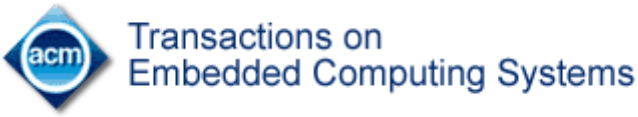

HIDC: Heterogeneous-ISA Dynamic Core Architecture

Journal:	*Transactions on Embedded Computing Systems*
Manuscript ID	TECS-2021-0235
Manuscript Type:	Paper for Domain-Specific System-on-Chip Architectures and Run-Time Management Techniques
Date Submitted by the Author:	03-Oct-2021
Complete List of Authors:	Boran, Nirmal; IIT Bombay Udeshi, Meet; IIT Bombay Rathore, Shubhankit; IIT Bombay Singh, Virendra; IIT Bombay
Computing Classification Systems:	
Topic - A primary topic is required. You may also select a secondary topic.:	18. Reconfigurable Embedded Systems

HIDC: Heterogeneous-ISA Dynamic Core Architecture

NIRMAL KUMAR BORAN, CADSL Lab, IIT Bombay, India
MEET UDESHI, CADSL Lab, IIT Bombay, India
SHUBHANKIT RATHORE, CADSL Lab, IIT Bombay, India
VIRENDRA SINGH, CADSL Lab, IIT Bombay, India

Heterogeneous-ISA architectures are emerging as a better alternative for performance and energy benefits. In such architectures, programs are executed using multiple ISAs considering phase-wise and application-specific affinity. Prior architectures were having higher area overhead, higher energy consumption and higher migration overhead. To overcomes these problems, we are proposing a new architecture: Heterogeneous-ISA dynamic core (HIDC). This architecture integrates support for multiple ISAs into a single dynamic core. This dynamic core is capable of dynamically changing its working ISA, while the program is under execution. Integrating multiple ISAs on the same chip improves single-threaded performance significantly by extracting ISA heterogeneity. We introduce a classification technique based on linear regression classifier which attempts to classify the program into the most suitable class of core (in terms of microarchitecture and ISA) and then that part of the program is scheduled on the most suited ISA. To achieve efficient migration between ISAs, we propose a migration strategy called Simultaneous Transformation which reduces migration overhead time by approximately 100×. Experimental results show that HIDC architecture gains an average performance speedup of 30% relative to x86 and 34% relative to previous implementations of Heterogenous-ISA CMP processor. On an average, 9% of energy saving is achieved with the proposed HIDC architecture.

CCS Concepts: • >**Multi core architectures** → **Heterogeneous-ISA multi-core chip**; • **Single thread performance**; • **Execution migration**;

Additional Key Words and Phrases: Regression techniques.

ACM Reference Format:
Nirmal Kumar Boran, Meet Udeshi, Shubhankit Rathore, and Virendra Singh. 2018. HIDC: Heterogeneous-ISA Dynamic Core Architecture. 1, 1 (October 2018), 19 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The Demand for high-performance computing is increasing rapidly with the increasing use of big data applications. Architects have proposed chip multiprocessors (CMPs) consisting of multiple cores to achieve high throughput. Multi-core architectures were able to increase the performance of multi-threaded programs. However, a single-threaded program's performance is still a bottleneck. Researchers have shown that only 60-80% of the code can be parallelized, and the remaining *20-40% is single-threaded code*. Hence, single-threaded performance is still a major bottleneck to enhance performance of the core.

Authors' addresses: Nirmal Kumar Boran, nirmalkboran@iitb.ac.in, CADSL Lab, IIT Bombay, Mumbai, Maharashtra, India, 400076; Meet Udeshi, mudeshi1209@gmail.com, CADSL Lab, IIT Bombay, Mumbai, Maharashtra, India, 400076; Shubhankit Rathore, shubhankitrathore@gmail.com, CADSL Lab, IIT Bombay, Mumbai, Maharashtra, India, 400076; Virendra Singh, viren@ee.iitb.ac.in, CADSL Lab, IIT Bombay, Mumbai, Maharashtra, India, 400076.

Manuscript submitted to ACM

To enhance single-threaded performance along with multi-thread performance, dynamic cores are proposed [13], [10], [15] in multi-core systems. These cores change their architecture dynamically to adapt to the program requirement. All the cores in such architectures support a single ISA. To improve the single-threaded performance further, Venkat et al. [16] explored a new dimension of Heterogeneity in ISAs for CMPs. They exploited the application's affinity towards a particular ISA. The authors discussed that ISAs have different characteristics such as, code density, dynamic instruction count, register pressure, native floating-point arithmetic vs emulation, decode logic and instruction complexity, and SIMD support. Due to such diversity in the ISA, it has been shown that different applications show affinity towards different ISAs. Running different parts of an application on their affined ISA can improve its performance significantly. We will call these parts as 'phases'. Multi-cores supporting heterogeneous-ISAs migrate an application from core 'A' to 'B', if the upcoming phase has more affinity to ISA supported by core 'B'.

The heterogeneous-ISA CMPs could enhance the single-threaded performance by up to 40% [4]. However, they still have a few limitations as follows: 1) In Heterogeneous-ISA CMPs, only one of the cores is active for the single-threaded program. The remaining idle cores dissipate the static power unnecessarily, 2) Resources of all idle cores are underutilized, and hence it is not area efficient, 3) The migration cost in heterogeneous-ISA CMPs [16], [6] for migrating the program from one ISA to another is significantly large. To overcome these three issues in heterogeneous-ISA CMPs, we propose a new architecture called Heterogeneous-ISA Dynamic Core (HIDC) along with a new migration mechanism. Taking inspiration from the earlier works in context of dynamic core [13], [10], [15] in single ISA, the proposed HIDC architecture has micro-architectural support for heterogeneous-ISAs within the core. By far, previous proposals have not considered incorporating support for RISC and CISC ISAs within a single core. The core is dynamic in nature as it changes its characteristics according to the ISA under execution. In order to restrain the power budget, the HIDC design is taken such that the peak power of HIDC is kept similar to the Heterogeneous-ISA architecture [16]. The memory hierarchy is shared across both ISAs in HIDC architecture. Therefore, migrating the state from one ISA to another through store-load causes minimum misses. Hence, the cost of migrating the program from one ISA to another ISA is reduced. To optimize performance and energy efficiency, we add a new dynamism in terms of core size. The big core exploits performance, and the small core provides energy-efficient execution in the proposed HIDC architecture.

The dynamic core in the proposed HIDC has dynamism in two forms:

(1) Dynamism for ISA: Core supports multiple ISAs such as ARM and x86 ISAs. This is to harness the benefit of affinity of a program to an ISA, which makes the system performance efficient.
(2) Dynamism in core: Big/Small core configurations are supported in the proposed architecture. This is done to make the system energy efficient.

Due to unified core, the migration cost can also be reduced by migrating the program with some better techniques. If the migration cost is reduced, then we can also take the migration decision for shorter phases then 100 millions (which was taken in previous proposals [16], [3]). To demonstrate the possible benefits from ISA heterogeneity in a single-threaded program at a finer level than 100 million, we executed a big phase 100 million of the *'bzip2'* benchmark from SPEC CPU2006 [9] on ARM (RISC) and x86 (CISC) after dividing it into 10 small phases of 10 million instructions approximately. The execution time for each phase is shown in Fig. 1. It can be seen that only change of phase length can affect the execution time significantly. The performance is better on x86 for phases 3-9 and 1, 2 and 10 for ARM. It is evident that having a bigger phase limits the potential in performance gain in heterogeneous-ISA multi-core architectures. The potential gain that could be extracted by Heterogeneous-ISA multi-core architectures on small phases

is shown in Fig. 2. A performance boost up to 29% could be achieved if the small phases are executed on their most suited ISA core.

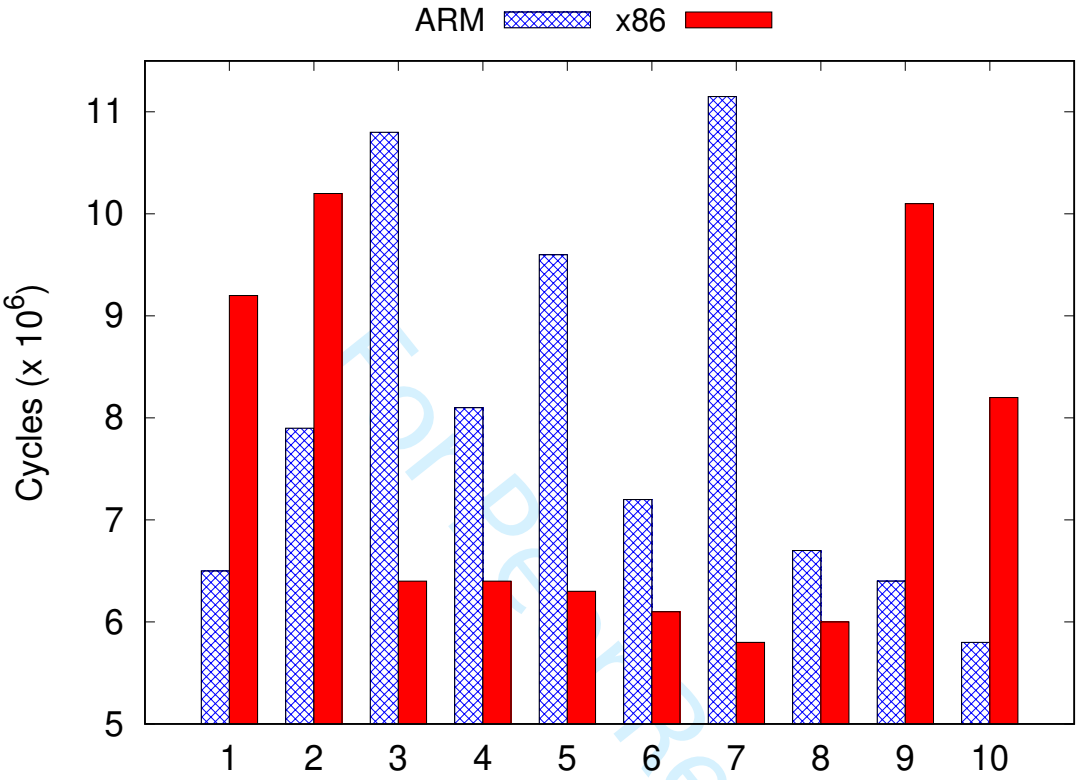

Fig. 1. Execution time of different phases of benchmark bzip2

To reduce the migration cost further, we propose a new simultaneous migration mechanism in HIDC. The proposed migration mechanism reduces the migration overhead by 100× compared to previous implementations by Venkat et al. [16]. For our current study, we use two widely used ISAs, namely ARM and x86, in big (8 wide fetch) and small (4 wide fetch) configuration to show the merit of the proposed scheduling algorithm and the benefits of incorporating support of Heterogeneous-ISAs within the HIDC. However, we believe that the proposed idea is scalable, and HIDC can incorporate support for further different ISAs and different configurations. Hence, in the present work, instead of using four cores to support namely the ARM and x86 ISAs with big and small cores, we use only one hybrid-core which has architectural features to support requirements of both the ISAs for both configurations.

The contributions of paper are as follows:

(1) A *Heterogeneous-ISA Dynamic Core* architecture which incorporates support for different ISAs within a single core. The core also changes dynamically in small and big core configurations. HIDC monitors the run-time requirement of the application running on the core to support changes in the application's ISA affinity by migrating it to most affine ISA.

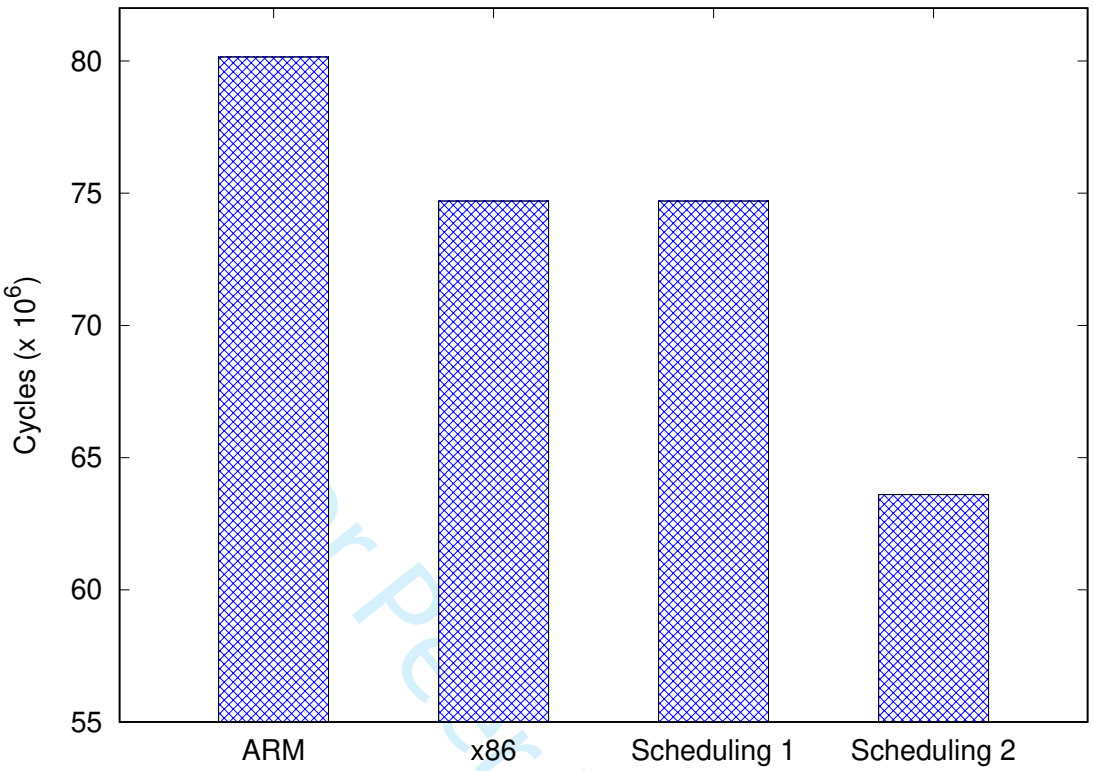

Fig. 2. Execution time when the program is run on ARM, x86, Scheduling on big phase big phase and execution time when each small phase is run on the core most suited to it

(2) Proposes a *linear regression based scheduler* which enables the execution of every phase on its most affine ISA, and micro-architecture among four different cores/ISAs.
(3) *Technique of simultaneous Transformation* for migration of stack-memory from one ISA to another with reduced migration overhead. Hence architecture has *potential of harnessing ISA diversity* at a finer granularity.

The remainder of the paper is organized as follows. Section 2 mentions related work. Section 3 describes the proposed HIDC architecture. Sections 4 and 5 state three aspects of migration namely when to migrate (granularity), which ISA to migrate to (target ISA) and how to migrate (strategy). Results of performance and energy of the HIDC using these methods are presented in Section 6. Finally, the paper is concluded in Section 7 with certain directions for future research. Researchers may also look into decreasing the FAT size using compiler level optimizations.

2 RELATED WORK

2.1 Multi-core architectures

Various proposals [10], [15], [16], [6], [14], [7], [12], [5], [2] have been proposed related to the heterogeneity present in the core and ISA. Kumar et al. [7] [12] have shown that single ISA heterogeneous multi-core architectures can greatly enhance power reduction as compared to homogeneous multicore. Yoshimura et. al [17] proposed an SMT processor

named OROCHI, where they have two different pipeline for ARM and VLIW architectures. Heterogeneity in ISAs in a processor was first introduced by Devuyst et al. [6]. In their proposal, they present that significant performance gain can be achieved if every phase of program runs on its most affine ISA. To gain maximum benefits from in Heterogeneous-ISA CMP, an efficient scheduling algorithm has been proposed by Boran et al. [3, 11]. They have proposed the scheduling which model the execution time and based on the execution time, it schedules the program on best affine ISA core.

2.2 Dynamic Core Architectures

To improve energy efficiency and performance, Dynamic Core architectures [10] have been proposed, where a single big core is morphed into many small cores (or vice-versa) depending on the behavior of the program. Ipek et al. [10] have proposed that energy can be saved if big cores are morphed into small cores based on the program's behavior. Mihai et al. [15] have also proposed to make the core dynamic based on ILP and MLP. Another work in this direction was proposed by Padmanabha et al.[14] where they run the program on Big core first time and remember the traces. On the second run of the same program, the small core executes the program as per the schedule of the big core.

2.3 Migration techniques in multi-core architectures

Once the most affine ISA is determined, the program state has to be migrated with minimum overhead. Devusyt et al. [6] described the memory layout of a program executing on two different ISAs. They observed that most of the memory image could be made consistent for different ISAs without significantly affecting the performance of any of ISA. A new migration scheme was devised by Venkat et al.[16] where the program migrates to the new ISA immediately and the previous ISA's instructions are binary translated to the new ISA until an equivalence point is reached. *An equivalence point is defined as a point where the program run-time state is consistent between the executables of different ISAs.* From the equivalence point onward, it starts executing using the new ISA's instructions. The program's memory state needs to be transformed from one ISA format to the other through migration. This transformation leads to significant migration overhead, which limits the granularity of migration. Our work proposes a *Simultaneous Stack Transformation* method to reduce the migration overhead for better performance benefits.

HIDC is a *dynamic core* as it can switch from one ISA to another as per the run-time requirements of the program.

3 HIDC ARCHITECTURE

The HIDC consists of a single out-of-order dynamic core which supports multiple ISAs. Fig. 3 shows the architecture of HIDC. The execution pipeline is shared among all the ISAs. The pipeline dynamically adjusts resources such as different functional units, decoders, etc., in the core according to the demand of current executing ISA. The decision pertaining to migration is taken by the Migration Engine Controller (MEC).

The MEC monitors multiple micro-architectural parameters and then dynamically decides the affined ISA based on these parameters for the program. The information of the currently executing ISA is stored in a 1-bit register which can be accessed in a manner similar to model-specific registers. We call this bit as Current-ISA Bit (CIB). CIB is accessed by all stages of pipeline except fetch. Fetch stage fetches independent of CIB, as the fetch width remains constant for all ISAs. MEC changes the CIB when the affinity of the program changes and CIB signals the core to switch its ISA. The HIDC architecture contains a superset of the resources that are required by all ISAs. It contains separate decoders for respective ISA, while all other pipeline stages are shared among all ISAs with appropriate resources. In order to save energy, the resources which are not required by the current ISA are clock-gated. In the proposed work, we explore x86 and ARM ISAs. Detailed stage-wise modifications in the out-of-order pipeline architecture are described as follows.

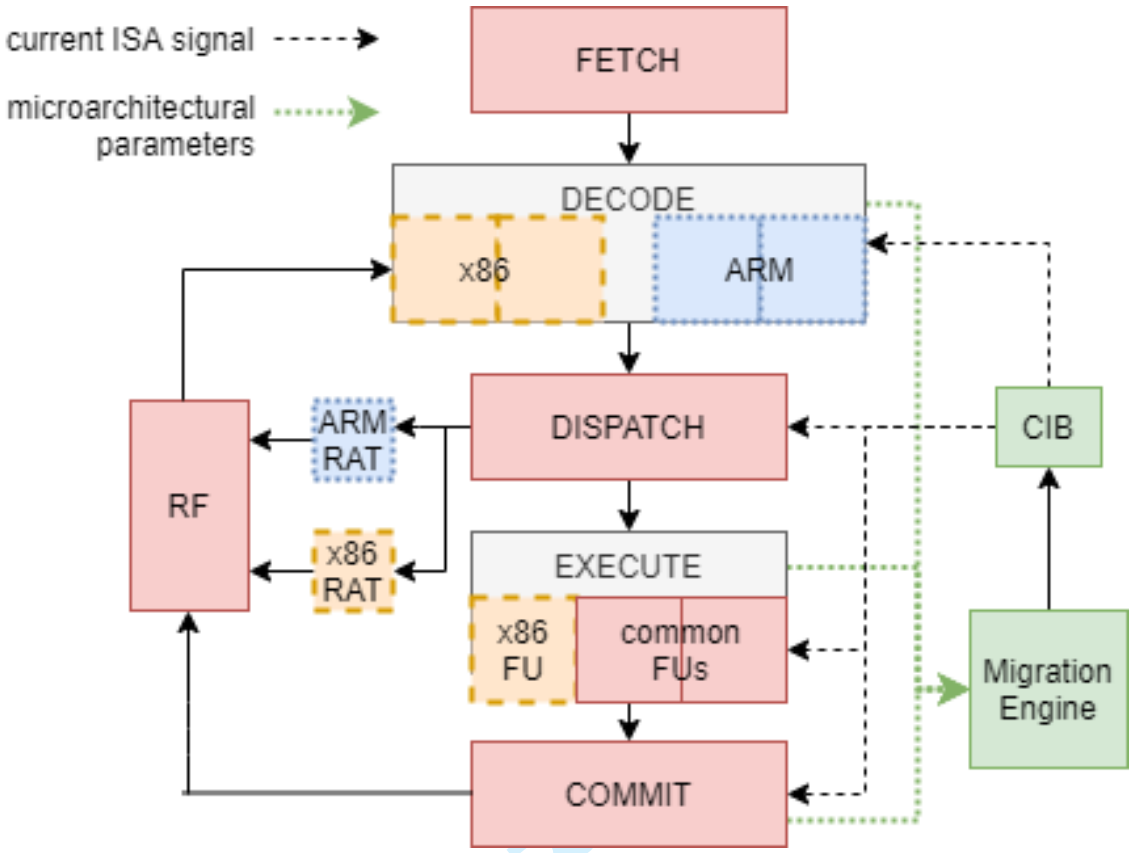

Fig. 3. High level architecture of HIDC pipeline

Fetch: This stage of the pipeline remains unaltered irrespective of the currently executing ISA. The unit fetches constant 64bytes from the i-cache in every cycle. This chunk of instructions may contain variable number of instructions for CISC ISA (e.g., x86) or constant number of instructions for RISC ISA (e.g., ARM).

Decode: HIDC consists of two separate set of decoders for decoding instructions of each ISA. The x86 being a CISC type ISA demands for the decoding of variable length instructions (macro-ops) into RISC-like micro-ops. Hence a complex multi-stage decoder is employed. The decoded micro-ops by the decoder are in the form of control signals and data, that are passed to the next stages of pipeline and have common format irrespective of ISA. The ARM is a RISC ISA, hence it requires decoding of fixed length instructions. The decoder is simple unit with parallel decoding of instructions equal to the width of superscalar pipeline.

Dispatch: The instructions from the decode buffer would be dispatched to the reservation station. Along with that, an entry in the store buffer and ROB is reserved for them. The reservation station would be commonly utilized by either of the ISAs. While dispatching the instruction, the register renaming takes place. Each ISA utilizes different number of architecture registers, hence separate register allocation tables (RAT) are used depending upon the value of CIB. The 64-bit ARM has a total of 32 registers whereas 64-bit x86 contains only 16 registers. However, a monolithic register file is maintained in the architecture.

Parameter	Purpose
L1-I misses	Stalls due to cache misses
L1-D misses	
L2 misses	
Instruction queue full	Execution pipeline stalls
ROB full event	
Store queue full event	
ILP	Parallelism of the executing program
MLP	
Branch mispredictions	Performance impact of branch predictor
Dynamic instruction count	ISA-specific parameters
Float instruction count	

Table 1. Micro-architectural parameters used for migration decision

Execute: The execution stage consists of a common/shared resource pool of all possible functional units required by any ALUs, shifters, multiplier are common between the ISAs. The ISA specific units like SSE extension, floating point units (x86 support) are also present in the pipeline and are clock-gated when they are not used. Therefore, the number and type of functional pipes changes according to the current ISA and Big/Small core configuration for single ISA.

Commit: This stage contains the Re-Order Buffer (ROB), that stores the instructions that are going to update the processor state. The ROB is common for both ISAs. The ROB is modified to keep all the information needed by both ISAs. From the top of ROB, the result is written back to the physical register and the mapping is updated in the corresponding RAT specific to the ISA in execution based on the CIB. Before migration is started, instructions present in the ROB are committed till the equivalence point, and remaining instructions that are fetched post equivalence point and are in ROB gets flushed to avoid any errors due to inconsistent execution. Different store buffer are employed for different ISAs, and the active store buffer would be decided by the CIB. This is done because store policy varies as per the ISA, e.g., x86 has total store order (TSO) and ARM retires in lazy order.

The HIDC is designed subjected to limit the total power consumption equal to Heterogeneous-ISA CMP [16] for single-thread program. Fundamentally HIDC replaces two cores of Heterogeneous-ISA CMP with one dynamic core. This allows HIDC to have double cache size and support higher fetch width compared to one core of Heterogeneous-ISA CMP.

4 SCHEDULING

Scheduling is done for phases with phase length approximately 10 million dynamic instructions with respect to x86 ISA. Each phase starts and ends on equivalence points. The scheduling algorithm is a hardware-based linear regression model and it is dynamic in nature similar to [3]. The proposed linear regression model makes use of eleven micro-architectural parameters shown in Table 1.

Fig. 4 is representing the schematic diagram of the flow of our scheduling model. Step 1-3 in Fig. 4 are for training purpose. The regression models are trained offline and incorporated into the processor with the coefficients of the model stored in special registers. Given a source ISA in execution (ISA_A) and a target ISA (ISA_B), our linear regression model for estimates the number of cycles as follows:

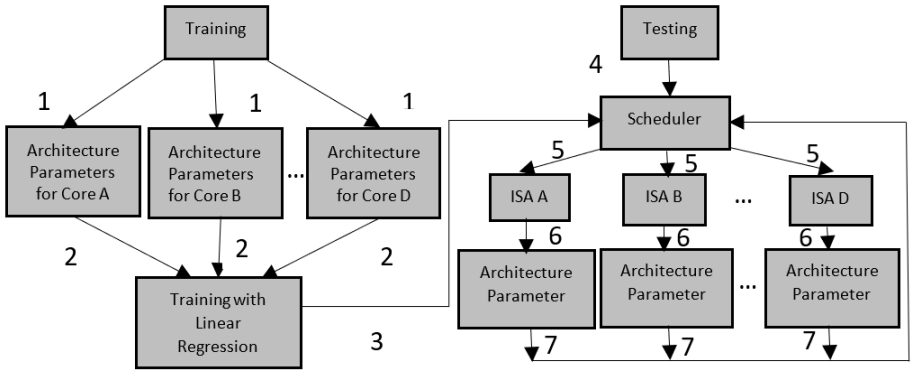

Fig. 4. A schematic representation of the scheduling model

$$\begin{aligned} Cycle_B = & K + a_1.(L1DcacheMiss_A) + a_2.(L1IcacheMiss_A) \\ & + a_3(L2cacheMiss_A) + a_4.(SQFullEvents_A) \\ & + a_5.(ROBFullEvents_A) + a_6.(IQFullEvents_A) \\ & + a_7.(BranchMissPrediction_A) + a_8.(MLP_A) \\ & + a_9.(ILP_A) + a_{10}.(FloatInstruction_A) \\ & + a_{11}.(DynamicInstructionCount_A) \end{aligned} \quad (1)$$

where a_1 to a_{11} are regression coefficients, $Cycle_B$ is number of cycles for ISA_B and K is a constant

During the run time of program, the migration will take place if it makes the system performance better even after considering migration overhead by an entity which we refer to as 'scheduler'. The scheduler estimates the cycle time of the phase for other ISAs by the model which was learnt during training using equation 1. On the basis of micro-architectural parameters of last one million instructions of any phase, the affinity of the next phase is predicted using the designed scheduler and migration will take place if required. Here, we are assuming that the behavior of the program will remain similar during the next 10 million instructions execution. Note that in the Fig. 4, the steps 1-3 are done offline whereas steps 4-7 are done online dynamically.

Please note that, we are migrating to small core if the performance loss is well within the predecided threshold. A limit of performance loss of 5% is kept in this work.

5 MIGRATION IN HIDC

As the affinity of a program changes, the program state has to be migrated efficiently and correctly from current ISA to target ISA. In order to continue smooth execution and avoid runtime errors, the state of all variables and data used by the program have to be kept as expected by the program. At compile time, a single *fat* binary is generated for the program and called as Combined Program Binary (CPB). Instructions are compiled for both ISAs from the source code (including libraries) and stored in the CPB and hence, this binary is common to both ISAs. To perform a faster migration, memory image consistency is achieved by keeping number of objects, their relative order, their sizes identical across the ISAs [6].

Code memory: The code section consistency is maintained by forcing the function's virtual address similar across ISAs. The size of the function is also kept identical by inserting NOP instructions [6].

Global variables: These variables are placed in a common location with the same alignment and size for both the ISAs. Hence these values stored in global memory are not affected as a result of migration.

Heap memory: Heap is allocated by functions like malloc and using the same implementation for both the ISAs. This will ensure that the heap is built in the same way. All heap variables will have the same addresses regardless of the ISA being used at the time of allocation. Hence heap memory does not require migration.

Stack memory: Stack is optimized by the compiler. It optimizes the location (stack-slot) of each variable according to number of registers available and size of the data. This is heavily dependent on the ISA. One approach would be to enforce the same optimizations in the stack for both the ISAs. This may lead to significant deterioration in the program performance. Hence during HIDC migration, the stack locations for each variable are decided by the compiler. The only constraint is to keep the function's stack frame size same. This is achieved by padding with zeros. By keeping stack-frame size same for both ISA, the overlap between stack-frame of two functions is avoided. A mapping between two ISA is generated at compile time between the stacks of the two ISAs. A stack transformation is required during migration. The procedure for stack transformation is described in detail in Section 5.1.

The migration in HIDC is only done at equivalence points. Functions' entry points are considered as equivalent points. HIDC will continue to execute the program in current ISA until an equivalence point is reached. If the affinity remains the same, the program keeps executing normally until the next equivalence point is reached. In case the affinity changes, the program execution is stopped. The necessary transformations are performed to make the memory image consistent with the affined ISA. Thereafter the program uses the affined ISA's compiled code to execute the program till the next equivalence point.

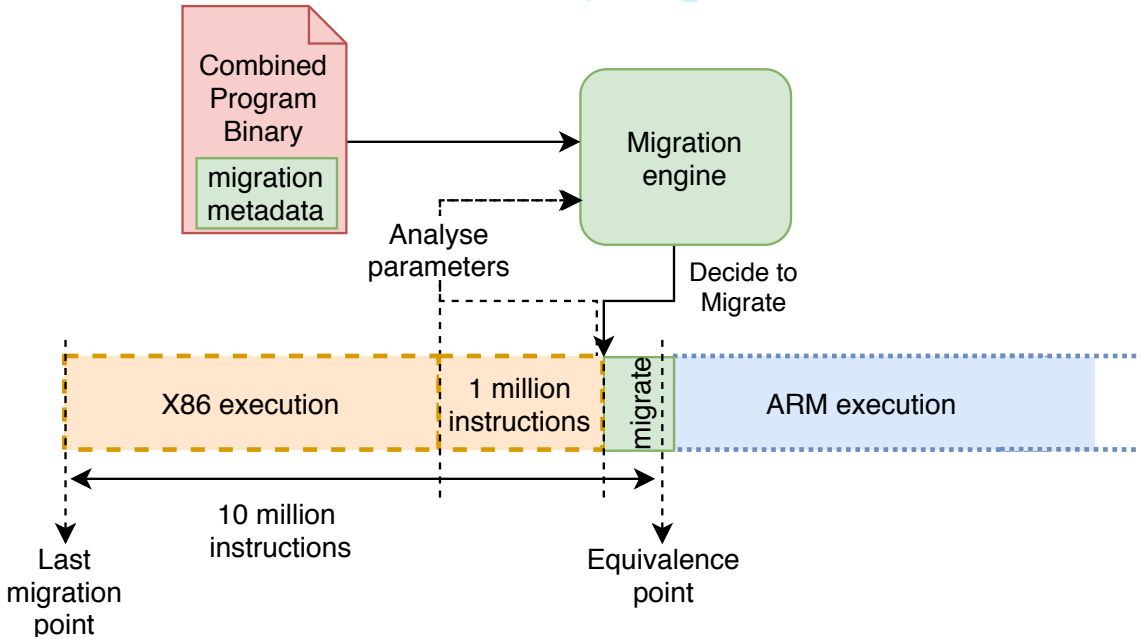

Fig. 5. Flow Chart for Migration

5.1 Inplace stack transformation

To avoid unnecessary migration overheads and for the simplification of the stack transformation, properties like direction of stack growth and endianness are maintained the same across ISAs. In x86 the stack growth is always downward, because push/pop instructions in ISA are implemented to decrement/increment respectively. In ARM, the stack growth is possible in both directions. ARM allows the direction of the stack growth to be fixed as downward stack similar to x86. We impose a restriction on programs compiled for HIDC to always follow downward growing stack convention. It is well supported by ARM using the 'LDMFD/STMFD' instructions and hard-coded into 'PUSH/POP' instructions for x86. Since x86 is little-endian while ARM supports both endianness and the compiler can be restricted to generate only little-endian code. This simplifies the data copies between ISAs as also suggested in [6]. Hence, we keep the stack growth direction and the endianness same for both the ISAs.

LLVM Compiler toolchain is used for stack and pointer analysis. Every function allocates it's own stack-frame on the stack which is generally independent of other function calls. Any access to local variables from outside the context of the function has to be done using pointers. The stack allocation for both ISAs is analysed and a migration mapping for each stack slot in the frame is created. At the time of migration, MEC reads the current Stack Pointer (SP) and Base Pointer (BP) to obtain last executing function's stack frame. MEC also reads the previous BP stored in the stack to determine the location of the previous stack-frame. In this way the MEC loops through the entire stack, one stack-frame at a time. The PC register and return addresses stored on the stack are used to determine the mapping of the stack-frame to the corresponding function. Once the stack frame and function is identified, the stack is modified in-place. This saves the memory-accesses reducing the migration overheads. The in-place modification requires going in a particular sequence such that no data on the stack is overwritten.

Listing 1. In-place migration using two temp variables

```
tmp1 = frame[loc[1]]
frame[loc[1]] = frame[loc[0]]

for i=2..len(loc):
    tmp2 = frame[loc[i]]
    frame[loc[i]] = tmp1
    tmp1 = tmp2
```

Pointer handling: Functions pass the pointers to local stack variables. The value of this pointer, i.e., the address it points to must be changed in case the variable has moved to another location after migration. We will have information of all the pointers by the end of first pass, hence we need a second pass to change the value of pointer with the help of metadata. This has to be done for all functions where this pointer is passed to, directly or indirectly via multi-level calls. Analysis of benchmark programs shows that there are only a few such instances of pointers to local stack variables which need to be handled. Therefore this step is skipped most of the time and does not add to migration time.

All the mappings generated at compile time are added to the program binary as migration metadata and loaded along with the program to be accessible to the migration engine as shown in Fig. 5.

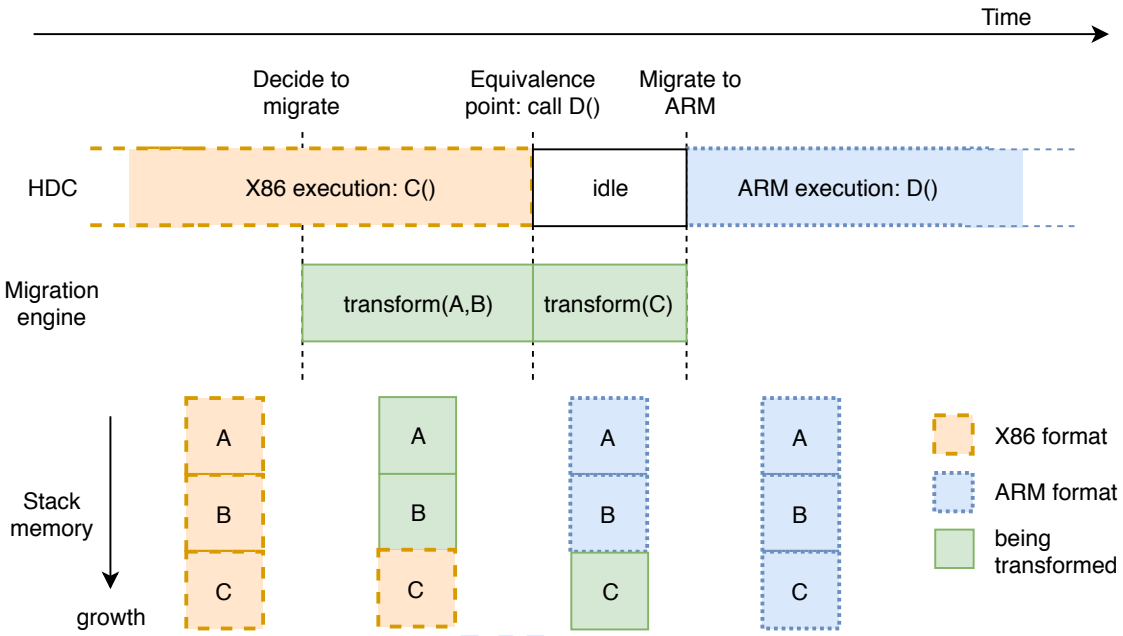

Fig. 6. Simultaneous Transformation

Table 2. Stack slots and location 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]

5.2 Reducing Migration Overhead

To migrate the entire process, every stack frame needs to be transformed individually and all pointers referring to stack memory have to be updated with correct addresses. Many functions can be active at the time of migration decision hence the stack can hold frames for multiple functions. In the previous proposals [6, 16], the migration process starts after the migration decision has been taken by scheduler. This leads to a high cost of migration in the range of few 100 microseconds [16].

Simultaneous transformation: In the proposed method, as shown in Fig. 6, when executing function `C()`, previous frames in the stack are untouched by the program. This allows us to run stack transformation on those frames in parallel to program execution. This reduces the time taken in stack transformation to the time of transforming last function's frame. The total time taken for transformation is similar to the method proposed by Venkat et al. [6], however majority of that time is hidden through execution in parallel to the function `C()`. Parallel stack-transformation in single dynamic core will require extra hardware because the core will be executing the program. This hardware will be described in a

Core	HISACMP		HIDC_Big		HIDC_Small	
	ARM	x86	ARM	x86	ARM	x86
L1 I/D	32 kB		64 kB		32 kB	
L2 Cache	4 MB		4 MB		4 MB	
Fetch Width	4		8		4	
Decode, Dispatch, Issue, Writeback, Commit width	4		8		4	
LQ Size	16	48	48		16	48
SQ Size	16	96	96		16	96
ROB Size	128	256	256		128	256
SIMD	No	Yes	No	Yes	No	Yes

Table 3. Core configurations

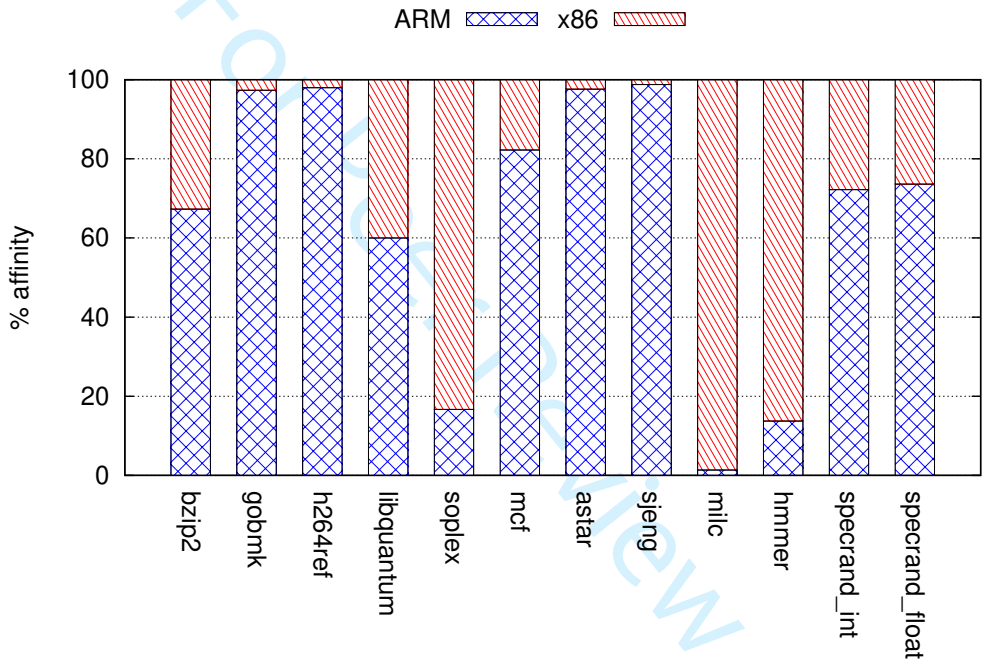

Fig. 7. Percentage ISA affinity for SPEC2006 benchmarks

later section and can be added to the migration engine which makes migration decisions. The cost of migration can now be taken as the time taken for stack-transformation for function `C()`.

Table 2 shows stack slots for both ISAs for function `BZ2_bzDecompressStream` from *bzip2* benchmark. The transformation of different variables/slots is done using the listing 1.

During simultaneous transformation, it is ensured that the data being modified is not in use by the executing the code. However, dirty blocks may exist in the cache for the stack being transformed, which will be in L2 cache if L1 cache is write-through. By connecting the transformation hardware to L2 cache, it is ensured that the most recent version of data is transformed. After simultaneous transformation and transformation of last stack frame, the data in L1

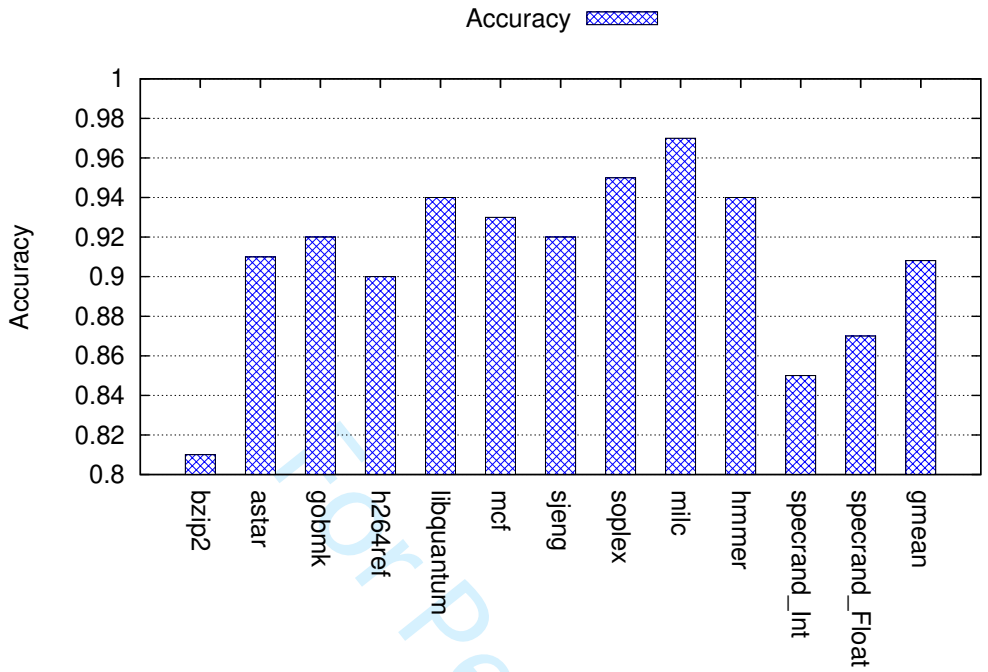

Fig. 8. Migration decision accuracy for different benchmarks by linear regression scheduler

data-cache may be inconsistent with the main memory if L1 is non-inclusive or mostly inclusive. Such case requires flushing of entire L1 data-cache, however the performance penalty is not very high because this is done every 10 million instructions. The scheduler takes 11 parameters as input and produces output. Thus it requires eleven 8-bit registers. It also and requires three 32-bit registers for bias term per scheduler. The scheduler requires twenty one 8-bit multiplier for applying weights to the input parameters along with one 32-bit adder.

6 RESULTS AND ANALYSIS

The experiments were performed using Gem5 [1] simulator with the SPEC CPU2006 [9] benchmark suite. These benchmarks were compiled in gcc for x86 and for ARM they were compiled using the cross compiler. Table 3 shows the detailed configuration of the cores. HIDC is compared with a dual core Heterogeneous-ISA architecture with one x86 core and one ARM core. This configuration is called 'Heterogeneous-ISA CMP' (HISACMP) and serves as a baseline for our work [16]. The power and area analysis was done using McPAT [8]. The migration overheads are obtained by running the migration code at the end of functions that needs to be migrated. We have two ISAs and two separate core types for each ISA. Thus, we have the following set of core configurations namely ARMBig, ARMSmall, x86Big and x86Small respectively. As our simulation methodology involves cross-compilation of benchmarks for ARM, benchmarks namely *gcc, tonto, sphinx3*, could not be cross-compiled successfully. A few other benchmarks namely *perlbench, gamess, cactusADM* could not run to completion after cross-compilation. However, we were able to identify and report results on the applications reported in [16]. We have studied twelve SPEC CPU2006 benchmark applications.

6.1 Program's affinity towards different ISAs

We first identify the affinity of phases within an application towards a different ISAs. Fig. 7 describes the percentage affinity shown towards these four configurations. Since we are focusing on ISA affinity for this section we will discuss the affinity along the lines of ISAs. From Fig. 7, we see that *gobmk, h264ref, sjeng* are more affined towards ARM ISA while *soplex* is affined towards x86. We also observed that some benchmarks like *libquantum, specrand_int* and *specrand_float* showed mixed behavior. This difference is affinity arises as a result of the program behavior, i.e., types of functional units, memory operations, number of instructions, etc. For example floating point intensive benchmarks like *milc, soplex, specrand_float* have affinity towards x86 as it supports floating point operation. The *libquantum* application efficiently utilizes the SIMD support of x86. The high ILP phases of *hmmer, bzip2* are affined towards ARM ISA due to less register pressure in ARM.

6.2 Dynamic Scheduling

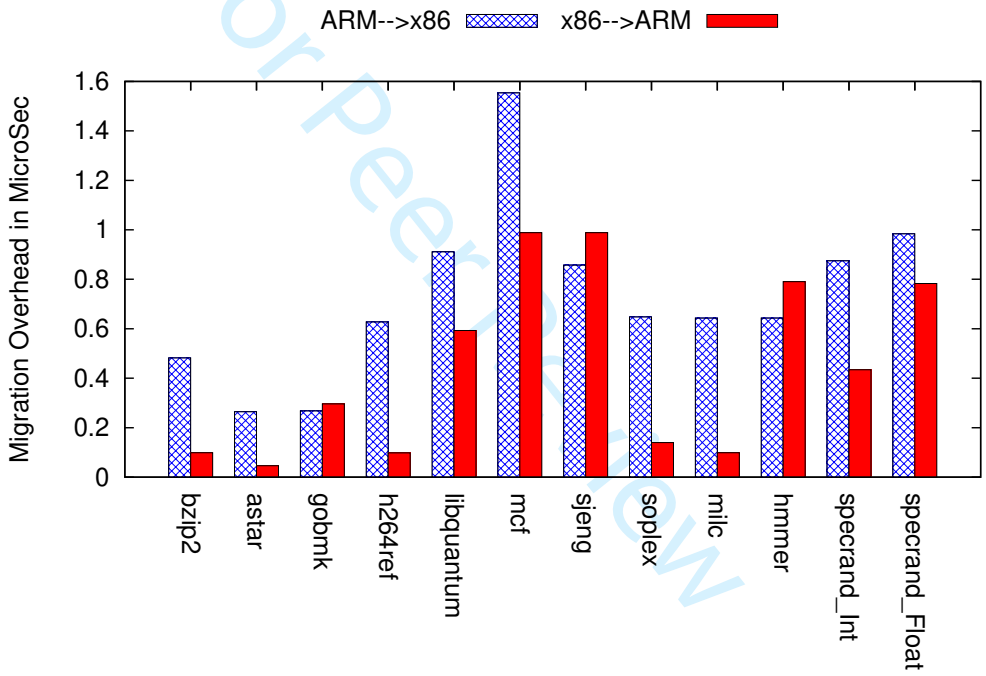

Fig. 9. Migration overhead for simultaneous transformation

In Fig. 8 the accuracy of the migration decision on the test data-set is shown. The efficiency is defined as the number of times we are able to predict the correct affinity for the next phase. The model is trained using a subset of the SPEC CPU2006 benchmark suite. We have used 3000 phases of 10 million instructions. 70% of these phases are used for the training and the rest of the 30% were used for testing. We want to highlight that the model was tested on application which were not used while training. This shows the resilience and generality of our migration framework. The scheduler gives affinity prediction accuracy of 71.4%.

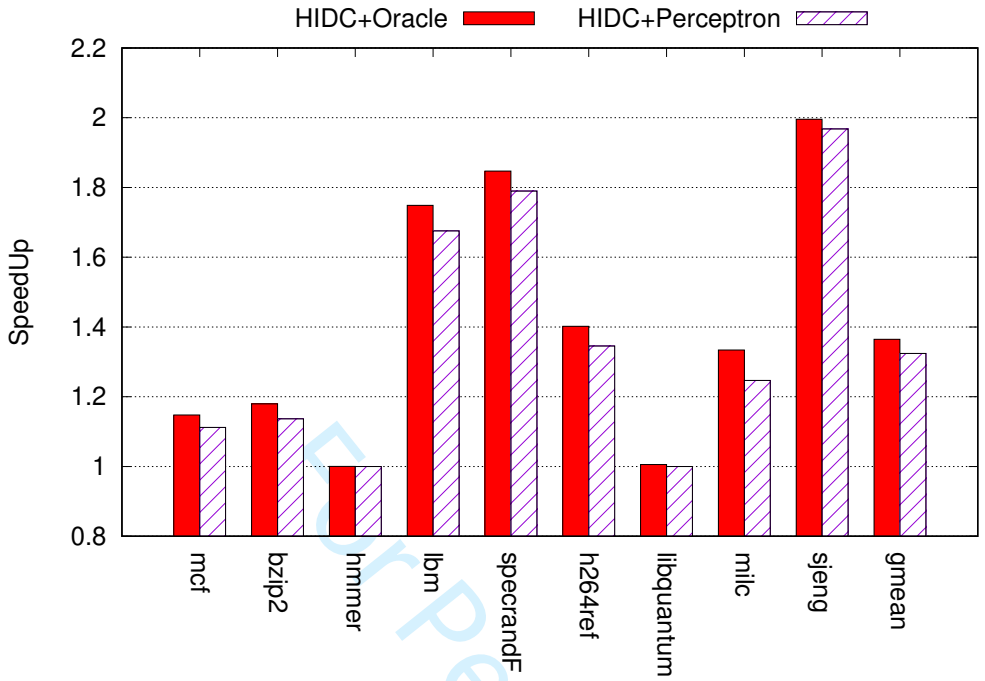

Fig. 10. Performance of HIDC architecture for benchmarks w.r.t. HISACMP

6.3 Migration overhead

The migration overhead lies in the range of 0.1 μsec to 1.5 μsec as shown in Fig. 9. Migration overhead is two orders of magnitude lesser in HIDC compared to [16]. This can be attributed to the hybrid-core, which reduces migration overheads. On top of that, HIDC allows to transform the stack alongside the programs execution. Due to different register pressure, stack to register movement for x86→arm is more, hence more load operations are performed. Whereas arm→x86 have the opposite behaviour which causes more store operations, resulting in more overhead. The migration cost depends on the cost of migration of last function before migration.

6.4 Performance and energy results for HIDC

Fig. 10 shows the speedup of benchmarks executed on HIDC and Heterogeneous-ISA CMP with respect to single ISA x86 core (8-wide fetch). Results are shown for oracle case and linear regression scheduling algorithm. The oracle is a hypothetical case in which each phase runs ideally on its best affine core. Benchmarks *soplex, milc, hmmer* are affined to x86 (shown in Fig. 7), so these benchmarks do not offer much performance gain(w.r.t. x86). However, *astar, gobmk, h264ref, sjeng, mcf* have more affinity towards ARM as shown in Fig. 7. *mcf* is highly affined towards ARM, however its higher migration overhead in ARM → x86 does not let it to migrate on x86 for few of x86 affined phases. *sjeng* is mostly ARM affined so program migrated very rarely. For *bzip2* benchmark, the scheduler could not predict phases with high accuracy. For *soplex_int* and *soplex_float* also the scheduler phase prediction accuracy is not very high. In the worst case, the complete program would run on a single ISA giving speedup almost equal to the ISA to which the program is affine. HIDC has lesser cache misses after migration, since the L1 caches are private in case of Heterogeneous-ISA CMP whereas shared in case of HIDC among all the ISAs. An average of 22.9% speedup over HISACMP is observed when the

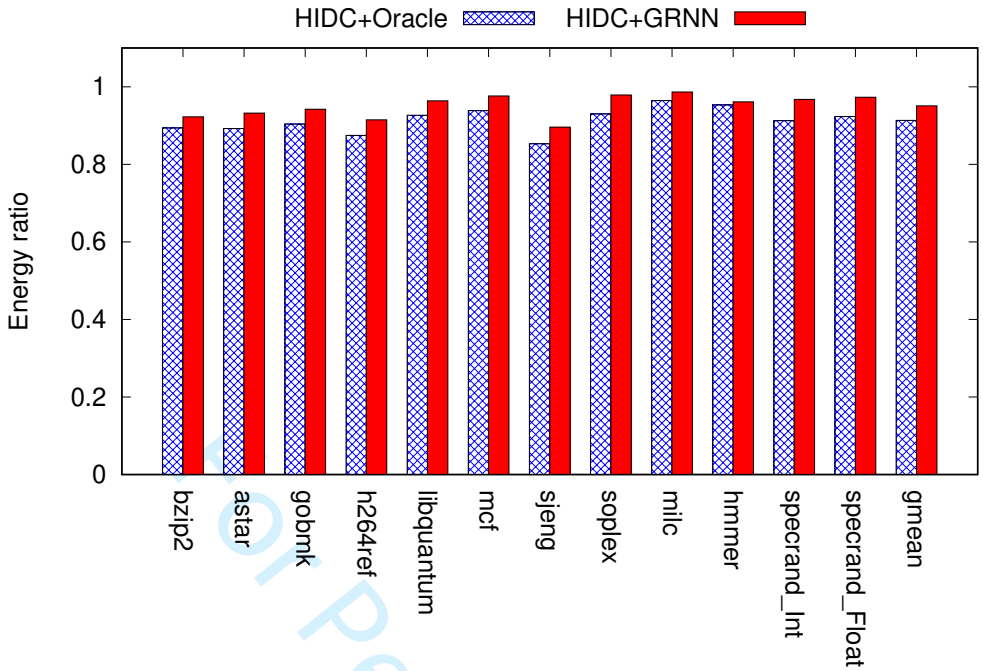

Fig. 11. Energy consumption ratios of HIDC architecture for benchmarks relative to HISACMP

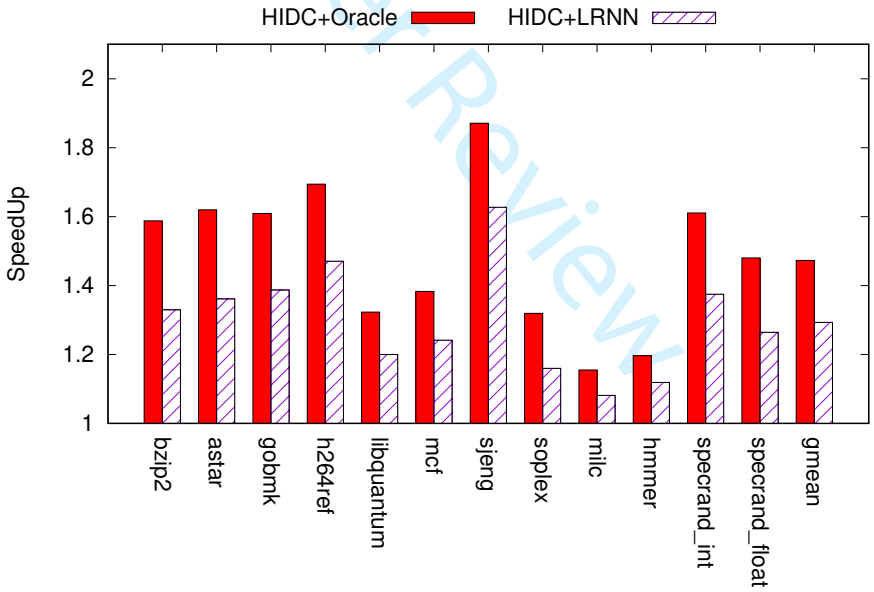

Fig. 12. Performance per Joule of HIDC for SPEC2006 benchmarks

linear regression scheduling algorithm is applied. In case of oracle, HIDC gains speedup of 34.5% relative to HISACMP (with one small core of x86 and one of ARM) and 30.2% relative to x86 (Big) core.

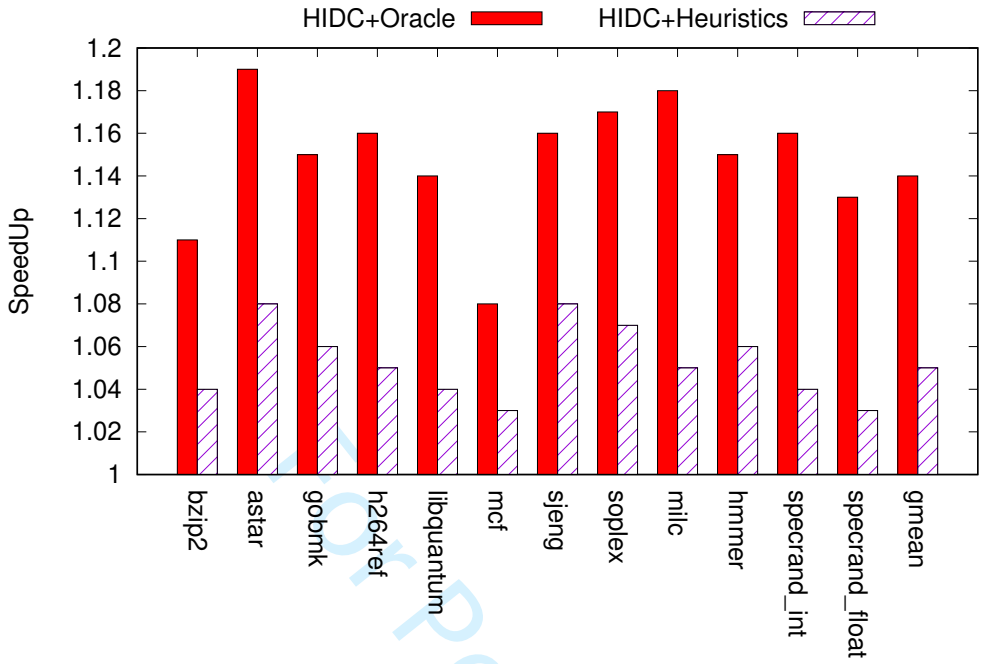

Fig. 13. Performance of HIDC with fine-grained scheduling relative to coarse-grained scheduling

HIDC also performs better in terms of energy consumption. In case of HIDC, the time taken for any phase is lesser than Heterogeneous-ISA CMP and x86 core. It is observed from the results that programs which are affined towards ARM have shown higher energy efficiency. As shown in Fig. 11, about 5.1% reduction is observed in energy consumption by HIDC over HISACMP. In case of oracle 8.9% energy is saved compared HISACMP.

To see the actual gain from a micro-architectural changes, performance per Joule or performance energy ratio (PER) is plotted in Fig. 12. The normalized PER for HIDC is 1.54, which indicates that HIDC will give more than 1.5 times performance for every Joule that is consumed by the processor compared to HISACMP.

6.5 Performance Results for fine-grained scheduling

All the above experiments are done for coarse-grained scheduling for phase length of approximately 10 million dynamic instructions. We have done experiments for fine-grained scheduling as well. Fine-grained scheduling is done for each function similar to [11]. The authors propose a scheduling heuristics where they first find the affinity of each function by sampling method and then store this affinity for next 20 calls of the function. The migration overhead is taken similar to [11], that is, in the range of 5 ns to 95 ns. A performance gain of approximately 13% in oracle case and 4% with scheduling heuristics on top of coarse grained scheduling is achieved as shown in Fig. 13.

6.6 Performance Results for Multi-workload

HIDC is proposed mainly to enhance the performance of single-threaded performance. However, to see the effectiveness of HIDC over multi-threaded programs, we have executed multi-workload benchmarks as well. Results are shown in Fig. 14. The reported speedup is compared to the case when each benchmark is set to run on one of the ISAs (the best performing choice out of two combinations). On an average a speedup of 27.4% is achieved.

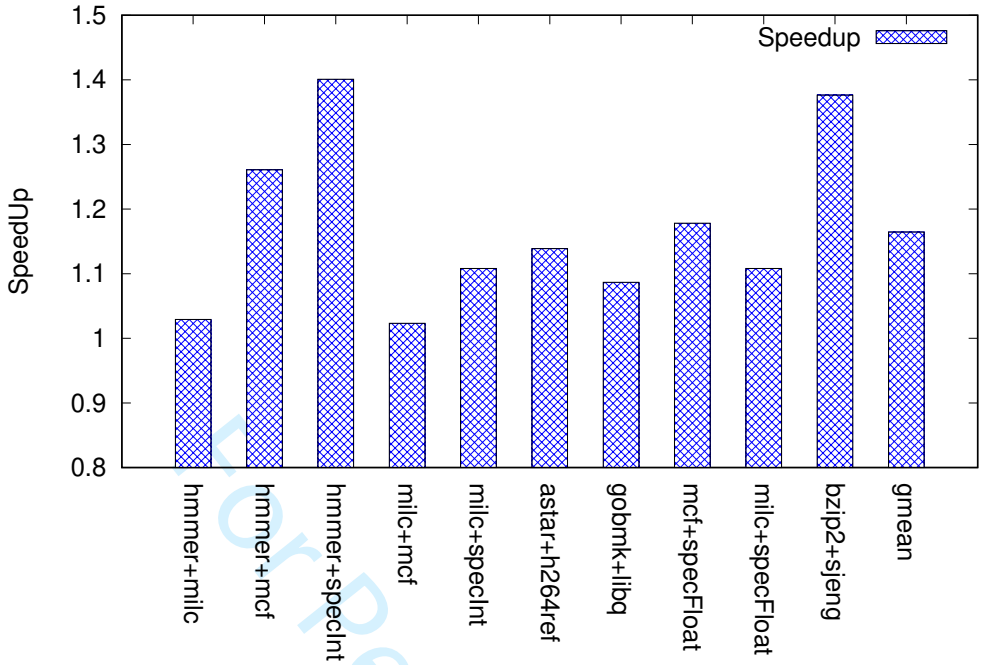

Fig. 14. Performance of multi workload benchmarks

6.7 Area Overhead

The area calculation is done by McPAT [8]. The result shows that the area is reduced by 20% in HIDC architecture compared to Heterogeneous-ISA CMP. The reduction in area has happened because in CMP the resources were dedicated to each core whereas in HIDC many resources are shared in the dynamic core. Our area calculation does not include the migration engine. However, the migration engine can be implemented using simple hardwares which would not add to the area overhead significantly.

7 CONCLUSION

With the increase in computing requirements, processor architecture needs timely modification. Utilizing the ISA affinity present in different phases of even a single program may give a significant performance boost and energy savings over single ISA cores. This work has proposed a novel architecture which supports multiple ISAs in a single core allowing us to improve single-threaded performance along with energy saving by executing program on different ISAs on a dynamic core. The core design does not modify any of the ISAs themselves and only provides methods to migrate between ISAs. To take scheduling decision, linear regression based scheduler is proposed. An improved migration strategy called Simultaneous Transformation reduces migration overhead time by approximately 100× with respect to previous implementations. HIDC has shown an increase in single-threaded performance up to 34% along with about 9% energy savings over Heterogeneous-ISA chip multiprocessor. Future scope lies in proposing better heuristics for fine-level scheduling.

REFERENCES

[1] Nathan Binkert et al. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[2] Emily Blem et al. 2013. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 1–12.

[3] Nirmal Kumar Boran et al. 2019. Performance Modelling and Dynamic Scheduling on Heterogeneous-ISA Multi-core Architectures. In *International Symposium on VLSI Design and Test*. Springer, 702–715.

[4] Nirmal Kumar Boran et al. 2020. Classification based scheduling in Heterogeneous ISA Architectures. In *2020 24th International Symposium on VLSI Design and Test (VDAT)*. IEEE, 1–6.

[5] Robert P Colwell et al. 2000. Computers, complexity, and controversy. *Readings in computer architecture* (2000), 144.

[6] Matthew DeVuyst et al. 2012. Execution Migration in a heterogeneous-ISA Chip Multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/2150976.2151004

[7] R. Kumar et al. 2003. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. *MICRO-36* (2003).

[8] S. Li et al. 2009. 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)*. 469–480.

[9] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* (Sept. 2006). https://doi.org/10.1145/1186736.1186737

[10] Engin Ipek et al. 2007. Core fusion: accommodating software diversity in chip multiprocessors. In *ACM SIGARCH computer architecture news*, Vol. 35. ACM, 186–197.

[11] Nirmal Kumar et al. 2020. Fine-grained Scheduling in Heterogeneous-ISA Architectures. *IEEE Computer Architecture Letters* (2020).

[12] Rakesh Kumar et al. 2004. Single-ISA heterogeneous multi-core architectures for Multithreaded workload performance. *ISCA* (2004).

[13] Andrew Lukefahr et al. Micro, 2012. Composite cores: Pushing heterogeneity into a core. (Micro, 2012), 317–328.

[14] Shruti Padmanabha et al. 2015. Dynamos: dynamic schedule migration for heterogeneous cores. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 322–333.

[15] Mihai Pricopi et al. 2012. Bahurupi: A Polymorphic Heterogeneous Multi-core Architecture. *ACM Trans. Archit. Code Optim.* 8, 4, Article 22 (Jan. 2012), 21 pages. https://doi.org/10.1145/2086696.2086701

[16] Ashish Venkat et al. 2014. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 121–132. https://doi.org/10.1109/ISCA.2014.6853218

[17] Kazuhiro Yoshimura et al. [n. d.]. AN ENERGY EFFICIENT SMT PROCESSOR WITH HETEROGENEOUS INSTRUCTION SET ARCHITECTURES. In *Proceedings of the 9th IASTED International Conference*, Vol. 676. 201.