Low Overhead Remote Procedure Call System for Saturn DSP

Christiaan Banister

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-144
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-144.html

May 19, 2022

Acknowledgement

We would like to thank the sponsors of SLICE lab for providing the infrastructure needed to do this research, and the staff for administering the lab space and coordinating meetings. We also thank Albert Ou for providing early direction and reading an early draft of this paper.

Low Overhead Remote Procedure Call System for Saturn DSP

by Max Banister

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Krste Asanović
Research Advisor

5/19/2022

(Date)

* * * * * * *

Professor Yakun Sophia Shao
Second Reader

5 | 9 | 2022

(Date)

Low Overhead Remote Procedure Call System for Saturn DSP

Max Banister
UC Berkeley
maxbanister@berkeley.edu

Abstract

System-on-a-Chip designs with specialized processors and domain-specific accelerators have grown in popularity over the last decade to meet ever-increasing compute demands. This is due to the superior performance-per-watt that can be obtained from such designs as well as the inherent limitations in scaling up traditional out-of-order superscalar CPUs. However, a heavy reliance on microarchitectural details and a lack of coordination between software writers and hardware designers makes targeting these specialized IPs highly non-trivial. In addition, offloading tasks from the CPU to an accelerator introduces overheads that may nullify the advantage of utilizing it in the first place. This paper investigates a transparent method for allowing programmers to run user code on digital signal processing units (DSPs) and quantifies the impact of such an offload.

We propose a lightweight framework at the system software level in the style of message passing Asymmetric Multiprocessing kernels. By pairing a large symmetric multiprocessing kernel with a lightweight real-time operating system, we find that we can achieve near speed-of-light accelerator communication while maintaining a relatively straightforward programming model.

1 Introduction

Modern SoCs sport dozens of programmable accelerators at the high end. The Apple A12 mobile SoC, for instance, has 42 specialized IP blocks. Although accelerators are often thought of as fixed-function (e.g. matrix multiplication), software is often needed to change configuration settings (weight station vs. output stationary), initiate DMA transfers, or perform I/O. Software also provides a degree of flexibility when application use cases change post-silicon. While FPGAs can solve the rigidity problem, they impose a stiff tradeoff in terms of power, performance, and area, and they are hard to justify when the space of accelerator designs is small and well-understood [16]. Digital signal processors, alternatively, provide a familiar software interface and may be used as a "control processor" for a larger accelerator. Therefore, attaching accelerators to a DSP core, or simply exposing a DSP at the system level, may be a sweet spot in accelerator controllability.

A growing problem with having a sea of accelerators on the same die is developing new programming models to exploit them [21]. In the status quo, this problem is currently solved by vertical integration, whereby the same company providing the hardware design also writes software targeting it, similar to how device drivers are developed today. Under this paradigm, accelerator programming is largely relegated to "experts", who use in-house expertise to hand-write kernels[1] targeting domain-specific accelerators. The ability to harness accelerators is thus gated behind APIs, which makes it hard for application writers to optimize their code beyond what the designer has envisioned it being used for, as well as making it more difficult to understand, profile, and debug their code.

Examples of this on mobile SoCs include Apple's Accelerate and CoreML frameworks [7][10]. The Accelerate framework provides library functions for image processing, DSP, neural nets, BLAS, and transcendental math functions. This library is agnostic to the underlying hardware platform, and will make use of the CPU, GPU, or domain-specific accelerators based on what is available to it. After setup, the API for a library call is: input operands are passed as arguments, and an output buffer (sometimes in place) is produced. The programmer has no control over which hardware block a kernel executes on. The API is purposefully high-level to encapsulate details about the hardware and provide backwards-compatibility, the latter of which is an issue that plagues accelerator programming models.

In the Android world, the tablets and smartphones made by OEMs are generally less vertically integrated than Apple, and SoCs are sourced from a variety of vendors. One particular SoC designer, Qualcomm, has shipped a DSP in their Snapdragon line of SoCs since 2006, leading to a well-cemented programming model and tooling that we will analyze here. The Qualcomm Hexagon DSP is a typical VLIW architecture, with 4 micro-instructions within a single instruction packet. It supports dynamic multi-threading, wherein the running context is swapped out on an L2 miss. The DSP runs at maximum 2GHz. Some configurations have HVX, a SIMD vector coprocessor. The most recent Hexagon, the

[1] Kernels in this context means tightly nested looped code, not an operating system kernel.

780, is capable of 4K video encoding and decoding at 120 frames per second [18].

To program, one must use the Hexagon SDK provided by Qualcomm [5]. The SDK consists of a C/C++ compiler for the Hexagon instruction set architecture, as well as tools for generating stub code to call a remote function. Hexagon uses a system called FastRPC to offload work to the DSP and transparently serialize/deserialize arguments. A programmer first declares the function they would like to use in an Interface Description Language (IDL), for which a compiler then generates C/C++ function definitions and stubs to be called by application code. Each interface function can specify input and output buffers, which may be optionally marked uncached (if the memory was not written by the CPU) and non-coherent, to avoid an expensive cache flush operation. These regions were allocated by the Android ION memory allocator, but presumably they will be changed to use DMA-BUF in the release of Android 12 after the discovery of security holes [22]. Usage of this allocator provides zero-copy buffers for RPC. The round-trip latency for a FastRPC call is 200-300 microseconds with highest clock settings. The overhead of launching a thread on the DSP is 5,000 cycles.

2 Background

Digital signal processors are specialized processing units designed for manipulating analog signal data. Compared to CPUs, DSPs are more power-efficient when executing certain algorithms and are architected with real-time latency considerations in mind. Because of their narrower domain, DSPs often have custom architectures that are optimized for digital signal processing. Certain operations, like discrete fourier transforms, convolutions, fused multiply-accumulates, and polynomial evaluation may be accelerated in hardware. There is also often support for fixed point and saturating arithmetic. The instruction set may be customized for such a processor: Very Long Instruction Word (VLIW) architectures are popular in the space, and hardware loops may be included to eliminate loop overhead. Scratchpads, on the order of a hundred of kilobytes large, may be used in addition to or in place of a cache when access patterns are known statically. A vector coprocessor may be attached. Applications include video encoding/decoding, image processing, and Simultantaneous Localization and Mapping (SLAM) for robotics and augmented reality.

Saturn is a parameterized, fully-synthesizable digital signal processing research core developed at UC Berkeley. It is written in the Chisel hardware construction language, and leverages the Rocket Chip SoC generator for many supporting components, such as caches, control and status registers, and debug functionality [3]. In addition to two scalar pipelines and floating point, Saturn has nascent support for the RISC-V Bitmanip extension. Saturn was taped out for the first time in May 2022 using the Intel 16 technology on the BearlyML chip. BearlyML is a heterogeneous SoC which will serve as the reference platform for our modeling. It features 4 general purpose cores having the in-order Rocket Core processor, and one instantiation of Saturn. The CPU and DSP run at 500MHz, while the memory subsystem and uncore run at 100MHz.

Rivet is a timing model for Spike, the RISC-V ISA simulator. While Spike provides a "golden model" reference for hardware implementations, it does not by default provide timing information that could be used for performance modeling. Rivet adds support for modeling the microarchitecture of a few cores, Saturn in particular. Rivet is able to capture salient microarchitectural details like pipeline hazards for use in counting cycles-per-instruction (CPI). Branches taken/not taken, read-after-write hazards, write-after-write hazards, and I$ misses all affect the cycle count of a code sequence. There is also support for a two-level cache hierarchy, with separate instruction and data caches and an inclusive L2. Timing parameters are constants that can be provided to Rivet in a configuration file. We believe that Rivet provides a level of timing granularity suitable for software development on experimental hardware that goes beyond simple dynamic instruction count. However, no model is perfect, and the shortcomings of Rivet will be addressed further on.

3 Related Work

3.1 Asymmetric Multiprocessing Kernels

At the advent of shared memory multiprocessor systems, a flurry of new operating system research emerged to deal with the new hardware. At the time, computer architects were faced with having to port millions of lines of operating system code, usually developed by an outside company, to support new multiprocessor systems. To get around this impracticality, it was proposed to have the system be partitioned per-processor and have an operating system kernel run on every core. The Disco virtual machine monitor [4], for instance, allowed multiple operating systems to run side-by-side on a cache-coherent non-uniform memory access system. The virtual machine monitor, which had a smaller code footprint, would be the only element needed to scale across all cores, and it would export hardware resources such as physical memory and virtual processors down to the operating system. Operating systems that didn't support NUMA or multicore yet would have the illusion they were running on a uniprocessor system. The Disco authors conclude, however, that with long enough development timelines, the operating system is the best place to manage such re-

sources.

Another early attempt at scaling the operating system across disparate hardware is CHORUS [6]. CHORUS is a distributed operating system based on microkernels, providing a familar UNIX-like interface. The CHORUS project originally supported running on multiple discrete machines, each running an instance of the core microkernel, called the nucleus, and with distributed system servers and actors[2]. The core nucleus handles interrupts, virtual memory, and IPC, while filesystem and I/O drivers are delegated to system servers. Like many microkernel implementations, actors communicate using asynchronous message passing, but uniquely to CHORUS, IPC is network-transparent, so actors can communicate across multiple sites. As the provenance of operating systems became more clear over time, and the methods of RPC more wrought-out, CHORUS shifted its focus to small embedded systems where it found a niche use-case for multi-OS configurations.

While AMP kernels have lost mindshare to symmetric multiprocessing (SMP) kernels, where a single operating system image manages all the application cores in a system, they may yet see a revival for heterogenous hardware platforms, where disparate hardware may have different supervisor needs, such as for hard real-time scheduling or for no-MMU processors. Similar to how microkernel research lives on in the form of virtual machine monitors [9], we believe that AMP kernels may find a fruitful future in heterogenous SoCs. The ecosystem of operating system kernels running on an SoC may include a large SMP kernel for high-throughput, general-purpose apps, and low-latency RTOSes for real-time and power-critical tasks.

3.2 OpenAMP

OpenAMP [2] is potentially the closest existing framework that attempts to solve the problem of OS communication in heterogeneous systems. Like our work, OpenAMP attempts to define a communication channel between a general-purpose operating system and a bare-metal RTOS, with a master and remote topology. OpenAMP itself is an umbrella project; the components of which primarily relate to our work are remoteproc and RPCMsg. The scope of OpenAMP is broader than the goals of our project, with high-level App Services built on top of its communication substrate as well as the System Device Tree, which attempts to standardize device tree bindings for processors with different bus views within a heterogeneous system.

Remoteproc is responsible for loading the remote image for the foreign agent. It partially specifies the contents of the ELF file used for loading firmware images. The resource table, optionally included in the ELF, specifies the load address of the image, shared memory regions for data, and trace buffers for remote software. Within the resource table is the VirtIO resource, which describes the vring, the circular queue used for master-remote data sharing. The master will parse this image, read the resource table to set up the shared memory regions, and start the remote processor at the entry address. If a resource table is not embedded in the ELF image, no subsequent inter-host communication is expected.

RPMsg is the protocol for interprocess communication used in OpenAMP. RPMsg defines a packet format, similar to Internet Protocol (IP), which includes a header describing the source and destination addresses, as well as the data being sent. These packets are sent over a bidirectional channel, which is generally established at runtime using a Name Service announcement. When the packet reaches an endpoint, a callback is invoked to consume that data. Underlying RPMsg is VirtIO, the collection of primitive shared memory data structures. VirtIO provides a way to access device memory space using virtqueues, which are queues of buffer descriptors that describe some region of memory. Virtqueues consist of some control data structures and the vring, which is a basic ring buffer implementation. VirtIO has its origins in a hypervisor, wherein the host and guest would exchange memory descriptors, such as the master and remote do in AMP kernel configurations.

In comparison to OpenAMP, our implementation defers the loading of the firmware image of the remote target, a concern of remoteproc, to some unspecified mechanism. The RTOS image is expected to be loaded early during system boot. Loading of user program code, on the other hand, is of primary concern to our implementation, and is a application-level service that could possibly be implemented on top of RPMsg-type communication channels. Channel discovery is also not dynamic in our implementation, but at a known location relative to the firmware image, to improve setup time. Finally, basic support for soft IRQ doorbells are implemented in our Linux kernelspace code.

3.3 Accelerator-Level Parallelism

As the number of accelerators grows, especially on mobile platforms, it becomes necessary to consider the system-level view of how they all fit together. Since the field is still new, the science is immature about how one should choose the accelerators for a particular SoC, allocate SRAMs amongst them, and expose programming them to the user. Researchers Mark Hill and Vijay Reddi, PhD. coined the term Accelerator-Level Parallelism to describe the interplay between multiple accelerators in a system [14]. In their work they put forth the

[2] An actor encapsulates resources, such as memory regions, threads, and communication ports, a concept similar to processes in UNIX.

Gables model for analyzing multiple accelerators, which expands on the standard roofline model to examine multiple accelerator compute units operating in a pipeline and sharing the same main memory interface. In their view, accelerators will be Swiss-Army Knives useful for one task, but many applications, and multiple of them will be strung together to build complex programs. An example is taking a picture in a camera app, which may involve the ISP, GPU, DSP, and CPU. They call for new programming models to be explored for many-accelerator systems. While this problem is too large to tackle in the general case, we seek to examine the system software-level components needed to facilitate such communicate among accelerators, and specifically look at the involvement of the CPU: should it be a command-and-control center for all accelerator activity, or merely the kick-off point for an autonomous program?

3.4 Cache Coherence Overheads

When evaluating loosely-coupled accelerators, it is relevant to look at how many cycles are spent in the memory subsystem. Research by Sophia Shao, PhD. et al. shows that as much as 40% of the total runtime of an offloaded kernel may be spent in data movement and cache coherence overhead [19] [20]. Thus, it is advantageous to co-design the memory subsystem alongside an accelerator to ensure that the optimum of low-leakage power and high-performance computation due to accelerator datapath parallelism is achieved. Accelerators are often incoherent with other agents in the system, and as such must communicate with CPU hosts via DRAM or last-level cache (LLC). As such, the CPU must first flush its useful data out of its local cache to be visible to the accelerator, and invalidate the shared output buffer.

Accelerators' interface to memory may be through DMA or cache. DMA can be considered a "push" based design where the data is preloaded upfront, while caches can be considered a "pull" design, whereby the data is only retrieved when it is first requested. DMA is generally more popular, because caches introduce hardware overheads in terms of tag checking and block management. However, caches provide fine-grain access to data, whereas DMAs perform block transfers, meaning that latency under cached designs is typically better, as the most critical data is delivered first. It is also easy to overlap accelerator computation with cache refills in a non-blocking hit-under-miss cache, which further reduces the latency. Optimizations to DMA, such as pipelining the CPU-side cache flush with the DMA transfer, may be employed, although they are not explored in our investigation.

This research shows that data movement overheads may contribute a significant fraction of a accelerator's overall offload time. If the overhead is too high, it may negate the runtime advantage of running a job on an accelerator. Though accelerators have been used for coarse-grain tasks in the past, scaling laws may lead to them being used for smaller tasks more frequently, necessitating minimizing overheads as much as possible. The paper shows a whole-SoC modeling using gem5-Aladdin, but only uses system call emulation, meaning overheads due to system software are not measured. We seek to quantify the "Linux tax" to better understand software's role in introducing overheads.

4 Implementation

We modeled the full-stack of hardware and software that might exist on an SoC to get an accurate picture of the overheads associated with the system layer. To this end, development was done at the firmware, operating system, and hardware simulator level. Our model SoC consists of two cores: Rocket and Saturn. Rocket is our stand-in for the application processor that runs a full SMP operating system, in our case Linux. Saturn is the domain-specific processor that handles DSP workloads and is managed by FreeRTOS. They communicate at the hardware level through shared memory and the software interrupt controller. Support for both operating systems had to be added, as well as changes to Rivet to enable two processors with independent attributes and cache hierarchies.

4.1 Linux Kernel Support

The general-purpose operating system used for our inquiry was Linux, version 5.15-rc5. Mild changes were needed in order to forward low-level access to shared memory and interrupts to user-space processes. As much as possible, we sought to offload code from kernelspace to userspace, to preempt security concerns by keeping a small footprint of privileged kernel code. Furthermore, almost no changes were necessary to core systems, as most functionality was confined to a singular driver.

Some discussion is warranted regarding interrupt handling. Our RPC mechanism uses a doorbell model, wherein Rocket and Saturn alert one another of changes to shared data structures via an interprocessor interrupt (IPI). Rocket Chip and Spike use the SiFive Core Local Interruptor (CLINT) for sending software IPIs. When Rocket or Saturn want to interrupt the other, they perform a write to an MMIO register in the CLINT's memory region that causes a software interrupt at the foreign hart[3]. If that interrupt is unmasked and interrupts are active, a trap will be taken on the foreign hart and control will transfer to the interrupt handler. Linux uses software IPIs for the purpose of scheduling, and since there is no in-band mechanism for specifying the sender, minor tweaks were needed to the default interrupt controller driver to

[3]Hart is short for hardware thread; in our system, every core is a hart.

differentiate a scheduling IPI from an RTOS-originating one. The flow of control during an incoming IPI to Linux would go: 1) arch-specific interrupt controller driver, 2) generic irqchip driver, and 3) Saturn device driver top-half handler.

4.2 Interrupt Controllers

Using the CLINT is not the only theoretical way to receive an interrupt coming from Saturn. In addition to the CLINT, there is SiFive's Platform Level Interrupt Controller [11]. Both interrupt controllers are usually global entities, serving every core in the system. While the CLINT is used mainly for software and timer interrupts, the PLIC handles external (I/O) interrupts. The CLINT runs at the same frequency as the core, while the PLIC may be decoupled from the core's clock, usually an integer quotient of the core's clock speed. The CLINT has a fixed set of priorities based on static privilege levels, whereas the PLIC has dynamic, configurable priorities. It is possible to connect Saturn to the PLIC to deliver interrupts. This would, however, require a more complex handshake with the PLIC: one for claiming an interrupt, and another for marking completion, adding some additional overhead.

Besides the PLIC, hardware FIFOs could be used to transmit messages under a so-called mailbox model. This would permit some small amount of data, such as a pointer, to be sent along with the interrupt request. This would trim down the overhead of interrupt processing even more, but requires hardware-level changes.

4.3 Linux Device Driver

The way most setups communicate with a system-level accelerator is through a device driver; ours is no different. We implement a small device driver embedded in Linux and running on the application processor that can communicate with the RTOS running on Saturn. It does all the work of setting up shared memory buffers and interpreting IPIs. We expose a filesystem binding at `/dev/saturn` to control it from userspace. Android vendors tend to protect access to their drivers to exclusive apps using a policy in SELinux; however, we are unopinionated about which users can access it. While our device driver uses Linux kernel APIs, the concepts involved in its implementation are generalizable to other OS kernels.

The device driver is rather small: only two methods need be implemented for it: `mmap`, to set up shared memory buffers, and `ioctl`, to instruct Saturn to begin running a task. When an `open` call is made to the device driver, it first setups up a context for the newly opened file. `mmap` is used to allocated physically contiguous memory for use on both Rocket and Saturn. `mmap` calls need to be page aligned in Linux, so we allocate memory in units of whole pages (4K). We use a simple `kzalloc` call to allocate and zero the backing memory, though a DMA or purpose-built allocator could be explored as an optimization. `kzalloc` has the additional benefit that power-of-2 size allocations will be naturally aligned, which fits with RISC-V PMP's idea of a NAPOT region (Naturally Aligned Power of Two). However, `kmalloc` allocations come from a common page pool, and to our knowledge no Rowhammer [15] mitigates have been added to the kernel allocator, meaning that it is ill-suited for production use as it may leak sensitive kernel data. Because we require allocations to be physically contiguous, there is a risk of fragmentation in the buffer pool causing large allocations to fail. We find that, empirically, long-running kernels will fail to allocate space above an order 10 (4MB) size. After an `mmap` call is placed, the user program has a view into a non-coherent array of memory shared with Saturn.

After creating up to 5 `mmap`ped regions, the user driver program can then make an `ioctl` call into the driver to instruct Saturn to begin running a job. The driver program must first set up an array of memory descriptors of every allocation it has made, and pass a buffer with the number of regions and the memory descriptors as the argument to the `ioctl` call. The device driver will then check each virtual memory region to ensure that they indeed have been previously allocated. It does this by iterating through a linked list of virtual memory areas (VMAs) belonging to the process and checking to see if to see if the provided memory falls within one of those regions. The VMA list is usually small, so this is not expected to create significant overhead. It then flushes the cache for each region marked as an input, and invalidates the cache for each output region. These memory descriptors are passed as pairs of physical memory pointers and lengths to Saturn's RTOS over the command queue, so this setup enables a completely zero-copy implementation.

4.4 OpenSBI

It was necessary to make mild modifications to the Linux bootloader, OpenSBI. OpenSBI is an implementation of the RISC-V Supervisor Binary Interface[4] as well as platform-level firmware [19]. OpenSBI supports more hardware than we make use of on our simple Rocket/Spike platform, such as the Advanced Core Local Interruptor, which adds support for supervisor-to-supervisor IPIs. Because upstream Linux does not yet have support for this, we did not include it in our implementation. Early on in our project, the decision had to be made whether OpenSBI would act as a layer between all harts and the hardware platform, or whether FreeRTOS would be a standalone kernel that loaded itself. In

[4]The Supervisor Binary Interface is the way supervisor code, like Linux, communicates with the machine mode Supervisor Execution Environment (SEE).

the end, it was decided to use OpenSBI for Linux SMP harts, but not FreeRTOS. This was motivated by the fact that OpenSBI uses PMP configuration registers to set up trust zones, which FreeRTOS-MPU would normally use for isolating task memory.

However, not allowing OpenSBI to boot all the kernels in the system added some complexity. The way IPIs are normally sent in RISC-V SMP systems is with SBI calls. Because OpenSBI is normally the SBI layer for all harts, it maintains per-hart metadata indicating the IPI operation to be performed, and may perform synchronization between harts when one is sent, e.g. when conducting a TLB shootdown. Allowing FreeRTOS to send IPIs directly through the CLINT meant that OpenSBI would no longer coordinate all IPI communication with shared data structures. Hence, an escape hatch was added to OpenSBI to send and receive "raw" IPIs, rather than ones accompanied with metadata. In production implementations, we encourage designers to think carefully about how platform firmware will setup and interact with the entire system.

4.5 FreeRTOS Port

We chose FreeRTOS to as the archetypical RTOS for our implementation. FreeRTOS is a real-time operating system maintained by Amazon Web Services with ports for dozens of MCU architectures [19]. FreeRTOS has an upgrade path to SafeRTOS, which provides certifications for safety-critical industries. While crash-safety is not of the utmost concern in application space, firmware RTOS crashes typically require a system reboot, and can be annoying to the end user. In addition, safety and security often go hand-in-hand, and security is still a concern to us: FreeRTOS is an operating system like any other, and needs to isolate processes from one another and from itself. Security vulnerabilities allowing for the execution of privileged code have been discovered on QuRT for the Hexagon DSP [13]. Because of the ubiquity, safety, and small size of the code base, we found FreeRTOS well-suited for our research needs.

Although there was already a port of FreeRTOS for RISC-V on Spike, support had to be added for the floating point and memory protection units. For the FPU, floating point state had to be saved and restored on a context switch, and the `fscr` and `mstatus` registers had to be updated based on their contents. For the MPU, every task switch required the RISC-V Physical Memory Protection (PMP) address and configuration registers to be swapped out, though for handling simple interrupts and system calls, it is unnecessary as the MPU is turned off. FreeRTOS was compiled using GCC with the `-O2` flag and the Newlib C standard library, and the heap_4 allocator was used with 64K of area. The executable was 150KB without debug information.

When microcontrollers have an MPU, they can optionally use FreeRTOS-MPU, a variant of FreeRTOS configured at compile time by setting the portUSING_MPU_WRAPPERS `#define` in the config file. In this mode, only the core parts of the kernel, like the scheduler, interrupt handler, and timer task run in M-mode, the highest privilege level. All other tasks run in user mode, under which they will be subject to physical memory protection through the PMP registers. When running in M-mode however, PMP checks are not enabled, so the processor has full access to all of physical memory, unless the L bit in the pmpcfg register is set, at which point the protection is permanent, until a system reset. User tasks can interact with the RTOS through system calls. Which system calls are implemented is ultimately left up to the designer, however, we do not implement any system calls for the moment, in part to derisk any possibility of privilege escalation, and in part to keep object code compatibility with normal userspace RISC-V programs.

Offloaded kernels are permitted 5 separate memory regions that they can use freely. The limitation of 5 regions is due to there being only 16 hardware PMP registers in Saturn. More PMP registers can be provided, but 16 is about the upper limit that can be obtained without negatively impacting cycle time. In FreeRTOS, 5 are already used: two for the bottom and top markers of the unprivileged data section, one 4-byte region for the privilege status, and two for each task's stack. There are 11 remaining PMP registers which can be used to protect 5 segments, given that the start and end each take up a register. Further optimizing this number is an area for future engineering, as not all of the FreeRTOS-provided PMP regions are strictly necessary, and it could be possible to allow the user to use NAPOT regions, each of which only use a single PMP address register. Nevertheless, five regions was sufficient for most kernels we explored.

4.6 End-to-End Offload Flow

A program wanting to make use of the Saturn DSP will begin its journey as an ordinary userspace program under Linux on the Rocket processor. From userspace, it will then open the Saturn device driver and create up to 5 shared memory buffers with `mmap` calls. It must load the DSP function code into the first of those buffers, with the entry vector at the top of the binary. Then, it will construct an array of memory descriptors using the returned virtual memory of those `mmap` calls. This array will be the argument to an `ioctl` that will call into the device driver. An example of the initialization procedure using the C interface is shown here:

```
int fd = open("/dev/saturn", O_RDWR);
if (fd < 0) {
      /* error */
```

```
}


volatile char *buf;
buf = mmap(NULL, 4096, PROT_READ |
    PROT_WRITE, MAP_SHARED, fd, 0);

if (buf == MAP_FAILED) {
      /* error */
}
```

```
uintptr_t taskDesc[] = {1, buf, buf +
    4096};

int ret = ioctl(fd, 0, &taskDesc);
if (ret) {
      /* error */
}
```

The `ioctl` call will block while the DSP code is running. Alternatively, one could implement a non-blocking ioctl, and have the user poll the file descriptor, or use a number of Linux's rich operations on file descriptors to await completion. However, we found the blocking code easiest for the programmer to reason about.

Most of the setup code is intentionally pushed to userspace. While somewhat tedious to manually set up shared memory regions and perform syscalls, we believe the interface is simple enough that an average programmer could be productive, and higher-level libraries could be easily written to encapsulate this functionality. Low-level implementation details need not be obscured from the programmer, as long as those details are sensible.

Inside the driver, each virtual memory address in the buffer array is mapped back to its physical address and checked to make sure it falls within a previously allocated region. Then, the buffer array is enqueued on the command queue, a standard ring buffer that is addressable by both Linux and FreeRTOS. It is assumed that the command queue exists in coherent memory with Saturn, i.e. updates to the command queue are made visible to Saturn through the cache coherence protocol, even though no similar assumption is made about the data regions. The buffers are then cautiously flushed from Rocket's local cache, including the DSP library code, and the output region is invalidated. Then, Rocket interrupts Saturn through the aforementioned IPI mechanism.

On the FreeRTOS side, the flow begins in the ISR with the buffer array being dequeued, and then quickly passed to the task spawning server via an internal queue. The task spawning server runs in task context and is itself a high-priority task that is awoken when a new task arrives. It creates the MPU-protected task with the memory descriptors passed to it by Linux, and assigns a priority to the user task (by default, all are equal). It also passes in the buffer pointers as arguments to the top-level, "main"

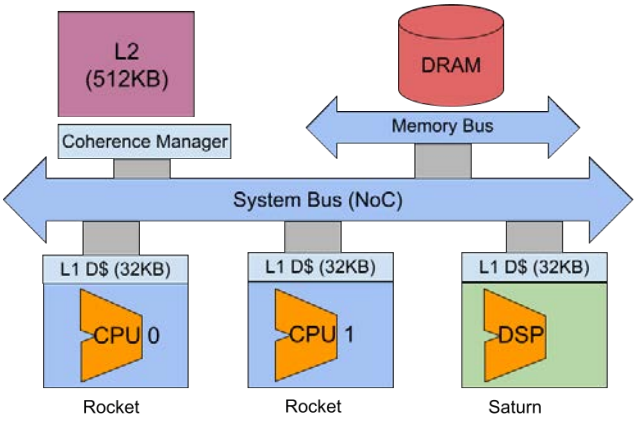

Figure 1: Overview of the model system.

function of the DSP code. Then, control is handed off to this newly spawned task to execute. It may use these buffers to perform some computation, and finally cache-flush the results in the output region. Once complete, the user task returns from its top-level function, and the wrapper code will make a system call that will perform the management duties of notifying Rocket of its completion, and marking the task as destroyed, deferring freeing its memory for a later time by a background server.

5 Evaluation

We quantify the system software communication overheads of our RPC mechanism using an SoC modeled in Rivet, a RISC-V software simulator and timing model. Rivet was configured with one in-order Rocket core, with a 32KB L1 data and instruction cache and a 512KB inclusive L2 data cache, and a Saturn core with 32KB local instruction and data caches. We chose to give Saturn a cache, rather than a scratchpad and DMA, to defer any questions about specializing the DMA, and to make the DSP as algorithm agnostic as possible. This also roughly approximates the design point achieved in the research chip BearlyML.

Saturn was configured with two pipelines, of which only the first can perform floating point operations. The latencies of the memory system are statically defined: a L1 data cache hit is 2 cycles, while a miss incurs a penalty of 16 cycles, and an L2 miss incurs a 58 cycle penalty. Not-taken branches have a penalty of 4 cycles. The one-stage branch predictor is similarly borrowed from Rocket.

First, we will briefly look at the overheads of the setup functions, `open` and `mmap`. For `mmap`, a cycle count is shown both for cold and warm caches, as it is likely to be executed consecutive times. Most of the time is spent zeroing out newly allocated pages. Each datum is the average of three trials, and rounded to the nearest integer. Measurements were obtained by post facto analysis

Op	Cycles (Cold)	Cycles (Warm)	Instructions
open	14,710	—	3,831
mmap	50,184	49,325	22,446

Table 1: Times to run open and mmap on twenty 4K pages in the device driver.

of the timing log in Rivet, or occasionally by reading the `cycle` and `instret` counters.

5.1 RPC Overheads for DGeMM

To examine the RPC latencies of a real workload, we benchmarked a dense general matrix multiplication kernel on our system. The DSP code was compiled with GCC with the `-O2` optimization level and `-mtune` set to SiFive's Series 7 processor family, which is dual-issue like Saturn. The data elements were double-precision floating point, and size of the input and outputs matrices were 100 by 100. Thus, the size of the two input matrices were each 80KB. Due to the compute intensity of the algorithm, $O(n^3)$, this is a good candidate for offload, as the linear data movement costs are quickly exceeded by the compute time.

Table 2 shows the costs associated with offload in both the outbound and inbound directions. While the costs of the cache management sections are proportional to the kernel sizes, the system overheads of Linux and FreeRTOS are fixed. Nevertheless, we see that operating system costs are a large portion of the total overhead, with the Linux irqchip, driver code, and scheduler taking up nearly a quarter of the overhead, and the RTOS taking over 5,000 cycles to start a new task. These costs represent fairly typical, unavoidable activity that one experiences when starting and stopping tasks on an operating system kernel. Altogether, overheads account for 36,500 cycles, or 73 microseconds on a 500MHz clock. We believe this is faster than commercial systems due to the pre-allocation of buffers and zero copies. In real world software, it may not be practical for data to originate in a specially allocated region, so a copy step may be unavoidable.

Our evidence tentatively suggests that even small kernels may be worthwhile to offload, so long as one is careful to place it in physically contiguous buffers from the start. Table 3 shows the various sized matrix multiplications and their corresponding overheads; matrices as small as 20x20 still exhibit small overheads in relation to the time spent doing useful work. This will, however, differ for other algorithms. In other words, operating on data that fits on just two 4K pages is amenable to cross-core acceleration, even with needing to flush/invalidate cache blocks.

	Cycles	Instructions	Percentage
Host Side			
Flush Code	34	16	0.0%
Flush Input	7,730	7,690	21.2%
Inv. Output	3,863	3,845	10.6%
Linux Ioctl	3,275	1,006	9.0%
Accel. Side			
FRTOS Spawn	5,502	3,686	15.1%
Inv. Input	5,052	7,518	13.8%
Compute	13,531,723	8,112,795	—
Flush Output	2,524	3,053	6.9%
Linux Sched.	8,542	3,053	23.4%

Table 2: RPC call latencies broken down by stage. Overheads are given as a percentage of total overhead cycles.

Size (N)	Total Cycles	% Overhead
100	13,565,721	0.03%
50	1,542,729	1.5%
30	336,195	5.8%
20	114,674	16%
15	60,448	30%
10	31,341	58%

Table 3: The runtimes, in cycles, of various size matrix multiplication jobs and the proportion which is overhead.

5.2 Cache Management

In order to flush the caches, we used the RISC-V Cache Block Operations (Zicbom) extension. Cache lines were assumed to be flushed with 64 byte granularity. Since GCC assembler did not have mainline support for this extension at the time of experimentation, instructions were inserted manually with inline assembly. Every cache instruction was assumed to execute with 1 IPC. Saturn was able to execute these sequences about 50% faster than Rocket, due to the extra issue slot, which allowed the pointer increment instruction to execute in the same bundle. In general, Saturn proved very effective at eliminating loop overheads from compiler generated code. Somewhat unoptimally, software on Saturn had to flush the entirety of the input buffers, even if they were larger than the small, local cache Saturn possessed. One optimization might be to include an instruction for flushing the entire cache, rather than doing so line by line, like what Rocket's non-standard `cflush.d.l1` does. This would

allow the cache flushes to be `O(1)` with respect to the size of the data operands.

There are some times when we conservatively invalidate memory when it is perhaps unnecessary. If we allocate a new buffer that we haven't used before, it is unlikely we will need to invalidate it, both on the Rocket and the Saturn side. However, because we are unaware of the previous contents of our cache, we must be conservative and invalidate it anyway. Linux could potentially track physical memory addresses used in allocation to determine is a process has ever seen it before, and not invalidate the region on an RPC if it is fresh. Likewise, Linux could communicate to FreeRTOS by attaching metadata to each buffer descriptor sent over the command queue to tell it whether it has been used before.

5.3 Sources of Variance

While we tried controlling for most factors outside of the critical path of the RPC flow, there were some unavoidable external factors that interfered with our measurements. Control was maintained by using a clean boot of Linux, with only kernel daemons running, and having tight control over the running tasks in FreeRTOS. Still though, there is variability to be expected in such measurements. In Linux especially, the presence of other tasks may enact strange behavior on the one you want to observe. In some cases, latencies will scale with the number of running threads on one's system. For example, to restart the user task after an IRQ, Linux must go through the scheduler, which runs in $log(n)$ time w.r.t. the number of threads. Furthermore, the device driver must make calls into `kmalloc` to set up the shared memory regions - these calls could block while the OOM killer runs if the system is out of memory. While this did not happen in our experimentation, we should mention that our cycle counts are based on linear, non-preemptive code scheduling. Furthermore, while theoretically the SLUB allocator that Linux uses can perform minimal block management if certain power-of-2 sizes are requested, here we request multiple pages, often in odd sizes. As such, the kernel may need to do free block coalescing or other structure management, which will run with slightly different times with differing heap state. FreeRTOS's heap_4 allocator suffers the same nondeterminism, though there runtime is bounded by the small absolute size of the heap.

5.4 Discussion

We found it easy to target the Saturn processor once the AMP communication framework was written. All that was needed to write on the driver program side was a few system calls, and the framework took care of the rest. The task that ran on the DSP was likewise easy to compile, as the entry point was a function that was passed its memory buffers as arguments, similar to existing threading libraries like pthreads[5]. Since the compiled code was compliant with the RISC-V LP64D ABI, we could transparently switch to running on the application processor to keep code compatibility with systems without a DSP (or not configured with such). While our compiled DSP library cannot make use of global or thread local data, we opted not to add them after not needing them in practice, while the lackthereof encouraged the writing of more functional style code. We find DSP offloaded tasks best suited for numerical code and image processing passes - in other words, programs that have explicit inputs and outputs. Programs that needed to communicate with the CPU often, however, were more difficult to write, as one had to either hand-write synchronization code with the CPU using incoherent memory buffers, or break up a long running task into several small ones, incurring the overhead of spawning a thread on the DSP each time.

We also found Rivet a good platform for the co-development of hardware and software. It was quick and easy to make a change to the simulator to test out e.g. different cache parameters and then see the evaluation on a relevant benchmark. While an FPGA-based evaluation platform like FireSim allows for cycle-accurate simulation of an entire system orders of magnitude faster than Rivet [12], changing hardware parameters requires the regeneration and reflashing of the FPGA image, slowing down iteration cycles. For small kernels or RPC microbenchmarks, Rivet is ideal for rapid testing.

5.5 Limitations

The Rivet timing model has a few shortcomings that impact the quality of our analysis. It especially fails to model the memory subsystem to high fidelity. Microarchitectually salient events like L2 bank conflicts and DRAM row refreshes are not accounted for individually. Rather, Rivet uses average latencies from real-world workloads, which tend to be accurate over long dynamic instruction counts. However, very short or pathological code sequences may induce timing irregularities. Also, contention in the memory subsystem is not accounted for: we only consider two cores whose traffic does not interact with one another. With greater development time, these details could be added to the Rivet timing model.

In addition, real mobile SoCs will make use of Dynamic Voltage and Frequency Scaling (DVFS), to optimize for power efficiency when not running compute intensive jobs. This makes real-world performance difficult to reason about, as a CPU-DSP combined job may cross multiple power and clock domains. For long running DSP tasks on the Hexagon, it often happens that the CPU

[5]The pthread_create(3) POSIX library call allows users to pass a void function as the entry point and a generic pointer to arguments.

enters a low power state, at which point a long wakeup sequence will occur when the task completes and notifies the CPU, which can take on the order of milliseconds. We control for this by only considering maxima clock frequencies (500MHz on BearlyML) for our analysis. Both the CPU and DSP are assumed to run at the same clock frequency, and measurements are given in terms of cycles.

6 Future Work

6.1 Page-Based Virtual Memory

Currently, our FreeRTOS port uses an MPU to protect tasks from one another, using a classic base-and-bound protection scheme. This works for certain classes of algorithms that operate on dense data; however, algorithms with less spatial locality will be harder to map onto our system, as many small data buffers will have to be independently allocated, which will be unwieldy to program and, in all likelihood, quickly exhaust the number of PMP registers available. Page-based virtual memory comes with its own set of drawbacks, such as increased memory usage overhead from page table allocations, and more latency when transferring page table mappings from the main operating system to the RTOS. Thus, DSPs with virtual memory typically employ large page sizes to mitigate some of these downsides. There is an optimization problem in finding the ideal unit size for memory protection such that the overhead of setting up virtual memory mappings doesn't swamp the execution time of small kernels. Also, one could consider sharing some page tables with the host kernel, though this introduces more invasive changes to Linux's memory management code.

6.2 Out of Order Core Modeling in Rivet

Many of our Linux benchmarks were pessimized due to the relatively simple design of the Rocket core that we modeled the application processor on. We achieved IPCs of around 0.8 on Rocket, even though 2+ could be expected of even a modest out-of-order core. A more refined model might simulate a large, out-of-order processor in Rivet, which could provide a comparison between the same task running on Saturn and the OoO core.

6.3 User-Level Interrupts

User-level interrupts are a popular design choice on microcontrollers, where individual tasks might want to subscribe to incoming interrupts. This concept could be applied to tasks running jointly on the CPU and DSP, which may want to notify one another, for example, when new data has arrived. User-level interrupts remove the need for a supervisor/user privilege transition by delivering interrupts to the lower privileged mode directly. However, it is unclear if the reduction in jitter could not be achieved in software by including a "fast path" in the interrupt handler. Until a clear use case for user-level interrupts arises, we will avoid relying on any hardware-level changes.

7 Conclusion

In this work, we have analyzed problems with offloading compute jobs to programmable DSP cores in a heterogenous processor system. We propose a mechanism for RPC between compute complexes, using Asymmetric Multiprocessing kernel communication, to coordinate the offloading of compute jobs to more specialized cores. With minimal system call setup, one can describe and launch a job on the DSP using a familiar programming interface in a sandboxed environment. Finally, we quantify the system call, RTOS, and cache coherence overheads to show where the opportunities for improvement lie. We hope to see more progress in enabling application developers to target DSP cores, as the performance uplift can be significant and as the landscape of purpose-built cores continues to grow.

8 Acknowledgements

We would like to thank the sponsors of SLICE lab for providing the infrastructure needed to do this research, and the staff for administering the lab space and coordinating meetings. We also thank Albert Ou for providing early direction and reading an early draft of this paper.

References

[1] Krste Asanović Andrew Waterman. 2019. The risc-v instruction set manual volume ii: Privileged architecture. https://riscv.org/technical/specifications/.

[2] Etsam Anjum and Jeffrey Hancock. 2020. Introduction to openamp library. https://www.openampproject.org/docs/whitepapers/Introduction_to_OpenAMPlib_v1.1a.pdf.

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John R. Hauser, Adam M. Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, Jack Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moretó, Albert J. Ou, David A. Patterson, Brian C. Richards, Colin Schmidt, Stephen Twigg, Huy D. Vo, and Andrew Waterman. 2016. The rocket chip generator.

[4] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. 1998. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems* 15. https://doi.org/10.1145/265924.265930.

[5] Lucian Codrescu. 2013. Qualcomm hexagon dsp: An architecture optimized for mobile multimedia and communications. https://developer.qualcomm.com/qfile/27696/qualcomm-hexagon-architecture.pdf.

[6] M. Gien. 1995. Evolution of the chorus open microkernel architecture: the stream project. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*. pages 10–16. https://doi.org/10.1109/FTDCS.1995.524963.

[7] Simon Gladman. 2019. Introducing accelerate for swift. https://developer.apple.com/documentation/ accelerate.

[8] Rich Goyette. 2007. An analysis and description of the inner workings of the freertos kernel. https://class.ece.uw.edu/474/peckol/doc/FreeRTOSPaper-1.pdf.

[9] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. 2006. Are virtual-machine monitors microkernels done right? *SIGOPS Oper. Syst. Rev.* 40(1):95–99. https://doi.org/10.1145/1113361.1113363.

[10] Apple Inc. 2020. Apple core ml framework. https://developer.apple.com/documentation/coreml/ mlmodel.

[11] SiFive Inc. 2019. Sifive interrupt cookbook version 1.2. https://www.starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf.

[12] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolić, Randy Howard Katz, Jonathan Bachrach, and Krste Asanović. 2019. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. *IEEE Micro* 39(3):56–65. https://doi.org/10.1109/MM.2019.2910175.

[13] Slava Makkaveev. 2021. Pwn2own qualcomm dsp. https://research.checkpoint.com/2021/pwn2own-qualcomm-dsp/.

[14] Vijay Janapa Reddi Mark D. Hill. 2021. Accelerator-level parallelism. In *Communications of the ACM*. volume 64, pages 36–38. https://doi.org/10.1145/3460970.

[15] Onur Mutlu and Jeremie S. Kim. 2020. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39(8):1555–1571. https://doi.org/10.1109/TCAD.2019.2915318.

[16] Ann Steffora Mutschler. 2018. What makes a good ai accelerator. https://semiengineering.com/what-makes-a-good-accelerator/.

[17] OpenSBI. 2021. Risc-v open source supervisor binary interface (opensbi). https://github.com/riscv-software-src/opensbi.

[18] Martin Saint-Laurent, Paul Bassett, Ken Lin, Baker Mohammad, Yuhe Wang, Xufeng Chen, Maen Alradaideh, Tom Wernimont, Kartik Ayyar, Dan Bui, Dwight Galbi, Allan Lester, Marzio Pedrali-Noy, and Willie Anderson. 2015. A 28 nm dsp powered by an on-chip ldo for high-performance and energy-efficient mobile applications. *IEEE Journal of Solid-State Circuits* 50(1):81–91. https://doi.org/10.1109/JSSC.2014.2371454.

[19] Yakun Sophia Shao. 2016. Design and modeling of specialized architectures.

[20] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. 2016. Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. pages 1–12. https://doi.org/10.1109/MICRO.2016.7783751.

[21] Mark Silberstein. 2017. Accelerators in data centers: the systems perspective. https://www.sigarch.org/accelerators-in-data-centers-the-systems-perspective/.

[22] Hang Zhang, Dongdong She, and Zhiyun Qian. 2016. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, CCS '16, page 1663–1674. https://doi.org/10.1145/2976749.2978320.