UNIFICO: Thread Migration in Heterogeneous-ISA CPUs without State Transformation

UNIFICO: Thread Migration in Heterogeneous-ISA CPUs without State Transformation

Nikolaos Mavrogeorgis
University of Edinburgh
United Kingdom
nikos.mavrogeorgis@ed.ac.uk

Christos Vasiladiotis
University of Edinburgh
United Kingdom
c.vasiladiotis@ed.ac.uk

Pei Mu
University of Edinburgh
United Kingdom
pei.mu@ed.ac.uk

Amir Khordadi
University of Edinburgh
United Kingdom
amir.khordadi@ed.ac.uk

Björn Franke
University of Edinburgh
United Kingdom
b.franke@ed.ac.uk

Antonio Barbalace
University of Edinburgh
United Kingdom
antonio.barbalace@ed.ac.uk

Abstract

Heterogeneous-ISA processor designs have attracted considerable research interest. However, unlike their homogeneous-ISA counterparts, explicit software support for bridging ISA heterogeneity is required. The lack of a compilation toolchain ready to support heterogeneous-ISA targets has been a major factor hindering research in this exciting emerging area. For any such compiler "getting right" the mechanics involved in state transformation upon migration and doing this efficiently is of critical importance. In particular, any runtime conversion of the current program stack from one architecture to another would be prohibitively expensive. In this paper, we design and develop Unifico, a new multi-ISA compiler that generates binaries that maintain the *same* stack layout during their execution on either architecture. Unifico avoids the need for runtime stack transformation, thus eliminating overheads associated with ISA migration. Additional responsibilities of the Unifico compiler backend include maintenance of a uniform ABI and virtual address space across ISAs. Unifico is implemented using the LLVM compiler infrastructure, and we are currently targeting the x86-64 and ARMv8 ISAs. We have evaluated Unifico across a range of compute-intensive NAS benchmarks and show its minimal impact on overall execution time, where less than 6% overhead is introduced on average. When compared against the state-of-the-art Popcorn compiler, Unifico reduces binary size overhead from ~200% to ~10%, whilst eliminating the stack transformation overhead during ISA migration.

CC '24, March 2–3, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641565

CCS Concepts: • **Software and its engineering → Compilers**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords: heterogeneous-ISA, thread migration, compilers

ACM Reference Format:
Nikolaos Mavrogeorgis, Christos Vasiladiotis, Pei Mu, Amir Khordadi, Björn Franke, and Antonio Barbalace. 2024. UNIFICO: Thread Migration in Heterogeneous-ISA CPUs without State Transformation. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3640537.3641565

1 Introduction

Heterogeneity in computing hardware (CPUs, GPUs, TPUs, and FPGAs) is common practice today in a multitude of deployments and configurations [46, 49, 69]. This has been driven by the ever-increasing computational demands of workloads, whose data-processing requirements boomed recently [52]. Yet, processing (large amounts of) data across heterogeneous processing units poses several problems, like hindering programmability.

Classical software compilation, targeting a single instruction set architecture (ISA) and the related programming models, e.g., shared memory programming, is not applicable as-is to heterogeneous-ISA platforms. Instead, the state-of-the-practice is to isolate (or mark) a set of functions to be run on a processing unit different from the main central processing unit (CPU), compile them for the specific ISA, and offload them at runtime. Bespoke programming frameworks exist to support the application programmer (e.g., OpenCL, CUDA), and include development and runtime environments [48].

To improve the programmability of such platforms, different solutions have been introduced, like the reuse of data pointers across CPUs and heterogeneous processing units. These required hardware memory management per processing unit, and motivated the introduction of coherent shared memory between the host CPU and heterogeneous processing units, either on the same chip [1], or via the peripheral bus [5, 7, 9, 13].

Emerging heterogeneous-ISA platforms. At the same time, the landscape of heterogeneous hardware computing is widening. While classical heterogeneous-ISA platforms, comprising a single general-purpose CPU and multiple special-purpose processing units (GPUs, TPUs, and FPGAs), are widely available, platforms with multiple general-purpose diverse-ISA processing units are emerging. Similarly to classical heterogeneous hardware, emerging heterogeneous-ISA platforms are also going to offer shared memory amongst ISAs.

While academia proposed single-chip, cache-coherent heterogeneous-ISA CPUs, e.g., BYOC [23] – which never reached the market, new peripheral interconnects like CXL [7] promise to enable coherent shared memory between the main CPU and the processing units of peripheral devices, such as SmartNICs [3, 37] or SmartSSDs [24, 64] (ARM- or RISC-V- based). Also, CXL will accommodate memory expansion cards that will likely integrate general- and special-purpose processing units for near-data processing (NDP) [59], as in UPMEM [35] processing-in-memory (PIM), where such processing units directly access the same memory as the main CPU.

Programming emerging platforms. Classic heterogeneous computing runs an application on the CPU and offloads a specific part of it to special-purpose processing units. However, when multiple CPUs of diverse ISAs lie on the same platform, thread migration has been shown to be more beneficial than offloading, enabling decisions at runtime rather than statically deciding on the function to offload at compile time [27, 28, 36, 54].

While earlier works on heterogeneous-ISA migration [47, 60] require the transformation of the entire application state before execution on another ISA, recent approaches transform only part of the state (e.g., the registers and stack), guided by metadata derived during compilation. Despite that, the transformation step and related metadata still incur execution time and binary size overheads, respectively, which are mostly linear to the number and size of active stack frames when migrating [27, 36]. Both overheads impact migration time, potentially hindering its benefits. Lastly, although outside the scope of this work, state transformation approaches constitute a potential attack surface for the binaries [30], e.g., by exposing the return addresses of functions that can be leveraged to create return-oriented programming (ROP) gadgets.

Unifico. Motivated by the emerging heterogeneous-ISA platforms with shared memory, with the goal of making programmability as simple as homogeneous-ISA platforms, and removing execution time and code size overheads of state transformation, we propose Unifico. Unifico is a compilation technique that generates multi-ISA binaries with a unified address space layout and application state (including stack, heap, thread-local storage, etc.) across different ISAs, enabling thread migration without transformations. We achieve this by rethinking how compiler backends generate code, extending the code generation passes that impact the application state to adhere to a common set of rules, without the need to graft any metadata in the binary.

We prototyped Unifico within the LLVM compiler, targeting the 64-bit versions of the ARM and x86 ISAs, and studied its efficacy on different benchmarks. Comparing against binaries generated using previous heterogeneous-ISA CPU migration projects, we demonstrate that Unifico adds on average no more than 6% execution time overheads and no more than 10% code size increases. We envision Unifico being integrated in existing compiler frameworks, and used in modern heterogeneous hardware platforms.

We make the following *contributions*:

- We introduce Unifico, a compilation technique that enables thread migration amongst heterogeneous-ISA CPUs *without state transformation*, removing its initialization, runtime, and code overheads.
- We prototyped Unifico within the LLVM compiler, targeting the x86 and ARM backends, and validate migration on the NAS Parallel Benchmarks (NPB) suite, utilizing the CRIU checkpoint and restore software. Unifico is released as open source software (OSS) at https://github.com/systems-nuts/unifico.
- We evaluated Unifico on different benchmarks, showing that heterogeneous-ISA CPU migration works, and demonstrate on average no more than 10% binary size overhead, no more than 6% overhead on execution time (without migration).

The rest of the paper is structured as follows: Section 2 sets the background and motivation for our approach, Section 3 provides an illustrative example of the current limitations and our solution; Sections 4 to 6 present Unifico's design, implementation and evaluation, respectively; we highlight related work in Section 7 and conclude in Section 8.

2 Background and Motivation

2.1 Heterogeneous-ISA Architectures

This work is motivated by the emerging compute heterogeneity, i.e., heterogeneous ISAs [23, 27, 28, 33, 36, 51, 65], coupled with next-generation memory architectures materialized by new interconnect technologies [5, 9, 13, 38, 50, 62], e.g., CXL [7, 20]. In such configurations, thread migration has been shown to be more advantageous than the traditional and prevalent offloading techniques [28, 36].

In particular, we focus on a combination of x86 and ARM CPUs inspired by a family of emerging platforms closely related to NDP [12, 25, 35, 59, 61]. These usually accommodate a brawny host processor (e.g., x86) plus one or more simpler reduced instruction set computer (RISC) processors (usually of different ISAs, e.g., ARM) near the memory, to avoid data movement and increase bandwidth utilization. Memory-intensive applications that exhibit weak locality are well suited for these architectures [42]. Their dominant programming paradigm is offloading.

```
queue.push(root)
while len(queue) > 0:
    for src in queue:
        for dst in out_edges(src):
            # Can migrate here with UNIFICO
            if parent[dst] == -1:
                parent[dst] = src
                queue.push(dst)
```

Figure 1. Pseudo-code of a queue-based BFS application. The for-loops cannot be used directly as offload kernels without major modifications.

2.2 Thread Migration Techniques

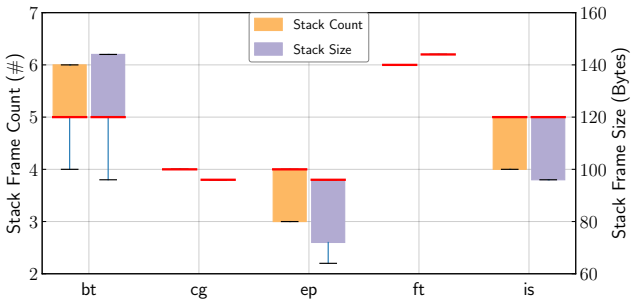
Dynamic software thread migration is the act of moving a thread's execution context (e.g., register state, stack contents, page mappings, etc.) between different processing units in a system [21, 39, 47, 60]. In shared-memory programming (SMP) systems, this can be achieved through hardware and operating system (OS) mechanisms. However, in the case of heterogeneous-ISA systems, additional compiler and runtime support is needed, since the thread state needs to be transformed in order to match the architecture-specific details of the target processor [26–28, 33, 36, 65, 67].

There are three main axes to consider when performing thread migration: First, *migration granularity*, which denotes where the program is able to migrate (e.g., function boundaries [27, 33, 66]). This property directly affects programmability and performance, as we explain in Section 2.3 and Section 2.4. Second, *state transformation cost*, which is incurred at runtime before the migration. This affects the overall performance of the migration. Finally, *compiler support*, which describes the changes made to the compiler in order not only to enable migration (i.e., generate code for both ISAs), but also control the granularity of migration and transformation costs. Unfortunately, trying to improve on two of the axes, means that compromise or more effort should be placed on the third one. For example, in order to maximize granularity but without modifying extensively the compiler, binary translation can be employed to a thread trying to migrate to another architecture but hasn't reached a valid migration point [36, 66]. However, this incurs extra overhead to the overall execution.

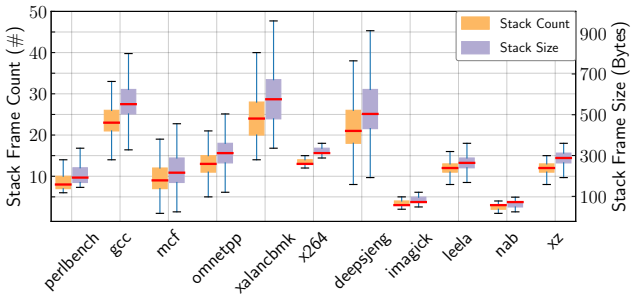
It is our belief, that in order to maximize programmability and performance in heterogeneous-ISA systems, more weight should be put in the compiler support, as described in the following sections.

2.3 The Programmability Problem

Utilizing offloading commonly requires modifications at the source code level of an application in order to use a specific supported application programming interface (API). This mainly involves i) the setup/teardown of communication with the accelerator device, and ii) code segmentation and data movement for the computation to be offloaded. This approach is inherently in tension with ease of programming, portability and programmer productivity [68].



(a) NAS Parallel Benchmarks (NPB) benchmarks.



(b) SPEC CPU2017 benchmarks.

Figure 2. Stack frame size and count distribution for benchmarks on ARM. Larger, more complex applications like those in SPEC CPU2017 also have higher frame sizes and frame counts and will benefit more from the elimination of stack transformation during migration by UNIFICO.

Figure 1 shows the main part of a small fragment of the breadth-first search (BFS) algorithm, as taken from [43]. Porting the algorithm to an offloading-based programming model, e.g., the recently introduced UPMEM PIM architecture [35], requires substantial modifications to the code and replicating the data structures among all processing units [6, 42]. However, with thread migration, there is no need of replicating data structures because threads can migrate where the data is, and UNIFICO also enables finer-grained migration, beyond just the function boundaries.

2.4 The State Transformation Overhead Problem

Related work shows that for a set of applications from the NPB, the cost of stack transformation is not significant [27]. However, since state transformation requires traversing all the stack frames for migrating between ISAs, overheads are proportional to the number of variables, the number of stack frames, and their sizes, therefore other applications might exhibit higher overheads. To this end, we examine the stack sizes and stack frame count of a wide range of high-performance computing (HPC) and compute-intensive benchmarks from the NPB [22] and SPEC CPU2017 [32] suites in Figure 2, showing their 5-point summary (min/max, lower/upper quartile, median values).

We observe that larger and more complex applications also have higher frame sizes and frame counts on average, e.g, up to 30 times more when comparing SPEC CPU2017 against

```
1  int i;
2  double x = 0, sum = 0;
3
4  for (i = 0; i < 16; ++i)
5    sum += hot_func(&x);
6
7  ...
8  // x also accessed here
9
```

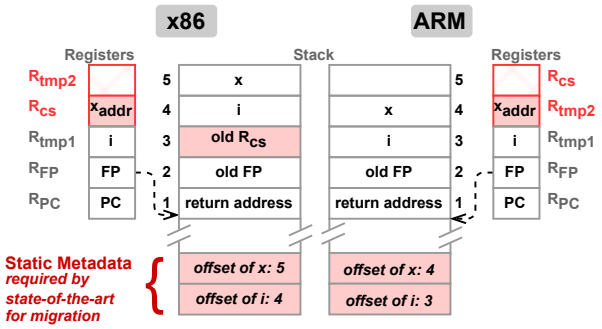
(a) Code snippet in C.

```
1  push Rcs // spill
2  load Rcs, xaddr // hoisted
3  Rtmp1 = 1
4  jmp end if Rtmp1 > 15
5 loop:
6  call hot_func() // arg: Rcs
7  Rtmp1++
8  jmp loop if Rtmp1 < 16
9 end:
```

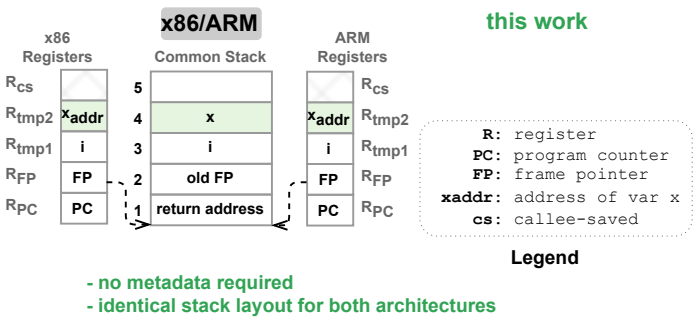
(b) x86 pseudo-assembly code for (a).

```
1  Rtmp1 = 1
2  jmp end if Rtmp1 > 15
3 loop:
4  // address recalculation
5  load Rtmp2, FP + 3
6  call hot_func() // arg: Rtmp2
7  Rtmp1++
8  jmp loop if Rtmp1 < 16
9 end:
```

(c) ARM pseudo-assembly code for (a).



(d) Unaligned stack layout for (a).



(e) UNIFICO aligned stack layout for (a).

Figure 3. Execution snapshot of the stack layout of a simple loop (a) on x86 (b) and ARM (c) ISAs just before a function call. Red highlighted areas in (d) show the differences and their required bookkeeping to enable migration using a state-of-the-art technique [27]. UNIFICO (this work) enforces a common layout (e) with minimal overhead for heterogeneous migration.

NPB applications. For example, XALANCBMK can have up to 40 stack frames with an average stack frame size of 500 bytes, amounting to a stack size of ~20KB. If we conservatively assume 8 byte variables, the runtime stack transformation will need to convert the state of 2500 variables. This is an order of magnitude larger when compared to the worst case of BT in NPB with only 6 frames, 120 bytes stack size, and 90 variables on average. Moreover, we see that there is a direct correspondence between the number of stack frames and the frame size. Lastly, frame count and size variability depend on the application and there is no correlation between programs even within the same benchmark suite. We conclude that larger applications can incur a higher overhead in execution time during migration. Finally, stack transformation also increases binary size due to additional metadata required (Section 6.2). Both costs are completely avoided by UNIFICO.

3 The Stack Layout Problem

Consider the top left-hand side of Figure 3 where a simple loop in C which accumulates the result of calls to function `hot_func` using as input local variable `x` passed by reference. We assume an ARM and x86 hardware setup which can facilitate migration and that `hot_func` represents a computationally intensive series of operations of a program which typically resides on the low-power ARM processor since it is not demanding outside of this function. Then, the function call boundary represents a natural point for migrating the long-running computation to the more powerful x86 processor which would significantly reduce the overall execution time of this program [29].

Despite its simplicity, this code snippet results in different stack layouts on x86 and ARM ISAs that would inhibit heterogeneous process migration. Figures 3(b) and 3(c) show, respectively, the simplified x86 and ARM pseudo-assembly generated for Figure 3(a) by the LLVM compiler backend code generators. The main difference between the two versions relates to the treatment of the pass-by-reference parameter. On x86, the compiler decides to hoist the load from memory operation outside the loop (Figure 3(b), line 2) since it is expensive to perform at every iteration, based on the compiler cost model for this ISA. This is done using a callee-saved register since the value of `x` needs to be preserved across calls to `hot_func`, which in turn forces its prior contents to be spilled on the stack (Figure 3(b), line 1). On ARM the compiler decides the opposite; the address is deemed cheap and hence kept in the loop and recalculated (i.e., rematerialized) using the frame pointer `FP` at every iteration (Figure 3(c), line 5).

Figure 3(d) shows a snapshot of the register file state and the top-most stack frame for each process when execution reaches the call to `hot_func`. It also shows the bookkeeping information required by the state-of-the-art POPCORN compiler [27] to perform heterogeneous migration. The main differences are highlighted in red. Comparing the two ISA execution states, we note the following:

i) the stack slot contents differ,
ii) the registers assigned to program variables differ, and
iii) the state-of-the-art Popcorn compiler [27] requires metadata embedded in the binaries to track differences for correct transformation during migration.

Popcorn Linux uses a runtime library to perform the transformation of the stack, guided by the metadata embedded in the binaries during their compilation.

Conversely, Unifico generates code that preserves the same stack layout across ISAs as seen in Figure 3(e). Hence, a binary compiled with Unifico does not require any metadata to account for the differences during heterogeneous migration. This obviates the stack transformation costs by significantly reducing binary size and state transformation overheads.

4 Unifico Design

Our goal is to provide the automatic generation of a unified memory stack layout that will simplify programming and allow faster heterogeneous migration without the associated overheads of metadata-dependent approaches. To this end, we have developed Unifico, a compiler backend technique which imposes a uniform stack layout between the targeted architectures and eliminates the need for stack transformation during migration.

We chose the 64-bit x86 and ARM ISAs as targets, being the most widespread architectures. Moreover, their primary application binary interface (ABI) properties partially overlap (e.g., alignment, register width, pointer size, endianness, etc.), simplifying an initial prototype.

A high-level overview of our approach is shown in Figure 4, where a modern modular compiler structure is assumed. The intermediate representation (IR) of the program is given as input to both compiler backends, in order to be lowered to the target assembly. By extending each of the backend passes, we mitigate the differences in the final stack layout. The code is lowered in every phase of the backend, so that the final binary (one for each architecture), when executed, will have the same stack layout for both architectures. Therefore, no metadata will be needed, as there will be no stack transformation.

The rest of the section presents the main design challenges of Unifico. First, we decompose the variations related to each architecture's ABIs. These are mostly straightforward to fix, since they are clearly documented as specifications and solving them first will also ease mitigating the rest of the differences. Then, we examine the differences that arise from the different instructions offered by each ISA. Finally, in every compiler implementation, certain decisions have gone into it that reflect tacit knowledge based on practice and experience, which is challenging to identify.

4.1 ABI Treatment

The first fundamental factor that causes stack layout mismatch is the register file. The number of registers is finite, so any values that the registers cannot hold need to be stored

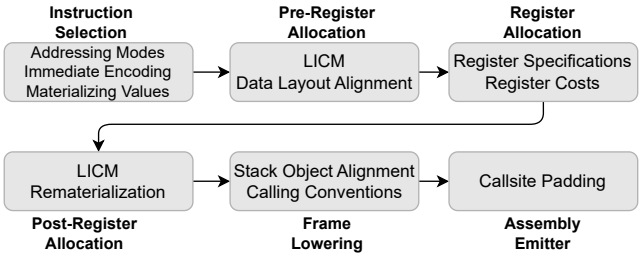


Figure 4. High-level design of Unifico. Each backend compilation stage shows the code generation passes that Unifico extends for both architectures.

(spilled) on the stack. Also, if a register is *callee-saved*, it needs to be pushed to the stack before being used by a function, affecting the stack layout (contrary to a *caller-saved* or *temporary*).

One of Unifico's guiding principles is to use the same number of registers between the architectures, while mapping the registers to the same functionality (we assume that such mapping exists, e.g., if one architecture supports 256-bit vector registers, the other one should support them too). However, removing registers that have a dedicated purpose in the ISA is challenging. For instance, in ARM, the registers `r16` and `r17` can be used to support long branches, and the register `r18` can be used to carry inter-procedural state (e.g., the thread context) [11]. Therefore, we give priority to those registers and keep them in the ISA, whereas we can freely remove others with no special use, e.g., simple temporary registers.

For the calling conventions [11, 53], we need to maintain: i) the same number of registers for function arguments since the extra arguments are passed through the stack, ii) the same return value registers, and iii) match the callee-saved registers which are also pushed on the stack. We apply this to both general-purpose and floating-point (FP) registers.

Table 1 shows the register sets of x86 and ARM after our mapping. Overall, we reduced the number of the ARM general-purpose and FP registers from 32 to 16, per category, which is compatible with x86, and modified the usage of a series of registers. For the callee-saved registers, we also ensured that they are saved in the same order on the stack by the called functions.

Moreover, if a register has a special use in one architecture (e.g., `r16`–`r18` in ARM), we preserve it and map the register to its counterpart in the other architecture, as long as there are no incompatibilities between the registers (e.g., if they both need to be reserved for a different purpose).

4.2 Instruction Treatment

Another contributing factor to stack layout mismatch stems from the instructions offered by each architecture. For example, loading a constant from memory in x86 and ARM requires different number of instructions (Figure 5). x86 calculates the address of the constant and loads the value from memory in one instruction. On the other hand, ARM calculates the address using a separate instruction and then loads the value

Table 1. UNIFICO's mapping between x86 and ARM register files which ensures a low execution-time impact. The table shows the callee-saved, the return value, the parameter passing, and the temporary registers. This is both for general-purpose and FP/vector registers.

Registers		Usage
x86	**ARM**	
Callee-saved		
`rsp`	`SP`	Stack pointer
–	`r30`	Link register
`rbp`	`r29`	Frame pointer
`rbx, r15`	`r19, r20`	General purpose
Caller-saved		
`rax, rdx`	`r8, r2`	Return
`rdi`	`r0`	Arg #1/return
`rsi,rdx,rcx,r8,r9`	`r1–r5`	Args #2–#6
`r10–r14`	`r6, r7, r16–r18`	Temp registers
`xmm0–xmm1`	`v0–v1`	FP args/return
`xmm2–xmm7`	`v2–v7`	FP args
`xmm8–xmm15`	`v8–v15`	Temp FP registers

```
1 R0 = constant          R0 = addressof(constant)
2                        R1 = load R0
```

(a) **(b)**

Figure 5. Pseudo-assembly for loading a constant value on x86 (left) and ARM (right). ARM requires an additional instruction for the same high-level operation.

in a subsequent instruction. Therefore, ARM requires an additional register and may spill one more value to the stack compared to x86.In addition, ARM uses two instructions to complete this behavior, which affects the register allocation regions and, hence, indirectly the stack layout.

These differences are tightly coupled with two important compiler problems: register allocation and instruction selection. The register allocator determines which values reside in registers, and which registers will hold these values. For different architectures, the register allocator might utilize a different number of registers for a specific operation, while keeping other values in memory. The instruction selection lowers the IR code into machine instructions. For each architecture, the instruction selection may choose different instructions for the same high-level operation, depending on the capabilities of the architecture, or implementation-specific decisions (Sections 4.3 and 5).

4.3 Compiler Backend Treatment

Finally, independently of the ABI or the instructions of an ISA, there are some differences caused by the manner the code is generated and optimized in the compiler. As shown in Figure 3, the code motion caused by the compiler reflects specific implementation decisions in the backend, affecting the stack layout even when the ABI differences between ISAs have been bridged.

These differences comprise alignment decisions (apart from the ABI specifications), register allocation decisions, and a series of optimizations, like rematerialization [31] and code motion (Section 5). We make up for the differences described in the last two sections by modifying the instruction selection (to select instructions with similar behavior), the register allocation, and other phases, like the optimizations of constants.

Table 2. UNIFICO's code generation extensions to the ARM and x86 backends maintaining a unified stack layout.

Category & Description	x86	ARM
Alignment (§ 5.1.1)		
Align symbols in code & data sections	✔	✔
Align return addresses after call sites	✔	✔
Allocate emergency spill slot	✔	✔
Align local stack objects to at least 4 bytes	✔	
Align callee-saved registers to 8 bytes		✔
Addressing Modes (§ 5.1.2)		
Do not encode complex addressing	✔	
Match legal address immediates	✔	✔
Immediate Encoding (§ 5.1.3)		
Do not encode immediates in multiplication	✔	
Encode same immediates for data-processing	✔	
Do not materialize non-zero FP constants		✔
Register Allocation (§ 5.1.4)		
Match register cost and allocation order	✔	✔
Hold the zero constant in temp registers	✔	
Optimized two-address format for integer instructions		✔
Match instruction input/output operand size	✔	✔
Rematerialization & Code Motion (§ 5.1.5)		
Rematerialize local variable loads		✔
Do not rematerialize `movss/movsd`	✔	
Rematerialize `lea`	✔	
Rematerialize `adrp`		✔
Reuse constants instead of rematerializing	✔	
Other Optimizations		
Disable heuristic for frame object ordering	✔	
Match optimization of special constants	✔	
Lower conditional *select* similarly	✔	✔
Vectorize pairs of *double*	✔	

5 Implementation

5.1 The Unifico Backend

We implemented Unifico by extending the ARM and x86 backends of LLVM. Having demonstrated how code generation can affect the stack layout, Table 2 enumerates how we implemented Unifico based on the LLVM backend infrastructure, in order to achieve a unified address space layout and application state. We have grouped the various backend parts based on their functionality and describe them in the following sections.

5.1.1 Alignment

Symbol alignment. Similarly to related work [27, 36], the symbols of the program, namely the functions and the data, need to lie on the same virtual addresses for both architectures. This way, accesses to global data will be consistent between the architectures, and the functions will be aliased to the same address, which is necessary when copying the memory images between ISAs. Therefore, we align the symbols in the code and data sections, by having one symbol per section and by adding padding between these sections during linking. The result is shown for `main` in the first lines in the snippets of Figures 6(a) and 6(b), which are all aligned to the address `1000`.

Call site alignment. When calling a function, the address of the instruction after the call is pushed to the stack (at least for the calling conventions in question). Typically, the return addresses pushed will differ for the two architectures, so if a migration happens inside a function the destination processor will return to a wrong address after the function returns. As shown in Figures 6(a) and 6(b), in the original version of the assembly, the instructions that follow the call to `hot_func` differ by three bytes (offset `1104` vs offset `1107`).

To address this issue, we align the call sites like we did with the program symbols, using `nop` instructions. Before the call instructions, we emit `nop` instructions to pad the return addresses (Figures 6(c) and 6(d)).

Stack object alignment. We ameliorate the alignment differences for the objects in the stack of the two architectures. These include the use of an emergency spill slot and the alignment of local stack objects and callee-saved registers, as listed in Table 2. To elaborate on the emergency spill slot, the ARM backend will scavenge an extra register in case it needs to materialize large stack offsets (i.e., more than 255 bytes), which do not fit in one instruction. If a register cannot be found, a special spill slot is reserved. For simplicity, Unifico conservatively reserves this slot and places it after the callee-saved registers for both architectures.

5.1.2 Addressing Modes. ARM does not support addressing modes of the form *[base + scaled register + offset]* for indexing arrays. When the x86 backend uses this mode, the ARM backend needs to reserve an extra register for the same operation. Usually, the ARM backend keeps the address of the array, i.e., *[base + offset]*, in a separate callee-saved register, to be able to reuse it for indexing the array multiple times. This usage of one extra callee-saved register may introduce extra spills, hence, we disable the former complex addressing mode in x86, to get the same behaviour in ARM.

```
1 1000: main:
2 1000: SP = SP - 32
3 ...
4 1100: call hot_func()
5 1104: // after call
```
(a)

```
1000: main:
1000: push FP
...
1102: call hot_func()
1107: // after call
```
(b)

```
1 1000: main:
2 1000: SP = SP - 32
3 ...
4 1100: four_byte_nop
5 1104: call hot_func()
6 1108: // after call
```
(c)

```
1000: main:
1000: push FP
...
1102: one_byte_nop
1103: call hot_func()
1108: // after call
```
(d)

Figure 6. Call site alignment in pseudo-assembly before (top) and after (bottom) Unifico's operation for ARM (left) and x86 (right). The `main` symbol is placed at the same address (offset `1000`) for both ISAs. Highlighted lines in top figures show the difference in return addresses, while in bottom figures show the code emitted after Unifico adds `nop` instructions for padding, resulting in same return address.

5.1.3 Immediate Encoding. The two ISAs do not support the same set of immediates. Due to smaller instruction size, ARM instructions can encode explicitly up to 21 bits of immediates for pc-relative addressing [4], 12 bits (with an optional shift of 12) for arithmetic operations [55], and up to 64-bit logical immediates [45]. Finally, the ISA allows moving up to 16-bit immediates, optionally shifted, to registers. Unifico keeps the same immediate encoding in x86.

5.1.4 Register Allocation. To limit possible overheads, we have kept the default *greedy* register allocator [10], which uses global live range splitting, minimizing the cost of spilled code. However, we need to make sure that the allocator will take the same decisions when assigning registers for the two backends, despite the heuristics it uses internally. Expanding on the first two corresponding entries of the table, we first assign the same order of preference to the registers. For example, `r15` is more expensive on x86 than `rbx`, due to encoding reasons, so we do the same for ARM's `x20` and `x19`, even though their cost is the same, to achieve similar allocation. Also, since ARM has a dedicated zero register that is not callee-saved, we avoid using callee-saved registers for x86 to hold the zero constant, as this would require extra spills in memory.

5.1.5 Rematerialization and Code Motion. Most of the changes in this category are related to how the compiler generates code for getting the values or the addresses of variables. As we showed in Section 3 and break down in Figure 7, we instruct the compiler to rematerialize the `load` instructions,

```
1  push Rcs
2  load Rcs, xaddr
3  Rtmp1 = 1
4  jmp end if Rtmp1 > 15
5 loop:
6
7  // arg: Rcs
8  call hot_func()
9  jmp loop if Rtmp1 < 16
```

(a)

```
3  Rtmp1 = 1
4  jmp end if Rtmp1 > 15
5 loop:
6  load Rtmp2, xaddr
7  // arg: Rtmp2
8  call hot_func()
9  jmp loop if Rtmp1 < 16
```

(b)

Figure 7. x86 pseudo-assembly from Figure 3 *with* (left) and *without* (right) rematerialization of `&x` using `load`. In (a) the address calculation is hoisted out of the loop (highlighted), occupying a callee-saved register whose contents are spilled to the stack. In (b), UNIFICO does not hoist the calculation, keeping the stacks among ISAs aligned.

Table 3. Accessing values (`x`) and references (`&x`) of variables per architecture.

	Scope	Access x	Access &x
ARM	*Local*	simple load	simple add/sub
	Global	`adrp` + load	`adrp`
x86	*Local*	simple load	`lea`
	Global	rip-relative `mov`	rip-relative `lea`

leading to the code in Figure 7(b), where `load` is kept inside the loop without the need for a callee-saved register. This way the address is calculated at every iteration (line 6) and passed to the first argument register (line 7). In general, there are eight different cases (Table 3).

Getting values of variables. Getting the value of local variables is done similarly in both architectures, with simple loads from the stack frame. For the case of global variables, x86 needs only one instruction (a `rip`-relative `mov`), whereas ARM requires two instructions (one to calculate the global address and one to load from it. Since LLVM cannot currently rematerialize multiple instructions, when ARM needs to reuse a global variable it will spill it in the stack, while for x86 it suffices to recompute the `rip`-relative `mov`. Regardless, we ensure that x86 will also spill the value in this case.

Getting references of variables. Getting the reference (i.e., address) of a local variable is done in ARM using simple arithmetic (`add`/`sub`) with the frame pointer, as we showed in Figure 3, whereas in x86 using `lea` instructions (termed `load` in our pseudo-assembly) in Figures 3 and 7. The `add`/`sub` instructions are usually not hoisted out of loops by the compiler, whereas the `lea` instruction is, since it is a little more expensive in the general case [2, 40]. Hoisting the instruction may consume a callee-saved register so, instead, our compiler rematerializes these instructions if it needs to reuse their result. Since we are not hoisting `lea` instructions, however, we need to do the same for the `adrp` instructions (the ARM counterpart), to also have an aligned behavior for the case of global variables (`adrp` vs rip-relative `lea`). Therefore, we additionally rematerialize `adrp` instructions.

Reusing constants. When the same constant is used multiple times in ARM, the compiler tries to place its value in a register. Instead, x86 tries to materialize the constant again by encoding it separately for every instruction. UNIFICO emulates the first behavior for x86.

5.1.6 Verifying Stack Layout.

In order to verify that the stack layout between an x86 and an ARM binary is the same and, hence, that migration will be successful, we use the LLVM StackMaps [17] to track the location of values in the stack. These are inserted in a separate Executable and Linkable Format (ELF) section, which is cross-checked after compilation for both binaries and then discarded. Compilation correctness is ensured by the LLVM backend tests, after they are ported to the new UNIFICO changes.

5.2 Other Considerations and Limitations

Libraries. First, we currently only support static linking for the libraries, to simplify the engineering effort required to align all the symbols, and to create consistent memory images of the binaries between the machines.

Also, we are not migrating inside library calls, so they do not need a consistent layout. However, some functions in the MUSL LIBC library (e.g., `setjmp`/`longjmp`) use inline assembly when handling signals/exceptions. In this case, we modified their inline assembly to obey our new ABIs for binary compatibility.

Interaction with the OS. We assume a replicated-kernel OS, where there is a kernel per core and each kernel loads the address space of the respective binary [27]. The address space of each binary has an identical layout, but the `.text` is natively compiled, and then aligned (Section 5.1.1), so each function in the two ISAs will have the same virtual address. The OS-specific details (e.g., page mapping, process scheduling, view of the OS by different processors, etc.) are out of scope and can be found in related work [28].

Backend and optimization flags. Regarding the backend infrastructure, we are not using the machine instruction scheduler since it can lead to instruction reorderings that invert the order of spilling. Identifying a good compromise between allowing machine scheduling and getting a predictable order of spilled values is left for future work.

Moreover, although the ARMv8 architecture is bi-endian, we keep the (default) little-endian setting to match with x86. The technical effort to support different endianness is outside the scope of this work.

Finally, migration is supported currently for up to the `-O1` flag. An open issue in the LLVM code generator[1] makes the allocator run out of registers for most of the NPB in `-O2`/`-O3`, when using stackmaps. Therefore, we cannot verify UNIFICO for the full suite in `-O2` or higher, although we are working on

[1] https://github.com/llvm/llvm-project/issues/56880

an LLVM patch to fix that. However, since the LLVM backend enables optimizations for any optimization level other than `-O0`, having supported `-O1` should cover from the outset many programs compiled with higher flags.

Applicability to other architectures. Even though the details described in this section are specific to x86 and ARM, our technique provides a blueprint and the high-level ideas for supporting other combinations of general-purpose processors (e.g., x86 and RISC-V). Much of the target-specific details, e.g., unifying the ABIs, the instruction formats, the register allocation costs, the rematerialization properties of values, etc., are usually encoded easily as a backend specification, e.g., in LLVM's TableGen [18] or GCC's Machine Description [15]. More elaborate implementation, e.g., which address immediates are legal, can be guided using the insights gleaned from this section and our OSS artifact. For target features not covered in this work, these can be detected through the stackmap machinery, e.g., a constant optimized specifically in one architecture will appear as a different architecture-specific value (or values) in the stackmaps.

Multithreading and memory consistency. Our evaluation is for a single thread, however Unifico's design is orthogonal to multithreaded execution. Since Unifico is based on a unified address space between the architectures, and the evaluated architectures support different memory consistency models [34, 56], an inter-device coherence protocol is assumed (e.g., CXL [7]), along with a fused memory model (e.g., compound models [41]) between the two architectures. In our setup, we assume that upon migration all threads are stopped and buffers/caches are flushed to memory, therefore migration points act as memory barriers. Investigating multithreaded execution for Unifico is left as future work.

5.3 Migration

There is no platform available today with x86 and ARM that share memory. Therefore, to validate migration under our approach, we are using CRIU [8] for a prototype. CRIU offers a checkpoint-restore mechanism in user space, by dumping a multi-file image of the application when pausing it, and restoring the state later, continuing the execution.

However, in our prototype, we are leveraging multi-ISA binaries, so we dump the state of the binary in the starting processor, rewrite the necessary CRIU images to be consistent with the target machine, transfer the images (via SSH), and continue execution of the other binary by restoring the rewritten state.

Contrary to related work [26, 27, 65], we do not transform the state at all, but only rewrite images like *core* [8], which contain core process and architecture-specific state information, e.g., the registers. For example, based on our mapping in Table 1, we need to simply copy the value of `rdi` to `r0`, if we are migrating from x86 to ARM. Rewriting happens through a script invoked by a simple runtime, but in our ideal use-case scenario, e.g., an x86/ARM machine with coherent shared memory, this is facilitated by a user- or kernel-space service without the need to transfer the state between the machines.

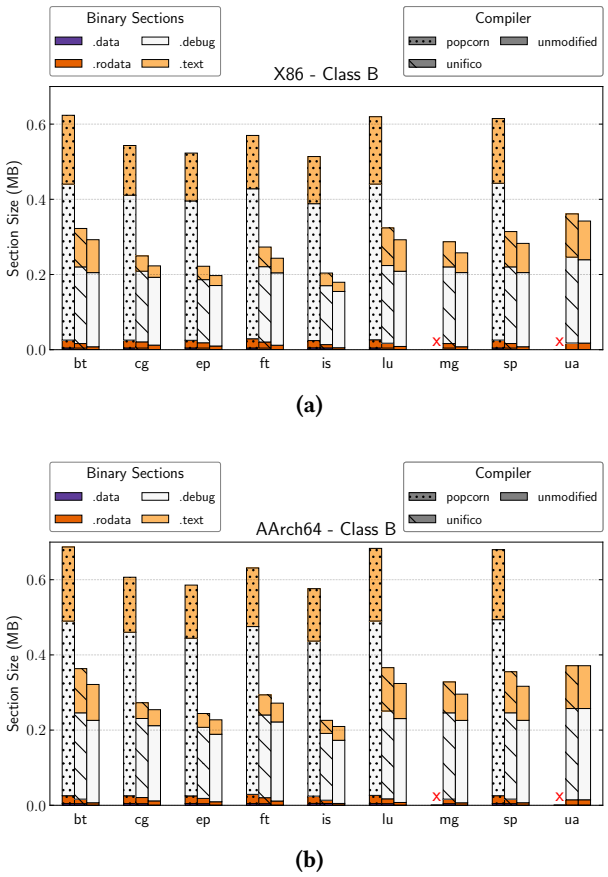


Figure 8. Size breakdown of binaries and their contents generated with static linking (unmodified LLVM), the Popcorn compiler, and Unifico, for x86 (top) and ARM (bottom) using the NPB suite. Unlike the Popcorn compiler, Unifico requires no metadata and its size overhead is minimal, within 10% of statically linked binaries generated with LLVM.

Virtual address consistency. We take some further steps to keep the memory mappings between the architectures consistent. First, we disable address space layout randomization (ASLR) on both machines, otherwise the start address of the stack will be different between the architectures, thus breaking the match of the stack layouts. We do this for simplicity and leave the effort of assigning the same random stack start address to both architectures as a future extension. Furthermore, by default, x86 programs map their stack to addresses starting from `0x800000000000`, whereas in ARM, frames start from the address `0xffffffffffff` [11, 53]. We modify the startup function from the C library, so that the frame of the `main` function starts from `0x800000000000` in both machines.

6 Empirical Evaluation

6.1 Setup

Hardware. We use two hardware configurations for evaluation: i) (ARM) GIGABYTE R181-T92-00 (SABER SKU), Dual Cavium ThunderX2 © CPU CN9980 v2.2 2.20GHz (32 cores/128 threads), and ii) (x86) Dell PowerEdge R440, Intel © Xeon © Silver 4110 CPU 2.10GHz (8 cores/16 threads).

Software. UNIFICO extends LLVM version 9.0.1, consisting of ~3000 lines of code (LoC) (1400 and 800 LoC for the x86 and AARCH64 backends respectively). The rest of the code relates to target-independent functionality, including a few changes on CLANG version 9.0.1. We reuse the modified linker and symbol aligner from the POPCORN compiler [16] (commit `4cc8805`). Regarding the migration mechanism, we prototype a method using HETCRIU[67], an extension of CRIU[14] at version 3.17.1. Finally, we employ a modified version of MUSL LIBC version 1.1.22, which we statically link to the binaries and release with our LLVM modifications.

Benchmarks. We focus on compute- and memory-intensive C benchmarks to explore the impact of UNIFICO's code modifications which may have been hidden otherwise (e.g., by system calls, I/O operations, etc.). In our experiments, we used a C implementation [58] of the NPB suite [22] (input classes A, B, C). All benchmarks are compiled with the `-O1` flag. For the class C of the FT and MG benchmarks, the enormous amount of static data declared lead to a relocation overflow error in ARM (hence we can't compile also the aligned x86 binary), so we omit these results. This is under investigation (e.g., using different code models), but does not affect our overall exploration in this section.

We will examine all the benchmarks for the binary sizes and the architectural overhead, and for direct comparison with the POPCORN LINUX compiler toolchain, we migrate the same subset as in previous state-of-the-art work [27]. However, although POPCORN and its compiler are open-source, we were unable to set up any execution migration functionality.

6.2 Size Comparison

We explore the sizes for binaries compiled with the following different methods: i) an unmodified CLANG/LLVM compiler (statically linked with the C library), ii) the POPCORN compiler toolchain (statically linked with the C library), and iii) UNIFICO (statically linked with the C library). We include the unmodified compiler in our evaluation to show the impact of our modifications to the code section size. These are shown in Figure 8[2]. The binaries are for the class B of NPB, but the trends are similar for the other classes.

We make three observations. First, in all the compilation categories, the binaries of x86 are smaller than the binaries of ARM. This can be attributed to the x86 being a complex instruction set computer (CISC) ISA, encoding more complex instructions that can lead to compact binaries. Second, in both architectures, the POPCORN binaries are up to 2*x* the size of the statically compiled binaries of the unmodified compiler. This is caused by additional metadata, debug information, as well as libraries (in the `.text` section) that are used to facilitate migration and state transformation. Finally, we observe that in both architectures UNIFICO leads to much smaller binaries relatively to POPCORN, and within 10% of the unmodified compiler. The size of the UNIFICO binaries is slightly larger than those compiled with the standard CLANG due to the constraints we imposed to the code generation, visible in the code and data sections. Overall, we reduced the binary size overhead from ~200% to ~10%.

[2] MG and UA cannot be compiled by POPCORN in `-O1` due a register allocation issue with the stackmaps, but this does not affect the overall analysis.

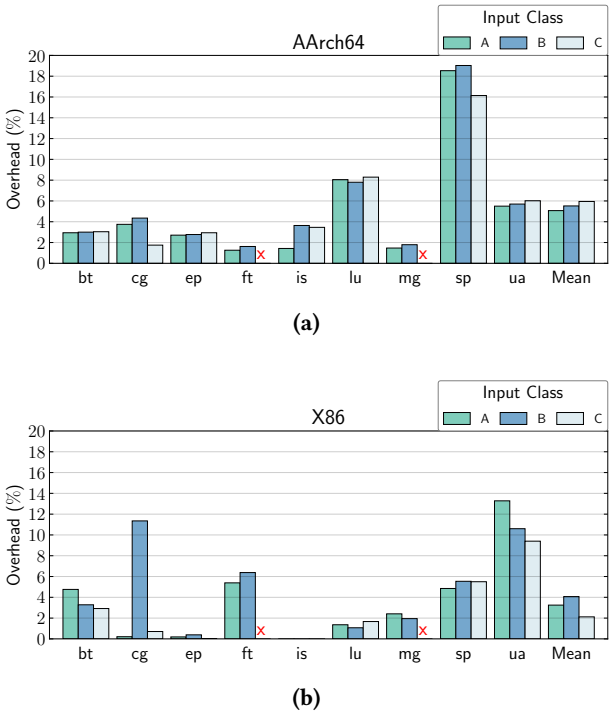


Figure 9. Overall execution overhead of NPB benchmarks compiled with UNIFICO over unmodified code for input classes A, B and C on ARM (top) and x86 (bottom). UNIFICO has on average low impact on the execution time of applications on both ISAs.

6.3 Impact per Architecture

For each architecture, we evaluate the impact of UNIFICO, relative to the unmodified compilation. We run each benchmark three times and all standard deviations were below 1%. The results are shown as overhead percentages in Figure 9.

In the case of ARM, the performance has not noticeably degraded, except for the cases of LU and SP. For SP, which has the most overhead, the main code region that dominates the runtime is a kernel with a deep-nested loop, manipulating 4-dimensional global arrays. Materializing the global addresses and getting the values of all elements before the calculations,

introduces more loads inside the inner loops which increase the overhead.

In the case of x86, almost all benchmarks compiled with Unifico demonstrate less than 5% difference from the unmodified compiler. The most noticeable outlier is cg for class B, where a close to 12% overhead is observed. The code region with the largest bottleneck is a sparse matrix computation in the conjugate gradient function, with indirect (because of the sparse representation) global array accesses inside nested loops. The simplified addressing in x86, with less available immediates to represent global addresses, cause greater impact in this case (similarly for ua). Interestingly, for the class C of cg, our investigation concludes that, for the much larger array sizes, the computation dominates again these less efficient memory accesses, utilizing fully all the cache levels.

Discussion. Further tuning cross-architectural tradeoffs is left as future work. For example, instead of imposing the two-address format for FP instructions in ARM, which was needed because the architectures had misaligned spill slots with FP values, we saw that padding the spill slots after the register allocation was sufficient, and caused much less overhead for ARM. Nevertheless, we envision end-to-end benefits at the system level enabled by finer migration granularity with no metadata and state transformation. Finally, if an overhead is not acceptable, techniques like Multi-Variant ELF (MELF) binaries [63] allow the runtime to pick the right compile-time variant of a function, exploiting the tradeoff between migration granularity and performance impact.

7 Related Work

Dynamic software migration. Migration among heterogeneous-ISA compute nodes during execution was initially explored in the context of distributed systems [21, 39, 47, 60]. DeVuyst et al. [36], were the first to examine the feasibility of heterogeneous-ISA migration for multicore CPUs, where state transformation dominates the migration overhead. Venkat and Tullsen [65] showed that ISA heterogeneity offers ~20% performance and ~23% power improvements over the homogeneous-ISA counterpart. Both works rely on the same compilation techniques, aiming to minimize the state transformation by providing a common data layout as much as possible and using a common "fat" binary format for the target architectures. The evaluation is done through simulation, but artifacts are not publicly available. The Popcorn Linux and compiler toolchain [27, 28], improve upon these techniques and provide an open source implementation. The Popcorn compiler toolchain [27] avoids "fat" binaries by embedding metadata in them to guide the transformation during migration. Later works based on Popcorn Linux, examined migration between heterogeneous-ISA systems and FPGAs[44], and at higher granularity among sets of processes in Linux containers [67]. Unifico, as a compilation technique, can be integrated in any of the aforementioned approaches.

Migration granularity. Von Bank et al. [66] provide a formal methodology to identify migration points among heterogeneous processes at varied granularity. A large body of work, along with Unifico, which operate at the process level, use call site boundaries as potential migration points [27, 28, 36, 65]. DeVuyst and Venkat [36, 65] propose to allow migration requests at any machine instruction by using dynamic binary translation (DBT) to fulfill them till a call site is reached, upon which actual migration occurs. However, the DBT overheads have shown to be excessively high [27]. Checkpointing has an affinity with process migration [8, 57], though HetCRIU [67] blurs this distinction.

8 Conclusion

We propose Unifico, a compilation technique that extends and innovates upon existing compiler support for heterogeneous-ISA CPU migration, by removing the need for runtime state transformation. Unifico removes state transformation overheads during migration, without creating large binaries and easing programmability, by extending the compiler backend to generate binaries for different architectures with a unique address space and stack layout. We show that Unifico does not substantially impact program execution, adding on average no more than 6% execution time overheads and no more than 10% code size increases, compared to the $2x$ size overhead introduced by related work. In future work, we will examine the applicability of our approach to other architectures, the automatic extraction of unification rules for the ABI and stack layout, as well their encoding to the high-level compiler specifications to ease adoption.

9 Data-Availability Statement

Our artifact is publicly available and can be found on Zenodo [19].

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and the artifact evaluation.

References

[1] 2012. APU 101: All about AMD Fusion Accelerated Processing Units. https://developer.amd.com/wordpress/media/2012/10/apu101.pdf
[2] 2014. Cortex-A7 Instruction Cycle Timings. https://hardwarebug.org/2014/05/15/cortex-a7-instruction-cycle-timings/
[3] 2020. Broadcom Stingray PS225 SmartNIC.
[4] 2022. Arm Architecture Reference Manual for A-profile Architecture.
[5] 2022. CCIX. https://www.ccixconsortium.com/
[6] 2022. CMU-SAFARI/prim-benchmarks/BFS. https://github.com/CMU-SAFARI/prim-benchmarks/tree/main/BFS
[7] 2022. Compute Express Link. https://www.computeexpresslink.org
[8] 2022. Criu Homepage. https://criu.org/
[9] 2022. Gen-Z Consortium: Computer Industry Alliance Revolutionizing Data Access. https://genzconsortium.org/
[10] 2022. The LLVM Target-Independent Code Generator. https://llvm.org/docs/CodeGenerator.html
[11] 2022. Procedure Call Standard for the Arm® 64-Bit Architecture (AArch64).

[12] 2022. SK hynix Introduces Industry's First CXL-based Computational Memory Solution (CMS) at the OCP Global Summit. https://news.skhynix.com/sk-hynix-introduces-industrys-first-cxl-based-cms-at-the-ocp-global-summit/

[13] 2022. What Is OpenCAPI? https://opencapi.org/about/

[14] 2023. Checkpoint-Restore/Criu. https://github.com/checkpoint-restore/criu

[15] 2023. Machine Descriptions. https://gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html

[16] 2023. Ssrg-vt/Popcorn-Compiler. Systems Software Research Group @ Virginia Tech. https://github.com/ssrg-vt/popcorn-compiler

[17] 2023. Stack maps and patch points in LLVM. https://llvm.org/docs/StackMaps.html

[18] 2023. TableGen Overview. https://llvm.org/docs/TableGen/

[19] 2024. *UNIFICO: Thread Migration in Heterogeneous-ISA CPUs without State Transformation.* Zenodo. https://doi.org/10.5281/zenodo.10567311

[20] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *Data Management on New Hardware (DaMoN'22)*. Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3533737.3535090

[21] G. Attardi, I. Filotti, and J. Marks. 1988. Techniques for Dynamic Software Migration. In *In ESPRIT '88: Proceedings of the 5th Annual ESPRIT Conference*. NorthHolland, 475–491.

[22] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[23] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, Kunal Gulati, Luca Benini, and David Wentzlaff. 2020. BYOC: A "Bring Your Own Core" Framework for Heterogeneous-ISA Research. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 699–714. https://doi.org/10.1145/3373376.3378479

[24] Antonio Barbalace and Jaeyoung Do. 2021. Computational Storage: Where Are We Today?. In *Conference on Innovative Data Systems Research 2020*.

[25] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. 2017. It's Time to Think About an Operating System for Near Data Processing Architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 56–61. https://doi.org/10.1145/3102980.3102990

[26] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. 2020. Edge Computing: The Case for Heterogeneous-ISA Container Migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*. Association for Computing Machinery, New York, NY, USA, 73–87. https://doi.org/10.1145/3381052.3381321

[27] 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)*. Association for Computing Machinery, New York, NY, USA, 645–659. https://doi.org/10.1145/3037697.3037738

[28] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/2741948.2741962

[29] Sharath K. Bhat, Ajithchandra Saya, Hemedra K. Rawat, Antonio Barbalace, and Binoy Ravindran. 2015. Harnessing Energy Efficiency of Heterogeneous-ISA Platforms. In *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower '15)*. Association for Computing Machinery, New York, NY, USA, 6–10. https://doi.org/10.1145/2818613.2818747

[30] Christopher Blackburn, Xiaoguang Wang, and Binoy Ravindran. 2022. Rave: A Modular and Extensible Framework for Program State Re-Randomization. In *Proceedings of the 9th ACM Workshop on Moving Target Defense* (Los Angeles, CA, USA) *(MTD'22)*. Association for Computing Machinery, New York, NY, USA, 3–10. https://doi.org/10.1145/3560828.3564008

[31] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1992. Rematerialization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. Association for Computing Machinery, New York, NY, USA, 311–321. https://doi.org/10.1145/143095.143143

[32] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*. ACM, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[33] Shenghsun Cho, Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. 2020. Flick: Fast and lightweight ISA-crossing call for heterogeneous-ISA environments. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 187–198. https://doi.org/10.1109/ISCA45697.2020.00026

[34] Nathan Chong and Samin Ishtiaq. 2008. Reasoning about the ARM weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. 16–19. https://doi.org/10.1145/1353522.1353528

[35] Fabrice Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2019.8875680

[36] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. 2012. Execution Migration in a Heterogeneous-ISA Chip Multiprocessor. *SIGPLAN Not.* 47, 4 (March 2012), 261–272. https://doi.org/10.1145/2248487.2151004

[37] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beránek, Luca Benini, and Torsten Hoefler. 2021. A RISC-V in-network accelerator for flexible high-performance low-power packet processing. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 958–971. https://doi.org/10.1109/ISCA52012.2021.00079

[38] Kevin Drucker, Dharmesh Jani, Ishwar Agarwal, Gary Miller, Millind Mittal, Robert Wang, and Bapiraju Vinnakota. 2020. The Open Domain-Specific Architecture. In *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, Piscataway, NJ, USA, 25–32. https://doi.org/10.1109/HOTI51249.2020.00019

[39] F. B. Dubach and C. M. Shub. 1989. Process-Originated Migration in a Heterogeneous Environment. In *Proceedings of the 17th Conference on ACM Annual Computer Science Conference (CSC '89)*. Association for Computing Machinery, New York, NY, USA, 98–102. https://doi.org/10.1145/75427.75437

[40] Agner Fog. 2022. Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD, and

VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf

[41] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1145–1168. https://doi.org/10.1145/3591267

[42] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. https://doi.org/10.48550/arXiv.2105.03814 arXiv:arXiv:2105.03814

[43] Eric Robert Hein. 2018. *Near-data processing for dynamic graph analytics.* Ph. D. Dissertation. Georgia Institute of Technology. http://hdl.handle.net/1853/60228

[44] Edson Horta, Ho-Ren Chuang, Naarayanan Rao VSathish, Cesar Philippidis, Antonio Barbalace, Pierre Olivier, and Binoy Ravindran. 2021. XarTrek: Run-Time Execution Migration among FPGAs and Heterogeneous-ISA CPUs. In *Proceedings of the 22nd International Middleware Conference (Middleware '21).* Association for Computing Machinery, New York, NY, USA, 104–118. https://doi.org/10.1145/3464298.3493388

[45] Dominik Inführ. 2017. Encoding of Immediate Values on AArch64. https://dinfuehr.github.io/blog/encoding-of-immediate-values-on-aarch64/

[46] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17).* Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[47] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. 1988. Fine-Grained Mobility in the Emerald System. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 109–133. https://doi.org/10.1145/35037.42182

[48] Nam Sung Kim, Deming Chen, Jinjun Xiong, and Wen-mei W. Hwu. 2017. Heterogeneous Computing Meets Near-Memory Acceleration and High-Level Synthesis in the Post-Moore Era. *IEEE Micro* 37, 4 (2017), 10–18. https://doi.org/10.1109/MM.2017.3211105

[49] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. 2005. Heterogeneous Chip Multiprocessors. *Computer* 38, 11 (Nov. 2005), 32–38. https://doi.org/10.1109/MC.2005.379

[50] George Kyriazis. 2012. *Heterogeneous System Architecture: A Technical Review.* Technical Report 1.0. AMD. 18 pages. https://developer.amd.com/wordpress/media/2012/10/hsa10.pdf

[51] Wooseok Lee, Dam Sunwoo, Christopher D. Emmons, Andreas Gerstlauer, and Lizy K. John. 2017. Exploring Heterogeneous-ISA Core Architectures for High-Performance and Energy-Efficient Mobile SoCs. In *Proceedings of the on Great Lakes Symposium on VLSI 2017 (GLSVLSI '17).* Association for Computing Machinery, New York, NY, USA, 419–422. https://doi.org/10.1145/3060403.3060408

[52] Jianshen Liu, Carlos Maltzahn, Craig Ulmer, and Matthew Leon Curry. 2021. Performance Characteristics of the BlueField-2 SmartNIC. https://doi.org/10.48550/arXiv.2105.06619 arXiv:arXiv:2105.06619

[53] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2021. System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models).

[54] Robert Lyerly, Alastair Murray, Antonio Barbalace, and Binoy Ravindran. 2018. AIRA: A Framework for Flexible Compute Kernel Execution in Heterogeneous Platforms. *IEEE Transactions on Parallel and Distributed Systems* 29, 2 (2018), 269–282. https://doi.org/10.1109/TPDS.2017.2761748

[55] Alisdair McDiarmid. 2014. ARM Immediate Value Encoding. https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/

[56] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22.* Springer, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

[57] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1995. Libckpt: Transparent Checkpointing under UNIX. In *USENIX 1995 Technical Conference (USENIX 1995 Technical Conference).* USENIX Association, New Orleans, LA. https://www.usenix.org/conference/usenix-1995-technical-conference/libckpt-transparent-checkpointing-under-unix

[58] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC).* 137–148. https://doi.org/10.1109/IISWC.2011.6114174

[59] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euiseok Kim, and Kyoung Park. 2023. Computational CXL-Memory Solution for Accelerating Memory-Intensive Applications. *IEEE Computer Architecture Letters* 22, 1 (2023), 5–8. https://doi.org/10.1109/LCA.2022.3226482

[60] Peter Smith and Norman C. Hutchinson. 1998. Heterogeneous Process Migration: The Tui System. *Software: Practice and Experience* 28, 6 (1998), 611–639. https://doi.org/10.1002/(SICI)1097-024X(199805)28:6<611::AID-SPE169>3.0.CO;2-F

[61] Xiaojia Song, Tao Xie, and Stephen Fischer. 2021. Two Reconfigurable NDP Servers: Understanding the Impact of Near-Data Processing on Data Center Applications. *ACM Trans. Storage* 17, 4, Article 31 (oct 2021), 27 pages. https://doi.org/10.1145/3460201

[62] Sajjad Tamimi, Florian Stock, Andreas Koch, Arthur Bernhardt, and Ilia Petrov. 2022. An Evaluation of Using CCIX for Cache-Coherent Host-FPGA Interfacing. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 1–9. https://doi.org/10.1109/FCCM53951.2022.9786103

[63] Dominik Töllner, Christian Dietrich, Illia Ostapyshyn, Florian Rommel, and Daniel Lohmann. 2023. MELF: Multivariant Executables for a Heterogeneous World. In *2023 USENIX Annual Technical Conference (USENIX ATC 23).* 257–273. https://www.usenix.org/conference/atc23/presentation/tollner

[64] Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. 2019. Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).* 430–437. https://doi.org/10.1109/EMPDP.2019.8671589

[65] Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 121–132. https://doi.org/10.1145/2678373.2665692

[66] 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

[67] Tong Xing, Antonio Barbalace, Pierre Olivier, Mohamed L. Karaoui, Wei Wang, and Binoy Ravindran. 2022. H-Container: Enabling Heterogeneous-ISA Container Migration in Edge Computing. *ACM Trans. Comput. Syst.* (March 2022). https://doi.org/10.1145/3524452

[68] Chenle Yu, Sara Royuela, and Eduardo Quiñones. 2020. OpenMP to CUDA Graphs: A Compiler-Based Transformation to Enhance the Programmability of NVIDIA Devices. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems* (St. Goar, Germany) *(SCOPES '20)*. Association for Computing Machinery, New York, NY, USA, 42–47. https://doi.org/10.1145/3378678.3391881

[69] Mohamed Zahran. 2016. Heterogeneous Computing: Here to Stay: Hardware and Software Perspectives. *Queue* 14, 6 (Dec. 2016), 31–42. https://doi.org/10.1145/3028687.3038873

Received 24-OCT-2023; accepted 2023-12-23