Cross-ISA Execution of SIMD Regions for Improved Performance

Yihan Pang
Virginia Tech
Blacksburg, USA
pyihan1@vt.edu

Robert Lyerly
Virginia Tech
Blacksburg, USA
rlyerly@vt.edu

Binoy Ravindran
Virginia Tech
Blacksburg, USA
binoy@vt.edu

Abstract

We investigate the effectiveness of executing SIMD workloads on multiprocessors with heterogeneous Instruction Set Architecture (ISA) cores. Heterogeneous ISAs offer an intriguing clock speed/parallelism tradeoff for workloads with frequent usage of SIMD instructions. We consider dynamic migration of SIMD and non-SIMD workloads across ISA-different cores to exploit this trade-off. We present the necessary modifications for a general compiler/run-time infrastructure to transform the dynamic program state of SIMD regions at run-time from one ISA format to another for cross-ISA migration and execution. Additionally, we present a SIMD-aware scheduling policy that makes cross-ISA migration decisions that improves system throughput. We prototype a heterogeneous-ISA system using an Intel Xeon x86-64 server and a Cavium ThunderX ARMv8 server and evaluate the effectiveness of our infrastructure and scheduling policy. Our results reveal that cross-ISA execution migration within SIMD regions can yield throughput gains up to 36% compared to traditional homogeneous ISA systems.

CCS Concepts

• **Computer systems organization** → *Single instruction, multiple data*; *Heterogeneous (hybrid) systems*; • **Software and its engineering** → *Scheduling*;

Keywords

System Software; SIMD; Heterogeneous Architectures; ISA; Scheduling; System Throughput

SYSTOR '19, June 3–5, 2019, Haifa, Israel

ACM ISBN 978-1-4503-6749-3/19/06…$15.00
https://doi.org/10.1145/3319647.3325832

ACM Reference Format:
Yihan Pang, Robert Lyerly, and Binoy Ravindran. 2019. Cross-ISA Execution of SIMD Regions for Improved Performance. In *The 12th ACM International Systems and Storage Conference (SYSTOR '19), June 3–5, 2019, Haifa, Israel.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3319647.3325832

1 Introduction

In recent years, the computer architecture landscape has seen the rise of systems integrating heterogeneous architectures as a possible solution to deal with the "end of Moore's Law" [13, 23, 80, 81, 86, 88]. Chip designers have explored the pairing of CPU designs that target vastly different use cases. For example, ARM's big.LITTLE technology [62] and its successor, DynamIQ [58], couple cache-coherent "big" cores (high clock speeds, advanced micro-architecture) for latency-sensitive workloads with "little" cores (high energy efficiency) for background and low-priority tasks. In the server space, Intel Xeon-Xeon Phi systems [24] integrate a small number of high-performance cores with many low-power cores to accelerate different workloads. Intel has also released plans for a heterogeneous x86 architecture with one big Sunny Cove core and multiple small Atom cores that uses a new 3D stacking technology [35, 70, 79, 91].

However, these designs do not have heterogeneity at the ISA level. At best, some cores used in these designs support extended instruction sets, but at their base, all cores share the same ISA (e.g., Xeon-Xeon Phi, ARM big.LITTLE). To date, there has not been any commodity-scale systems with heterogeneity at the ISA level (a notable exception is MPSoCs [37]). However, the research community has been exploring experimental heterogeneous-ISA designs, showing that they provide better performance and energy efficiency than single-ISA heterogeneity. Exploration in this design space includes many forms – shared-memory chip multiprocessors [10, 85, 86], multiprocessors with multiple cache-coherent domains (and no coherence between domains) [53], and composite-ISA cores [84] – across many settings, ranging from cluster architectures [68] to mobile settings [50].

Industry trends are also changing. With the advent of ARM-based high-end servers [20, 82, 83] capable of powering high-performance computing (HPC) applications, third-party organizations such as data-center providers and cloud

providers are increasingly integrating machines of different ISA families in their computing installations [3]. Chip vendors are also integrating CPUs of different ISA families in the same SoC, e.g., the Intel Skylake processor with in-package FPGA [25, 40] is capable of synthesizing RISC-V and x86 soft cores, or on the same platform, e.g., smart NICs integrate ARM [30, 65] or MIPS64 [60].

To understand the performance difference between heterogeneous-ISA servers, we ran applications from four popular HPC benchmark suites and calculated the slowdown of each benchmark when executed on an ARM core compared to an x86 core. We used both servers' factory-specified settings. Figure 1 shows the relative performance of single-threaded applications using one core of the Cavium ThunderX machine (96 cores, ARMv8 ISA) [82] in comparison to one core of the Intel Xeon machine (12-core/24-thread, x86-64 ISA) [40]. The benchmarks include the NAS Parallel Benchmark (NPB) suite [9], Phoenix Benchmark suite [77], Livermore Loops suite [61], and Test Suite for Vectorizing Compilers (TSVC) [15].

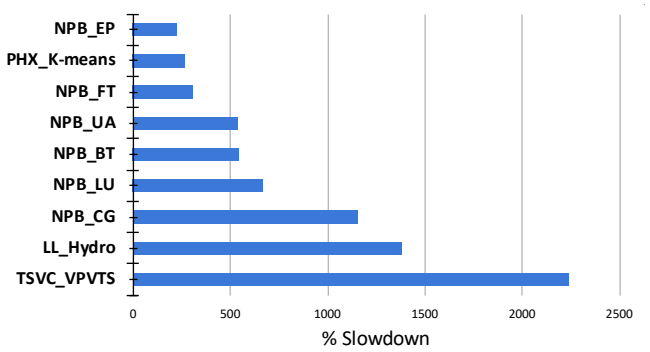


Figure 1: % Slowdown of NAS Parallel Benchmarks [9], Livermoore Loops [61], Phoenix [77], and Test Suite for Vectorizing Compilers [15] when run on a Cavium ThunderX core vs. an Intel Xeon Gold 5118 core.

Figure 1 reveals that all benchmarks are slower on ARM but differ in the degree of slowdown. It is worth noting that the ARM core slowdown is not surprising given that each individual Xeon core is clocked faster and ThunderX cores have different micro-architectural design goals, trading off single-core performance for massive parallelism. As a result, there are significantly more ARM cores in the ThunderX CPU, and the unit price of x86 cores is usually higher than its ARM counterparts ($1,300 for 12 x86 cores [39] vs $800 for 96 ARM cores [16]). This means that, for applications that experience a relatively low degree of slowdown on ARM cores, it is not immediately clear which CPU provides the best system throughput within the same price budget.

To illustrate that no ISA is the clear winner in throughput when running diverse workloads, we performed another study. We selected five benchmarks, EP, FT, BT, LU, and Hydro, based on the results from Figure 1, and ran them with a variety of different composition ratios on the same set of servers we used for the slowdown experiment. We measured the system throughput, i.e., *the number of benchmarks completed in a 75 minute period.* For each experiment, both servers executed as many benchmarks as possible without exceeding the number of available physical threads.

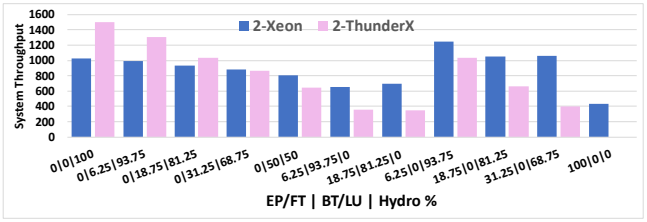


Figure 2: System throughput under different ratios of EP/FT [9], BT/LU [9], and Hydro [61] on two Xeon servers and two ThunderX servers.

Figure 2 shows the result of our study. Neither x86 nor ARM has a clear advantage in all tested scenarios. x86 CPUs have a higher throughput when there is an increasing number of relatively high-slowdown benchmarks (BT/LU/Hydro), whereas ARM CPUs perform better when the workload is mainly composed of low-slowdown benchmarks (EP/FT). Further inspection of the highest slowdown benchmarks in Figure 1 reveals that benchmarks like Hydro and VPVTS contain large portions of single-instruction-multiple-data (SIMD) instructions, accounting for more than 80% of benchmark execution time.

With SIMD instructions gaining more attention from chip designers as a means to extract additional data parallelism in various application domains (e.g., HPC [8], ML [75, 89], computer vision [22, 69], cryptography [4, 44]), coupling together heterogeneous machines that have significant differences in micro-architecture and SIMD extensions provides an interesting platform for running diverse applications. While the Xeon processor executes individual applications faster than the ThunderX (especially for SIMD workloads), the ThunderX integrates a much larger number of cores and can therefore execute more applications in parallel.

For designers aiming to optimize the performance of a heterogeneous-ISA system, this raises interesting questions:

(1) *How should a workload consisting of a mixture of SIMD and non-SIMD applications be scheduled to maximize throughput?*
(2) *How does an application consisting of SIMD regions impact the execution of co-executing workloads?*
(3) *Are there throughput advantages in migrating applications executing SIMD code across ISAs with different SIMD widths?*

In this work we investigate scheduling batch workloads consisting of SIMD and non-SIMD applications on heterogeneous-ISA systems. We analyze the effects of how SIMD and non-SIMD workloads interact on both x86-64 and ARMv8 systems, which have SIMD widths of 512 and 128 bits, respectively. Additionally, we analyze the impact of micro-architecture on system utilization, including where applications should be scheduled to maximize throughput. In order to conduct this investigation, we extend Popcorn Linux [10] – an OS/compiler/run-time system infrastructure for executing and migrating shared-memory applications across non-cache-coherent heterogeneous-ISA CPUs.

This paper makes the following contributions:

- We develop a cross-ISA SIMD migration compiler/run-time framework that enables applications containing SIMD instructions to be migrated between heterogeneous-ISA CPUs with different SIMD widths. The framework is built as an extension of Popcorn Linux's [10] compiler/run-time system infrastructure.
- We analyze the effects of co-executing SIMD and non-SIMD workloads on a heterogeneous-ISA system to understand the impact of micro-architectural heterogeneity and SIMD/non-SIMD workload composition on system throughput.
- Using insights gained from our analysis, we develop a SIMD-aware scheduler that monitors system workload and migrates applications executing SIMD regions between heterogeneous-ISA CPUs to improve system throughput. Our evaluations reveal up to 36% throughput gains over homogeneous-ISA CPUs.

2 Background

2.1 SIMD Instructions

SIMD instructions are instructions where a single operation is performed on multiple data elements. They help perform multiple calculations in parallel as well as amortize CPU core front-end costs (instruction cache pressure, decoding/issue latency) and are thus an attractive addition to ISAs. For example, Intel continues to widen its AVX vector extensions to 512 bits [38] and add new capabilities like neural network instructions [14]. Similarly, ARM has introduced its Scalable Vector Extension [5] with width-agnostic instructions to complement its existing NEON SIMD extension [6].

Extending SIMD width does not come for free. Since the CPU executes multiple instructions at once, significantly more power is consumed and thus the CPU must reduce its clock speed to avoid overheating. Intel's CPUs are a perfect example of this trade-off. Intel CPUs use dynamic voltage frequency scaling (DVFS) to adjust CPU frequency during runtime. When encountering SIMD-intensive code, the CPU dramatically scales down frequency (known as the AVX frequency [38]) to limit power consumption and reduce heat.

2.2 Cross-ISA Execution Migration Infrastructure

Currently, there does not exist a commodity heterogeneous-ISA chip multiprocessor with cache-coherent shared memory. To approximate such a machine, we connected an Intel Xeon server (x86-64) to a Cavium ThunderX server (ARMv8) via Infiniband. Table 1 shows the server details. To support cross-ISA execution, the system software must provide two capabilities: i) the ability to migrate threads and ii) the ability to migrate an application's data between the servers. We build on Popcorn Linux [10], an operating system, compiler, and runtime that provides system software support for cross-ISA execution migration.

Table 1: Server configurations.

Name	Intel Xeon	Cavium ThunderX
Generation	Gold 5118	1
ISA	x86-64	ARMv8
Micro-architecture	OoO	IO
Number of Cores	12	96
Number of Threads	24	96
RAM Size	48 GB	128 GB
SIMD Register Width	512 bit	128 bit

Popcorn Linux uses a replicated-kernel OS design, where kernels on separate machines communicate via message passing to provide user applications the illusion of a single machine. The OS provides a thread migration service, whereby threads call into the kernel and transparently resume execution on the destination architecture. Underneath, the originating kernel (origin) transfers the thread's context to the destination kernel (remote) to be re-instantiated and returned to user-space. Additionally, the OS provides a page migration service that transfers an application's data between machines on demand. When initially migrated to a new machine, the application has no pages mapped into memory – all memory accesses result in page faults. The OS's page fault handler is modified to intercept the faulting accesses, allowing the kernels to observe data accessed by threads and coordinate to migrate page data between machines. Pages are unmapped from the origin and mapped into the thread's address space on the remote.

While these capabilities are sufficient for migrating threads and application data pages between machines, they are not sufficient for cross-ISA execution. Popcorn Linux's compiler generates multi-ISA binaries that are suitable for cross-ISA execution. In multi-ISA binaries, the application's virtual address space is aligned so that references to symbols (global data, function addresses) are identical across

all architectures. For execution state that is tailored to each ISA's capabilities (stack, register set), the compiler generates metadata describing the function activation layout.

When migrating between architectures, Popcorn Linux's run-time parses the metadata for all function activations currently on the stack and transforms them between ISA-specific formats. After this state transformation, the runtime hands the new stack and register set to the OS's thread migration service to be restarted on the destination architecture. By aligning as much of the virtual address space as possible and only transforming the pieces that are tailored to each ISA, most of the application's state is valid across architectures, incurring minimal thread and data migration time.

In order to generate multi-ISA binaries, Popcorn Linux's compiler builds on LLVM's modularity to generate object files for all target architectures. The compiler's front- and middle-ends generate optimized LLVM bitcode from the application source. Threads can only migrate between architectures at equivalence points [87], i.e., points where execution has reached a semantically equivalent location and there exists a valid mapping between each ISA's execution state. A pass inserts call-outs to a migration library at equivalence points to enable thread migration between architectures. Because threads can only migrate at the inserted call-outs, there is a tradeoff between how many migration points are inserted into the code (and the associated call overheads) versus how long it takes a thread to respond to a migration request. Finally, a pass tags each call site with metadata describing all the live values at that location, as the run-time must be able to recreate the sequence of all function activations in the destination ISA's format.

The LLVM bitcode instrumented with migration points and live value metadata is passed to all target ISA backends for code generation. As the bitcode is lowered to machine code for each ISA, the backend records the locations of the live values specified by the middle-end, i.e., stored in a register or a stack slot. Note that the same set of live values is passed to each backend, and thus the same values are alive at each call site, meaning the runtime only needs to determine where to copy each live value for each ISA. In addition to live value locations, the compiler records per-function information such as callee-saved register locations and frame sizes. Each call site is also tagged with a unique ID to correlate call sites across architectures at run-time. The linker takes the object files for each architecture as input and emits a multi-ISA binary with an aligned virtual address space and the transformation metadata.

At run-time, threads execute like normal. When threads reach a migration point, they call out to the migration library to check if migration was requested, and if so, migrate to the requested destination. Threads check for migration requests via syscall – the kernel maintains a per-thread flag that can be set within the application or by external processes. If a migration is requested, the thread takes a snapshot of its current register set and begins transforming its stack and register set. First, the thread unwinds its stack to determine which activations are currently alive and to load each function's metadata. Next, the thread goes frame-by-frame from the most recently called function inwards, copying live values to the correct destination-ISA location. After transformation, the runtime passes the transformed register set for the outermost function to the OS's thread migration service. The kernel transfers the register set to the destination, which instantiates a new thread with the transformed register set and returns the thread to userspace. The thread exits the runtime on the destination and resumes normal execution as if it were still executing on the same machine.

3 Cross-ISA SIMD Migration Framework

3.1 Definitions

We define a *SIMD region* as a piece of code within a program that includes usage of SIMD instructions, such as a vectorized matrix computation. A SIMD region's size can vary in execution time and a program can contain any number of SIMD regions. Nested SIMD regions are considered a single SIMD region. A *SIMD workload* is defined as a set of applications in which every application has at least one SIMD region. In contrast, a non-SIMD workload is a set of applications that have no SIMD regions. Our work enables migration capability within SIMD regions; the framework can migrate any program in a SIMD workload across ISA-different CPUs for increased system throughput. We define a program to be *SIMD-intensive* if 50% of program execution time is in SIMD regions.

3.2 Basic SIMD Region Migration

The main obstacle for enabling SIMD region migration is identifying a suitable location inside each SIMD region for migration. The compiler's intermediate representation (IR) provides a means to achieve this. Upon inspection of multiple SIMD IRs generated by the LLVM compiler [49], we observe that SIMD computation at the LLVM IR level follows a very predictable code flow, as shown in Figure 3. In this figure, "Baseline" and "PGO-based" (explained in Section 3.4) are two different approaches for inserting migration points.

The majority of SIMD computations occur within a single basic block. In cases where a SIMD region spans multiple basic blocks, at least one of those basic blocks contain this code flow. In Popcorn Linux [10], equivalence points [87] are identified as "migration points" – i.e., program points where execution can be migrated across ISAs. Function boundaries

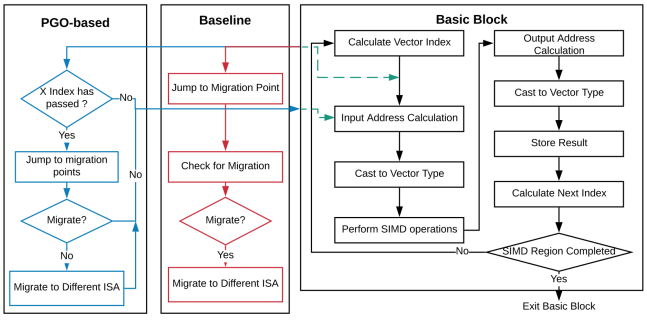


Figure 3: Cross-ISA SIMD migration approach. Green dotted lines indicate modifications applied for both approaches.

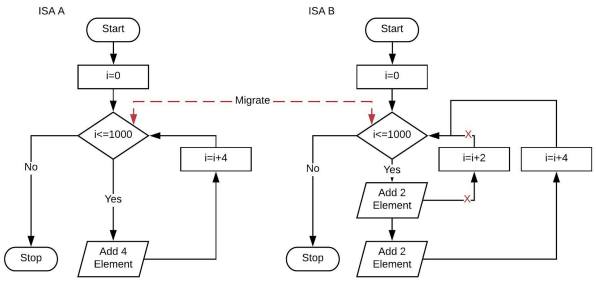


Figure 4: Vector unroll example.

are naturally occurring equivalence points. Popcorn Linux's compiler automatically inserts migration points at function entries and exits. We extend this approach. We insert new equivalence points by providing new function boundaries inside a SIMD region via "dummy" function calls after the vector index is calculated.

3.3 Vector Unrolling

After inserting migration points, we must ensure that the program executes correctly after (potential) migration. Because each ISA implements SIMD operations with varying widths, the number of loop iterations needed for each SIMD region varies. Our framework needs to account for combining different numbers of loop iterations for x86 and ARM due to their different SIMD widths.

Consider the example in Figure 4, in which one SIMD region is vectorized for ISA A and ISA B, whose SIMD widths vary by a factor of two. Suppose there are 1000 element need to be computed, thus, ISA A will take half as many iterations (250) as ISA B (500) to complete the task. This discrepancy in the number of elements calculated at each iteration is inefficient when migrating between different ISAs inside SIMD regions near the end of iterations. This because a single element calculation (inefficient usage of available SIMD instruction) needs to be performed if the remaining element is not large enough for a single SIMD loop iteration (calculating element 998-1000 can use SIMD instruction on ISA B but not on ISA A). There can also exist corner cases such as, the framework can try to migrate from ISA B to ISA A at the start of the last iteration on ISA B and exit on ISA A. In this case, the result will perform two unnecessary calculations at the end

One way to prevent this is by unrolling the loop as many times as the least common multiple (LCM) of both ISAs' SIMD width so that the same number of calculations are done in a single loop iteration. In LLVM, the loop vectorizer uses a cost model to decide on unroll factor and users can force the vectorizer to use specific values [55]. The LCM approach is compatible with any compiler optimization. The LCM is generated based on the final number of elements that are processed in each iteration after compiler optimization (a SIMD width of 2 unrolled three times is equivalent to a SIMD width of 6). Hence, for Figure 4's example, the problem is solved by unrolling twice on ISA B. Therefore, each time ISA B performs operations on four elements, it is identical to the number of elements performed in a single loop iteration for ISA A. This technique also helps to reduce the instrumentation overhead (discussed in detail in Section 3.4) by increasing the migration check interval.

3.4 SIMD Region Optimization

SIMD operations are meant to speed up computation, and in most cases, each SIMD loop iteration executes relatively quickly. However, if the compiler blindly inserts migration points at the beginning of each SIMD loop iteration as discussed in Section 3.2, the program will suffer significant execution overhead. This overhead is mainly due to the additional system calls to check for a migration decision (i.e., whether or not to migrate). For SIMD-intensive programs with a large number of loop iterations that never actually migrate, naïvely executing system calls to check at every loop iteration can harm performance. Nonetheless, applications should be able to quickly respond to migration requests to efficiently leverage heterogeneous-ISA systems.

In order for the program to quickly respond to migration requests as well as incur low instrumentation overhead when migrated , we propose a two step profile-guided optimization (PGO) approach, similar to LLVM's built-in PGO [12, 56, 57], to guide the insertion of migration points. In step one, we compile the program with instrumentation around SIMD regions to calculate each SIMD region's execution time. The results are logged in a file. Based on the results, in step two, we perform two actions. First, we eliminate migration points inside smaller SIMD regions. Second, we adjust the granularity at which the application checks for migration decisions by only executing migration checks at certain iteration intervals (this iteration number can be varied based on the

user's desire for responsiveness and loop unrolling factors). The program is then recompiled such that migration points only trigger when certain iterations are reached. Figure 3's blue lines illustrate the PGO approach's second step being applied. Our evaluation reveals that the instrumented SIMD benchmarks when using PGO suffer only 5% overhead on average.

3.5 SIMD-aware Scheduling

Compiler-level infrastructural support for cross-ISA execution migration within SIMD regions allows us to explore the possibilities of leveraging ISA affinity [85] to increase system throughput. This, however, requires a SIMD-aware scheduler – i.e., one that can decide when to migrate an application with SIMD regions from one ISA to another to increase the overall throughput.

We propose a scheduling policy to achieve this goal. Our policy assumes that the final slowdown (taking into considerations all factors, such as clock speed and micro-architectural differences), for each application on each different platform is known through profiling in our two-step PGO approach.[1] Our policy is centered around the idea that *the speedup gained from executing an application on the optimal ISA core should outweigh the slowdown other applications suffer from not running on that core.* In other words,

$$\frac{speedup\_of\_app\_X}{slowdown\_of\_app\_Y} > 1$$

For example, consider two applications A and B, where A runs $10x$ slower on an ARM core than on an x86 core, and B runs $5x$ slower on an ARM core than on an x86 core. Since the speedup gained from running A on the x86 core ($10x$) is greater than the slowdown B experiences from running on the ARM core ($5x$), it is likely effective to schedule A on the x86 core and B on the ARM core (assuming there is only one ARM core and one x86 core available) for improving the system throughput. To reduce the complexity of comparing every application based on their ISA slowdown, it is reasonable to categorize applications into three slowdown groups based on their individual ISA slowdown and the hardware thread count (ht) difference between the two ISA-different servers, as follows:

$$App = \begin{cases} G\_High\_Slowdown, & \text{if } Slowdown \geq \Delta 2ht \\ G\_Medium\_Slowdown, & \text{if } \Delta ht < Slowdown < \Delta 2ht \\ G\_Low\_Slowdown, & \text{if } Slowdown \leq \Delta ht \end{cases}$$

The reason behind using hardware thread count difference as a classification metric is that hardware thread difference can compensate the total throughput by executing more in parallel. This model can be extended for future multi-ISA systems (i.e., more than two ISAs) as long as the user identifies a "baseline" platform to compare. In the above equation, Δht represents the hardware thread count difference between the two ISA-different servers. Low slowdown applications have low affinity to either CPU and are most likely to benefit from the extra cores. Medium slowdown applications have higher affinity towards the faster CPU, but the slowdown is likely equal to or close to the hardware thread count difference between the two servers. Therefore, they are likely to experience a smaller degree of throughput degradation. Lastly, high slowdown benchmark applications have extremely high affinity towards the faster CPU and thus the speedup gain of running on the faster cores easily outweighs the maximum hardware thread count difference.

Our scheduling policy prioritizes executing high slowdown group members on the faster server (x86) as often as possible, followed by medium slowdown group members, and finally the low slowdown group. Our scheduler is implemented using an event-driven client-server model. Each application communicates with the scheduler upon three events: (1) arrival into the system queue, (2) upon application completion, and (3) after a migration is completed. Migration points, which are essentially callbacks for the scheduler, are inserted into the application.

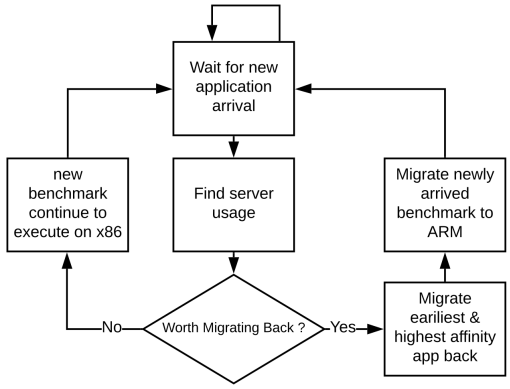


Figure 5: Scheduler's decision making process.

The application communicates its slowdown information to the scheduler upon arrival, which then determines its slowdown group. The scheduler tracks applications in both servers and makes migration decisions whenever the migration callback is triggered. The scheduler runs on one of the server cores (ThunderX in our experimental setup due to having more cores). The scheduler always tries to schedule applications with a higher x86 affinity on the x86 server without overloading it, and migrates applications to the ARM server after all the x86 cores have been utilized. The extra communication between the scheduler and the processes incurs a 5% overhead on average. Figure 5 illustrates the scheduler's decision tree.

[1] We restrict the scope of our work to platforms with two ISAs.

4 Experimental Setup

Table 1 describes the key characteristics of the heterogeneous-ISA servers that we used in our experiments. We considered four server configurations for comparison: (1) a homogeneous system composed of two Xeon servers, called "x86-static"; (2) a homogeneous system composed of two ThunderX servers, called "ARM-static"; (3) a heterogeneous system composed of one ThunderX server and one Xeon server, called "het-static", wherein applications are statically pinned to the next available core with the highest ISA affinity and run to completion on that core (i.e., no migration); and (4) a heterogeneous system composed of one ThunderX server and one Xeon server with cross-ISA SIMD migration enabled using the aforementioned techniques, called "het-dynamic". x86-static, ARM-static, and het-static serve as a baseline to evaluate the effectiveness of het-dynamic and do not migrate applications between servers. For het-dynamic, the two servers are interconnected via RDMA over Infiniband (56Gbps). We use factory-specified settings because we are trying to evaluate the maximum performance of each machine despite their micro/macro-architectural differences and are not interested in trying to isolate difference due solely to SIMD.

We used the same benchmarks as in Figure 1, compiled in two different ways for our experimental study. For het-dynamic, the benchmarks are compiled with the migration instrumentation described in Section 3. For the baseline configurations, the benchmarks are compiled without any instrumentation to avoid unnecessary overhead and to ensure a fair comparison (applications do not migrate in these cases). Currently, this PGO-based approach is done manually. We profiled the SIMD benchmarks and instrumented the SIMD regions to check for migration once every second. Because each benchmark was profiled beforehand, the relative slowdown of all applications is known up front when setting up classification groups for schedulers.

Our evaluation workload is generated by a script that starts a workload batch with a predefined SIMD/non-SIMD ratio of benchmark composition. To ensure fairness, in the first iteration, the workload script assigns benchmarks based on affinities. Then when all cores are fully occupied, the script randomly assigns benchmarks remaining in the workload batch to the next available free core to best mimic a dynamic workload scenario. In a dynamic workload scenario, the incoming benchmarks can not be predicted but the overall ratio can be estimated. If a workload batch is finished, the script regenerates an identical batch from which to select. This process continues until the evaluation period ends. To ensure a fair comparison, we used the same random seed so that each configuration has the same benchmark selection outcome for every run. Each experiment is run for a duration of 75 minutes. The rationale behind this is that most of the benchmarks execute in about 3 to 5 minutes when running on the x86 CPU; a period of 75 minutes is large enough to mitigate the impact of noise.

5 Experimental Results

We evaluate our proposed framework to understand the effectiveness of het-dynamic on improving system throughput on mixed non-SIMD/SIMD workloads. To this end, we conducted two sets of experiments. The first set considered workloads composed of two applications: one non-SIMD benchmark and one SIMD benchmark. The goal of this experiment is to determine the workloads that yield throughput gains for het-dynamic. By focusing on only two applications, we can carefully control individual benchmarks with different slowdowns.

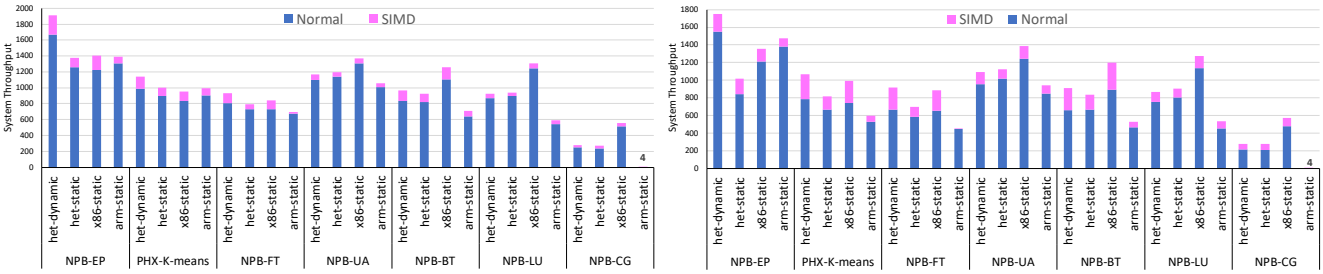
Using the insights gained from these experiments, our second set of experiments considered a more realistic workload consisting of multiple non-SIMD and SIMD benchmarks. For easier comparison, we used the Hydro benchmark from the Livermoore Loops suite [61] as the designated SIMD benchmark for both sets of experiments.

5.1 Two-application Workload

Figures 6a and 6b show the system throughput of a two-application workload under two different SIMD/non-SIMD ratios: 12.5/87.5% and 25/75% respectively. The x-axis represents the benchmarks tested as the non-SIMD benchmark, arranged in increasing slowdown order according to Figure 1, with the leftmost being EP from the NPB suite [9], which has the lowest slowdown, and the rightmost being CG from NPB, which has the highest slowdown. The y-axis is the total number of benchmarks completed in a testing period of 75 minutes, where higher is better. The upper portion of each bar represents the number of SIMD benchmarks completed and the bottom portion represents the non-SIMD benchmarks completed in the testing period.

These figures reveal that het-dynamic consistently outperforms het-static. For the 12.5% SIMD scenario, het-dynamic outperforms all baselines in three cases: EP, K-means, and FT. EP has the best performance gain of ~36% over the next best baseline. However, the performance gain shrinks to ~14% for K-means and ~11% for FT. For other benchmarks with increasing slowdown, x86-static consistently outperforms het-dynamic and also het-static and ARM-static. An average of 6.3% of non-SIMD benchmarks migrated to ARM in order to vacate x86 cores for SIMD applications and no SIMD application are spilled onto ARM.

Similar trends occur in the 25% SIMD scenario – het-dynamic still has better performance for EP, K-means, and



(a) 1/8 SIMD/non-SIMD ratio. **(b) 1/4 SIMD/non-SIMD ratio.**

Figure 6: Throughput of two-application workloads with different SIMD/non-SIMD ratios.

FT. However, the performance gain drops down to ~19%, ~8% and ~3% respectively. This performance decrease can be attributed to an average of 5.1% SIMD benchmarks spilled onto ARM. However, every spilled SIMD benchmark eventually migrates back to x86 cores. The average percentage of non-SIMD benchmarks migrated to vacate x86 cores for SIMD benchmarks is around 12.8%. This almost doubled percentage is likely due to the 2x increase in SIMD ratio. We also tested larger SIMD ratios of 50% (i.e., 50% SIMD applications) and 100% (i.e., all SIMD). In both cases, het-dynamic shows no performance gain over x86-static.

From these results, we can draw several conclusions. First, het-dynamic performs well with low SIMD ratio workloads. For workloads with larger SIMD ratios, het-dynamic lacks enough high affinity x86 cores to execute SIMD applications; thus, spilling SIMD applications onto ARM has a negative impact on throughput. The spilled SIMD applications are forced to execute on a significantly slower architecture with narrower SIMD widths until one of the existing SIMD applications on an x86 core finishes. This degrades throughput.

Second, het-dynamic yields better throughput for low slowdown applications. In both Figures 6a and 6b, all three benchmarks either belong to or close enough to the low slowdown group. Low slowdown applications allow het-dynamic to compensate for the slowdown with the higher number of ThunderX cores. This allows the system to execute more applications in parallel, thereby outperforming x86-static. Despite a low SIMD ratio, het-dynamic is also better than ARM-static because there are still a few SIMD benchmarks in the mix. In the ARM-static case, the SIMD applications become stragglers due to extreme slowdowns, which ultimately harms throughput. However, het-dynamic's scheduler better matches applications to cores for improving throughput.

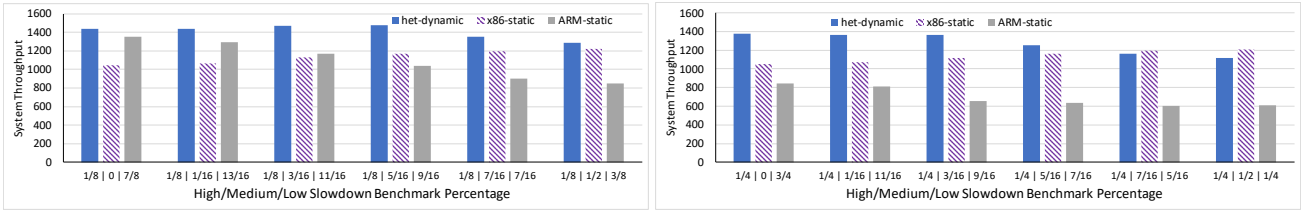
Lastly, the impact of het-dynamic's scheduler on system throughput cannot be ignored. The scheduler allows het-dynamic to better allocate resources based on the incoming application's ISA affinity. The impact of the scheduler can also be reflected by the fact that the EP-Hydro (SIMD) workload combination has the best performance gain in both 1/4 and 1/8 scenarios. The EP-Hydro workload combination contains two benchmarks that have the most diverse CPU affinity. Thus, migrating processes across ISAs to match their ISA affinities enables het-dynamic to obtain the largest throughput gain (36%) over the baselines.

5.2 Multi-application Workloads

Figures 7a and 7b show the system throughput of workloads composed of more than two applications with the high slowdown benchmark (SIMD only) fixed at 12.5% and 25%, respectively. For both scenarios, we fixed the ratio of the high slowdown group benchmarks and only selected SIMD benchmarks belonging to that group. This is done to further evaluate SIMD's impact on system performance. For each scenario, we varied the ratio of benchmarks belonging to low and medium slowdown groups. The x-axis represents each group's ratio and the y-axis shows the system throughput.

In the 12.5% high slowdown group (SIMD) ratio scenario, het-dynamic outperforms both x86-static and ARM-static in all tested cases with an average gain of 14.6% and a maximum gain of ~26% over the next best homogeneous baseline with workloads consisting of 12.5%, 31.25%, and 56.25% of high, medium, and low slowdown group members respectively. However, similar to the two-application workload experiments, the performance gain shrinks as the percentage of medium slowdown benchmarks increases. This is due to our scheduling policies reduce effectiveness when workloads consist of more benchmarks biased towards a single ISA. An average of 23.1% of total benchmarks are migrated to ARM during the experiment to vacate cores for benchmarks with better affinity. 24.7% of total benchmarks on average are eventually migrated back to x86. This larger migrate back percentage indicates that our scheduler is able to utilize the system to migrate back applications that started earlier to faster cores if there is no difference in affinity.

Similar trends also occur in the 25% scenario – het-dynamic still has the best performance in four out of six test cases, achieving a maximum performance gain of 31.1%.



(a) 1/8 SIMD ratio. **(b) 1/4 SIMD ratio.**

Figure 7: Throughput of multi-application workloads with different SIMD/non-SIMD ratios.

An average of 28% of total benchmarks are migrated to ARM during the experiment to vacate cores for applications with better affinity. An average of 24.4% are eventually migrated back to x86. We tested larger high slowdown group ratios (e.g., 50%, 100%). In both cases, het-dynamic has no performance gain over x86-static in all workload cases.

From the multi-application workload experiment, we gain further understanding of our heterogeneous-ISA systems design. Het-dynamic is best equipped to handle workloads that contain large low slowdown and medium slowdown application ratios. This further expands the "sweet spot" of het-dynamic because in the two-application scenario, we do not see any benchmark that has medium slowdown actually benefiting from het-dynamic.

The multi-application experiments thus reveal that in a more realistic workload with multiple diverse applications, het-dynamic can achieve increased performance for applications with slightly higher x86 affinity. het-dynamic can therefore achieve higher throughput over a broader workload spectrum than comparable homogeneous setups. This is achieved by better matching each application's ISA affinity to optimal cores with additional migration and scheduling capabilities, improving the system throughput.

6 Related Work

The vast majority of past efforts in heterogeneous computing focus on CPU/GPU systems [28, 63] in which a GPU accelerator device is attached to a host CPU [51, 66, 67, 78]. Gad, et al. [28] propose a similar context migration process between CPU and GPU. Past efforts have also focused on single-ISA heterogeneous systems [45–47] which use cores of the same ISA but with different ISA extensions or micro-architecture (e.g., ARM's big.LITTLE [29], NVidia's Kal-El [67]).

More recently, there have been several efforts on heterogeneous-ISA systems [10, 50, 53, 84, 85]. Venkat, et al. [85] investigates the design space of heterogeneous ISAs using general-purpose ISA cores (e.g., x86, Thumb, Alpha) to evaluate their effectiveness for improving performance and energy. Their work reveals that many applications, especially in the HPC domain, exhibit ISA affinity, often in different program phases. ISA affinity was further studied in [1, 2]. Exploiting ISA affinity for performance and energy gains requires a cross-ISA execution migration infrastructure, which can transform the program state from one ISA format to another and migrate execution to the optimal-ISA core. Barbalace, et al. present a complete software stack – Popcorn Linux – that supports a cross-ISA execution migration infrastructure which we summarize in Section 2.2 [10]. These efforts do not consider cross-ISA migration inside SIMD regions – exactly the problem that we study. We extend [10]'s compiler and run-time for cross-ISA SIMD migration.

Lee, et al. [50] investigate the offloading of compute-heavy workloads from mobile platforms, which often use RISC-style ISAs such as ARM, to server platforms that use CISC-style ISAs such as x86. They present a compiler infrastructure that generates binaries that can execute on multiple ISAs such as those with uniform memory layouts, address conversion code, endianness translation/conversion code, and a run-time system for orchestrating the offloading of computations across ISA-different platforms. We do not consider offloading tasks because our goal is to maximize throughput, which requires us to utilize both servers as much as possible, rather than having one platform wait for the completion of the offloaded tasks. Offloading and migrating processes are also different in terms of process execution. Offloaded processes require synchronization at the end of the offloaded computation with their parent processes and exit at the original platform. In contrast, a migrated process is able to exit on the migrated architecture since its call stack is transformed. Lin, et al. [53] is very similar to [10], but focuses on mobile incoherent domain SoCs whereas our work focuses on the server space. Venkat, et al. [84] focuses on leveraging a single large superset ISA composed of fully custom ISAs but scopes out cross-ISA migration and lacks a real prototype.

With SIMD gaining more attention from chip vendors [6, 38, 48] as a means to extract additional data parallelism, research interest in this space has surged. Most efforts focus on redesigning algorithms to leverage SIMD instructions [17, 31, 34, 36, 72], exploring SIMD usage in new application domains [18, 21, 32, 90], and improving SIMD code generation at the compiler level [7, 27]. SIMD instructions have also been considered in dynamic binary translation (DBT) efforts [19, 26, 33, 52, 54, 71], which focus on efficient

translation of SIMD registers between ISAs. In addition to cross-ISA SIMD translations at migration points, our work also conducts SIMD-aware scheduling to maximize system throughput of workloads composed of SIMD and non-SIMD applications – entirely out of scope for DBT efforts.

Past efforts have also focused on scheduling in heterogeneous systems. However, most of these efforts focus on single-ISA heterogeneous systems, where the heterogeneity is in terms of execution frequency [59, 74], micro-architecture [43], cache sizes [41], or performance goals [73]. These schedulers follow the same principle of allocating resources to applications based on their resource demand – e.g., if an application requires a higher usage of a resource (frequency, micro-architectural features, cache line), an attempt is made to grant that resource.

Scheduling in heterogeneous-ISA systems has received less attention. Beisel, et al. [11] extend the Linux CFS to support cooperative multitasking for heterogeneous accelerators (CPU/GPU). Barbalace, et al.'s [10] scheduler balances thread counts across ISA-different cores. Prodromou, et al. [76] presents a machine learning-based program performance predictor that drives an ML-based heterogeneous-ISA job scheduler. These works have scoped out migration costs as well as SIMD workloads. Karaoui, et al. [42] presents schedulers for heterogeneous-ISA systems considering migration costs, but scopes out SIMD workloads.

7 Discussion and Future Directions

We believe that our work only scratches the surface of the heterogeneous-ISA SIMD space. Many promising future directions exist. For example, we scoped out optimizing energy costs largely due to the process node gap between the two servers that we selected. The Cavium ThunderX server (initially released in 2014) uses a more power consuming 28 nm process, whereas the Xeon server (released in 2018) uses a more recent 14 nm process. Combined with the fact that the ThunderX does not implement many energy saving features such as low-power states, DVFS, and clock gating, the ThunderX is not an energy efficient CPU.

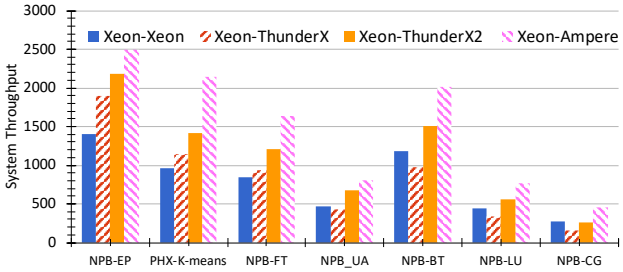


Figure 8: Throughput of het-static and x86-static on ThunderX2 [83] and Ampere [20] servers.

Recent ARM servers such as Marvell's ThunderX2 [83] and Ampere's eMAG server [20] are likely more energy efficient and have higher performance than ThunderX. We could not evaluate het-dynamic on a Xeon-ThunderX2 or a Xeon-Ampere configuration as that requires significant amount of engineering effort in porting the Popcorn Linux infrastructure necessary for cross-ISA migration on these platforms. However, we measured het-static on both these configurations as it does not involve cross-ISA migration; Figure 8 shows these results.

het-static on Xeon-Ampere shows the best performance with average throughput gain of 100% over het-dynamic on Xeon-ThunderX and 82% over x86-static on Xeon-Xeon. From our evaluation, we show that het-dynamic outperforms het-static on similar heterogeneous-ISA servers. Thus, extrapolating from Figure 8, het-dynamic will likely outperform het-static and x86-static on these newer servers. Another promising direction is scheduling. Recent results such as [64] reveal that machine learning-based approaches can accurately predict program performance for superior scheduling policies. This can be leveraged in the heterogeneous-ISA SIMD scheduling space as well.

8 Conclusion

We championed the usability of het-ISA system compared to mainstream homogeneous-ISA systems. We explored whether in the space of SIMD, het-ISA systems can be leveraged for performance gains. We extended the Popcorn Linux [10] framework to support migration inside SIMD regions. Efficiently using ISA affinity and dynamically migrating applications to use ISA optimal cores in heterogeneous-ISA systems can result in significant performance gains over homogeneous architectures. Enabling migration inside SIMD regions allows us to tap into an important instruction set that is widely used. Our work's main conclusion is that there is "no one ISA/micro-architecture that fits all." The fact that het-dynamic allows two heterogeneous-ISA servers that are five years apart in production to outperform two 2018-released x86 servers is a strong validation of our results.

Acknowledgements

We thank the reviewers for their insightful comments which have significantly improved the paper. This work is supported in part by ONR under grants N00014-13-1-0317, N00014-16-1-2711, and N00014-18-1-2022, and NAVSEA/NEEC under grants 3003279297 and N00174-16-C-0018.

References

[1] Ayaz Akram. 2017. A Study on the Impact of Instruction Set Architectures on Processor's Performance. Master Thesis, Western Michigan University.

[2] Ayaz Akram and Lina Sawalha. 2017. The Impact of ISAs on Performance. In *Workshop on Duplicating, Deconstructing and Debunking (WDDD) co-located with 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada.*

[3] Amazon. 2018. EC2 Instances Powered by ARM-based AWS Graviton Processors. https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-processors/.

[4] Kazumaro Aoki, Fumitaka Hoshino, Tetsutaro Kobayashi, and Hiroaki Oguro. 2001. Elliptic Curve Arithmetic Using SIMD. In *Information Security*, George I. Davida and Yair Frankel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 235–247.

[5] ARM. 2018. *ARM HPC tools for SVE.* Technical Report. https://developer.arm.com/products/software-development-tools/hpc/sve.

[6] ARM. 2018. *NEON.* Technical Report. https://developer.arm.com/technologies/neon.

[7] Mehmet Ali Arslan, Flavius Gruian, Krzysztof Kuchcinski, and Andréas Karlsson. 2016. Code generation for a SIMD architecture with custom memory organisation. In *Design and Architectures for Signal and Image Processing (DASIP), 2016 Conference on.* IEEE, 90–97.

[8] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.* 26, 4 (Dec. 1994), 345–420.

[9] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. 1992. NAS parallel benchmark results. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing.* 386–393.

[10] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17).* ACM, New York, NY, USA, 645–659. https://doi.org/10.1145/3037697.3037738

[11] Tobias Beisel, Tobias Wiersema, Christian Plessl, and André Brinkmann. 2011. Cooperative multitasking for heterogeneous accelerators in the linux completely fair scheduler. In *ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors.* IEEE, 223–226.

[12] Chandler Carruth Bob Wilson, Diego Novillo. 2007. PGO and LLVM, Status and Current Work. https://llvm.org/devmtg/2013-11/slides/Carruth-PGO.pdf

[13] Shekhar Borkar. 2007. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference.* ACM, 746–749.

[14] Dennis Bradford, Sundaram Chinthamani, Jesus Corbal, Adhiraj Hassan, Ken Janik, and Nawab Ali. 2017. Knights Mill: New Intel Processor For Machine Learning. In *Hot Chips 29.*

[15] D. Callahan, J. Dongarra, and D. Levine. 1988. Vectorizing Compilers: A Test Suite and Results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing (Supercomputing '88).* IEEE Computer Society Press, Los Alamitos, CA, USA, 98–105.

[16] Cavium. 2013. ThunderX CN8890 - Cavium. https://en.wikichip.org/wiki/cavium/thunderx/cn8890

[17] Hao Chen, Nicholas S Flann, and Daniel W Watson. 1998. Parallel genetic simulated annealing: a massively parallel SIMD algorithm. *IEEE Transactions on Parallel and Distributed Systems* 9, 2 (1998), 126–136.

[18] Chi Ching Chi, Mauricio Alvarez-Mesa, Benjamin Bross, Ben Juurlink, and Thomas Schierl. 2015. SIMD acceleration for HEVC decoding. *IEEE Transactions on circuits and systems for video technology* 25, 5 (2015), 841–855.

[19] Nathan Clark, Amir Hormati, Sami Yehia, Scott Mahlke, and Krisztian Flautner. 2007. Liquid SIMD: Abstracting SIMD hardware using lightweight dynamic mapping. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture.* IEEE, 216–227.

[20] Ampere Computing. 2018. Ampere Processors. https://amperecomputing.com/product/

[21] Gregory W Cook and Edward J Delp. 1995. An investigation of scalable SIMD I/O techniques with application to parallel JPEG compression. *Journal of Parallel and distributed computing* 30, 2 (1995), 111–128.

[22] R. Cypher and J. L. C. Sanz. 1989. SIMD architectures and algorithms for image processing and computer vision. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37, 12 (Dec 1989), 2158–2174.

[23] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA).* 365–376.

[24] Michael Feldman. 2017. Intel Dumps Knights Hill, Future of Xeon Phi Product Line Uncertain. https://www.top500.org/news/intel-dumps-knights-hill-future-of-xeon-phi-product-line-uncertain/.

[25] Michael Feldman. 2018. Intel Ships Xeon Skylake Processor with Integrated FPGA. https://www.top500.org/news/intel-ships-xeon-skylake-processor-with-integrated-fpga/.

[26] Sheng-Yu Fu, Ding-Yong Hong, Jan-Jan Wu, Pangfeng Liu, and Wei-Chung Hsu. 2015. Simd code translation in an enhanced hqemu. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS).* IEEE, 507–514.

[27] Sheng-Yu Fu, Jan-Jan Wu, and Wei-Chung Hsu. 2015. Improving SIMD code generation in QEMU. In *Proceedings of the 2015 design, automation & test in europe conference & exhibition.* EDA Consortium, 1233–1236.

[28] Ramy Gad, Tim Süß, and André Brinkmann. 2014. Compiler Driven Automatic Kernel Context Migration for Heterogeneous Computing. In *2014 IEEE 34th International Conference on Distributed Computing Systems.* IEEE, 389–398.

[29] P. Greenhalgh. 2011. big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. Technical report, ARM.

[30] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-data Processing of Big Data Workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16).* IEEE Press, Piscataway, NJ, USA, 153–165. https://doi.org/10.1109/ISCA.2016.23

[31] Arthur Hennequin, Ian Masliah, and Lionel Lacassagne. 2019. Designing efficient SIMD algorithms for direct Connected Component Labeling. In *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing.* ACM, 4.

[32] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein. 2014. Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi-and manycore chips. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing.* ACM, 57–64.

[33] Ding-Yong Hong, Sheng-Yu Fu, Yu-Ping Liu, Jan-Jan Wu, and Wei-Chung Hsu. 2016. Exploiting longer SIMD lanes in dynamic binary translation. In *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on.* IEEE, 853–860.

[34] IK Hong, ST Chung, HK Kim, YB Kim, YD Son, and ZH Cho. 2007. Ultra fast symmetry and SIMD-based projection-backprojection (SSP) algorithm for 3-D PET image reconstruction. *IEEE transactions on medical imaging* 26, 6 (2007), 789–803.

[35] Joel Hruska. 2018. Intel Uses New Foveros 3D Chip-Stacking to Build Core, Atom on Same Silicon. https://bit.ly/2SSQ62R

[36] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. 2007. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*. IEEE, 189–198.

[37] Texas Instruments. 2004. OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide.

[38] Intel. 2013. Intel Advanced Vector Extensions 512 (Intel AVX-512). https://intel.ly/2SyYl4i.

[39] Intel. 2017. Intel Xeon Gold 5118 Processor (16.5M Cache, 2.30 GHz) Product Specifications. https://ark.intel.com/products/120473/Intel-Xeon-Gold-5118-Processor-16-5M-Cache-2-30-GHz-

[40] Intel. 2018. Intel Xeon Processor Scalable Family. https://intel.ly/2t1apTH.

[41] Xiaowei Jiang, Asit Mishra, Li Zhao, Ravishankar Iyer, Zhen Fang, Sadagopan Srinivasan, Srihari Makineni, Paul Brett, and Chita R Das. 2011. ACCESS: Smart scheduling for asymmetric cache CMPs. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 527–538.

[42] Mohamed Karaoui, Anthony Carno, Robert Lyerly, Sang-Hoon Kim, Pierre Olivier, Changwoo Min, and Binoy Ravindran. 2019. Scheduling HPC Workloads on Heterogeneous-ISA Architectures. In *24th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. Poster paper.

[43] David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*. ACM, 125–138.

[44] Vlad Krasnov. 2017. On the dangers of Intel's frequency scaling. https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[45] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 81.

[46] Rakesh Kumar, Dean M Tullsen, and Norman P Jouppi. 2006. Core architecture optimization for heterogeneous chip multiprocessors. In *Parallel Architectures and Compilation Techniques (PACT), 2006 International Conference on*. IEEE, 23–32.

[47] Rakesh Kumar, Dean M Tullsen, Parthasarathy Ranganathan, Norman P Jouppi, and Keith I Farkas. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*. IEEE, 64–75.

[48] Chris Lamont. 2011. *Introduction to Intel Advanced Vector Extensions*. Technical Report. https://intel.ly/2ETCWyt.

[49] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[50] G. Lee, H. Park, S. Heo, K. Chang, H. Lee, and H. Kim. 2015. Architecture-aware automatic computation offload for native applications. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 521–532. https://doi.org/10.1145/2830772.2830833

[51] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2013. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 245–256.

[52] Jianhui Li, Qi Zhang, Shu Xu, and Bo Huang. 2006. Optimizing dynamic binary translation for SIMD instructions. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 12–pp.

[53] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. 2014. 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 '14)*. ACM, New York, NY, USA, 285–300.

[54] Yu-Ping Liu, Ding-Yong Hong, Jan-Jan Wu, Sheng-Yu Fu, and Wei-Chung Hsu. 2017. Exploiting Asymmetric SIMD Register Configurations in ARM-to-x86 Dynamic Binary Translation. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 343–355.

[55] LLVM. 2003. Auto-Vectorization in LLVM. https://llvm.org/docs/Vectorizers.html

[56] LLVM. 2003. How To Build Clang and LLVM with Profile-Guided Optimizations. https://llvm.org/docs/HowToBuildWithPGO.html

[57] LLVM. 2007. Profile Guided Optimization. https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization

[58] Arm Ltd. 2017. Technologies | DynamIQ Arm Developer. https://developer.arm.com/technologies/dynamiq

[59] Luca Lugini, Vinicius Petrucci, and Daniel Mosse. 2012. Online thread assignment for heterogeneous multicore systems. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE, 538–544.

[60] Marvell. 2019. LiquidIO II 10/25GbE Adapter family. https://bit.ly/2H7NWLk.

[61] F H McMahon. 1986. *The Livermore Fortran kernels: a computer test of the numerical performance range*. Lawrence Berkeley Nat. Lab., Berkeley, CA. https://cds.cern.ch/record/178064

[62] Sparsh Mittal. 2016. A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors. *ACM Comput. Surv.* 48, 3, Article 45 (Feb. 2016), 38 pages. https://doi.org/10.1145/2856125

[63] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* 47, 4, Article 69 (July 2015), 35 pages. https://doi.org/10.1145/2788396

[64] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, and A. Cristal. 2017. A Machine Learning Approach for Performance Prediction and Scheduling on Heterogeneous CPUs. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 121–128.

[65] Netronome. 2019. Agilio SmartNICs. https://www.netronome.com/products/agilio-cx/.

[66] Nvidia. 2010. The Benefits of Multiple CPU Cores in Mobile Devices. https://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf

[67] Nvidia. 2011. Variable SMP - A Multi-Core CPU Architecture for Low Power and High Performance. https://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0911b.pdf

[68] Pierre Olivier, Sang-Hoon Kim, and Binoy Ravindran. 2017. OS Support for Thread Migration and Distribution in the Fully Heterogeneous Datacenter. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*. ACM, New York, NY, USA, 174–179. https://doi.org/10.1145/3102980.3103009

[69] OpenCV. 2018. *OpenCV: Introduction*. Technical Report. https://docs.opencv.org/3.3.1/d1/dfb/intro.html.

[70] Edson Luiz Padoin, Laércio Lima Pilla, Márcio Castro, Francieli Z Boito, Philippe Olivier Alexandre Navaux, and Jean-François Méhaut. 2014. Performance/energy trade-off in scientific computing: the case of ARM big. LITTLE and Intel Sandy Bridge. *IET Computers & Digital Techniques* 9, 1 (2014), 27–35.

[71] Alex Pajuelo, Antonio González, and Mateo Valero. 2002. Speculative dynamic vectorization. In *ACM SIGARCH Computer Architecture News*, Vol. 30. IEEE Computer Society, 271–280.

[72] Szilárd Páll and Berk Hess. 2013. A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications* 184, 12 (2013), 2641–2650.

[73] S. Panneerselvam and M. Swift. 2016. Rinnegan: Efficient resource use in heterogeneous architectures. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 373–386. https://doi.org/10.1145/2967938.2967964

[74] Vinicius Petrucci, Orlando Loques, and Daniel Mossé. 2012. Lucky Scheduling for Energy-Efficient Heterogeneous Multi-Core Systems.. In *HotPower*.

[75] Ioannis Pitas (Ed.). 1993. *Parallel Algorithms: For Digital Image Processing, Computer Vision and Neural Networks*. John Wiley & Sons, Inc., New York, NY, USA.

[76] Andreas Prodromou, Ashish Venkat, and Dean M Tullsen. 2019. Deciphering Predictive Schedulers for Heterogeneous-ISA Multicore Architectures. (2019).

[77] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 13–24. https://doi.org/10.1109/HPCA.2007.346181

[78] Greg Sadowski. 2014. Design challenges facing CPU-GPU-Accelerator integrated heterogeneous systems. In *Design Automation Conference (DAC'14)*.

[79] David Schor. 2018. Intel Reveals 10nm Sunny Cove Core, a New Core Roadmap, and Teases Ice Lake Chips. https://bit.ly/2NLEbTg

[80] J. M. Shalf and R. Leland. 2015. Computing beyond Moore's Law. *Computer* 48, 12 (Dec 2015), 14–23. https://doi.org/10.1109/MC.2015.374

[81] Michael B Taylor. 2013. A landscape of the new dark silicon design regime. *IEEE Micro* 33, 5 (2013), 8–19.

[82] Marvell Technology. 2013. ThunderX ARM-based Processors. https://www.marvell.com/server-processors/thunderx-arm-processors/.

[83] Marvell Technology. 2018. ThunderX2 ARM-based Processors. https://www.marvell.com/server-processors/thunderx2-arm-processors/.

[84] Ashish Venkat, H. Basavaraj, and D. M. Tullsen. 2019. Composite-ISA Cores: Enabling Multi-ISA Heterogeneity Using a Single ISA. HPCA.

[85] Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 121–132. http://dl.acm.org/citation.cfm?id=2665671.2665692

[86] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 205–218. https://doi.org/10.1145/1736020.1736044

[87] David G. von Bank, Charles M. Shub, and Robert W. Sebesta. 1994. A Unified Model of Pointwise Equivalence of Procedural Computations. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1842–1874. https://doi.org/10.1145/197320.197402

[88] R. S. Williams. 2017. What's Next? [The end of Moore's law]. *Computing in Science Engineering* 19, 2 (Mar 2017), 7–13. https://doi.org/10.1109/MCSE.2017.31

[89] Michael Witbrock and Marco Zagha. 1990. An implementation of back-propagation learning on GF11, a large SIMD parallel computer. *Parallel Comput.* 14, 3 (1990), 329 – 346. https://doi.org/10.1016/0167-8191(90)90085-N

[90] Demin Xiong and Duane F Marble. 1996. Strategies for real-time spatial analysis using massively parallel SIMD computers: an application to urban traffic flow analysis. *International Journal of Geographical Information Systems* 10, 6 (1996), 769–789.

[91] Yuhao Zhu and Vijay Janapa Reddi. 2013. High-performance and energy-efficient mobile web browsing on big/little systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 13–24.