HIDC: Heterogeneous-ISA Dynamic Core

Nirmal Kumar Boran[1,2], Prakhar Diwan[1,3], Meet Udeshi[1], Shubhankit Rathore[1] and Virendra Singh[1]
[1]Indian Institute of Technology Bombay, India [2]National Institute of Technology Calicut, India [3]Texas Instruments, India
nirmalkboran@nitc.ac.in, p-diwan@ti.com, {mudeshi1209,shubhankitrathore}@gmail.comviren@ee.iitb.ac.in,

***Abstract*—Heterogeneous-ISA architectures are emerging as an alternative for better performance and energy efficiency. These systems exploit ISA affinity within and across programs by executing them dynamically on the most suited ISA core. However, heterogeneous-ISA chip multiprocessor incurs significant power and area overheads compared to single-ISA systems. Also, due to switching overhead between different ISA cores, the migration opportunities are constrained to coarse-grained program phases (order of $100M$ instructions), limiting the potential of harnessing ISA diversity. We propose Heterogeneous-ISA Dynamic Core (HIDC), an architecture which reduces the migration overhead by supporting different ISAs within a single core and an improved migration strategy called Simultaneous Transformation. HIDC improves single-threaded performance and energy efficiency by exploiting ISA diversity at a finer granularity and reducing power consumption compared to heterogeneous-ISA CMP. To optimally schedule program phases as per their ISA affinity, we present a low-overhead perceptron-based scheduling mechanism along with its implementation in hardware. Overall, HIDC improves by 52.89% and 33.93% in performance per joule metric over heterogeneous-ISA CMP and single x86 ISA core, respectively. It achieves 4.62% and 19.06% performance improvement relative to heterogeneous-ISA CMP and single x86 ISA core. Also, it leads to $\sim$20$\times$ reduction in cross-ISA migration latency.**

***Index Terms*—heterogeneous architectures, single-thread performance, energy efficiency, ISA, execution migration**

I. Introduction

With transistors shrinking due to innovations in VLSI technology, Moore's law persists with rising transistor counts on single chip. However, corresponding decline in threshold voltages of transistors saturated. This restricted active number of transistors on chip for remaining within power budget.

Heterogeneous CMPs [1] were proposed to effectively trade area for energy efficiency by synthesizing cores with different resources on the same chip. With cores of different performance and power characteristics, energy efficiency over homogeneous CMPs was improved by allocating programs and their phases to most efficient core. Later, execution semantics was utilized as a heterogeneity dimension in ARM's big.LITTLE [2], which consisted of in-order and out-of-order cores. Researchers proposed techniques to better utilize this diversity and designed accurate schedulers [3].

Heterogeneous architectures were restricted to a single ISA, due to large cross-ISA migration overheads encountered by Tui system [4]. DeVuyst et al. [5] proposed heterogeneous ISA CMP, which consisted of different ISA cores; and presented a faster method to switch program execution between different ISA cores. Migration latency was reduced by (a) compiler modifications to store most of program state in ISA-independent format and (b) shared main memory between different ISA cores. Migrations were performed at equivalence points (where memory state was consistent across different ISA executables) or dynamic binary translation was performed till an equivalence point was reached. Venkat et al. [6] explored the design space of heterogeneous-ISA CMPs and highlighted significant performance and energy benefits over single-ISA heterogeneous CMPs. ISA affinity was attributed to different ISA characteristics such as code density, register pressure, native floating-point arithmetic vs. emulation, instruction complexity, and SIMD support. However, for single-threaded execution it faces two drawbacks. First, only the ISA-affine core remains active with other ISA cores remaining idle, leading to significant power costs compared to single-ISA systems. Also, these overheads magnify with number of ISAs considered, which is crucial as larger number of ISAs enhance the potential of ISA affinity [7]. Second, the cross-ISA switching overheads ($\sim 100\mu s$ [5], [6]) restrict the migration opportunities to coarse phases of $\sim 100M$ instructions.

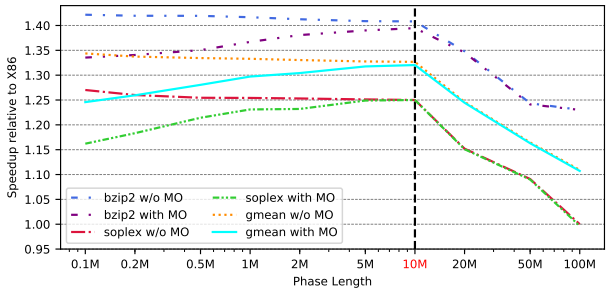

Fig. 1. Phase length sensitivity analysis for *bzip2* and *soplex*, with and without migration overhead (MO)

To find optimal phase length for cross-ISA switching, we performed a sensitivity analysis for two benchmarks from SPEC2006 [8] suite, namely *bzip2* and *soplex*; with analysis shown in Figure 1. We use two widely used ISAs: ARM and x86; and observed notable performance improvement moving from phase length of $100M$ to $10M$ x86 instructions, due to ISA-impacted parameters such as code density and register pressure fluctuating at finer granularity over execution. We used oracle scheduling and our proposed migration strategy's overhead for all phase lengths. For phases finer than $10M$ instructions, we observed that overall performance degraded due to migration overhead, as number of switches increased at finer phases. For ARM ISA's execution equivalent phase lengths were obtained via high-level language mapping. Hence, we use phase lengths equivalent to $10M$ x86 instructions for HIDC (spaced between function call sites).

The contributions of the paper are as follows:

- *Heterogeneous-ISA Dynamic Core*, an energy-efficient dynamic architecture that support ARM and x86 ISAs within single core. It introduces a shared cache hierarchy for program execution across ISAs.

- Technique of *simultaneous transformation* for faster program state migration between ISAs, enabling finer cross-ISA switching to harness potential of ISA diversity.
- A low-overhead *perceptron-based scheduler* for affine-ISA execution of program phases.

II. Related Work

Various works have been proposed related to heterogeneous architectures. Kumar et al. [1] showed that single ISA heterogeneous CMPs can significantly reduce power compared to homogeneous CMPs. Resource-sharing [9] in CMPs has been investigated to improve energy efficiency and area. *Composite Cores* [3] improves energy efficiency over heterogeneous CMPs by supporting in-order and out-of-order execution within a core. Researchers studied ISA diversity to improve performance and energy efficiency further. Heterogeneous-ISA CMP was introduced by Devuyst et al. [5], where they presented a system consisting different ISA cores and cross-ISA migration techniques. Blem et al. [10] claimed that ISA being RISC or CISC does not impact systems' performance and energy efficiency. However, the claim had several issues, such as non-uniform hardware platforms and infrastructural limitations. Venkat et al. [6] performed design space exploration on heterogeneous-ISA CMPs, showcasing their performance and energy benefits. For deriving maximum benefits from heterogeneous-ISA CMP, efficient scheduling algorithms have been proposed [7], [11]. For achieving heterogeneous-ISA gains from single ISA, [12] proposes a heterogeneous CMP consisting of cores built on composite instruction feature sets. Barbalace et al. [13] extended the energy benefits of ISA heterogeneity to server space by presenting system software for execution migration between ARM and x86 servers.

III. HIDC Design

A. Overview of HIDC architecture

HIDC is an out-of-order core that dynamically supports execution with ARM(v7) and x86 ISAs by sharing the execution pipeline between ISAs. It contains a superset of resources required by both the ISAs with detailed configuration compared to heterogeneous-ISA CMP listed in Table III from Section VI. It employs separate decoders for each ISA, whereas other pipeline stages are shared between the ISAs with minor changes (explained in following subsections). Pipeline resources not required by the executing ISA are clock-gated to conserve energy. For indicating the executing ISA we add a 1-bit register called Current-ISA Bit (*CIB*). It is reset by default, with 0 and 1 denoting x86 and ARM respectively. Based on *CIB*, the superscalar pipeline is dynamically adjusted as per the needs of executing ISA, which includes changing the functional unit pool, activating appropriate decoder, and clock-gating the structures unused by running ISA. In the proposed work, we explore two diverse and widely used ISAs: x86 and ARMv7; with the idea extensible to other ISAs. We do not modify either of the ISAs, hence, supporting legacy codes.

For extracting maximum benefits from ISA affinity of program phases, we require optimal mapping of program phases to their affine ISA. This is achieved with the migration engine integrated alongside the core on same chip, to facilitate scheduling and migration to the affine ISA. It monitors multiple micro-architectural parameters and then dynamically decides the affined ISA for the program phase. We propose *simultaneous transformation* strategy for faster migration of program execution between ISAs. We introduce two 1-bit registers: (a) migration decision (*MD*) to indicate if migration is scheduled at upcoming equivalence point, and (b) back-reference flag (*backref-flag*) to indicate back-references of open functions during migration. These are reset by default and are accessible like model-specific registers. Details about migration methodology used in HIDC are mentioned in Section IV. Section V details the migration engine. Figure 2 provides an architectural overview of HIDC. Next, we describe stage-wise changes to the out-of-order pipeline for HIDC.

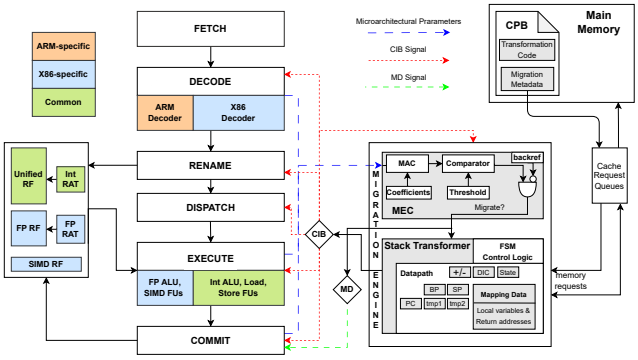

Fig. 2. Architectural overview of HIDC

B. Fetch and Decode Stages

The fetch stage of the pipeline behaves identically irrespective of the executing ISA (indicated via *CIB*). The fetch unit accesses 64 bytes based on the program counter from the L1 I-Cache every cycle. This stage is modified for HIDC's operation by halting fetch for instructions after reaching an equivalence point in case a migration is planned at that point (indicated via commit stage). The decode stage comprises of separate decoders corresponding to ARM and x86 ISAs for decoding instructions from both instruction sets. The incoming chunk of instructions is fed to the appropriate decoder based on *CIB*. x86, being a CISC ISA, demands decoding variable length instructions (macro-ops) into RISC-like μops. Hence, a complex multi-stage decoder is employed, whereas ARM being a RISC ISA requires decoding of fixed-length instructions. Therefore, a simpler decoder suffices. The decoded μops, in form of control signals and data, are passed to subsequent pipeline stages and have a common format irrespective of ISA.

C. Rename and Dispatch Stages

After decoding, the decode stage buffer's instructions (μops) undergo register renaming. There is a minor change in the rename stage as ARM and x86 utilize different number of architectural registers, 32 and 16, respectively. This distinction leads to a difference in the number of bits used for the register index, which is used in re-order buffer (ROB) and reservation station (RS) entries, and for

accessing the register alias table (RAT). We overcome this by using renaming logic for 32 architectural registers for both the ISAs.

A unified integer architectural register file is used by both ISAs. The entries in integer RAT are updated upon migration as per the target ISA. As x86 supports floating point (FP) and SIMD instructions, the core consists of FP register file, SIMD register file and FP register alias table (RAT). These are clock-gated during the execution of ARM ISA (signaled via *CIB*), as ARMv7 supports floating point instructions using software emulation technique, hence, it uses only integer register file. After register renaming, the μops are dispatched to reservation station, an entry in ROB is reserved for them and for store operations in store buffer as well.

D. Execute and Commit Stages

In execute stage, once source operands of an instruction are ready and required functional unit is available, it is selected for execution. In HIDC, the execute stage consists of a superset of functional units required by ARM and x86 ISAs. Integer ALUs, integer multiply/divide units, and shifters are used by both ISAs, whereas SIMD and floating point units are specific to x86 ISA. Both are present in functional unit pool, with SIMD and floating point units clock-gated during ARM ISA's execution (as they are supported using software emulation), thereby, making clock-gating logic the only modification to this stage. Hence, the number and type of active functional units in execution stage change based on executing ISA.

Commit stage contains the re-order buffer (ROB), which maintains program order of instructions that are executed out-of-order. ROB is shared by both ISAs and is modified to keep information required for execution of both ISAs. Instructions are committed from the head of ROB in conventional manner for both ISAs. For committing SIMD and floating point (FP) x86 instructions, the logic for updating FP register alias table, FP register file and SIMD register file remains as in traditional x86 out-of-order pipeline. For memory reads/writes, since ARM and x86 use different memory consistency models, lazy retire and total store order (TSO), HIDC enforces stricter TSO model for unified execution of both ISAs. We observed $<$2% performance penalty due to this restriction from simulations. The commit stage is modified to use *MD* for identifying migration points. If *MD* is '1' and an instruction indicating function call is committed, then HIDC identifies it as a migration point. After finding a migration point, *MD* is reset and in-flight instructions are flushed. We store the base addresses for ARM and x86 ISA binaries in two additional registers. The displacement for any function is maintained same across the ISAs via padding (as in prior works). Hence, on migration to target ISA we add corresponding displacement to base address of target ISA binary for obtaining program counter.

E. Cache Hierarchy

In heterogeneous-ISA CMPs, shared main memory led to a reduction in cross-ISA migration overheads by avoiding memory image transfer, which was the bottleneck in earlier work [4]. However, they used private LLCs, which led to a substantial overhead. As HIDC unifies the execution of different ISAs within a core, it leads to a shared cache hierarchy across different ISA executions. Due to the code section being ISA-specific, the stale instruction blocks (of source ISA) are invalidated from cache hierarchy upon migration (to avoid aliasing). Instruction blocks are identified with the help of a 1-bit flag. However, as data sections are made target-independent, corresponding data blocks in cache hierarchy remain valid even post-migration. This lowers the migration cost by reducing cache misses post-migration.

IV. Migration Methodology

Upon a change in the ISA affinity of the executing program, it must be migrated from the executing/source ISA to the affine/target ISA. We borrow some techniques from DeVuyst et al. [5] to facilitate migration. However, it incurred large migration overheads, hence, we present a novel technique called *simultaneous transformation*. We compile a *fat* binary called combined program binary (CPB). CPB is used during execution with both ISAs and consists of target-independent data sections and target-specific code sections. It is appended with migration metadata, which contains compilation information about function call sites, frame layouts, variable locations, caller-saved and callee-saved registers' spill areas for both ISAs and mapping between program objects generated via intermediate representation. The metadata also comprises an extra bit (*backref*) corresponding to each function to indicate if it back-references to any ancestor functions' stack frame through its pointers arguments. Overall, the metadata negligibly adds to the binary size. We use *unified* address space and common address translation scheme as in prior works. To avoid fixing function pointers after migration, every function's definition is placed at an identical virtual address, along with the order of functions in binary being identical across ISAs. This requires function size to be identical, which is achieved by padding *NOP* instructions.

As we do not switch to a different core in HIDC, migration involves program state transformation for execution using target-ISA. *This is required so that target ISA finds the program state identical to the case where the program was executed using target ISA till the migration point.* To minimize the program state transformation, memory image consistency is required. It is achieved by keeping the number of objects, relative order, sizes, alignment, and padding rules identical within each section across the ISAs. We enforce the same endianness and stack growth direction across ARM and x86 ISAs; and choose little-endian and downward direction, as x86 is little-endian and supports downward direction, whereas ARM supports both endianness and stack growth directions. We choose equivalence points as function call sites because the memory image is consistent at these points across different ISA executables. These modifications reduce the overhead for program state transformation.

We describe the overview of migration in HIDC with an example shown in Figure 3. Due to reduced migration overhead, we shrunk the phase length from hundred million [6] to ten million instructions. It is assumed that source

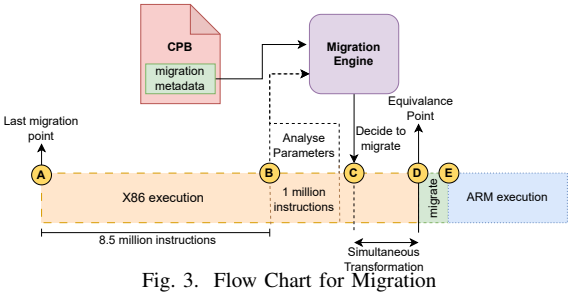

Fig. 3. Flow Chart for Migration

ISA is x86 and the following function call site after C is at D. Upon crossing $8.5M$ instructions from point A (indicated via 24-bit dynamic instruction counter (DIC) in migration engine), HIDC resets performance counters, which measure the microarchitectural parameters required by MEC (listed in Section V-A). Upon reaching the $9.5M$ instruction count, the performance counters' values are transferred to MEC, which makes the migration decision. If chosen to migrate, *MD* is set and simultaneous transformation begins from point C by the stack transformer (ST), while HIDC concurrently executes the program. HIDC identifies the migration point at D via the commit stage and halts. Once program state transformation is finished (at point E), the current-ISA bit (*CIB*) is appropriately written by migration engine, whereas migration decision (*MD*) 1-bit register is reset. Clock-gated structures required for ARM's execution are activated at point D. The structures not required by ARM are clock-gated at point E. Thereafter, program execution resumes natively on ARM. It should be noted that from point C to D, HIDC executes atomically, deferring any occuring interrupts.

A. Program State Transformation

Due to the compiler modifications described previously, only the register state and stack portions must be transformed upon migration. Return addresses and local variables of open functions are the stack portions, which must be modified upon migration. The register state must also be changed for the target ISA's execution. We use LLVM (*llc* and *clang*) [14] for analysis of function stack-frames for stack transformation.

TABLE I
STACK SLOT MAPPING FOR *BZ2_bzDecompressStream*

Slot	Size	Align	Location	
			x86	ARM
fi.0	4	4	[SP-28]	[SP-20]
fi.1	8	8	[SP-24]	[SP-32]
fi.2	4	4	[SP-36]	[SP-36]
fi.3	4	4	[SP-32]	[SP-40]
fi.4	8	8	[SP-16]	[SP-48]

For transforming the program state as needed by the target ISA's execution, we use the migration metadata. As the number of architectural registers and calling conventions vary across ARM and x86, the variables may be expected at different locations across ARM and x86 codes. Hence, a transformation of the architecture-specific program state (stack and register state) is necessary, which requires two passes, as described in prior works. 1^{st} pass goes from the innermost to the outermost frame, obtaining caller-saved registers spilled and local variables saved on the stack. Once it obtains these for a frame, it uses the cross-ISA mapping to modify the stack. The compiler reduces stack changes by allocating the same stack locations to large variables whose addresses are taken. Hence, only stack slots corresponding to small variables are modified. The local variables mapped to registers are stored to be used later in the 2^{nd} pass for generating register state at equivalence points. Apart from local variables, return addresses must be changed for correct execution using target ISA. Return addresses are stored in stack frames at a compiler-known offset. They are fetched, mapped to their counterparts in target ISA using migration metadata, and modifications are stored for transformation in 2^{nd} pass. The 2^{nd} pass goes from outermost frame to innermost frame, obtaining data of callee-saved register spills. Along the traversal, it builds the program state at visited function call sites as expected by target ISA. We provide an example of stack slots mapping between ISAs for the function *BZ2_bzDecompressStream* from *bzip2* in Table I.

B. Simultaneous Transformation

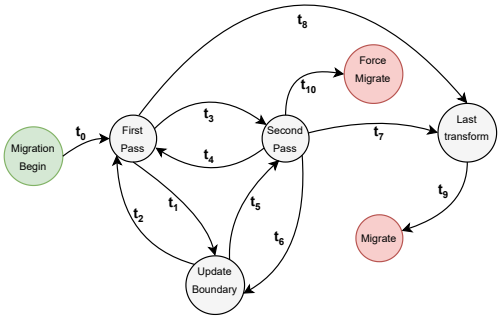

Fig. 4. State Diagram for Simultaneous Transformation

We propose a novel migration technique, 'simultaneous transformation'. The migration metadata, which includes variable mappings from the compiler's live analysis, is appended to the binary. It is accessed by the migration engine as shown in Figure 3. The data obtained from the 1^{st} pass is used for transforming the register state and callee-saved spilled stack region as expected by the target ISA. It is stored with stack transformer to be used for 2^{nd} pass.

Event	t_i
Setting P_{bottom} and collecting *backref* information	t_0
HIDC executes function return (1^{st} pass)	t_1
P_{bottom} is updated (1^{st} pass)	t_2
1^{st} pass complete	t_3
2^{nd} pass complete, recursive 1^{st} pass	t_4
P_{bottom} is updated (2^{nd} pass)	t_5
HIDC executes function return (2^{nd} pass)	t_6
2^{nd} pass complete, single untransformed frame left	$t_7(a)$
HIDC reaches equivalence point	$t_7(b)$
HIDC calls function with *backref* (during 2^{nd} pass)	$t_7(c)$
Pre or during 1^{st} pass, single untransformed frame left	$t_8(a)$
HIDC reaches equivalence point	$t_8(b)$
HIDC calls function with *backref* (during 1^{st} pass)	$t_8(c)$
backref-flag == *true* & HIDC executes function call	$t_8(d)$
backref-flag == *true* & 1^{st} Pass Complete	$t_8(e)$
Register values shifted to register file, ready to migrate	t_9
HIDC tries to enters a transformed frame	t_{10}

TABLE II
MAPPING OF EVENTS TO TRANSACTIONS OF FIGURE 4

Stack transformer (ST) initiates program state transformation upon receiving migration decision from migration engine controller (MEC). ST's hardware description

is provided in section V-B. It recursively performs the transformation for inactive open functions in parallel to HIDC's execution (as shown in Figure 4) before migration point. Let f_{curr} be the function being currently executed by HIDC. ST utilizes data from f_{curr} call site, required for transforming ancestor stack-frames. We store f_{curr} as the bottom boundary (P_{bottom}), which is updated every time HIDC removes the corresponding stack frame upon completion of a function (i.e. execution of return instruction), shown via t_1 and t_2 in Figure 4 and Table IV-B. ST begins the 1^{st} pass from innermost frame to outermost frame by reading the current stack pointer (SP) and base pointer (BP) to obtain previous open function's stack frame, which is now inactive. The program counter (PC) is used to determine mapping of the stack frame to the corresponding function name and its call site. ST stores the local variables' values, which map to registers in the target ISA. ST then reads the previous BP stored on stack to go to the previous stack frame, traversing the entire stack while collecting migration metadata for transforming each stack frame visited during 2^{nd} pass as required by target ISA.

Upon finishing the 1^{st} pass, ST begins the 2^{nd} pass (t_3 in Figure 4 and Table IV-B) by using the information collected during the 1^{st} pass. ST starts 2^{nd} pass going from the outermost function to P_{bottom} and modifies the register state, local variables (mapped to stack for target ISA), caller and callee-saved spill locations and return addresses in-place on stack, saving on memory accesses and migration overhead.

Once the 2^{nd} pass is also completed, ST recursively begins the 1^{st} pass and 2^{nd} pass (t_3 and t_4 in Figure 4 and Table IV-B.) of the remaining un-transformed frames similarly. These recursive iterations continue until either HIDC reaches the migration point, or a single frame is left for transformation during HIDC's execution (t_7 and t_8 in Figure 4 and Table IV-B). It was empirically observed that ST completes transformation of frames and catches up to HIDC with a single frame left ($t_7(a)$ and $t_8(a)$) occured more frequently than ($t_7(b)$ and $t_8(b)$). In both these cases, HIDC is halted and ST enters Last Transform, a state where HIDC is halted and ST finishes it's remaining transformations; hence, a *last transformation* is performed. After this, HIDC is finally migrated to the target ISA before the migration point (coming from $t_7(a)$ or $t_8(a)$ in Figure 4 and Table IV-B), or at equivalence point ($t_7(b)$ or $t_8(b)$ in Figure 4 and Table IV-B). Upon completion of the transformation, the architecture-specific program state (register and stack) is as per the expectations of the target ISA at the migration point. (t_9 in Figure 4 and Table IV-B)

During the execution of either passes, the Bottom Boundary (P_{bottom}) needs to be updated to synchronize HIDC with ST and prevent transformation of any frame which has been removed by HIDC (t_1, t_2, t_5, t_6 in Figure 4 and Table IV-B). This is done by stalling both ST and HIDC for few cycles (order of tens); necessary to properly update the P_{bottom} and preventing HIDC from crossing ST. Furthermore, during the 2^{nd} pass, a case may arise where HIDC tries to use a transformed stack frame. In this case, ST enters Forced Migration, a state where HIDC is stalled and moved to target ISA post-transformation by ST (t_{10} in Figure 4 and Table IV-B). The terminating states for the migration process are shown with red colour in Figure 4. Upon reaching these states, the CIB is modified by ST as per target ISA. The transformation of memory state as per the requirements of target ISA is performed in 2^{nd} pass in in-place fashion. The in-place change requires going in a sequence such that no stack data is overwritten, requiring two temporary variables.

However, in-place and simultaneous transformation raised a challenge. As stack-frames of ancestor functions are transformed simultaneous to f_{curr}'s execution, f_{curr}'s reference to any data from previous frames via pointers may become invalid. This issue is tackled by only performing the 1^{st} pass if *backref* is '1' for f_{curr} and starting 2^{nd} pass after HIDC reaches the equivalence point. In case any of the open function frames (at beginning of migration) have any back-references, the *backref-flag* is set. With back-references in play, two cases could arise - HIDC tries to call a new function which contains a back reference or ST tries to transform a frame which is back-referenced by another recent frame. In former case, HIDC is immediately halted, ST enters Last Transform state; ST finishes it's current transformation cycle and performs the *last transformation* ($t_7(c)$, $t_8(c)$ in Figure 4 and Table IV-B). However, in latter case, we prevent the situation from happening by only allowing HIDC to run during 1^{st} pass of transformation. In case ST completes the 1^{st} pass and HIDC hasn't arrived at an equivalence point, it stalls until HIDC reaches equivalence point, after which ST enters Last Transform ($t_8(e)$ in Figure 4 and Table IV-B). On other hand if HIDC executes a call instruction during ST perfoming the 1^{st} pass, HIDC is halted and ST enters Last Transform ($t_8(d)$ in Figure 4 and Table IV-B). In this case, we do not allow HIDC to execute *call* and stall it, to reduce the number of frames whose transformation is exposed. This can be improved further, however, we rarely observed *backref* being set for the executing function across the equivalence points considered.Overall, the migration latency was effectively hidden behind HIDC's program execution (migration overhead is exposed for a single frame only), using simultaneous transformation. As many functions can be in open state during migration decisions; hence, the stack can consist of multiple function frames. In previous proposals [5], [6], migration process starts after program execution is halted, leading to high migration costs ($\approx$ 100 μs). We reduce the cost by (a) making migration decisions early and performing simultaneous transformation and (b) sharing cache hierarchy across different ISA executions

V. Migration Engine

The migration engine is additional hardware integrated alongside the dynamic out-of-order core. It is responsible for decision-making and program state migration to affine ISA; and allowed access to performance counters and cache request queues. A bit is added to all memory requests indicating whether to bypass L1D called BL_1. BL_1 is set to '1' for memory accesses made by migration engine,

providing it access to the L2 cache. The migration engine also has read/write access to current-ISA bit (*CIB*) and migration decision (*MD*) bit, and is composed of two components: the Migration Engine Controller and Stack Transformer, described below.

A. Migration Engine Controller (MEC)

To predict the affine ISA for upcoming phase, we use a classification-based scheduling method [11], which is based on a single-layer perceptron model. We construct it to make accurate decisions at finer phases with lower number of microarchitectural parameters, thereby reducing hardware overhead. The predictor dynamically schedules upcoming phase (of $\approx 10M$ instructions) via online analysis of micro-architectural parameters from $1M$ instructions of the current phase. We observed that accuracy of the predictor drops negligibly even after reduction of phase length from $100M$ to $10M$ instructions. If the scheduler predicts migration, the *backref-flag* bit register (updated on every function entry using migration metadata) is checked. If it is '1', then simultaneous transformation is performed only for 1^{st} pass, and then once HIDC reaches the equivalence point, stack transformer performs the 2^{nd} pass and completes migration; otherwise, the decision to migrate is propagated. The perceptron model is incorporated inside MEC along with registers (memory-mapped) to store the threshold and model coefficients corresponding to source ISA. It utilizes ten microarchitectural parameters to predict the affine ISA for next phase. We use L1-I, L1-D, L2 (last-level cache) misses to quantify stalls due to them. We model execution pipeline stalls using instruction queue, store queue and re-order buffer full events. We utilized branch misprediction count and instruction mix (floating point instruction ratio) to model impact of branch predictor and ISA-specific nature, respectively. Most of these parameters are available through performance counters in modern processors [15]. ILP and MLP were used to measure parallelism of executing program, and estimated using prior works [3], [11].

For each benchmark, we divide the microarchitectural parameters observed from execution into training and test data, both of them being mutually exclusive. The perceptron model consists of ten nodes corresponding to the microarchitectural parameters used. Perceptron model is trained offline with training data. After training, we obtain two sets of coefficients corresponding to ARM or x86 as source ISAs. The initial phase is executed using x86 ISA (as indicated by *CIB* reset to 0). For upcoming phases, model input parameters are extracted using performance counters and appropriate coefficients are fetched from the memory-mapped registers. After this, a sum of product is taken between model input parameters and their coefficients, then compared with the threshold (obtained from training). The result decides the affine ISA for next phase.

The hardware overhead for MEC includes the required performance counters (most are already existent in modern processors [15]), coefficients and threshold values (memory-mapped programmable registers), a multiply-and-accumulate (MAC) unit and a comparator.

B. Stack Transformer (ST)

Stack transformer's high-level working is described in Section IV-B. This logical behavior is achieved through a finite state machine (FSM), with hardware implementation consisting of FSM control logic and the datapath. The datapath includes registers for the FSM state, two temporary variables for in-place transformation, dynamic instruction counter (DIC), base pointer (BP), stack pointer (SP), program counter (PC). It also includes an adder/subtractor unit and space for cross-ISA mapping data. To store the cross-ISA mapping data we utilize a 2KB register file.

VI. Results

The experiments were performed using Gem5 [16] architectural simulator with SPEC CPU2006 [8] and CPU2017 [17] benchmark suites. The benchmarks were compiled using gcc for x86 and cross-compilation was done for ARM(v7). Table III shows the detailed configuration of the cores. HIDC is compared with a dual-core heterogeneous-ISA CMP, containing an x86 core and ARM core, each with private last level caches. This configuration is called 'Heterogeneous-ISA CMP' (HISACMP) [6] and serves as a baseline for our work. We provide comparisons of HISACMP and HIDC relative to x86 core. The power and area analysis was performed using the McPAT [18] framework. All the cores are modeled using 32nm technology, with the clock rate fixed at 3GHz. Traditionally, ARM cores are designed to achieve energy efficiency, whereas x86 cores are geared towards performance. We try to reflect this using the configuration chosen for the respective ISA cores in HISACMP configuration. HISACMP configuration uses an oracular schedule (identical to their proposal), i.e., it knows the ISA affinity of future phases. Also, during HISACMP's single-threaded execution, the idle core is power-gated. Some benchmarks could not be cross-compiled for ARM successfully or were incompatible with our simulation infrastructure. However, we analysed results for most benchmarks from prior work [6] and added *h264ref, specrand, soplex* alongwith *deepsjeng, leela, namd, xz* from SPEC CPU2017 suite.

TABLE III
Core configurations

Configuration	**HISACMP**		**HIDC**	
ISA	ARM	x86	ARM	x86
Out-of-order Core with 3GHz clock				
Superscalar Width	4			
LQ size	16	36	36	
SQ size	16	48	48	
RS size	36	64	64	
ROB size	108	192	192	
Cache Hierarchy				
L1-I/D	32 KB, 8-way, 2-cycle latency			
L2 (LLC)	2MB 16-way, 15-cycle latency			

A. ISA affinity of different programs

We identify ISA affinity of program phases based on performance. Figure 5 describes the percentage affinity between ARM and x86 ISAs. From Figure 5, we see that *gobmk, h264ref,leela* are heavily affined towards ARM ISA, as are *hmmer, libquantum,deepsjeng* towards x86. We also observed that benchmarks like *bzip2, mcf, milc,*

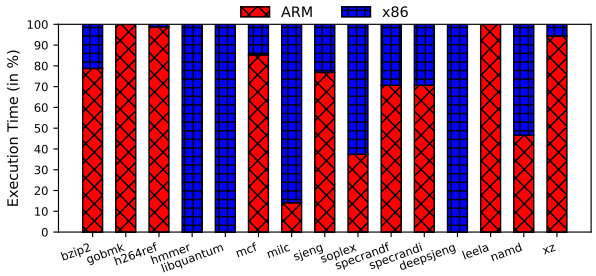

Fig. 5. Percentage ISA affinity for benchmarks

sjeng, soplex, specrandi, specrandf, xz, namd displayed mixed ISA affinity. This difference in affinity arises due to varying program behavior for different ISAs, i.e., types of supported operations, varying complexity of memory operations and number of instructions. For example, *libquantum* and *milc* applications efficiently utilize the SIMD support of x86. The high ILP phases of *bzip2, xz* are affined towards ARM ISA due to lower register pressure encountered during execution.

B. Dynamic Scheduling

Figure 6 shows the accuracy of the migration decision on the test dataset. *Accuracy is defined as the number of times the scheduler can correctly predict affinity for the upcoming program phases.* The model is trained using a subset of the SPEC CPU2006 and CPU2017 benchmark suites. We speculate performance of $10M$ dynamic instruction phases based on parameters from $1M$ instructions. We utilized phases of $10M$ instructions, with 70:30 as training and test split. The last $1M$ instructions' microarchitectural parameters (mentioned in Section V-A) were collected as data. We observed an average prediction accuracy of 90.26%.

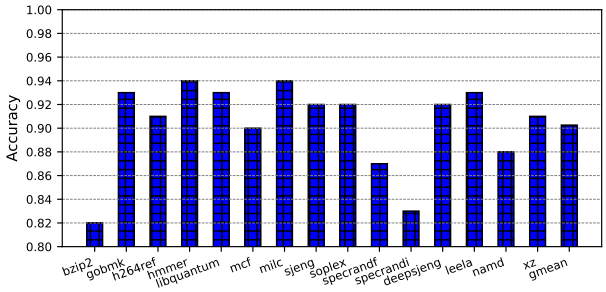

Fig. 6. Prediction accuracy of perceptron scheduler

C. Migration overhead

The migration overhead ranges from 0.3μs to 8μs as shown in Figure 7. Migration overhead is more than an order of magnitude lesser in HIDC compared to heterogeneous-ISA CMP. This can be attributed to a) the hybrid core, which leads to shared caches, and b) simultaneous transformation, which hides a major part of migration latency behind program execution by HIDC. Due to different register pressures, the stack-to-register movement for x86→ARM is higher; hence, more load operations are performed. In comparison, ARM→x86 has the opposite behavior, which causes more store operations, leading to higher migration overhead. The migration cost depends on the cost of migration of the last function before migration, as we observed that ST always finishes the 1^{st} pass before HIDC reaches the equivalence point.

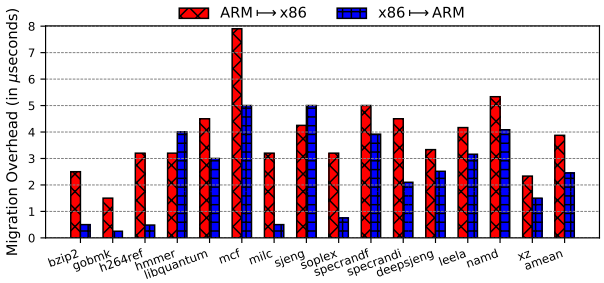

Fig. 7. Migration overhead for HIDC

D. Performance and Energy of HIDC

Figure 8 shows the performance gains achieved using HIDC over HISACMP, using x86 core's performance as reference. Oracle is the case where each phase runs on its affine ISA core. Benchmarks *bzip2, h264ref, mcf, xz* and *soplex* achieve maximum benefits of finer phases due to quicker migration. Due to the low accuracy of scheduler for *bzip2*, a significant performance drop from oracle was observed. With oracle scheduling, HIDC achieves 4.62% and 19.06% improvement relative to HISACMP and x86 core, respectively. We observe HIDC with perceptron-based scheduling outperforms HISACMP with oracle scheduling.

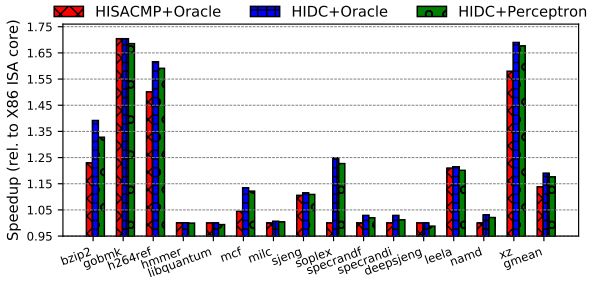

Fig. 8. Performance of HIDC relative to x86 ISA core

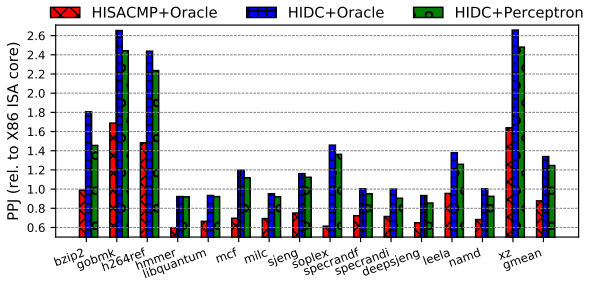

Fig. 9. Performance Per Joule of HIDC relative to x86 ISA core

HISACMP was evaluated on single-threaded benchmarks relative to single-ISA heterogeneous CMP, showcasing energy efficiency improvements. However, when compared to x86 core, from Figure 9 we observed that HISACMP suffers 12.40% performance per joule (PPJ) degradation, despite performance improvement of 13.80%. HIDC achieves significant PPJ gains over x86 core for heavily ARM-affined benchmarks *h264ref, xz* and *gobmk*, owing to significant speedup and small power overheads. *bzip2, mcf, soplex* and *sjeng* display mixed ISA affinity, which is leveraged at finer phases by HIDC to outshine both HISACMP and x86 core. HIDC faces minor PPJ loss for x86-affine benchmarks *hmmer, liquantum, deepsjeng* and *milc*, due to minor power overhead over x86 core. With oracular schedule HIDC improves PPJ by 33.93% and 52.89% relative to x86 core and HISACMP.

E. Evaluation of multi-programmed workloads

HIDC is a single core architecture to improve the energy efficiency of heterogeneous-ISA execution. However, we claim the proposal is scalable to multiple cores and evaluate multiprogrammed workloads to support the same. In Figure 10, we provide average performance, power, energy and performance per joule results for all the 1365 quad-combinations and 105 pair-combinations generated from the 15 SPEC benchmarks considered. For evaluating the quad-core and dual-core configurations, we utilized private LLCs (as in heterogeneous-ISA CMP) to reduce the number of simulations. With oracle scheduling in a dual-core system, we observed an average improvement of 32.55% and 50.36% over HISACMP (with power-gating) on energy consumption and performance per joule metrics respectively. We achieve better performance to multicore HISACMP configuration over the combinations studied.

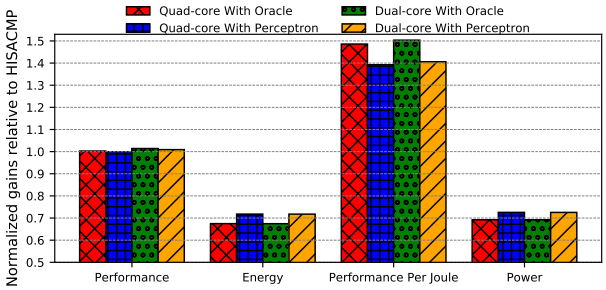

Fig. 10. Evaluation over multi-programmed workloads

F. Migration effect on hardware structures

After switching to a different ISA in HIDC, instruction blocks in the cache hierarchy (and i-TLB) are invalidated, and the branch predictor state is flushed. We modeled the effects of performing these operations on phase length of $10M$ instructions. This analysis is conservative as the migrations are not requested after every phase. Overall, we observed that phase-wise performance is degraded by $<1\%$ on average, hence, a negligible impact.

G. Hardware and Area Overhead

The area calculation was performed using the McPAT [18] framework. The result shows that the area is reduced by 42.82% in HIDC compared to heterogeneous-ISA CMP. The reduction in area is due to the sharing of core resources in HIDC across ISAs, whereas in HISACMP, resources are dedicated to specific ISA cores. The hardware overhead for migration engine components, MEC and ST are mentioned in Sections V-A and V-B.

VII. Conclusion

With the increase in computing requirements, processor architecture needs timely modification. Utilizing the ISA affinity present in different program phases gives significant performance boost and energy savings with heterogeneous ISA architectures over single ISA counterparts. This work proposes a novel architecture HIDC, which dynamically supports multiple ISAs within a core. It improves energy efficiency and single-threaded performance by executing programs with different ISAs within single core. It utilizes a novel method called *simultaneous transformation* to migrate program execution between different ISAs. Overall, HIDC improves performance per joule by 52.89% & single-threaded performance by 4.62% on average over heterogeneous-ISA CMP. The cross-ISA migration in HIDC is $\sim$20$\times$ faster than prior works. A perceptron-based scheduler is utilized to observe ISA affinity changes and migrate precisely. Compared to x86 ISA core, HIDC improves performance and performance per joule by 19.06% and 33.93%, respectively.

Acknowledgement

This work was supported in part by Indo Japanese Joint Lab Grant and AI powered adaptive cyber defence framework sponsored by NSCS, Government of India.

References

[1] R. Kumar et al., "Single-isa heterogeneous multi-core architectures: the potential for processor power reduction," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 81–92.

[2] B. Jeff, "Big.little system architecture from arm: saving power through heterogeneous multiprocessing and task context migration," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1143–1146.

[3] A. Lukefahr et al., "Composite cores: Pushing heterogeneity into a core," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 317–328.

[4] P. Smith et al., "Heterogeneous process migration: The tui system," *Software: Practice and Experience*, vol. 28, no. 6, pp. 611–639, 1998.

[5] M. DeVuyst et al., "Execution migration in a heterogeneous-ISA chip multiprocessor," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 261–272.

[6] A. Venkat et al., "Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 121–132.

[7] P. Diwan et al., "Mist: Many-isa scheduling technique for heterogeneous-isa architectures," in *2024 37th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID)*, 2024, pp. 348–353.

[8] "Spec 2006," 2006. [Online]. Available: https://www.spec.org/cpu2006/

[9] R. Kumar et al., "Conjoined-core chip multiprocessing," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004, pp. 195–206.

[10] E. Blem et al., "Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 1–12.

[11] N. K. Boran et al., "Classification based scheduling in heterogeneous isa architectures," in *2020 24th International Symposium on VLSI Design and Test (VDAT)*, 2020, pp. 1–6.

[12] A. Venkat et al., "Composite-isa cores: Enabling multi-isa heterogeneity using a single isa," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 42–55.

[13] A. Barbalace et al., "Breaking the boundaries in heterogeneous-isa datacenters," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 645–659.

[14] C. Lattner, "Llvm and clang: Next generation compiler technology," ser. The BSD Conference, 2008.

[15] D. Terpstra et al., "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173.

[16] N. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011.

[17] "Spec 2017," 2017. [Online]. Available: https://www.spec.org/cpu2017/

[18] S. Li et al., "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.