Flick: Fast and Lightweight ISA-Crossing Call for Heterogeneous-ISA Environments

Shenghsun Cho, Han Chen, Sergey Madaminov, Michael Ferdman, Peter Milder
Stony Brook University
{shencho, smadaminov, mferdman}@cs.stonybrook.edu, {han.chen.2, peter.milder}@stonybrook.edu

***Abstract*—Heterogeneous-ISA multi-core systems have performance and power consumption benefits. Today, numerous system components, such as NVRAMs and Smart NICs, already have built-in processor cores with ISAs different from that of the host CPUs, making many modern systems heterogeneous-ISA multi-core systems. Unfortunately, programming and using such systems efficiently is difficult and requires extensive support from the host operating systems. Existing programming solutions are complex, require dramatic changes to the systems, and often incur significant performance overheads.**

To address this challenge, we propose Flick: Fast and Lightweight ISA-Crossing Call, for migrating threads in heterogeneous-ISA multi-core systems. By leveraging hardware virtual memory support and standard operating system mechanisms, a software thread can transparently migrate between cores with different ISAs. We prototype a heterogeneous-ISA multi-core system using FPGAs with off-the-shelf hardware and software to evaluate Flick. Experiments with microbenchmarks and a BFS application show that Flick requires only minor changes to the existing OS and software, and incurs only 18µs round trip overhead for migrating a thread through PCIe, which is at least 23x faster than prior work.

***Index Terms*—Heterogeneous-ISA, Multi-Core, Thread Migration, Virtual Memory, FPGA**

I. Introduction

Multi-core processors, including CMPs and MPSoCs, have dominated CPU architectures for over a decade. To keep improving processor performance within constrained energy budgets, researchers and industry have proposed and deployed heterogeneous multi-core processors, including single-ISA heterogeneous multi-cores [1], [2] and heterogeneous-ISA multi-cores [3], [4]. Such processors are deployed in systems ranging from massive warehouse-scale datacenter servers to smartphones and small embedded systems. Among the heterogeneous multi-core processors, heterogeneous-ISA multi-core CMPs and MPSoCs require the highest effort to effectively utilize the diversity of both the architecture (ISAs) and the microarchitecture for performance and power efficiency.

In addition to the intentionally heterogeneous-ISA CMPs and MPSoCs, many modern system components, such as Smart NICs [5], NVMes [6], and FPGAs [7], [8], include built-in general-purpose processors. These *NxPs* (Near-x-Processors), such as near-to-communication (Smart NICs), near-to-data (NVMes), or near-to-accelerator (FPGAs), typically use ISAs different from the host processors, with their microarchitectures tailored to run the target workloads efficiently. As a result of incorporating NxPs, numerous modern computing systems are already heterogeneous-ISA systems. This fact presents a new opportunity to spread computation and communication across cores with different ISAs, achieving better performance and power efficiency.

However, although the benefits of heterogeneous-ISA multi-core systems are attractive, these benefits come with significant challenges for the software developers of such systems. These challenges are particularly pronounced for NxPs, which have their own private memory, calling for software developers to use the NxPs in an "offload engine" programming style, similar to CUDA or OpenCL for GPUs. The offload-engine software organization differs greatly from the conventional general-purpose multi-core programming environment familiar to many developers. In such NxPs, developers typically handle not only the communication between the host processors and the NxPs but also the data addressing and movement, further increasing the complexity (and potentially reducing the performance) of the resulting systems.

To address the programmability challenges, researchers have proposed new operating systems [9], [10] and modifications to existing operating systems [11], aiming to hide the complexity of heterogeneous-ISA multi-core systems and provide a traditional multi-core environment familiar to developers. These approaches typically assume that software threads must be permitted to freely migrate between cores at any execution point, enabling thread migration based on, for example, dynamically profiled system load. To support such thread migration capabilities, prior work employs many complex techniques, including run-time binary translation and thread state transformation [11], integrating these mechanisms into heavily modified or even brand new operating systems. Unfortunately, although they generally achieve their goal of simple programmability and convenience, these approaches impose major performance overheads on the execution of the heterogeneous-ISA software.

In this work, we observe that conveniently-programmable heterogeneous-ISA systems are possible, and high performance is achievable, by sacrificing a small amount of purity associated with the migrate-any-time systems. Particularly for NxP systems, where the software developers have a clear idea about the partitioning of the software and the data, many of the techniques from the prior work introduce unnecessary complexity and overhead. The system should provide the illusion of a simple and efficient shared memory multi-core environment to users, enabling them to easily develop software

that uses cores with different ISAs. We, therefore, take a principled approach to identify the minimum requirements for a heterogeneous-ISA multi-core programming environment that is easy to use, requires minimal deviation from traditional single-ISA multi-core programming, and can be achieved without introducing major modifications to the existing operating systems and hardware.

We develop Flick: Fast and Lightweight ISA-Crossing Call for migrating threads in heterogeneous-ISA environments. To maintain expectations of conventional multi-core systems, Flick guarantees a unified physical memory address space, and the same software thread observes the same virtual memory address space on all cores, regardless of ISA. Given hardware that can support such a memory organization, we create a convenient and lightweight generic thread migration mechanism capable of transferring threads between cores with different-ISAs at function-call boundaries. Flick uses mechanisms that are readily available in modern systems, such as access-control bits in page table entries, and makes non-intrusive use of existing software techniques such as page fault handling and dynamic library loaders. Ultimately, Flick allows software developers to take conventional software that targets multi-core systems and indicate which functions should run on which ISA, thereby directing the system to transparently migrate the running thread to the desired core when the program makes function calls across the designated ISA boundaries.

We prototype and evaluate Flick on a heterogeneous-ISA multi-core platform using a modern off-the-shelf server with a PCIe connected FPGA. On this real system, we show that Flick achieves 18µs migration times with fewer than 2K LoC modifications in the off-the-shelf Linux operating system, as opposed to hundreds to thousands of microseconds overhead and tens of thousands LoC modifications in the prior work. Our experiments with a BFS application show that Flick achieves up to 2.5x speedup when compared to a system that does not perform thread migration.

The rest of this paper is organized as follows. Section II provides an in-depth discussion of the background and motivation of this work. Section III describes the architecture of Flick. Section IV describes the details of the implementation. Section V presents our case studies and evaluation. Section VI discusses related work and Section VII concludes.

II. Background and Motivation

Today's systems use heterogeneity to improve performance and efficiency. While the most common forms of heterogeneous systems are still general-purpose cores with accelerators such as GPUs, researchers are also exploring heterogeneous multi-cores, which integrate general-purpose cores with different characteristics in one system. By leveraging each core's strengths, such as high performance or low power consumption, heterogeneous multi-core systems achieve high overall execution efficiency, while maintaining the flexibility to support a wide range of tasks because the execution units are still "general purpose." Among the various types of heterogeneous multi-core systems, heterogeneous-ISA multi-core systems represent an extreme case for combining cores with different architectures (ISAs) to achieve efficiency with both specialization (heterogeneity) and parallelism (multi-core). Studies have shown that heterogeneous-ISA multi-core systems can achieve 21% speedup and up to 23% energy savings compared to homogeneous multi-core systems [3].

A. Heterogeneous-ISA Multi-Core Integration

There are several ways to integrate cores with different ISAs, primarily depending on the physical locations of the cores. On one end of the spectrum, different cores can be tightly coupled into one chip to form a heterogeneous-ISA CMP or MPSoC. The cores can share the same memory hierarchy and have low-latency communication with each other. However, although heterogeneous-ISA CMPs have been proposed in academia [3], they are still rarely seen in the field. On the other end of the spectrum, a data center that connects processors with different ISAs using a high-speed network is also a heterogeneous-ISA system [12]. However, because the cores are spread across servers, developers are limited to distributed programming techniques, and the communication overheads are high.

An interesting integration option, which lies in the middle, spreads heterogeneous-ISA cores across multiple chips, but still within one machine. Many components of modern systems already have their own cores, using different ISAs from the host cores, executing jobs specifically targeted for those components. Common examples include servers with Smart NICs, NVMe controllers, and FPGAs with integrated cores. The cores are typically used as NxPs, as they reside close to the subjects they are working on, such as network, storage, or accelerators, providing low data access latencies for their tasks. Such tailored architectures make the NxPs highly efficient for their target workloads. More importantly, because these NxPs are located within the same machine as the host CPUs, it is relatively easy for them to share the memory space and communicate with the host CPUs. As a result, if the NxPs can be exposed to and utilized by developers, many of today's systems can provide new opportunities to achieve better overall system performance and efficiency.

B. Programming Style and Challenges

Unfortunately, although hardware designers are making their best effort to improve system efficiency through heterogeneity, heterogeneous-ISA multi-core systems inherit all of the programming challenges from heterogeneous systems, in addition to facing the obstacles of concurrently using multiple ISAs. The difficulty of developing software for heterogeneous-ISA multi-core systems introduces a heavy burden on the developers, limiting adoption.

Unlike GPUs, which have widely adopted programming frameworks such as CUDA and OpenCL, there are no generally-accepted solutions for software on general-purpose cores with different ISAs. In some cases, multiple independent operating systems are run across the heterogeneous-ISA

cores. Developers must partition the software into completely independent applications, compile and run them separately for each copy of the operating system, and manually handle the communication and data movement using RPCs. For NxP systems, developers usually treat the NxPs as slave processors and explicitly use the offload engine programming style, where the host CPUs prepare data and job commands in the form of descriptors and let the slave cores fetch the descriptors and execute the jobs.

Notably, adapting the GPU programming frameworks for heterogeneous-ISA multi-core systems does not provide a general programming solution for most NxPs. GPU frameworks are designed around computational kernels, targeting massive parallelism, bulk data transfer, and coarse grain control from the host. Software for traditional multi-core systems that would be ported to NxPs either does not share these characteristics or would require significant changes to adapt it for the GPU-centric programming frameworks.

Without a convenient NxP programming framework, developers must tear apart the original software and handle each piece separately, breaking the integrity of the software and adding extra work for software maintenance. Rewriting software in different programming styles and frameworks is time-consuming and error-prone, and may require additional debugging and verification efforts. Manually handling communication and data movement between cores requires careful work to avoid performance loss. As a result, although heterogeneous-ISA multi-core systems sound attractive in theory, the problems faced by software developers make developing such NxP systems less attractive in practice.

C. Heterogeneous-ISA Operating Systems

To address the programmability challenges, researchers have proposed creating new or modifying existing operating systems specifically for heterogeneous-ISA multi-core systems [11], [13]. The goal is to provide a shared-memory homogeneous multi-core environment to software developers while hiding all complexity within the operating systems. Developers can design software with their convenient and familiar shared-memory programming style, or compile and run existing software, ideally without any modification.

The key to hiding the complexity is the ability of the operating system to migrate threads between cores with different ISAs, transparently to the users. When the software encounters code that needs to run on a core with an ISA different from the current one, the operating system suspends the thread on the current core, transfers the thread state to the other core, and resumes execution on the target core. The operating system must handle the ABI (Application Binary Interface) differences between ISAs, including calling convention, stack layout, and system calls, to ensure that the thread runs correctly on the target core. In addition to handling the heterogeneity, operating systems must also manage shared resources, such as memory and storage, so the threads can access the data and instructions they need from all cores they are running on.

While offering a friendly programming environment to developers, operating systems must minimize the impact on the efficiency that comes along with the convenience. The abstraction layer between the software and the heterogeneous-ISA multi-core hardware introduces overhead and affects the system efficiency. Thread migration requires extra cycles to suspend and resume threads, as well as handle the ABI differences. Transferring thread state consumes bandwidth and time. Managing system resources and providing availability also requires effort from the operating system. These overheads can reduce the benefits of the heterogeneous-ISA multi-core systems, forcing developers to return to the complicated offload engine programming style to achieve higher efficiency.

Although prior works provide useful abstractions to developers, they suffer from high overhead. To enable threads to freely migrate between cores with different ISAs at any execution point, prior work employed many complex techniques, such as binary translation and stack frame transformation, which can take hundreds of microseconds to several milliseconds. Compared to context switches or page faults, which usually take a few microseconds, high overhead prevents frequent thread migration, limiting usefulness.

In addition to the runtime overhead, to implement the thread migration techniques and support data management, prior work either develops new operating systems from scratch, which limits their use to academia or research labs, or heavily modifies existing operating systems, which creates difficulties for maintenance. Either situation hampers the adoption of heterogeneous-ISA multi-core systems.

D. Opportunities

High-overhead techniques for integrating heterogeneous-ISA cores are typically not appropriate for NxP systems. Because NxPs are usually less powerful than the host CPUs, the system should migrate a thread to an NxP only when the thread is working on something that the NxP is close to. For example, when running graph workloads where the graph is stored in NVMe, only the graph traversal function should run on the cores close to the NVMe storage. The rest of the program, including the operations after the desired nodes have been found, should still run on the host CPUs. This characteristic largely reduces the number of migration points in the program, and also makes the migration points static, usually at the function call and return boundaries. By utilizing this fact, the operating system no longer needs most of the techniques used in the prior work, eliminating the overhead and complexity that comes with those techniques.

The unified memory space between the host and NxP provides another opportunity to reduce the migration overhead for NxP systems. Conventional software threads must have access to the shared data and instructions from any core where they run. The system bus offers unique opportunities for fast and efficient unified memory in NxP systems because system interconnect protocols such as PCIe already allow the CPUs and devices to share the same global memory view. Interfaces such as OpenCAPI [14] and Gen-Z [15] will provide even

lower latency and better cache coherency than PCIe. With a unified memory space, the thread migration process does not require address translation and can avoid the serialization and transfer of data between different memory islands.

Restricting the migration points to function call and return boundaries and utilizing a unified memory space for the host and system components enables Flick, a Fast and Lightweight ISA-Crossing Call mechanism for migrating threads in NxP systems. Flick (1) attains low-overhead thread migration, (2) retains the same programming and execution environment as a homogeneous multi-core system, and (3) achieves this with minimal changes to the existing compiler toolchains and operating system.

III. The Flick Architecture

The Flick architecture comprises several inter-dependent components, including the hardware and software to support a unified memory space, the thread migration mechanism, generation and execution of multi-ISA binaries, and modifications to the operating system. This section describes each of the components in detail.

A. Unified Memory Space

Flick's thread migration mechanism relies on a unified memory space, where virtual addresses within a thread's address space point to the same physical addresses from both the host CPUs and NxPs, and all physical addresses point to the same system component, such as host memory or NxP local memory. A unified memory space allows software threads to access the same instructions and data while running on different cores, without the need to modify application source code. Moreover, it is possible to directly pass pointers between the host and NxP without instrumenting the code with address translation.

To enable a unified physical memory space, Flick uses a PCIe bridge to map the host memory into the NxP address space, and at the same time exports the NxP memory and peripherals to the host system as PCIe BAR regions. For a unified virtual memory space, the NxP utilizes the local TLB to translate the virtual addresses to the same physical addresses as the host CPUs, with the help of a local MMU that walks the host page table structures on NxP TLB misses, using the same Page Table Base Register (PTBR)[1] as the host CPU, as shown in Figure 1. By using a local MMU to populate TLBs on demand, Flick avoids host interaction while the NxP is performing context switches.

The main challenge in having MMUs local to the NxP is the TLB miss penalty, due to the long latency of page table walks on page tables stored in the host memory. However, although it is difficult to reduce the page table walk latency, the TLB miss rate is mitigated in modern systems by using huge pages, which amortizes the cost of the high latency operations across many accesses. More specialized NxPs also have the freedom to implement other address translation mechanisms, such as segments [16], [17], to further reduce TLB misses.

[1]For example, the CR3 register in the x86 architecture.

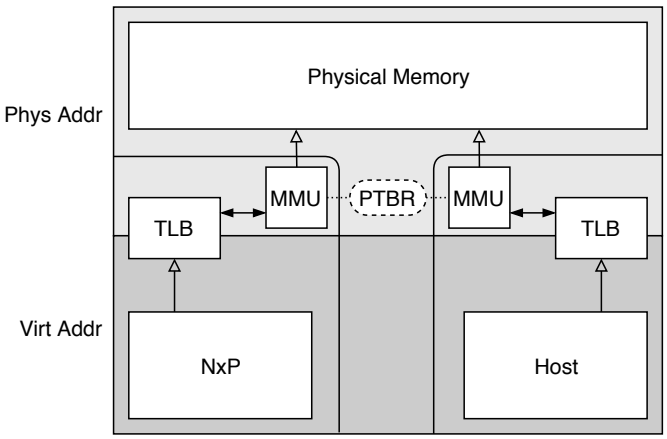


Fig. 1. When executing a thread, both the NxP and the host use the same physical address as the Page Table Base Register (PTBR) and therefore have the same view of virtual memory. This allows addresses to be shared between cores seamlessly.

B. Thread Migration on No-Execute Page Fault

Migrating threads between the host and NxP cores requires extra code at the function call and return boundaries. To make the migration transparent to the software developers, the migration code must be inserted into the program by the compiler, or at runtime by the operating system.

Although letting the compiler insert migration code using customized system calls appears straightforward, it has two major drawbacks. First, software often uses function pointers, which can point to any function, with no way for the compiler to know whether the pointer targets a host function or an NxP function at runtime. To handle function pointers, the compiler would be forced to insert migration code at the beginning and end of all functions. Moreover, some functions can be invoked by both host cores and the NxP cores, requiring such migration code to incur additional overhead to check where the thread is currently running to determine whether or not to trigger a migration. Second, typical software routinely calls functions in pre-compiled shared libraries (e.g., the standard C library), which do not have migration code inserted. This makes supporting migration for programs that link against shared libraries challenging with a static compiler-based approach.

Instead of inserting migration code using a compiler, Flick uses the OS to trigger and manage thread migration. We use the *Non-Executable (`NX`)* bit of the x86 page table entries to implement page fault triggered migration, shown in Figure 2. For functions that should run on the NxP, we load binaries into their own memory pages and mark the `NX` bit in the page table entries for these pages. When the host CPU tries to execute such functions, an instruction page fault occurs. The operating system page fault handler gets the function address and passes it to a user-space migration handler linked into the application binary. The migration handler gathers the parameters of the function call and other necessary information, such as the values of PTBR and PID, and performs a "call migration" from the host to the NxP. The NxP scheduler uses the provided

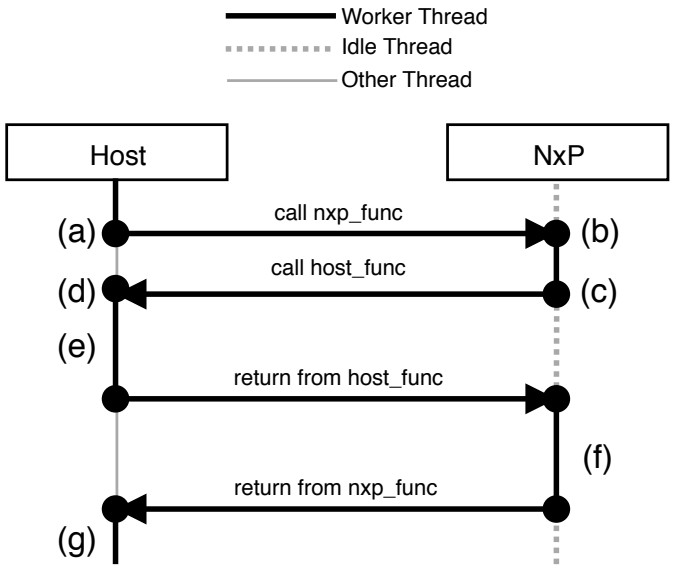


Fig. 2. (a) The host calls an NxP function. It encounters an NX page fault and the thread migration process begins. The migration handler on the host sets up the host-to-NxP call descriptor, sends it to the NxP side, and deschedules the thread from the host processor. (b) The descriptor is picked up on the NxP side and the thread is context switched in to execute the target function. (c) During the NxP function execution, the NxP calls a host function, which triggers a page fault on the NxP side and sends an NxP-to-host call descriptor. (d) The host receives the descriptor and starts executing the target host function. (e) The host finishes the function and sends the host-to-NxP return descriptor. (f) The NxP resumes the original target function and eventually sends the NxP-to-host return descriptor to the host. (g) The host receives the return value from the descriptor and continues execution.

information to call and execute the target function. When the function finishes execution, the NxP scheduler performs a "return migration" from NxP to the host, which includes triggering an interrupt on the host CPU to take back the thread and get the function's return value.

Similarly, the NxP can call functions that should run on the host CPU using the same mechanism. The difference is that, on the NxP side, we use the NX bit in the opposite direction. When the NxP is trying to call a function that resides in a page without the NX bit set, a page fault occurs and the page fault handler of the NxP performs a "call migration" from the NxP to the host CPU. After the function finishes execution, the host CPU performs a "return migration" from the host to the NxP and the thread resumes execution on the NxP.

Although a page fault for a function call may appear to be a significant overhead, because a developer explicitly designated the function to run on the NxP side, the benefit of running on the NxP must be significant enough to mitigate this overhead. Notably, gathering information such as the function call arguments and passing them to the NxP is a necessary overhead even for the conventional offload style programming model. The combination of the unified memory space and the NX-bit-triggering thread migration serves as a low-overhead mechanism that allows transparent migration from the host to the NxP core, and vice versa.

C. Multi-ISA Binaries

Flick aims to provide the same software execution environment as the single-ISA system, where users expect to launch just one binary file for their application. As a result, the binary file running on heterogeneous-ISA systems must contain instructions for both ISAs. This idea is similar to Universal Binary [18] or FatELF [19]. However, for Universal Binary and FatELF, the entire program is compiled into instructions for each ISA and stored in one binary file, whereas in Flick, the program is logically partitioned into different ISAs at function granularity, and each function has its target ISA and will only run on a core supporting that ISA. Instead of letting the linker prepare several copies of binaries for the entire program in one executable, the Flick linker handles ISAs simultaneously, resolving addresses of the functions for different ISAs within one shared address space. To prepare and execute multi-ISA binary files, the toolchain and compilation process require changes, such as separating the `.text` section of each ISA, relocating the symbols using each ISA's relocation methods, and marking the NX bits of the page table entries while loading the binary. These toolchain changes are transparent to the application developers, requiring only the addition of command-line flags to the compilation and linking processes.

D. Instruction and Data Placement

In NxP systems, each core can access all memory regions in the system, including the host memory and the NxP local memory, which creates a NUMA environment. The host cores can access the host memory with lower latency than accessing the NxP local memory, and visa versa. Ideally, each core should access data within its NUMA region as much as possible. Moreover, today's dominant system bus protocol, PCIe, has limited cache coherence capabilities, requiring special care when determining the placement of the instructions, stack, heap, and data for multi-ISA executables.

For instructions, because the instruction footprint for the NxP cores can be relatively small, the `.text` sections for the NxP cores can reside in the host memory, relying on the I-cache of the NxP core to minimize access latency.

For the stack, because the thread migration in Flick happens only at function call boundaries, we assume that each core is unlikely to access data that other cores store in the stack. Based on this assumption, we extend all migrating threads with their own stack space in the NxP local memory. Threads typically use their local stacks, avoiding costly memory accesses over the system bus in most cases. In the rare event that a callee function uses pointers to access data on the caller's stack frame, the unified address space ensures correct execution, even if the parent executed on a different core.

For the heap, the system has separate memory allocators for each core's local memory. While linking the program, the linker relocates the memory allocation calls in each core's `.text` section to the corresponding memory allocator, avoiding application-level code changes. If software developers want to allocate memory in a particular memory region, the allocation can be annotated in the source code to indicate

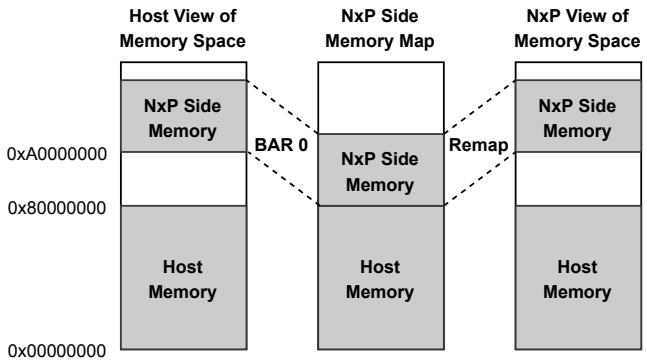


Fig. 3. The host system and NxP platform share the same physical address space through remapping of the BAR region on the NxP side.

which allocator to use. The latter can be useful, for example, for the near-storage computation case, where the host cores allocate and initialize the data for the near-storage-processors to operate on.

Finally, in Flick, the `.data` and `.bss` sections are still placed in the host memory, necessitated by the limited PCIe cache coherency capabilities. NxP cores cannot have coherent D-caches whose contents are automatically invalidated or written back on host access, forcing global variable accesses to cross the system bus. As a workaround, if coherence with the host is not required for a frequently-accessed variable or data structure, software developers can use directives in the source code to indicate the variables that should be allocated in the NxP local memory. The compiler can then place these variables into NxP-specific `.data` and `.bss` sections, enabling the loader to map and copy these sections into the NxP local memory. Notably, unlike PCIe, emerging system bus protocols such as CCIX [20], OpenCAPI [14], and GenZ [15] can avoid such workarounds.

IV. Implementation Details

To prototype Flick, we built a heterogeneous-ISA multi-core system comprising x86-64 host cores and a 64-bit RISC-V NxP core [21]. The choice of the x86-64 ISA was dictated by its dominance in modern servers. The RISC-V ISA, while being less ubiquitous, is an open and free ISA and has many available implementations, which allowed us to tailor some of the NxP core components, such as the TLB and MMU, for our specific needs. We adapted the Linux kernel and standard GNU tools to enable compiling, linking, and loading multi-ISA application binaries.

A. Unified Memory Space

To share the same view of the unified memory space between the host system and the NxP cores, we customized the RISC-V core MMU and TLB by replacing the built-in TLB and MMU with their x86-like counterparts. These counterparts understand and use the host x86-64 page table structure including huge page support (i.e., 2MB and 1GB pages). The L1 I-TLB and D-TLB each have 16 entries with one cycle latency to support fast accesses. This is especially important for data accesses due to the PCIe restrictions. The data cache can only be enabled for the NxP memory regions that do not require coherence with the host cores. Notably, our TLB and MMU cause the NxP cores to always use virtual addresses. To bootstrap the system in this environment, we initially set one I-TLB entry that maps the NxP core reset address to a small ROM containing the NxP bootstrap instructions. The bootstrap instructions set up the NxP stack pointer and jump to the NxP scheduler, which starts triggering TLB misses for normal execution in the virtual address space shared with the host.

On a TLB cache miss, the TLB blocks the memory access until the MMU traverses the page table structure in the host memory to update the virtual-to-physical translation in the TLB. Instead of implementing a fixed-function MMU that walks the x86 page tables, we implemented the MMU as a tiny micro-controller core. This helps to ease development efforts and will enable future research on the optimization of the NxP address translation.

A programmable MMU opens up an opportunity for run-time optimization. The MMU can be configured to open holes in the NxP virtual address space, bypassing the page table traversal. These holes allow the NxP core to directly access NxP components for debugging purposes or to access a large region of local physical memory without traversing page tables in the host memory, allowing implementation of high-performance private scratchpad memories.

Other than performing the virtual-to-physical translation, the TLBs also ensure that the NxP has the same view of the physical memory space as the host system. The host memory directly maps to the same physical addresses on the NxP platform starting at address 0x0. However, the NxP local memory and peripherals are exposed to the host using PCIe BARs, whose addresses are assigned dynamically by the host system. To match the view of the physical address space on the NxP core, we add remapping functionality to the TLB. This remapping shifts the addresses of the local memory and peripherals to the correct NxP-side local address instead of the host-side address. An example of the remapping is shown in Figure 3. The NxP's local memory is designed to start at address 0x80000000 and is exposed to the host as BAR0. However, the host system maps BAR0 at address 0xA0000000. Because the BAR addresses are dynamically assigned by the host, a driver running on the host calculates the mapping offset 0x40000000 and writes it into a control register in the NxP TLB. During execution, when the TLB translates a virtual address into a physical address that falls into the BAR address range starting at 0xA0000000, the TLB uses the control register content to adjust the physical address, forming the actual address (starting with 0x80000000) used by the NxP local address mapping for the NxP local memory and peripherals. Adding this address offset into the TLB enables the NxP to observe the same physical memory space as the host cores for both host memory and the NxP components.

Listing 1. Host migration handler pseudocode.

```
1  host_migration_handler(arg1, ...)
2  {
3      if (first_time_migration)
4          allocate_nxp_stack();
5      prepare_host_to_nxp_call(arg1, ...);
6      ioctl_migrate_and_suspend();
7      while( nxp_to_host_call ) {
8          args = fetch_nxp_call_desc();
9          host_rtn = call_target_host_func(args);
10         prepare_host_to_nxp_return(host_rtn);
11         ioctl_migrate_and_suspend();
12     }
13     nxp_rtn = fetch_nxp_return_desc();
14     return nxp_rtn;
15 }
```

Listing 2. NxP migration handler pseudocode.

```
1  nxp_migration_handler(arg1, ...)
2  {
3      prepare_nxp_to_host_call(arg1, ...);
4      migrate_and_suspend();
5      while( host_to_nxp_call ) {
6          args = fetch_host_call_desc();
7          nxp_rtn = call_target_nxp_func(args);
8          prepare_host_to_nxp_return(nxp_rtn);
9          migrate_and_context_switch();
10     }
11     host_rtn = fetch_host_return_desc();
12     return host_rtn;
13 }
```

B. Thread Migration

This section details the Flick thread migration mechanism. We describe the software and hardware components that ensure correct execution and low overhead when performing thread migration across cores with different ISAs. We split the description into two scenarios of thread migration: migration that happens when the host calls an NxP function, and migration that happens when the NxP calls a host function.

1) Host Calls NxP Function: With Flick, a thread should always start its execution on the host. The migration to the NxP core happens when there is a need to call an NxP function. As a result, the first migration of a thread is always a host-to-NxP call migration.

When a thread running on the host fetches NxP core instructions from memory, the `NX` bit in the page table entry triggers an `INST` page fault, transferring control to the page fault handler. In the page fault handler, the faulting address is the address of the function that should run on the NxP. This address is automatically stored by the host core on the stack as the return address of the page fault handler. We save the faulting address in the `task_struct` of the thread and replace this address on the stack with the address of Flick's host migration handler. When the page fault handler returns, instead of fetching the NxP instructions again, the thread resumes execution in the host migration handler and starts the migration process from the host to the NxP.

The host migration handler, shown in Listing 1, first examines whether the NxP stack pointer has been initialized—i.e., it is not `NULL`. An uninitialized stack pointer indicates that migration is occurring for the first time for the current thread. This causes the migration handler to allocate and set up a block of NxP local memory as the thread's NxP stack (lines 3–4 in Listing 1). Because we hijack a function call and redirect execution to the migration handler, the handler receives the original function's arguments as the handler's own arguments. The host migration handler collects its arguments and passes them to the kernel using an `ioctl()` system call (line 6 in Listing 1). Internally, the `ioctl()` collects auxiliary information, such as the target address, stack pointer, CR3 value (the x86 PTBR), and PID (to identify which process to wake up after execution eventually returns to the host). These values, taken from the thread's `task_struct`, are packaged together with the function arguments into a host-to-NxP call descriptor. Finally, the `ioctl()` system call concludes by suspending the thread on the host and triggering the transfer of the descriptor to the NxP. The thread will next resume execution on the NxP.

To minimize the overhead of transferring the descriptor using multiple memory operations across PCIe, Flick uses a DMA controller to copy the entire descriptor using one PCIe burst transfer. The NxP scheduler polls a DMA status register to discover when at least one migration descriptor has been transferred from the host. When the NxP scheduler observes a change in the DMA status register, it reads the host-to-NxP migration descriptor and extracts the NxP stack pointer, using the stack pointer to context switch to the target thread. As the NxP stack is carefully initialized by the host beforehand, the target thread starts execution inside the `while()` loop of the NxP migration handler (line 6 in Listing 2). Inside the `while()` loop, the NxP calls the target function using the target address and arguments from the host-to-NxP call descriptor. After the target function call returns, the return value and PID are stored in an NxP-to-host return descriptor, and the thread performs a context switch back to the NxP scheduler (line 9 in Listing 2). The scheduler then transfers the NxP-to-host return descriptor to the host memory using the DMA controller.

The DMA controller triggers an interrupt on the host system to wake up the thread using its PID. The thread wakes up inside the `ioctl()` system call of the migration handler and returns to the host migration handler with the received NxP-to-host descriptor (line 6 in Listing 1). The migration handler checks if this is an NxP-to-host call migration (explained further) or a host-to-NxP return migration. For a return from the NxP core, the migration handler returns the return value stored in the return descriptor (lines 13–14 in Listing 1). Because we hijack the original function call, this return will be just like a normal return from the target function as though execution never left the host core. The migration process and execution on the NxP core is therefore entirely transparent.

Any future attempts to call functions compiled for the NxP ISA will similarly cause the `INST` page fault and trigger a migration, reusing the NxP stack allocated previously. The NxP system will perform a context switch and the thread will resume inside the `while()` loop of the NxP migration handler where it previously stopped (line 9 in Listing 2), re-

peating the process of receiving descriptors, calling functions, and returning back to the host.

2) NxP Calls Host Functions: Just as the host can call functions that should run on the NxP, the NxP can call functions that should run on the host, requiring migration of the thread back to the host system. Recall that our host system relies on the `INST` page fault. Notably, migration from the RISC-V NxP core to the host can be triggered by an additional mechanism. First, similar to the host, the `INST` page fault can be triggered by the MMU when the NxP attempts to fetch instructions of a function compiled for the host ISA. In Flick, the meaning of the `NX` bit in the page tables is inverted for the NxP, requiring the TLB to trigger this exception on pages where the `NX` bit is *not* set). Second, as x86 has variable-length instructions, the instruction fetch of a host function can trigger a RISC-V misaligned instruction address exception. We treat both exception types as indicators of a need for migration, using the exception mechanism to transfer control to the RISC-V migration handler shown in Listing 2.

The NxP migration handler is similar to its host counterpart, except that the NxP migration handler does not perform stack initialization, because all threads originate on the host and have a stack there. The migration handler prepares an NxP-to-host call descriptor and uses the DMA controller to copy the descriptor to the host memory (lines 3–4 in Listing 2), triggering a host core interrupt whose handler wakes up the corresponding thread (based on its PID), which was previously suspended inside the `ioctl()` call.

Execution on the host resumes in the user space migration handler (line 6 in Listing 1), which checks whether this is a return from a call to an NxP function or if it is an NxP-to-host call. In the latter case, the host migration handler goes into a `while()` loop and retrieves the target function address and arguments from the descriptor. Then it calls the function on the host (lines 8–9 in Listing 1). When the host function returns, the migration handler prepares a host-to-NxP return descriptor with the return value to send back to the NxP. The migration handler then calls `ioctl()` to initiate the DMA transfer of the descriptor. Similar to the host-to-NxP migration, the thread on the host (lines 10–11 in Listing 1) is suspended.

To allow the NxP core to make any number of calls to the host functions, the host thread remains in the `while()` loop as long as it continues being woken up by the NxP-to-host call migrations. However, if the host thread is woken up with a request for an NxP-to-host return migration rather than NxP-to-host call migration, the host breaks out of the `while()` loop, allowing the thread to resume execution on the host at the original call site where the first migration to the NxP occurred.

On the NxP, when the NxP scheduler receives a migration descriptor, it performs a context switch to the corresponding thread, which was previously suspended inside the NxP migration handler (line 4 in Listing 2). The migration handler checks if this is a return migration from the host or a call migration from the host. For a return, the NxP migration handler initiates the return procedure with the value received from the descriptor (lines 11–12 in Listing 2). For a call, the NxP migration handler enters a `while()` loop to keep executing function calls from the host, and remain in the loop until it receives a host-to-NxP return descriptor to break out of the loop and return control to the original NxP caller function.

Nested Bidirectional Function Calls: Both the host and the NxP migration handlers are designed to be reentrant. Notably, we rely on all functions that can trigger a migration to follow the standard function call convention. This ensures that multiple copies of the migration handlers' local variables can reside in different locations of a single thread's stacks simultaneously and not interfere with each other. Consequently, the page-fault-triggered migration mechanisms we describe above allow the application code to freely call any function in the executable at any time, even recursively, as the system transparently migrates execution back and forth between cores with different ISAs as needed.

C. Multi-ISA Binary Generation and Execution

To generate and execute multi-ISA binaries on a conventional Linux system, we introduced several changes to the compiler, the linker, and the loader toolchain.

1) Compiler: To compile code for multiple ISAs, we rely on user annotations in the source code. We use scripts to partition the annotated source code, placing functions that target different ISAs into separate temporary source files, allowing us to separately invoke unmodified compilers for each ISA target. Because Flick migration happens automatically at runtime through a transparent page fault mechanism, there is no need to perform any source or binary instrumentation. Furthermore, as both the host and NxP cores share the same virtual address space, the compiler never needs to convert any addresses or make any special provisions for pointers. The only notable change we introduce is to the RISC-V toolchain to change the names of the resulting ELF sections. For example, the `.text` section containing compiled code is named `.text.riscv`. This approach provides a clean way to handle the different ISA targets without complex compiler modifications such as merging the compilers for multiple ISAs.

2) Linker: After generating all object files, we use the native host linker to merge these object files using a custom linker script. The linker script prevents the NxP sections from being automatically merged with the host sections and maintains the section names that indicate the target ISA. Our custom linker script also ensures 4KB alignment for all `.text` sections to ensure that pages holding code for each ISA have different page table entries.

After the object files are merged, the linker performs symbol relocation. We modified the linker sources to include the relocation functions for both the host (x86-64) and the NxP (RISC-V) cores. The linker invokes the corresponding relocation functions based on the section name, resolving symbols across sections and populating all internal references in the resulting multi-ISA executable. Because all cores share the same virtual address space, the linker links all section normally, without any special address handling. Notably, the

TABLE I
SYSTEM SPECIFICATION

Host System	Dual Xeon E5-2620v3, 64GB DDR4
FPGA Board	NetFPGA SUME (Xilinx xc7vx690tffg1761-3)
FPGA Memory	4GB DDR3
NxP Core	In-order Scalar RV64-I @ 200MHz
Interconnect	PCIe 3.0 x8
Operating System	Ubuntu 18.04.2, Linux 5.2.2
Toolchain	GCC 8.3.0, binutils 2.30, glibc 2.27

executable produced after linking has all of its internal references resolved, including instruction and data references that cross ISA boundaries; host code directly refers to the code and data in the NxP sections and NxP code directly refers to the code and data in the host sections of the binary.

3) Dynamic Linker & Loader: The Flick multi-ISA binaries contain one `.text` section for each ISA, with the load address of each section aligned to a page boundary. We added support for these multi-ISA binaries to the GLIBC user-space dynamic linker. When loading a `.text` section of the executable, the linker uses an extended `mprotect()` system call to configure the appropriate bits in the page table entries based on the section name. This enables triggering thread migration on their respective page faults. For example, for the dual-ISA executables with RISC-V as the NxP ISA and x86-64 as the host ISA, the loader sets the `NX` bit for the `.text.riscv` section and leaves it cleared for the x86 `.text` section. Once loading is complete, the program starts its execution as usual, jumping to the application entry point. The page fault handler and migration handler will transparently migrate the thread to an appropriate NxP or host core at any point when ISA boundaries are crossed throughout the program execution. Notably, for executables with more than two ISAs, the loader would have to use additional bits in the page table entries to distinguish between the different NxP ISAs.

D. Kernel Changes

The Flick thread migration mechanism requires changes to the Linux kernel. Flick relies on page faults to trigger the thread migration and exploits the `mprotect()` system call to mark pages as non-executable. We modified the default Linux page fault handler to recognize `NX` page faults and use them to trigger migration. When an `NX` page fault occurs, a handler manipulates the stack such that it returns to Flick migration handler as described in Section IV-B.

In addition to the `NX` page fault handler, Flick requires modification to the kernel ELF loader to support multi-ISA kernel modules. Similar to the multi-ISA executables, the multi-ISA kernel modules include functions that should run on the host (e.g., NxP platform initialization and host `ioctl()`) and functions that should run on the NxP (e.g., NxP scheduler and NxP migration handler). We implement a similar mechanism for multi-ISA modules in the kernel as for the user-space programs. The compilation flow links together a multi-ISA kernel module, and the kernel module loader is

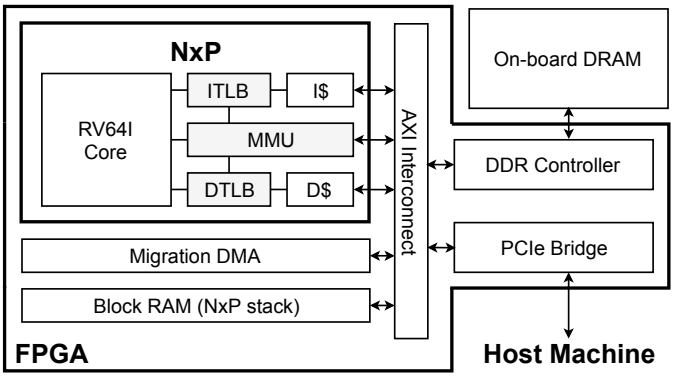


Fig. 4. FPGA-based evaluation platform for Flick.

modified to handle the relocation functions for both the host and the NxP cores according to the section name.

Finally, Flick requires a minor modification to the Linux scheduler. When a user-space migration handler issues a migration request via an `ioctl()`, the kernel creates a descriptor for executing the target function, suspends the thread on the host core by setting its state to `TASK_KILLABLE`, runs the scheduler to context switch away from the thread on the host, and triggers a DMA transfer of the descriptor to the NxP. Notably, the Linux scheduler needs to be modified to trigger the DMA controller *after* context switching away from the running thread because, by the time the transfer must be triggered, the thread execution has already been suspended. To facilitate this, we add a "migration" flag to the `task_struct`, which is set prior to suspending the thread, and used to trigger the descriptor transfer from the scheduler after the thread is suspended. Although this appears to be a rather intrusive mechanism, it is necessary to trigger the descriptor transfer after the thread has been suspended to avoid a race condition where the NxP can receive the descriptor, execute the target function, and return, all before the host completes suspending the thread on the host core.

V. EVALUATION

To serve as a testbed for evaluating Flick, we prototyped a heterogeneous-ISA system using an off-the-shelf Intel Xeon server as the host and a PCIe-connected FPGA emulating a device with an NxP. Details of our test environment are presented in Table I, and Fig. 4 depicts the FPGA platform we created. An in-order scalar RV64-I core [21] serves as an example of NxP. It connects to on-chip block RAM for its local stacks. On the FPGA board, we use a 4GB on-board DDR3 SDRAM DIMM as the NxP side data storage. The FPGA platform uses a PCIe bridge to connect to the host and allows the RISC-V core to access the host memory. In the other direction, the entire 4GB NxP side data storage is memory-mapped on the host via a PCIe BAR region through the PCIe bridge, enabling the host cores to directly access the NxP side data storage with load and store instructions. Together, this forms a shared memory heterogeneous-ISA multicore system. Measurements show that the round-trip time for the host x86

TABLE II
THREAD MIGRATION OVERHEAD FROM PRIOR WORK AND FLICK.

Work	Fast Cores	Slow Cores	Interconnect	Overhead
ASPLOS'12 [11]	MIPS @2GHz	ARM @833MHz	Not Considered	≈600µs
EuroSys'15 [13]	Xeon E5-2695 @2.4GHz	Xeon Phi 3120A @1.1GHz	PCIe	≈700µs
ISCA'16 [6]	Xeon E5-2640 @2.5GHz	ARM Cortex R7 @750MHz	PCIe Gen3 x4	≈430µs
ARM Big-LITTLE [2]	ARM Cortex A15 @1.8GHz	ARM Cortex A7	Onchip Network	22µs
Flick (this work)	Xeon E5-2620v3 @2.4GHz	RISC-V RV64I @200Mhz	PCIe Gen3 x8	18.3µs

TABLE III
FLICK THREAD MIGRATION ROUND TRIP OVERHEAD.

Host-NxP-Host	NxP-Host-NxP
18.3µs	16.9µs

cores and the NxP RISC-V core to access the NxP side storage are approximately 825ns and 267ns, respectively.

We use a Xilinx MicroBlaze soft core as a programmable MMU to handle TLB misses by walking the page tables in the host memory. The TLB miss penalty is high due to the cross-PCIe memory accesses, but 1GB huge pages used for the 4GB NxP side data storage allow only four TLB entries to cover the NxP local storage, avoiding most time-consuming TLB misses during normal operation.

For evaluation purposes, we first study Flick using microbenchmarks, providing a closely controlled environment to understand and precisely characterize the system and measure migration overhead. Then, we present an example breadth-first search application, demonstrating the behavior of the Flick technique on a practical problem.

A. Thread Migration Overhead

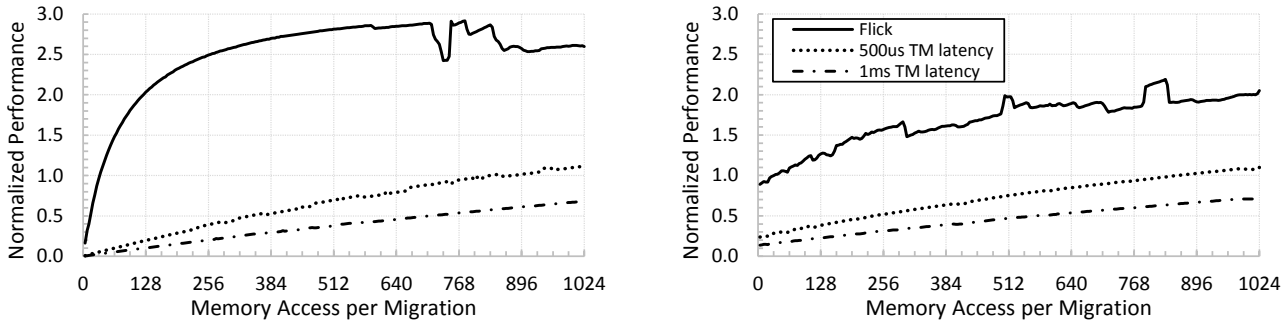
We first examine the thread migration overhead of Flick, which is critical to the overall system performance. We created a microbenchmark where the host calls a function on the NxP that immediately returns. The microbenchmark calls this function 10,000 times, and we measure the average round-trip overhead from host to the NxP. Similarly, to measure the overhead of the NxP calling a function on the host processor, we let the NxP function call a host function, which also immediately returns. We then subtract the host-to-NxP call overhead from the measurement to get the NxP-to-host overhead. The results are shown in Table III. We see that Flick requires only 18.3µs and 16.9µs to do a Host-NxP-Host and NxP-Host-NxP thread migration round trip, respectively. Further investigation indicates that the host side page fault only incurs 0.7µs of the total migration overhead, showing the efficiency of the page fault triggered thread migration. Table II compares this overhead to prior work, showing that Flick's migration overhead is already 23x to 38x less than prior work on heterogeneous-ISA thread migration [6], [11], [13]. In fact, Flick's migration over PCIe is even faster than homogeneous-ISA migration using the on-chip network within commercial SoCs [2]. Notably, our NxP core is a simple soft core running at only 200MHz. We anticipate that the overhead of Flick can be further reduced when using hardened cores.

B. Microbenchmark: Pointer Chasing

To study the performance of Flick in a controlled environment, we developed a pointer chasing microbenchmark, which traverses variable-length linked lists stored in the NxP side storage. We chose pointer chasing as a representative example of a data structure where it is beneficial to migrate the thread to be close to the data being accessed. Our goal is to understand how the overhead of thread migration can affect potential speedup. The microbenchmark uses a loop to call a function on the NxP to traverse the linked lists in the data storage, where the nodes' addresses are 8-byte aligned and randomly spread across the 4GB space. We swept the number of traversed nodes per function call from 4 to 1024, in increments of 4, and measured the average time to finish the traversal. Traversing lists of different lengths emulates different amounts of work performed per migration. We then normalized the performance to the baseline, where the host core directly traverses the linked lists over PCIe. Additionally, to better understand how thread migration overhead affects the overall performance, we also measured the effect of introducing extra migration latency to mimic the larger overheads incurred in the prior work.

Figure 5a shows the normalized performance when the host frequently migrates the thread to the NxP, as no delay is inserted between the function calls. At 32 memory accesses per migration, Flick (shown as a solid black line) is able to achieve the same performance as the baseline. As the number of memory accesses increases, amortizing the thread migration overhead, the Flick performance benefits increase and stabilize at 2.6x, which is the relative difference in latency of the host core and the NxP when accessing the NxP side storage. The two dashed lines below Flick show the performance of alternative systems with 500µs and 1ms migration latency. Higher migration latency prevents the system from achieving the baseline performance within a reasonably-small number of accesses per migration. This demonstrates the crucial importance of fast migration; we see that systems with higher migration overheads require many more accesses to be done per migration to show any benefit from NxP.

Figure 5b shows the result of infrequent migrations, where the system only migrates a thread every 100µs. This corresponds to a scenario where more work is done on the host and therefore a smaller portion of the overall execution time is spent on the pointer chasing. In this case, the migration overhead becomes less significant and causes less penalty before reaching the baseline performance. After 32 memory accesses per migration, although Flick still outperforms the baseline,



(a) No Migration Interval

(b) 100µs Migration Interval

Fig. 5. Pointer chasing microbenchmark with different thread migration latency and migration interval.

TABLE IV
BFS DATASETS AND EXECUTION TIME IN SECONDS.

Dataset	Vertices	Edges	Size	Baseline	Flick
Epinions1	76k	509k	16.7 MB	1.8s	2.4s
Pokec	1,633k	30,623k	1.0 GB	107.4s	90.3s
LiveJournal1	4,848k	68,994k	2.2 GB	240.5 s	220.9s

the benefit of thread migration is reduced to approximately 2x. Overall, the results highlight the importance of thread migration latency, showing that Flick's low overhead makes it effective even in case of frequent migration of small jobs.

C. Application: BFS

The BFS (breadth-first search) benchmark is a graph traversal benchmark from Graph500 [22]. BFS begins with a source vertex and recursively explores its neighbors until all reachable vertexes are discovered. Graph traversal is a central component of many analytics problems, such as recommendation systems, social media modeling, and route optimization.

We use three social network datasets from the Stanford Network Analysis Project (SNAP) [23], shown in Table IV, to evaluate Flick with different graph sizes. The graphs generated from the datasets are stored in the NxP side DRAM. Flick migrates the entire traversal function to the NxP. The host calls the traversal function 10 times in a loop and calculates the average execution time of one iteration. To emulate a common scenario where the host software must perform a task for each vertex, the traversal function running on the NxP calls a dummy host function for each newly discovered vertex. This causes execution to migrate to the host and return.

We compare the average execution time to a baseline where the host core directly traverses the graph via PCIe, shown in the two rightmost columns of Table IV. For the smallest dataset, because the vertex-to-edge ratio is higher than the two larger datasets, the migration overhead of Flick causes the performance to be lower than the baseline. However, for the two larger datasets, even though the thread must migrate for every new vertex, Flick still outperforms the baseline by 9% to 19%. Such a speedup would not be possible for an approach with higher migration overhead. The result further highlights the need for low overhead, showing how Flick enables applications with frequent thread migration.

VI. Related Work

Heterogeneous-ISA Thread Migration. Prior work on heterogeneous-ISA thread migration demonstrates the benefits of heterogeneous-ISA systems [11], [13]. However, the migration overhead of these techniques is on the order of several hundred microseconds, which limits their use to infrequent thread migrations. In this work, we showed that Flick's migration overhead is significantly lower, cutting overheads from hundreds of microseconds to 18 microseconds, facilitating frequent thread migration. Beyond the performance benefits, Flick also requires much fewer modifications to the operating system. For example, prior work reports 37K LoC changes to the Linux kernel and 5K LoC changes to the compiler [13], while Flick requires less than 2K LoC in total. Although not targeting heterogeneous-ISA multicores, one prior system used the UD fault (Invalid Opcode) to trigger unidirectional thread migration for overlapping-ISA heterogeneous multicore systems [24], whereas Flick uses the *Non-Executable (NX)* bit in the page table entries to support bidirectional thread migration between cores with different ISAs.

Heterogeneous Multi-core Operating Systems. Several works on heterogeneous multi-core operating systems also support cores with different ISAs in one system [9], [10]. However, these systems completely redesigned the OS, having limited impact on the acceptance of heterogeneous-ISA systems. On the other hand, Flick requires only minor changes to the existing operating systems and shows benefits on widely available commodity servers.

Unified Memory Space. One of the requirements of Flick, although not a focus of this work, is the unified memory space for the entire system. Many designs already target virtual memory support for GPUs, both from industry [25], [26] and academia [27]–[29], as well as virtual memory support for accelerators [30], [31]. Flick can benefit from the advances in accelerator TLB design and improved support for accelerator virtual memory. A distinction from prior work is that Flick unifies both the virtual and physical address spaces.

NxPs. Works on near-storage-processing [6] and near-memory-processing [32] demonstrated those NxPs either improving system performance or power efficiency. However, the conventional offload engine programming style to utilize

NxPs breaks the integrity of the software and requires manual orchestration of communication and data movement between cores. Flick utilizes virtual memory support and page fault triggered thread migration to enable transparent utilization of NxPs and still achieves their performance benefits.

VII. Conclusions

In this work, we proposed Flick, a Fast and Light-Weight ISA-Crossing Call mechanism, for transparent thread migration in heterogeneous-ISA multi-core systems. By restricting the migration points to function boundaries, unifying the physical address space, and leveraging hardware virtual memory, Flick eliminates the majority of the thread migration overhead while requiring only minor changes to existing operating systems. We evaluated Flick using a PCIe-connected FPGA with off-the-shelf hardware and software. Experiments with microbenchmarks and a BFS application showed that Flick's thread migration overhead is 23x to 38x less than prior work.

Acknowledgments

This research was supported by the National Science Foundation through grants CCF-#1452904 and CCF-#1725543, by the Semiconductor Research Corporation, and by a Google Faculty Research Award.

References

[1] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[2] P. Greenhalgh, "big.LITTLE processing with ARM Cortex-A15 & Cortex-A7," ARM, Tech. Rep., 2011.

[3] A. Venkat and D. M. Tullsen, "Harnessing ISA diversity: design of a heterogeneous-ISA chip multiprocessor," in *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[4] A. Venkat, H. Basavaraj, and D. M. Tullsen, "Composite-ISA cores: enabling multi-ISA heterogeneity using a single ISA," in *Proceedings of the 2019 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2019.

[5] Marvell. LiquidIO II Smart NICs. Accessed: April 16, 2020. [Online]. Available: https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/

[6] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: a framework for near-data processing of big data workloads," in *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[7] Xilinx. Xilinx Zynq UltraScale+ MPSoC. Accessed: April 16, 2020. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[8] Intel. Intel SoC FPGAs. Accessed: April 16, 2020. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/soc.html

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[10] F. X. Lin, Z. Wang, and L. Zhong, "K2: a mobile operating system for heterogeneous coherence domains," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[11] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-ISA chip multiprocessor," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[12] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-R. Chuang, V. Legout, and B. Ravindran, "Breaking the boundaries in heterogeneous-ISA datacenters," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[13] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: bridging the programmability gap in heterogeneous-ISA platforms," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, 2015.

[14] OpenCAPI Consortium. (2016) OpenCAPI overview. Accessed: April 16, 2020. [Online]. Available: https://opencapi.org/wp-content/uploads/2016/09/OpenCAPI-Overview.10.14.16.pdf

[15] The Gen-Z Consortium. Gen-Z technology. Accessed: April 16, 2020. [Online]. Available: https://genzconsortium.org/about-us/gen-z-technology/

[16] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[17] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan, "SpaceJMP: programming with multiple virtual address spaces," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[18] Apple Inc. (2006) Universal binary programming guidelines.

[19] R. Gordon. FatELF: universal binaries for Linux. Accessed: April 16, 2020. [Online]. Available: http://icculus.org/fatelf

[20] CCIX Consortium. Cache coherent interconnect for accelerators (CCIX). Accessed: April 16, 2020. [Online]. Available: http://www.ccixconsortium.com

[21] Roa Logic. RV12 RISC-V processor. Accessed: April 16, 2020. [Online]. Available: https://roalogic.com/portfolio/riscv-processor

[22] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray Users Group (CUG)*, May 2010.

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[24] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *Proceedings of the 16 International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[25] Intel. (2014) OpenCL 2.0 shared virtual memory overview. Accessed: April 16, 2020. [Online]. Available: https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-overview

[26] M. Harris (NVIDIA). (2013) Unified memory in CUDA 6. Accessed: April 16, 2020. [Online]. Available: https://devblogs.nvidia.com/unified-memory-in-cuda-6/

[27] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[28] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *Proceedings of the 2014 IEEE 20th International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.

[29] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: a case for software address translation on GPUs," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[30] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *Proceedings of the 2017 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2017.

[31] B. Pichai, L. Hsu, and A. Bhattacharjee, "Address translation for throughput-oriented accelerators," *IEEE Micro*, vol. 35, no. 3, 2015.

[32] M. Alian, S. W. Min, H. Asgharimoghaddam, A. Dhar, D. K. Wang, T. Roewer, A. McPadden, O. O'Halloran, D. Chen, J. Xiong, D. Kim, W.-m. Hwu, and N. S. Kim, "Application-transparent near-memory processing architecture with memory channel network," in *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.