Fast Thread Migration in a Heterogeneous ISA System

Max Banister
maxbanister@berkeley.edu
University of California, Berkeley

John Fang
zitaofang@berkeley.edu
University of California, Berkeley

Charles Hong
charleshong@berkeley.edu
University of California, Berkeley

ABSTRACT

Heterogenous systems-on-chip (SoCs) are becoming more popular across different levels of computing. However, the modularity of extensible ISAs like RISC-V can also cause fragmentation and the coexistence of cores in the same system that support different ISA extensions. Currently, no OS scheduler is designed for managing threads on such heterogeneous systems. Users must use poorly-suited mechanisms to manually set the cores on which their software should run, hurting performance and increasing development time. We propose an automatic thread migration mechanism that schedules accelerated programs to the appropriate cores in a shared memory heterogeneous system, adapting system software to an increasingly common hardware environment. The fast thread migration system, implemented for SOCs implementing the RISC-V vector extension on only certain cores, speeds up thread migration over Linux `sched_setaffinity()` system call by 2.1× and eliminates the need of the application to be developed with awareness of which cores in the system support specific ISA extensions.

1 INTRODUCTION

Many-core System-on-a-Chips (SoC) designs have proliferated due to their simplicity of manufacturing and improved intra-chip communication latencies. Miniaturization in IC technology has meant that more components of the system can be added to a single chip, including the CPUs, caches, NICs, and wireless components. In addition, because of scaling limitations with general-purpose processor cores, designers have used their increasing transistor budget to add domain-specific hardware accelerators, to efficiently run certain applications. Such accelerators have greatly enhanced the performance of tasks such as video encoding and decoding, audio digital signal processing, cryptography, and deep neural networks. With the possibility of fitting dozens of such accelerators on a chip, system designers are faced with the challenge of controlling a diverse set of IP blocks in a uniform fashion, as to allow firmware, operating system, and application writers to easily exploit their capabilities. In the past, attempts at doing so have resulted in ad hoc implementations and divergent architectures, greatly increasing the amount of development and verification effort. In this paper, we explore a greenfield system architecture and software stack aimed at solving this problem. We leverage existing research in interprocessor communication, and build on the RISC-V philosophy of having all cores belonging to the same base ISA. Such a design, we believe, cuts down on the amount of repeated work for verification and minimizes the number of software toolchains one must support.

Tightly-coupled accelerators, which integrate with the processor's control logic, or loosely-coupled accelerators, are two models of accelerators that build on the traditional scalar unit and extend it with additional functionality. Such designs are advantageous because they allow tight feedback between the accelerator and the scalar unit for computations that may need to alternate between the two, allowing data to reside in a shared SRAM or introducing intermixing of control flow with data-parallel operations. While simple devices may be programmed by foreign processor MMIO, more complex operations will inevitably require the control processor to run bespoke logic, and attaching a scalar core is a simple alternative to a complicated state machine or microcoded implementation. For example, a deep neural network engine might feature acceleration for common operations such as matrix multiplication and 2D convolutions, but a host processor might be necessary for layer-wise scheduling and performing operations that weren't anticipated prior to chip fabrication. As such, software will now be running on accelerator cores, necessitating a communication mechanism to the general purpose application processors that may need to retrieve data or initiate a work job. We do not examine the challenges involved in facilitating this communication at a hardware level, such as with a network-on-a-chip architecture, but instead look at software's view of such accelerators.

These accelerator control processors often have custom instructions, separate from the base ISA, to enable operations particular to the application domain they are attempting to accelerate. These instructions may be long-latency, CISC-style instructions that deviate from the simpler instructions they are nested between. This enables the type of fine-grained acceleration that tightly-coupled accelerators are targeting. Other software models may exist to enable accelerator support, such as at the framework or the OS level, but here we focus on ISA-level support, upon which higher level abstractions can be built if desired. Adding OS support for accelerator functions, such as with a device driver, incurs significant overhead because of various user-kernel memory copies and system call overhead. As such, it is desirable that a

user application be able to reach out and control an accelerator, as if it were just another functional unit on the processor it is running on. In our model, an application can leverage platform firmware, with little modification to the OS, and run part or all of a thread on an accelerator. The operating system only needs to know about the migration insomuch as it needs to put the process to sleep during accelerator execution. It is expected that a program may need to switch between the accelerator host processor and the application processor multiple times during its lifetime - although for coarsely schedulable timeslices, to amortize the overhead of migrating CPUs.

This design is also useful for solving the issue of fragmentation in the RISC-V ecosystem. Since RISC-V is designed to be extensible, vendors may add proprietary features to the ISA in order to support accelerator execution. In addition, there are over 20 ratified "well-known" extensions already in use, and building software for the power set of all of these poses a challenge. In addition, system designers have to decide whether to have homogenous ISA support, potentially bloating each processor's decode unit and datapath, or have piecemeal ISA extensions on each core, complicating build systems and operating system scheduling. We propose a mechanism for thread migration whereby the second option becomes more feasible, as knowledge of each core's innate capabilities is deferred to the platform firmware, which by necessity is customized for each SoC anyway. In this way, the operating system itself can have a view of a homogenous set of CPUs upon which to schedule, but there are hidden, special purpose cores for use only by platform firmware. Our system is unique in that it requires no special system calls or linked libraries in order to make use of an accelerator device on the SoC. An application writer can merely use the custom instructions expected by the accelerator, and a combination of firmware and OS kernel will transparently handle migrating the running thread across hardware threads. It does so by recognizing that, if both the application cores and the accelerator processor support the same base ISA, such as RV64I, a thread can start its life on one core, and migrate to the other depending on characteristics of the program, such as which instructions it's executing or whether it needs to perform I/O. It starts on an application core, and upon detecting an illegal instruction that matches the accelerator's supported opcodes, it initiates a migration to the other processor, whose datapath is equipped to handle such instructions. This allows the application to be extremely flexible in how it is written, although performance is more of a blackbox, which is why library authors should make use of knowledge of this behavior in order to prevent migration thrashing and achieve more predictable results.

The remainder of this paper is organized as follows: In Section 2 we discuss the accelerator software support ecosystem and prior system migration work. In Section 3 we give an overview of the RISC-V system software environment and the steps we take to implement fast, automatic thread migration. In Section 4 we discuss the benchmarks used and the metrics of success for fast thread migration. In Section 5 we evaluate the effectiveness and performance of fast thread migration against baseline Linux. In Section 6 we discuss future work and conclude in Section 7.

2 RELATED WORK

2.1 Hardware Accelerator Control

Hardware accelerators are specialized hardware for some particular algorithm that has far better performance than general purpose processor. [8] has a detailed description of the taxonomy of accelerator communication type. Depending on the method to control the accelerators, we can form a spectrum of accelerator classification, from direct memory access engine (DMA) to tightly coupled co-processor like FPU. DMA type accelerators can also be further divided based on the location of DMA engine (as a part of the accelerator or located somewhere else on the bus), although they are pratically the same for the purpose of accelerator programming.

The most naive ways to control them are to treat accelerators as IO devices and access them through memory instructions, or to treat them as a part of the application processor and issue specialized instructions to them. However, the former method is slow and inflexible due to the memory access latency and instruction format, and the latter is difficult to implement when the number of accelerators grow. For slow DMA accelerator that doesn't requires much interaction with the core, control through memory address is sufficient even with a large number of such accelerators. Standalone GPUs, for example, often exist as a hardware accelerator for graphic rendering connected through PCIe to the core. They are far away from the core, and they can execute code on their own, therefore we can get acceptable performance with the current communication method. But, it is not ideal for computational intensive accelerator with fine-grained control, which is what we intend to tackle.

As of today, most hardware accelerator built in RISC-V system have a small control core that can execute some control code and issue specialized instructions to the specialized hardware part. Rocket core is a popular open-source control core, which come with a standard accelerator interface Rocket Custom Co-processor (RoCC) [3]. Some examples of accelerators using this platform include Gemmini [9], Hwacha [11], and SHA3 accelerator [12]. SHA3 is the accelerator for cryptographic algorithm, and it is a typical

DMA-style accelerator that perform computation on the data in the built-in scratchpad and use an DMA engine to transfer data from and to the memory. Hwacha, on the contrary, is a non-standard vector unit that predates the RISC-V standard extension, therefore it requires much more interaction with the general cores despite using the same interface as SHA3. Gemmini is a neural network accelerator with a big systolic array for matrix multiplication and some nonlinear operations. This accelerator is located between the two extremes represented by SHA3 and Hwacha on the DMA-tightly coupled accelerator spectrum. There are also other accelerators like NVDLA [1], an open-source neural network accelerator developed by NVIDIA that has been integrated with RISC-V and the RoCC interface in the past [7].

There are some prior works regarding the control of accelerators. Among them, OpenAMP [2] is the most popular framework to control systems with heterogeneous cores. Such system is prevalent in today's system-on-chips which comprise of central application core, accelerators and IO cores, which OpenAMP intends to address according to their introduction. This frameworks allows different cores to run different OS, and it treats them as remote cores which can be controlled through RPC mechanism. Essentially, OpenAMP is a collection of inter-processor communication tools, and it requires special programming model in which the programmer must know the differences between cores and the system configuration. Our goal is to allow programmers to create application for large-scale networks of cores with their familiar programming models, and OpenAMP doesn't satisfy this requirement. OpenAMP is designed to support communication between cores with completely different ISA, like ARM and x86; however, we assume that all cores in the system share the same base ISA, which lead to a more efficient accelerator control mechanism.

2.2 Type-II Hypervisor

In this project, we intend to take over the management of some hardware resources from the commodity OS that does not recognize the heterogeneity of the hardware without extensive modification. Such design is very similar to Type-II hypervisor, which runs on bare-metal machines and control the access to hardware resources. Disco [5] is the earliest implementation of such system. It was originally designed to address the memory access problem of regular OS, which assume that all memory accesses have the same cost, in NUMA system. It virtualizes all IO devices as well as the physical CPU and multiplex them between each virtual machines, and it applies second-level address translation for virtual memory so that every OS see a continuous "physical" memory space. This also allows the hypervisor to move memory pages closer to the core to improve NUMA performance. At this point, the hardware platform we are assuming is not large enough to have significant effects on memory access latency differences, but we predict that we will eventually need to tackle the NUMA problem in our implementation for larger SoC or distributed systems.

Xen [4] is another type-II hypervisor with support of paravirtualization, which also exposes raw physical devices to virtual machines for higher performance. If the OS knows what kind of devices they are working with, they can apply proper optimization to improve the performance. This improvement comes with a price, though; we need to modify OS kernels so that they can recognize the devices besides the code changes to support Xen's page table translation. There are also some other commercial implementation of Type-II hypervisor, like VMWare's ESXi server [**?**], although obviously they cannot be easily used for research purposed without their assistance.

2.3 VM Live Migration

VM live migration is a technique to move the OS between different machines while allowing them to continue to serve incoming request except for a very short down time [6]. This technique can be used in the same data center for load balancing and removing machines for maintenance. It can also facilitate edge computing by moving VM with the application from a mobile device to a computationally more powerful machine, as in the case of Kimberly. Essentially, the idea of VM migration is to move the tasks to the machine that can best serve it. We adopt this idea for our project, although we are only moving the tasks, or threads, that need to be moved.

The original paper iteratively copies dirty memory pages to the new machines while the VM is still running, and it will stop the VM when the hypervisor determine there is no additional benefit to continue copying, then move the remaining pages over. Currently, we are targeting a shared memory environment, therefore explicit memory copying is not required. However, as we move into distributed or NUMA system, we may need to introduce this function to reduce the down time when the thread is being migrated.

3 IMPLEMENTATION

Our implementation of the proposed thread migration technique targets an SoC running Linux on heterogeneous RISC-V cores. The Linux and OpenSBI modifications made for this work are viewable on Github [1] [2].

[1] Linux: https://github.com/zitaofang/linux (on the migrate branch)
[2] OpenSBI: https://github.com/MaxBanister/opensbi-hetero (on the master branch)

3.1 The RISC-V System Software Environment

RISC-V has four privileged levels: user, supervisor, hypervisor, and machine [13]. Machine level is the level with the highest privilege and normally only reserved for a system's firmware. Hypervisor is the second level, primarily used by virtual machine hypervisors (both type-I and type-II). Supervisor mode is the privilege level where the most operating system code runs, and user mode is where user programs run in.

We choose not to implement this feature in the OS for two reasons:

(1) Thread migration in the OS kernel requires one to modify the scheduler, and most OS schedulers are coupled with the rest of the kernel to a high degree. An intrusive change to the OS scheduler leads to bugs and is not friendly to kernel developers.
(2) We would need to implement thread migration completely in every OS kernel we intend to support, which adds unnecessary complexity to our project. A small code footprint facilitates porting greatly.

Therefore, we opted to implement this feature underneath the OS level, either in H-level or M-level, with minimal glue code in the OS kernel.

At the point of writing, the RISC-V hypervisor extension has only just been approved. Since there is no hardware implementation that supports the ratified version of the spec, we choose to implement the thread migration function in RISC-V's platform firmware, OpenSBI.

To create a system for rapid thread migration between a general purpose core and an accelerator processor, we leverage OpenSBI, a RISC-V based runtime environment with buildable platform firmware. It serves the role of a second-stage bootloader, which is adequate for our purposes. We do not worry about an OS loader, because all of our tests take place in simulation, and the Linux kernel image can be pre-loaded into RAM. A platform-specific first stage loader that performs hardware initialization is also not necessary in a simulation environment. OpenSBI runs in machine mode (M mode), which is the highest privilege level in the RISC-V architecture. As such, it is the first entry point for exceptions, such as illegal instruction traps, barring delegation to a lower privilege mode. One of the purposes of the SBI runtime is to emulate unsupported instructions, so it is natural to use it to run foreign instructions by migrating the thread to a core that supports them. We also modify Linux as an example of how easily transparent accelerator thread migration can be supported by the operating system. The changes to Linux consist primarily of putting the waiting thread to sleep, and communicating with M-mode software through SBI calls.

3.2 Migration Flow

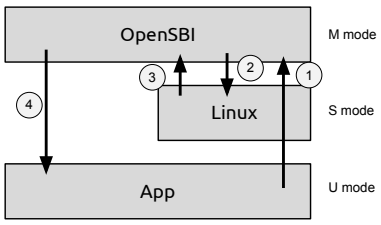

Figure 1: Migration Flow to the Accelerator

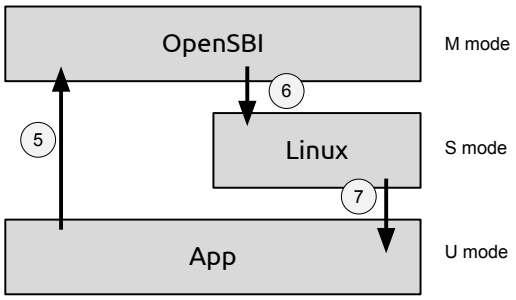

Figure 2: Migration Flow Back to Linux

In order to initiate a cross-hart thread migration, a thread needs only use an accelerator specific custom instruction (knowledge that a migration will occur may not be known at compile time), which will cause a privilege mode escalation to OpenSBI (1) in order to service an illegal instruction trap. The illegal instruction will be unrecognized by OpenSBI, and redirected to Linux (2) to (presumably) crash the program. This is still part of the normal illegal instruction trap flow. In Linux, instead of sending SIGILL to the process, the trap handler will decode the instruction and recognize that the opcode matches a pattern known to belong to a certain class of accelerator. It will then make an SBI environment call (3), to initiate the migration. Before leaving the kernel, it will put the current thread on a wait queue, with a condition to wake up that will be set upon completion. There are a few essential elements of the execution context that must be communicated in order to restart the thread on a different hart:

(1) Memory pages - We assume a shared memory SoC. Our demonstration relies on the fact that both the accelerator hart and the application hart have the same

address map, although this is not necessary for the system to work.
(2) Page table mappings - The address of the root page table (the satp register in RISC-V) is passed as an argument with the SBI call. Both processors must support the same Sv39 MMU with 3 levels of page table.
(3) Scalar registers - The architectural registers will be saved on the kernel stack upon taking an exception. Even with Meltdown mitigations in place, this part of the kernel's virtual address space is shared with the user, so it can be conveniently accessed alongside user memory.
(4) Accelerator state - This is state specific to the accelerator. Different accelerators may have different types and sizes of state, ranging from special registers, such as with a vector unit, or entire local scratchpads for intermediate computations. This memory is saved alongside the scalar registers on the kernel stack and passed to the accelerator during migration.

After making the SBI call, OpenSBI will start executing. It will collect the architectural register state from a pointer, passed as an argument, which points to the the kernel stack where the user registers are stored, and put it in a `task_context` structure. It will also record which application core interrupted it. This structure will then be passed as a message to the accelerator core. The message passing structure is abstracted away to allow for different hardware communication mechanisms. We use shared memory and interprocessor interrupts (IPIs) to achieve it in our simulator. We assume that the accelerator and the application core are part of the same cluster, and can thus send and receive IPIs to and from each other. Other SoCs may pass messages differently, such as through the use of hardware mailboxes. For this reason, the specific message queue implementation is abstracted away to allow for easy adaption to other hardware. For our implementation, we use software FIFOs to pass messages, and IPIs to indicate their arrival. Once the `task_context` structure is enqueued in the FIFO, the accelerator hart is interrupted by an IPI. OpenSBI running on the accelerator hart will dequeue this `task_context` and prepare to run it (4). OpenSBI uses a trick to run the new program, in which it overrides its return trap registers, so that when it returns to its pre-trap execution state, it runs the migrated thread. Execution finishes when the running thread encounters a synchronous exception, either because of a system call (usually to perform I/O), or because of a page fault. At this point, migration back to the application core occurs.

Migration back follows much in the same vein as migration to. Again, the architectural state is captured upon taking the exception (5) and passed as a message back to the Linux-running core that initiated the request. The interprocessor interrupt is taken in OpenSBI, now on the application processor, which writes a special exception cause value, so Linux knows what to do with this interrupt. It will also override the saved registers on the kernel stack, using the pointer that was saved before, with the new values of the architectural registers. Then, the exception is redirected to Linux (6), by jumping to its exception entry address. Linux will switch on the exception cause and see that a thread stopped running on the accelerator and should be restarted on the general purpose CPU. Additionally, we pass in the thread ID of the awakened task in the `stval` register, which is used in RISC-V to provide additional trap information. It can use the thread ID to wake up the thread on the wait queue, upon which it will continue running normally under a supervisor environment (7). The accelerator core, if there are no more tasks to run in its queue, jumps to an idle loop.

3.3 Page Fault Handling

One important implementation detail that deserves its own discussion is the handling of page faults. Although a programmer can avoid synchronous exceptions that trigger a migration back by e.g. not making any system calls, they cannot predict when a page fault will be taken, since the valid bits in a page table are soft state, and the operating system is free to page in and out memory as it sees fit. We have found that empirically, programs will have a period of frequent page faults as their working set size increases, and a period of relative tranquility as it works on that memory. Thus, for compute-heavy programs, migrations should not dominate the overall cycles of a program, though they are important to keep track of when analyzing performance. Since page faults are a common occurrence, it may be useful to have a separate message format to pass to Linux in order to allocate a page/check for an out of bounds access. We currently handle page faults uniformly with other types of exceptions, i.e., we restart the thread running under the supervisor, where the page fault can be serviced. This contributed to the ease of implementation, but in the future it may be useful to introduce a fast path for physical page allocations. Another solution might be having a shared global memory pool that is shared between different system software, although we do not explore this in this paper. The presence of thrashing in the face of frequent page faults, wherein a tight interleaving of accelerator ops and allocating new memory occur, can be ameliorated by using large TLB entries, although at the expense of greater internal fragmentation. It is also necessary that while the thread is executing on the accelerator hart that Linux not swap out or move the thread's memory. That is, the memory should remain pinned to physical RAM. This is possible to do in Linux natively by invoking the syscall `mlockall` with the `MCL_FUTURE` option. We experienced this

Scalar state

Exception PC	epc
Page Table Root Pointer	satp
Scalar register	x1-x31

Vector state

Vector start	vstart
Vector type	vtype
Vector length	vl
Vector Control and Status Register	vcsr
Vector registers	v0-v31

Table 1: Execution state captured at migration

syscall having a one time cost of 100,000's of cycles, which is paid at the start of a program as all the pages are faulted in. For applications in which this is an unacceptable delay, we came up with a mechanism by which a process can optimistically defer locking in the common case that pages are not evicted but still guarantee that that its pages are not moved. When Linux invalidates a page table entry, it performs a TLB shootdown to force every hart in the system to flush that entry from its TLB, so as to not have stale data. Linux makes an environment call into OpenSBI to do this. In OpenSBI, the shootdown is performed by sending a flush request to each hart via an IPI and waiting for them to acknowledge. If a flush request is received by the accelerator hart while an accelerated thread is running and the address space identifiers (ASIDs) match, it can stop the thread from running at that point in time and begin the migration back to Linux. This way, before Linux swaps out a page, the accelerator thread stops using the contents of that memory, and correctness is preserved.

3.4 RISC-V Vector Extension

For our implementation, we chose the RISC-V vector extension to illustrate how accelerator ops may work in conjunction with a regular program. Vector machines have several properties of tightly-coupled accelerators that we believe well fit the type of application we are targeting. They exhibit fine-grain data parallelism, and are intermixed with scalar instructions for things like pointer bumps, branches, and function calls/returns. The vector extension also has a rich, expressive ISA that eludes a simple MMIO control pattern. It is a reconfigurable architecture, meaning that it may change the data length, data types, or a number of other datapath control settings at runtime. It is hard to specify every type of computation one may want to perform with vectors, which is why our system does not abstract away the ISA and gives the programmer full control at the assembly level. The relevant state that must be saved and restored on a migration are the configuration registers vstart, vtype, vl, and vcsr, in addition to the vector registers themselves.

3.5 Linux Kernel Modifications

We modified the Linux kernel so that the threads are put to sleep when they are executing on the accelerator cores. This allows the Linux kernel to re-allot the application core to other thread. Due to the restriction on available memory in OpenSBI, we have to create a queue on the Linux side, implemented as a wait queue of a semaphore, to store the threads waiting to be migrated when the OpenSBI queue is full.

Essentially, the process can be understood as forking a bare-metal thread on the accelerator core from the OS thread, and the bare-metal thread will join the OS thread once it finishes running on the accelerator cores. The thread state of the sleeping OS thread will be replaced by the bare-metal thread upon joining.

Currently, the outbound migration is triggered by an illegal instruction exception. We modify the trap handler to detect whether the offending instruction is possibly part of the custom extension available on other cores in the system but not on this core. If it is not the case, we fall back to the regular illegal instruction handler, which will terminate the program normally. Otherwise, we initiate the migration process and put them in the OS outbound queue. If the OpenSBI queue is not full (whose size is represented by a semaphore in Linux kernel), we will pin all memory pages used by the cores and perform an SBI call (which is similar to a syscall but evaluates our permission from the S mode to M mode). This call will pass the necessary info to recover execution to OpenSBI, like pointer to the thread control block. After the SBI call returns, the thread will enter a wait queue to wait for the incoming migration flag of the thread coming back. The flag is actually a global variable marking the thread ID of the returning thread, and the thread will be checking if the global variable matches its thread ID during a wake-up check.

Although the accelerator core will use an IPI to notify the main/application core that the thread is coming back, OpenSBI will intercept and handle the interrupt. Instead, we will receive a platform-specific interrupt from OpenSBI once it finishes writing the state back to the data structure of our threads. The only thing that Linux kernel needs to do is to set the global variable above to the incoming thread and increment the queue length semaphore, then wake up the thread.

As this is an RISC-V specific feature, we restrict our kernel modification to the RISC-V architecture section.

4 METHODOLOGY

4.1 Spike

We benchmark our fast thread migration implementation using Spike, the RISC-V ISA-level simulator. Spike is not cycle accurate to hardware implementations of RISC-V, but is capable of accurately counting instructions during events relevant to thread migration. We decided to use Spike for two main reasons. First, Spike allows us to quickly test our system software modifications within the order of seconds, much faster than if we were to use a cycle-accurate simulator like FireSim [10]. It is also flexible enough to be configured with a variable number of RISC-V cores, each with its own set of extensions. A real RISC-V Second, there is no currently available RTL implementation of a RISC-V heterogeneous ISA SoC. There is one under development in Berkeley's ADEPT computer architecture lab, but it is not mature enough to boot Linux, which is a prerequisite for running fast thread migration.

4.2 Benchmarks

We examine the behavior of fast thread migration under a set of various micro- and macro-benchmarks. The micro-benchmarks used for this project, which mainly tested the functionality and performance of single migrations, are expanded on in Section 5. One main macro-benchmark used in this work is a RISC-V vector program called `dgemm`, short for double-precision, general matrix-matrix multiplication. We use an assembly-level implementation of `dgemm` because compilers for the RISC-V vector extension are still at an early stage. `dgemm` is a typical workload that would be accelerated by a RISC-V core supporting the vector extension, and is a good stress test of the functionality and correctness of fast thread migration. Though we would have liked to have tested fast thread migration on larger workloads such as neural networks, current implementations of neural networks for the RISC-V vector extension are experimental and were not able to be compiled within the time constraints of this project. We leave this level of testing as future work.

End-to-end performance, as well as progress through the `dgemm` benchmark was measured by instrumenting the code with reads of the `cycle` RISC-V Control and Status Register, which is available to read in the user space. The benchmarking code used for this project is publicly viewable on Github [3].

4.3 Baseline and Metrics of Success

When programs such as `dgemm` run in the fast thread migration environment, they automatically migrate to the accelerator core when vector instructions are encountered, and back to the application core exposed to Linux when a Linux kernel event such as a syscall or page fault occurs. However, in an unmodified RISC-V Linux system environment under the same hardware architecture, the program would simply fail to execute on the application core. Therefore, the baseline performance we compare to is one where the Linux syscall `sched_setaffinity()` has been manually added to the program, so that accelerated code executes on the correct core as necessary. An example is shown in Listing 1. This lends itself to one of the metrics of success of this project, the burden removed from the application developer.

Listing 1: Example of Linux core pinning baseline. Code below is rendered unnecessary by fast thread migration.

```
#include <sched.h>

int main() {
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(1, &mask); // Assumes CPU 1 is accelerator core
    sched_setaffinity(0, sizeof(mask), &mask);

    ...
}
```

The second main metric of success for this project is performance. Since fast thread migration may necessitate some migration not needed in the baseline case, we do not expect fast thread migration to significantly outperform our baseline. However, we do expect competitive performance as migration cost is low, and not running Linux on the accelerator may actually provide a performance benefit since other kernel-level tasks cannot be scheduled on the accelerator core and interleaved with the benchmark program.

5 EVALUATION

5.1 User Experience

As discussed in Section 4.3, one of the main goals of this project is to reduce the burden on application developers targeting heterogeneous ISA systems. Listing 1 shows the process to pin a program to a certain core exposed to Linux. Though the number of lines of code is small, the baseline requires the program to be aware of which core has the correct extension (in the example, core 1). If the program is pinned to the wrong core, the Linux scheduler will be unable to resolve this issue and the program will crash. Additionally, in a real world scenario like a datacenter, where many cores may be distributed across different sockets and servers, managing these variables becomes more and more complex. In fast

[3] https://github.com/charleshong3/262-proj-benchmarks

thread migration, only the platform firmware must be aware of which cores support which ISA extensions.

5.2 Individual Migration Performance

We first evaluate the performance on one individual migration across a core, to verify that our implementation functions and will not add excessive performance cost to the program. In order to measure individual migration performance, we run a simple program that runs one vector instruction, causing the firmware to migrate the program to the accelerator core, then the syscall `getpid()`, causing a migration back to the application core. We migrate back and forth between the application core and the accelerator core in order to get accurate cycle count measurements on taken on the same core, since in Spike, each core has its own cycle counter that constantly increments. In the baseline, the program is first pinned to the application core to ensure that migration occurs. A cycle measurement is taken, and the program is pinned to the accelerator core, then back to the application core. As shown in Figure 3, the results of this measurement show that fast thread migration is faster than core pinning with `sched_setaffinity()` in Linux, by a magnitude of about 2.1×.

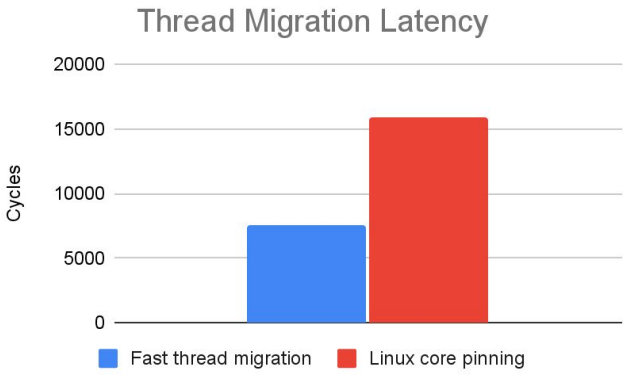

Figure 3: Latency of one thread migration under fast thread migration, and under the baseline of Linux's `sched_setaffinity()`.

5.3 Scaling

Another important consideration of any system software implementation is scaling to parallel execution. Since we implement fast thread migration using a simple first-in-first-out queue for multiple programs that attempt to migrate to the same accelerator core, when multiple programs content for access to the same accelerator core, we do not expect any significant increase in runtime beyond the sum of the runtimes of each program, and the migration time. In order

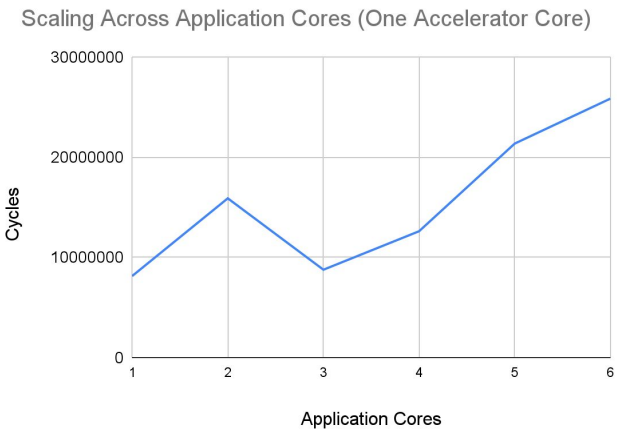

Figure 4: Runtime of the benchmark program on a variable number of application cores. This benchmark shows that there is no additional cost associated with multiple programs attempting to access the same accelerator.

to confirm this, we benchmark fast thread migration with a program that spawns one UNIX `pthread` for each available application cores. Each `pthread` runs for a short duration on the same accelerator core, and we vary the number of application cores in the system (and therefore the number of parallel program instances spawned. Figure 4 shows the results of this benchmark. The program performs mostly as expected, with the cycle count of the whole program increasing mostly linearly. We are able to consistently reproduce a slowdown at 2 application cores, but are unable to identify the root cause of this unexpected performance issue due to the time constraint of the project. Nonetheless, this benchmark achieves its goal of showing that the runtime of the program does not scale worse than linearly with the number of parallel processes vying for the same accelerator core. In fact, the program scales better than linearly with a small number of parallel processes, likely due to program starting and process spawning overhead.

5.4 End-to-End Performance

It is also important to examine the performance of fast thread migration on macro-benchmarks that show the effectiveness of the system in real-world applications. As discussed in Section 4.2, the main macro-benchmark we use to evaluate fast thread migration is `dgemm`, a double-precision general matrix multiplication. As discussed on Section 4.2, we measure the progress of the program by instrumenting `dgemm` with reads of the `cycle` RISC-V Control and Status Register. We estimate progress through the program by calculating the proportion

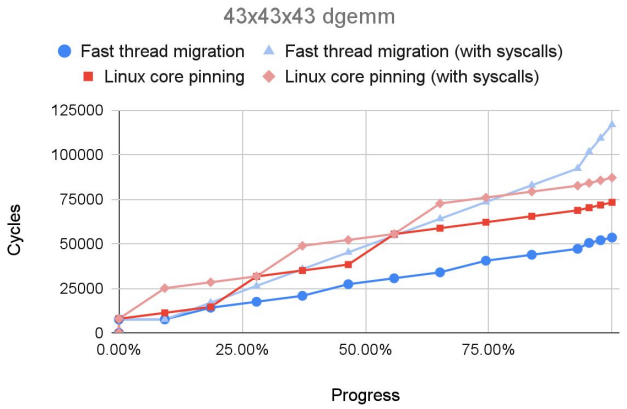

Figure 5: Runtime of a 43 by 43 by 43 matrix multiply in each of the test environments. `dgemm` is also run with 14 syscalls interspersed throughout the program, in order to simulate the usage of kernel mode program calls like `printf`.

of multiply-accumulate operations that have occurred so far, relative to the total number of multiply-accumulate operations in the whole program.

Because our `dgemm` implementation does not have any additional system calls inside the program, and causes only a relatively small number of page faults, we find that `dgemm` actually runs faster with fast thread migration than it does in the baseline, by about 27%, or 20,000 cycles, as shown in Figure 5. Since the startup cost of both programs (initial thread migration, Linux core pinning) is within a few thousand cycles, we suspect that this difference is caused by additional overhead imposed by Linux, which in the baseline case is running on both the application core and the accelerator core. This is evidenced by the fact that although `dgemm` proceeds at a similar rate most of the time in both the fast thread migration and baseline cases, the number of cycles needed to proceed to the next cycle count measurement tends to jump a couple times throughout the program, likely when another programs is being scheduled by Linux on the accelerator core.

Because our base `dgemm` implementation does not have trigger many thread migrations, we decided to simulate the effect of the user adding system calls like `printf()` by interspersing calls to the `getpid()` syscall throughout the program. Specifically, this system call is made 14 times throughout the program. When these system calls are added to the program, `dgemm` predictably begins to perform worse in the fast thread migration environment than in the baseline environment. As shown in Figure 5, `dgemm` with fast thread migration has a runtime of about 34%, or 30,000 cycles longer

Average cost of `getpid()` syscall

Baseline Linux	988 cycles
Fast thread migration	4524 cycles

Table 2: System call cost incurred in baseline and fast thread migration environments. Additional cost of syscall in fast thread migration: 3536 cycles.

than it does in the baseline case. This benchmark also allows us to estimate system call cost in fast thread migration. As shown in Table 2, we measure the average cost of the `getpid()` system call as 988 cycles on average in the baseline, and 4524 cycles on average when a system call run from the accelerator core causes a thread migration in the fast thread migration environment. This means that on average, each migration from the accelerator core to the application core, then back (since `dgemm` continues to run vector instructions) incurs a cost of 3536 cycles. Interestingly, this is less than the originally measured cost of a single thread migration, which shows that it is likely that there is some additional overhead associated with the first thread migration that occurs in a program.

A last macro-benchmark that we carried out on the fast thread migration system was a version of `dgemm`, repeated 1000 times. This benchmark verifies the functionality of and demonstrates the performance of a program running for an extended period of time under fast thread migration. The results of this benchmark are shown in Figure 6. We see that the program has an almost identical runtime under both fast thread migration and the baseline, further showing that there is no performance cost to fast thread migration unless there are a significant number of additional system calls made by the program.

6 FUTURE WORK

Due to the time constraint, we only implemented the basic function for this system. We are planning the following in the future:

6.1 Hypervisor Implementation

The hypervisor extension for RISC-V was formally ratified two weeks before this paper is written. At this time, there is no hardware platform that implement this instructions. Once our targeted hardware platform is updated to support hypervisor, we will move the implementation from the firmware to the hypervisor level so that it can be portable without modifying the code.

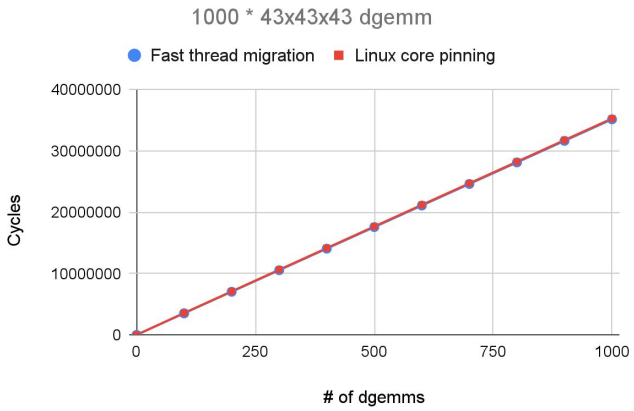

Figure 6: Runtime of the same `dgemm` program above, repeated 1000 times.

6.2 Migration Hint

Currently, we initiate a migration when there is an instruction the current core cannot handle. This may cause long latency issue after the thread finishing an accelerator code segment or IO segment and attempt to execute latency-critical codes not supported by the current core. Also, if some intensive general-purpose computation immediately follows the accelerator code, leaving the thread on the accelerator control core harm our performance since accelerator control cores are normally small in-order core with low performance. By introducing hint call, we can not only avoid the unwanted latency spike in some code segment but also encourage programmers to optimize their code with the thread migration mechanism in mind, like avoiding interleaving IO and accelerator function calls. We don't intend to make the call mandatory but allows the legacy program to also run on the thread migration system without code change, though with some potential performance degrade. Besides, in the case where memory copying is needed, an early hint can allow us to employ iterative copying to reduce the thread down time during migration.

6.3 Distributed System Support

Shared memory space is not a requirement for thread migration, although it does simplify the implementation and allow us to obtain initial results quickly. We hope to extend this mechanism to support migration over network, like in a data center setting. The only difference in this case is that we can no longer directly access the memory pages on another machine, and we can solve this problem in the hypervisor level by modifying page fault handler to fetch content from another machine.

6.4 Advanced Scheduling

Our current implementation only use basic FIFO scheduler when deciding which core to send the task to. We are considering implementing a more sophisticated scheduler for tasks in the future. Also, since every accelerator has a control core that can be used to execute general-purpose tasks, we are also planning to offload some tasks from the application core to these accelerator control cores when there is no accelerator tasks, therefore improving the hardware utilization of the entire system.

7 CONCLUSION

As we approach the end of Moore's Law, hardware accelerators are becoming more important to continue the performance improvement in computer system. The increasing number of hardware accelerators needed for modern SoCs presents a unique challenge for designers to control the heterogeneous computing resources in such systems. In this work, we show that it is possible to implement a thread migration system that automatically delivers a program to the correct core at runtime, through firmware modifications and some changes to the Linux kernel. This system adds little end-to-end performance cost and even implements the same operation as Linux's `sched_setaffinity()` system call with 2.1x faster performance. With fast thread migration, programmers and hardware designers can target SoCs with a control processor per accelerator, allowing higher accelerator availability, especially in the case of tightly coupled accelerators where the processor must keep issuing instructions to keep the accelerator busy. With some improvement on the hint mechanism, we believe that we can create a practical, high performance accelerator control framework to tackle the increasingly complex heterogeneous environment.

8 ACKNOWLEDGEMENTS

We would like to thank:

- Jerry Zhao, Albert Ou, and Professor Krste Asanovic for their assistance in navigating the RISC-V heterogeneous ISA system environment.
- Stephanie Wang and Professor John Kubiatowicz for helping develop this project from an idea to a practical, scalable implementation.

REFERENCES

[1] 2017. NVIDIA Deep Learning Accelerator (NVDLA). https://nvdla.org/.

[2] 2021. The OpenAMP Project. https://www.openampproject.org/.

[3] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[5] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. 1997. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 412–447.

[6] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 273–286.

[7] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2019. Integrating nvidia deep learning accelerator (nvdla) with risc-v soc on firesim. *arXiv preprint arXiv:1903.06495* (2019).

[8] Antoine Fraboulet and Tanguy Risset. 2007. Master interface for on-chip hardware accelerator burst communications. *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 49, 1 (2007), 73–85.

[9] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. 2019. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. *arXiv preprint arXiv:1911.09925* (2019).

[10] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 29–42.

[11] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and K Asanovic. 2015. The Hwacha vector-fetch architecture manual, version 3.8. 1. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262* (2015).

[12] Colin Schmidt and Adam Izraelevitz. 2015. A Fast Parameterized SHA3 Accelerator. (2015).

[13] Asanovic Krste Waterman, Andrew and John Hauser. 2021. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture. *RISC-V International* (2021).