

affect the performance is if a resource which should be protected is put into a group with a resource which is commonplace to access. If such a situation occurs, the trap-and-emulate overheads could be felt not only when accessing the resource that needs to be protected, but also when other resources grouped in the wrong protection level are accessed.

In this test, we use access to the user dynamic network as an example of a resource which can be protected with CFP, but there are many other resources that may need fine-grain protection. In fact the TILE64 has 48 different fine-grain protectable protection groups. Other examples where applications may need fast access to fine-grain protectable resources include timers, in-core DMA engines, hardware accelerators, counters, profiling hardware, fast access to memory state such as LRU information, and memory watch point hardware. These resources are especially important for auto-tuners or JITs which want to test how an optimization affects performance without incurring trap overhead.

# 7. Related Work

Heterogeneous multicores such as those used in cell phone systems and multimedia consumer applications [25] are good examples of embedded systems containing multiple cores in a system-on-a-chip fashion. While many of these systems execute multiple OSes, the manner in which they are connected is typically ad hoc and governed by how they are wired together. CFP and the spatial partitioning of the TILE64 processor provides a structured approach to isolating multiple systems which share a single chip versus that of ad hoc embedded SoC design.

Many architectures contain protection systems with two or more protection levels. Having multiple protection levels arranged in a set of rings was originally proposed in the Multics system [10, 21]. Since that time, many systems have supported more than two levels of protection. The x86 architecture supports 4 protection levels [14]; Alpha [8] and MIPS [17] support 3. But unlike CFP, the allocation of resources to protection levels is fixed. We have seen a resurgence of additional protection levels to support virtualization with Intel's VT technology [13] and AMD's Pacifica [3].

Architectures such as SPARC [23] have the ability to disable the floating point unit. When access occurs, an interrupt is taken at the supervisor level. This type of hardware bears resemblance to CFP, but is only for one resource. Also, SPARC does not allow the reassignment of the FPU to a different protection level which CFP enables.

The use of a memory management unit to restrict access to memory mapped I/O (MMIO) devices can be used to restrict access to I/O devices much in the same manner that CFP can restrict access to in-core hardware resources. While similar, protection of MMIO has some disadvantages and differences when compared to CFP. The largest problem with MMIO is that it requires protection management to be handled by the piece of software (hypervisor) that is in control of the MMU. Because of this, any change to the protection levels of any hardware in the system require the hypervisor to get involved. In CFP, these changes can be taken care of by lower protection levels and delegated. Also, MMIO does not allow direct delegation of interrupts, therefore with MMIO any interrupt or protection violation must enter the hypervisor before being proxied to the relevant system software. CFP, in contrast, enables hardware to directly dispatch the interrupt to the relevant system software. Proxying through the hypervisor is especially problematic when a real-time system is involved because it can introduce an absolute performance problem along with additional performance jitter.

Protection of MMIO devices is typically used to protect out-of-core devices. The control mechanism for MMIO protection is different than CFP. MMIO utilizes loads and stores to access devices and is not able to use any other means, by definition. CFP can protect in-core registers which can be much faster to access. For example, a CFP protected SPR access in the TILE64 takes only two cycles, while MMIO needs an address translation to occur, and typically involves a core communicating with the I/O bridge. Because MMIO utilizes page based protection, the granularity of protection is quite coarse and removes address space from any process attempting to access MMIO devices while CFP does not. MMIO also pollutes the TLB with an entry per resource protected. Most TLBs are an associative structure, while CFP utilizes indexed structures to restrict access. Finally, MMIO cannot directly control the privilege level at which other in-core entities, such as a DMA engine, execute; CFP can.

An example of the MMIO approach to restricting access to resources include the IBM Power 5 [4], Power 6, Cell [11], and zSeries [9] through their use of MMIO to control logical partitions (LPARs). LPARs also spatially protect differing OSes via memory mapped isolation but do not solve the problem of protecting register mapped on-chip network isolation as presented in this paper.

Capability based machines provide much of the same fine-grain protection that CFP provides. In a capability system, typically a capability is stored in memory that is not modifiable without special privilege. This non-user-modifiable memory can then describe an action that the holder of that capability can perform such as accessing memory or other hardware. While capability based systems can provide much flexibility, their implementation can be expensive as most capability based systems require an extra protection bit for every word in main memory. They also require special hardware to access memory to validate a capability when it is used, and they typically require drastically different system software for effective use. Example

systems include System/38 [5], AS/400 [22], and i432 [7]. We feel that CFP provides many of the benefits of capability based systems in that the protection system can be modified to meet the needs of the system software without the cost and complexity of a full capability based system. In addition, CFP is designed to work with standard OSes.

Hypervisors such as VM/370 [9], VMWare [6, 26], Sun's xVM [29], and POWER's hypervisor [4] can run multiple protected OSes on a single multiprocessor system. The classic approach to virtualization takes a trap-and-emulate model. As discussed in Section 2, a trap-and-emulate model can cause performance overhead when a resource is shared across multiple OSes. CFP moves many of the occurrences where trap-and-emulate are necessary into hardware checked flows. With CFP, the hypervisor does not need to get involved when interrupts are destined for lower PL software or when MPLs need to be adjusted or delegated. Because resources can be split to such a fine-grain with CFP, the likelihood of device sharing is reduced.

Memory protection and management can be done not only in the processor, but also to restrict memory access made by I/O devices. Workstations and mainframes have long had I/O memory management units (IOMMUs), but we have seen this technology make its way to the PC market with the introduction of Intel's VT-d [15] and AMD's IOMMU [1] hardware. CFP is orthogonal to IOMMUs as CFP protects in-core resources and IOMMUs protect memory access done by I/O devices. The VT-d IOMMU approach along with page based MMIO access to I/O devices can allow x86 processors protected access to off-chip devices, but these techniques introduce large amounts of latency; this makes them unsuitable for use in protecting in core resources like inter-core on-chip networks.

# References


[1] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification*, 1.26 edition, 2009.

[2] A. Agarwal et al. APRIL: A Processor Architecture for Multiprocessing. In *Proc. Int. Symp. Computer Architecture*, pages 104–114, June 1990.

[3] AMD. *AMD64 Virtualization Codenamed Pacifica Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005.

[4] W. J. Armstrong et al. Advanced Virtualization Capabilities of POWER5 Systems. *IBM J. Res. Dev.*, 49(4/5):523–532, 2005.

[5] V. Berstis. Security and Protection of Data in the IBM System/38. In *Proc. 7th Ann. Symp. Computer Architecture*, pages 245–252. ACM, 1980.

[6] E. Bugnion et al. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proc. ACM Symp. Operating System Principles*, pages 143–156, 1997.

[7] R. P. Colwell, E. F. Gehringer, and E. D. Jensen. Performance Effects of Architectural Complexity in the Intel 432. *ACM Trans. Comput. Syst.*, 6(3):296–339, 1988.

[8] Compaq Computer Corporation. *Alpha Architecture Handbook*, fourth edition, Oct. 1998.

[9] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, pages 483–490, Sept. 1981.

[10] J. B. Dennis and E. C. V. Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3):143–155, 1966.

[11] M. Gschwind et al. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[12] C. Hales, C. Milsted, O. Stadler, and M. Vagmo. *PowerVM Virtualization on IBM System p: Introduction and Configuration*. IBM, fourth edition, May 2008.

[13] Intel Corporation. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, 2005.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, Nov. 2007.

[15] Intel Corporation. *Intel Virtualization Technology for Directed I/O*, version 1.2 edition, 2008.

[16] T. Johnson and U. Nawathe. An 8-core, 64-thread, 64-bit Power Efficient SPARC SOC (Niagara2). In *Proc. 2007 Int. Symp. Physical Design*, ISPD '07, pages 2–2. ACM, 2007.

[17] MIPS Technologies. *MIPS R4000 Microprocessor Users Manual*, second edition, 1994.

[18] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-machine Multicomputer: An Architectural Evaluation. *SIGARCH Comput. Archit. News*, 21(2):224–235, 1993.

[19] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[20] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proc. Int. Symp. Computer Architecture*, pages 422–433, June 2003.

[21] M. D. Schroeder and J. H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Commun. ACM*, 15(3):157–170, 1972.

[22] F. G. Soltis. *Inside the AS/400*. Duke Press, second edition, 1997.

[23] Sparc International, Inc. *The SPARC Architecture Manual*, version 8 edition, 1991.

[24] M. B. Taylor et al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the International Symposium on Computer Architecture*, pages 2–13, June 2004.

[25] S. Torii et al. A 600MIPS 120mW 70A Leakage Triple-CPU Mobile Application Processor Chip. *Proc. IEEE Int. Solid-State Circuits Conf.*, Feb. 2005.

[26] VMWare, Inc. *VMWare Website*, 2009. http://www.vmware.com/.

[27] D. Wentzlaff et al. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, Sept. 2007.

[28] C. Whitby-Strevens. RISC and the I1 Instruction Set for the Transputer. *12th Int. Symp. Computer Architecture, Boston*, pages 17–19, June 1985.

[29] C.-H. Yen. *Solaris Operating System Hardware Virtualization Product Architecture*. Sun, fourth edition, Nov. 2007.