Architecting High Performance, Energy Efficient & Secured Multi-Core Systems

Submitted in partial fulfilment of the requirments

of the Degree of

Doctor of Philosophy

by

Nirmal Kumar Boran

(Roll No. 154050006)

Supervisors:

Prof Bernard Menezes

Prof. Virendra Singh

Department of Computer Science and Engineering

Indian Institute of Technology Bombay

Mumbai – 400076

AUGUST 2021

Dedicated to

my gurus, my family, and friends

Thesis Approval for Ph.D.

The thesis entitled **Architecting High Performance, Energy Efficient & Secured Multi-Core Systems** is approved for the degree of **Doctor of Philosophy**.

Examiners

- -

- -

- -

Co-supervisor

- -

Supervisor

- -

Chairman

- -

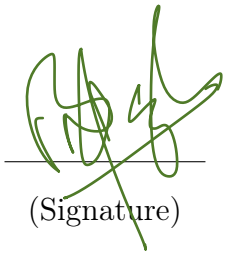
- -

Date:

Place:

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.



(Signature)

Nirmal Kumar Boran

154050006

Date: 30-08-2021

Acknowledgements

It gives me immense pleasure to express my sincere thanks and gratitude to all those who have kindly helped me in carrying out this research work. I express my deepest gratitude to my advisors Prof. Virendra Singh & Prof Bernard Menezes. There guidance has instilled a sense of doing independent research in me. I will remain indebted to them for his guidance in all spheres of life.

Publications

Included in Thesis

Peer reviewed conferences

1. **Nirmal Kumar Boran**, Dinesh Kumar Yadav, and Rishabh Iyer, "Classification based scheduling in Heterogeneous ISA Architectures", *VDAT -2020*, (pp. 1-6) IEEE, https://ieeexplore.ieee.org/document/9190559

2. **Nirmal Kumar Boran**, Dinesh Kumar Yadav, and Rishabh Iyer, "Performance Modelling and Dynamic Scheduling on Heterogeneous-ISA Multi-core Architectures ", *VDAT -2019* (pp. 702-715), Springer, https://link.springer.com/chapter/10.1007/978-981-32-9767-8_58. ***Best paper award***,

3. **Nirmal Kumar Boran**, Rameshwar Prasad Meghwal, Kuldeep Sharma, Binod Kumar, and Virendra Singh. "Performance modelling of heterogeneous ISA multi-core architectures." In 2016 IEEE East-West Design & Test Symposium (EWDTS), pp. 1-4. IEEE, 2016, https://ieeexplore.ieee.org/document/7807641

4. **Nirmal Kumar Boran**, Kenrick Pinto, and Virendra Singh, "On Disabling Prefetcher to Amplify Cache Side Channels", *VDAT 2021, 2021*

5. **Nirmal Kumar Boran**, Pranil Joshi, and Virendra Singh, "PASS-P: Performance And Security Sensitive Dynamic Cache Partitioning", *DATE 2022* [To be submitted]

Journal publications

6. **Nirmal Kumar Boran**, Shubhankit Rathore, Meet Udeshi, and Virendra Singh, "Fine-Grained Scheduling in Heterogeneous-ISA Architectures", *IEEE Computer architecture letter, 2021*, Volume: 20, Issue: 1, pp. 9 - 12, https://ieeexplore.ieee.org/document/9294082

7. **Nirmal Kumar Boran**, Meet Udeshi, Shubhankit Rathore, and Virendra Singh, "HIDC- Heterogeneous ISA Dynamic Core architecture, *Transactions on Computers* [Submitted]

Abstract

Multi-core architectures have the ability to enhance system's throughput, however single threaded programs' execution can not be improved by multi-core architectures. According to Amdahl's law, the performance of an application can be significantly hampered by the limited performance of a single threaded program. Given the ever increasing demand for improved computational capabilities, heterogeneous-ISA multi-core architectures have emerged as a promising alternative to improve single-threaded performance. Such architectures comprise of multiple cores that differ not just in micro-architectural parameters but also in their Instruction Set Architectures (ISAs). These architectures extract previously latent performance gains by executing different phases of the program on the core (and ISA) best suited to it, as opposed to executing the entire program on a single ISA. In such a computing paradigm, maximum performance can be extracted when we ensure that at every point in the program's execution, the program runs on the core best suited to it. If a program is divided into multiple phases, then scheduling of each phase has to be done at run time. While there have been prior works which have looked into heterogeneous ISA multi-core architectures, none of these works have addressed the problem of run time scheduling. To the best of our knowledge, we are the first one in looking into dynamic scheduling in heterogeneous ISA architectures. We have performed scheduling at both coarse-grained and fine-grained levels. In addition to this we also provide a low overhead migration framework which further improves the performance of heterogeneous ISAs. We achieve a speed up of 35% speed up with respect to state-of-art in coarse-grained scheduling. In fine-grained scheduling, we achieve speed up of 22 % on top of state-of-art coarse-grained scheduling.

For single threaded program, only one core of heterogeneous ISA multi-cores is active during the entire execution. The rest of the cores do not contribute anything to the performance of the single threaded program. On top of this, the idle cores keep dissipating static power leading to an overall increase in the power consumption of system. Hence, for single threaded programs having multiple cores leads to under utilization of resources. Also on affinity change within a program from one ISA to other, the program needs to be migrated to the other core which causes large migration overheads. To alleviate the issue of power consumption and migration overhead, we propose Heterogeneous-ISA dynamic core (HIDC) architecture. HIDC integrates support for multiple ISAs into a single dynamic core. HIDC is capable of dynamically changing its working ISA, while the program is under execution. Integrating multiple ISAs on the same core not only improves the energy efficiency but also significantly improves the performance of a single-threaded program. Since, HIDC is a single heterogeneous core, unlike multi-core heterogeneous systems, the migration overheads are reduced. The migration overhead is reduced 100× with respect to state-of-art. We gain a speed up of 30 % with respect to state-of-art. The energy consumption is also reduced by 15% for HIDC architectures. With respect to state-of-art [99], multi-threaded programs achieve a performance gain of 16.2% in HIDC architectures.

In multi-core architectures the last level cache is shared. Though HIDC supports multiple ISAs in single core, however to improve the throughput, multiple HIDC can be used in a chip. Shared caches are vulnerable to side channel attacks. In this work, we show a side channel attack that can be mounted by disabling the prefetchers in multi-core systems. In addition to this, we also reveal the vulnerability in the utility based cache partitioning protocol. To mitigate side channel attacks, we propose PASS-P (Performance And Security Sensitive Protocol) to improve robustness against side-channel attacks. We are proposing a new cache replacement technique to overcome the performance concerns. The proposed work enhances the hardware security of the last level cache. With PASS-P, a performance gain of 7.2% relative to static partitioning is achieved, which is less then by just 0.35% relative to UCP[81].

Overall in this thesis, we investigate three important aspects in multi-core systems:

1. Improving Single-thread performance of the system.

2. Improving energy efficiency of the system.

3. Providing better hardware security.

Keywords

Multi-core, Single-threaded program, Heterogeneous ISA, Dynamic migration, Dynamic scheduling, Scheduling technique, Coarse-grained scheduling, Fine-grained scheduling, Linear regression neural network, General regression neural network, Perceptron, Migration overhead, Dynamic core, Side-channel attacks, Prime-probe attack, Flush-reload attack, Cache partitioning, Static partitioning, Prefetchers, Stride prefetchers

Notations and Abbreviations

UCP	:	Utility based Cache Partitioning	HIDC	:	Heterogeneous ISA Dynamic Core
ISA	:	Instruction Set Architecture	CMP	:	Chip Multiprocessor
LRNN	:	Linear Regression Neural Network	ARF	:	Architectural register file
GRNN	:	General Regression Neural Network	PRF	:	Physical register file
HISACMP	:	Heterogeneous ISA Chip Multi Processor	TLP	:	Thread level parallelism
PASS-P	:	Performance And Security Sensitive Partitioning	ILP	:	Instruction level parallelism
DC	:	Dynamic core	HeMC	:	Heterogeneous multi-cores
HLL	:	High level language	HoMC	:	Homogeneous Multi-cores
RISC	:	Reduced Instruction Set Computer	ROP	:	Return oriented programs
CISC	:	Complex Instruction Set Computer	AES	:	Advanced Encryption Standard
IQ	:	Instruction Queue	RSA	:	Rivest, Shamir, Adleman
ROB	:	Reorder buffer	MSHR	:	Miss Status handle registers
LQ	:	Load Queue	SIMD	:	Single instruction multi data
SQ	:	Store Queue	PC	:	Program counter
DCP	:	Dynamic cache partitioning	SecDCP	:	Secured Dynamic cache partitioning

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The past decades have witnessed very competitive dynamism in the advancements occurring in the field of microprocessor technology. Technological advancements and research have been focused on enhancing processor performance along with improving energy efficiency. Still, the demands for high-performance computing and energy efficiency are increasing rapidly nowadays with the increase in the amount of computations required in various applications.

During last century, since the invention of processors, researchers could achieve approximately 50% performance gain every year. This was achieved due to both technology improvements and micro-architectural innovations. The focus in the past few decades was mainly on making the core faster. “The faster, the better" was the tagline during this time. By the end of the century, power dissipation reached the limit of the air-cooled systems. Also, power density became equivalent to the power density of the hot plate. Hence the frequency of the system could not be increased further due to power density constraints, ending Dennard scaling. Therefore, the performance could be improved until saturation of the frequency scaling. This limitation is generally referred as power wall.

As the designers hit the power wall [76], to cater the need of current applications, the designers had to shift to multi-core architectures as making the core faster was not feasible. Multi-core systems have multiple cores on a chip. These systems can

execute multiple threads simultaneously. Thus, multi-core architectures exploit thread-level parallelism (TLP), enhancing system throughput. This led to change in the tagline from " The faster, the better" to "The more, the better".

Since single-threaded applications employ only one core and the remaining cores are not utilized. Therefore, in multi-core systems, throughput of the system is limited by single-threaded performance [5]. Many applications such as python interpreter, LAME: the open-source audio encode, Gem5, recursive programs mostly run as single-threaded application. So improving the *performance of single-threaded programs* is the area of prime interest. As the number of cores increases, the energy consumption of the system also increases. Therefore, the energy efficiency of a system still remains a problem.

Multi-core architectures share different resources among the cores to provide better energy efficiency. This may enable information leakage. Therefore, hardware security also become another concern in multi-core architectures. Thus, in this thesis, we focus on the following three main factors.

1. Single-threaded performance
2. Energy efficiency
3. Hardware security

1.1 Single-threaded performance

In multi-core architectures, various innovations have been done to improve the single-threaded performance (for a single core), such as branch predictions, prefetcher, value predictions, cache replacement policies etc. Venkat et al. [99] proposed that a single-threaded application's performance can be further increased when the affinity of a program towards an ISA is exploited. They found that the affinity of a program towards an ISA arises due to multiple factors such as code density, register pressure, dynamic instruction count, floating point and SIMD (Single instruction multiple data) support. To demonstrate the ISA affinity exhibited by programs and the proposed benefits of

exploiting the same for single-threaded performance, we divide the *'astar'* benchmark from SPEC CPU2006 [39] into 15 phases (phase length is 10 million instructions) and execute it on ARM (RISC) and x86 (CISC) cores. Different phases have an affinity towards different ISAs. Figure 1.1 shows the execution time of each phase on both cores. The results show that by only changing the ISA, there is sufficient variability across each phase's execution time. Further, neither core is dominant throughout, with x86 performing considerably better on phases 1, 7-14 and ARM performing considerably better on phases 2-6 and 15. Clearly, running the entire program solely on any one of these cores does not lead to optimal performance.

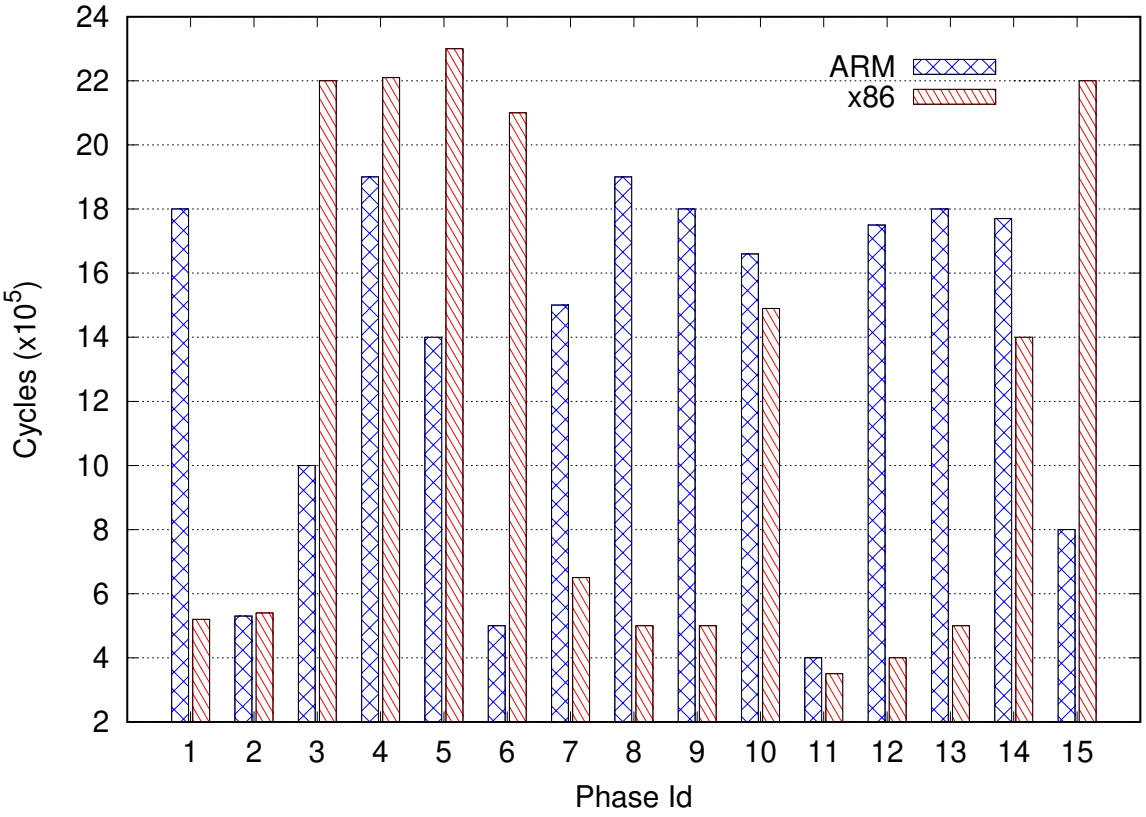


Figure 1.1: Execution time of different phases of benchmark astar

Figure 1.2 shows the latent performance benefits that can be extracted by Heterogeneous ISA multi-core architectures (accurate-migrations). When running each phase of the program on the core (and ISA) it is most suited to, the program's performance can

increase by up to 39%. On the other hand, if the scheduling is flawed and mispredicts where to execute each phase of the program, it can lead to unacceptable performance deterioration of up to 26%. Clearly, maximum performance benefits are achieved only when the program is migrated across the cores at the right time. Venkat et al.[99], worked on showing the feasibility of prospective gains; however, the focus was NOT given on *how to achieve* that. Naturally, this requires a prediction and scheduling mechanism, which decides when this migration is supposed to occur. The best affine core for each phase of a program can be predicted using some mathematical models. Given the need for an accurate cross-ISA performance modelling technique, we have solved this problem at a coarse-grained and fine-grained level. We have done all the studies on two commonly used ISAs: RISC type ARM and CISC type x86. However, the proposed work can be used as it is on any number of ISAs on a chip. All the experiments are done on SPEC CPU2006 [39] benchmarks. Please be noted that we have done the scheduling at two different levels coarse-grained, and fine-grained levels. These are discussed in sections 1.1.1 and 1.1.2.

1.1.1 Coarse grained scheduling

In coarse-grained scheduling, a phase of 100 million consecutive instructions is taken into consideration for scheduling purpose. The affinity of each phase is predicted, and the program is scheduled on that ISA. The migration decision is taken on the equivalence point (a point where the memory state is the same across the executable of both ISAs) [28]. If there is a need for migration, then the program's context switching takes place from one core to another core.

Since the program runs on only one of the ISA, we know the execution time and behaviour of the program on the ISA it is running. However, we do not have idea of the program's behaviour on the ISA on which it is not running. Hence, to take the scheduling decision, we need to estimate the program's behaviour on other ISA as well. To achieve this, we propose the following two scheduling techniques to execute the application on affined ISA based on the micro-architectural parameters.

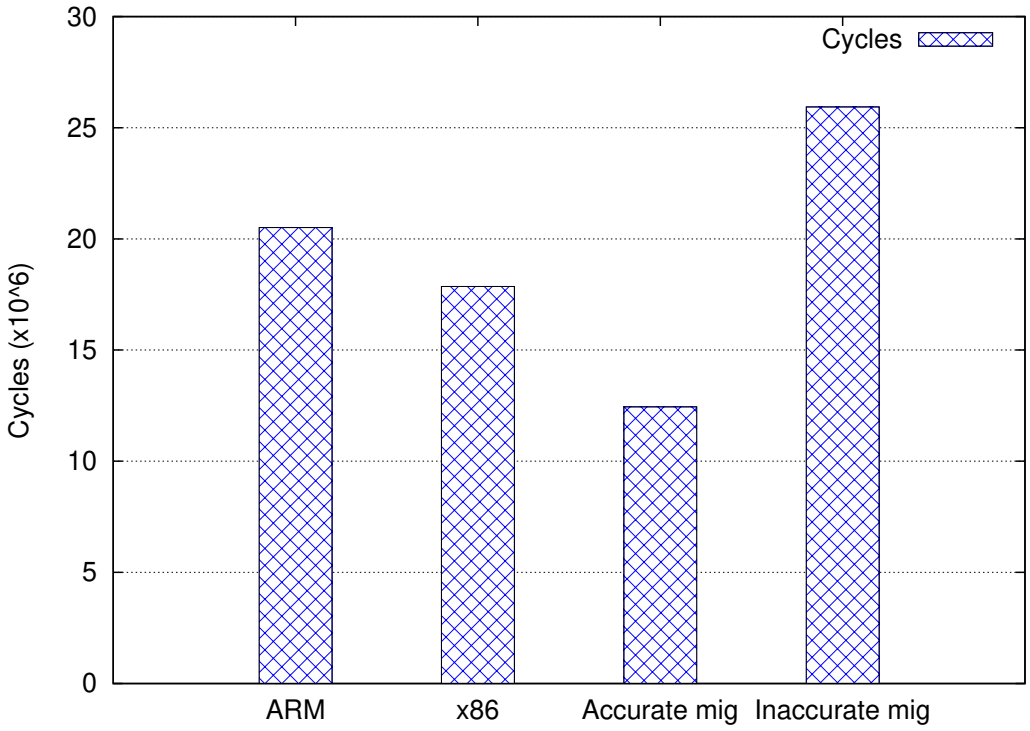


Figure 1.2: Execution time when the program is run on ARM, x86 and execution time when each phase is run on the core most suited to it ("Accurate mig") and least suited to it ("Inaccurate mig")

1. Performance based scheduler

2. Classification based scheduler

1. Performance based scheduler

In this work, the scheduler is designed based on the performance estimation of the cores. *Performance estimation is defined as estimating a program's execution time on a core on which it is not running.* Once the performance estimation is done, the scheduling of the application is done on the ISA, which has a lesser execution time after considering migration overhead.

For performance estimation, we trained different models which learn the relationship among various micro-architectural parameters of target ISA (ISA on which program is not running) and source ISA (ISA on which program is running). We also model the

relationship between micro-architectural parameters and corresponding execution time for both ISAs. All these models are trained offline. Using these trained models, the system estimates the execution time of target ISA at run-time by first predicting the micro-architecture parameters of the target ISA from the source ISA's.

Based on this predicted execution time, the affinity of the phase is decided by the scheduler. The scheduling decision is taken after considering the migration overhead due to context switching. The accuracy of proposed scheduler is 82.9%, which translated to a performance benefit of 29.6% relative to state-of-art.

2. Classification based scheduling

As our interest lies in obtaining a dynamic mapping of the program phase to an ISA core, we approached the scheduling problem as a classification problem in the second proposal of scheduling. We achieve a performance gain of 35.7% relative to x86 ISA, which is 6.1% more than the state-of-art i.e. performance based schedulers [13].

In all the coarse-grained scheduling mechanisms, affinity is identified for a phase of 100 million dynamic instructions.

1.1.2 Finer-grained scheduling

For long phases, the heterogeneity can not be exploited effectively, as the program behaviour within a phase varies. This work shows that shorter phases can give better performance if we reduce migration overhead (which is up to 100 µs in existing schemes [28], [99]). To exploit the program's heterogeneity more effectively, we introduce a fine-grained function-wise scheduling technique, in which every function is proposed to be scheduled to its most affined ISA. Function-based scheduling reduces migration overhead as only function arguments have to be transformed from one ISA format to the other in this scheduling technique. This avoids a complete stack transformation which was required in previous proposals [28], [99]. We have reduced the migration overhead to a range of {0.05 µs - 0.15 µs}, that is three orders of magnitude less compared to previous proposals [28], [99]. With reduced migration overhead and making the scheduling

decision on function levels, a performance gain of 65% relative to x86 can be achieved. We have proposed a heuristics for finding the affinity of a function at run-time. The performance gain with the proposed heuristics is 34.8% relative to x86. The proposed fine grained function wise scheduling achieved 22.9% in comparison to the state-of-art [15]. We have also shown that with the help of this scheduling algorithm, performance increases for the multi-threaded programs by 15.8% compared to state-of-art [99].

Now that we have looked into improving the performance of single-threaded applications using three different techniques. In the next section, an energy efficient architecture is proposed.

1.2 Energy efficient heterogeneous ISA dynamic core

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.

To enhance single-threaded performance along with multi-thread performance, dynamic cores are proposed [59], [43], [79] in multi-core systems. These cores change their architecture dynamically to adapt to the program requirement. However, all the cores in such architectures support a single ISA. Heterogeneous ISA architectures emerged as a good alternative to enhance single-threaded performance (and multi-threaded programs as well). If any program is divided into multiple parts and each part is scheduled on most affined ISA, the significant performance gain is achieved. For any single-threaded program, only one core is active in these architectures. Due to this following are the limitations in these architectures.

1. The resources of idle cores are not utilized optimally.

2. All idle cores dissipate static power continuously.

3. On affinity change of a program, the context switching is done from one core to other core. This leads to larger context switching (migration) overhead and leads to reduction in performance.

To overcome these limitations, under peak power-constrained model we propose an architecture called HIDC (Heterogeneous ISA Dynamic Core architecture). This architecture has a single core that supports multiple ISAs. This core changes its ISA support and architecture dynamically. Since the core is capable of supporting multiple ISAs, it contains the union of functionalities of multiple ISAs, which it supports. We studied four configurations (big x86, big ARM, small x86, small ARM). Bigger configurations are taken to improve the performance of the system. Smaller configurations are taken to improve the system's energy efficiency further. In the proposed architecture, all the level of caches are shared among all the ISAs, hence the migration cost is reduced. The scheduling is done at the coarse-grained and fine-grained level. We have reduced the migration overhead for coarse-grained level scheduling significantly with a "simultaneous migration mechanism". The migration cost is reduced by 100× times compared to state-of-art [99]. On coarse-grained scheduling, we achieve a performance gain of 30.8% relative to state-of-art heterogeneous ISA architectures [99]. Since we could reduce the static power dissipation in HIDC, the energy consumption is reduced by 15.4% relative to [99]. These architectures give energy efficiency (Performance to energy ratio) of such architectures up to 54%. For fine-grained level scheduling, the performance gain of 5.2% is achieved on top of coarse-grained scheduling. Multi-threaded programs achieve a performance gain of 16.2% relative to state-of-art [99].

HIDC architectures help not only in improving the energy efficiency but also gives higher performance. Given that we have looked into performance and energy-efficient architectures in Section 1.1 and Section 1.2 respectively. In the next section, we discuss hardware security in the context of multi-core architectures.

1.3 Security in multi-core architectures

In all the multi-core architectures generally last level cache (LLC) is shared between the cores. This causes a security vulnerability in the form of side-channel attacks. Side channel attacks look at the unintended hardware footprints of the victim program to infer information. The different modes in which an attack can be mounted are referred to as 'channels'. These channels include analysis of execution time, memory accesses, power consumption and electromagnetic radiation of the hardware resources being used by the victim program. The attacker can infer the victim's memory access patterns by monitoring the cache lines in the shared cache. This is done by forcing collisions with the victim and making accesses to that alias with the same lines. This is possible as all the cores can access shared libraries. Since the memory access patterns in most security protocols are dependent on the private key, leaking information about these patterns may compromise the key. In any multi-core system, scopes of side-channel attacks always exit when multiple processes are running on different cores. Flush+Reload [107] and Prime+Probe [58] are common side-channel attacks that use differential cache access timing-analysis on lines modified by the victim process. By the timing-analysis on victim's cache lines, the attacker can deduce the addresses accessed by the victim.

Researchers have proposed various mitigation methods for side-channel attacks. Prefetchers are claimed [94], [101] to be mitigate the side-channel attacks. They have shown that prefetcher works adversely for the attacker in timing-based side-channel attacks. However, we show the feasibility of attack even in the presence of prefetchers by disabling them. Another source for the attack in multi-core architectures is partitioning the cache dynamically. The last level cache of multi-cores is dynamically partitioned for performance enhancement. In this thesis, we will show that dynamic partitioning also increases the vulnerability in multi-core architectures.

1.3.1 Disabling prefetchers to attack side-channel attack

A key feature in side-channel attacks is their susceptibility to noise. Data fetched into the cache but not used by the victim, will be observed as noise. A source of noise for an attacker is hardware prefetchers. Since the prefetcher brings data into the cache, it interferes adversely with the side-channel. Only the cache accesses corresponding to the victim are of interest to the attacker. However, the attacker will observe data fudged by the prefetcher and cannot distinguish them from accesses made by the victim. The true source of the access (whether it is due to victim or prefetcher) cannot be deciphered. Thus, in addition to boosting performance, prefetchers act as good mitigators of cache side channels. In this proposal, we claim that hardware prefetching can be circumvented when performing an attack. We present an attack that is successfully capable of disabling the prefetcher and opening up cache-side channel vulnerabilities. In the presence of a prefetcher, we extracted the private key with only 21% increment in the time taken compared to a noiseless system having no prefetcher.

Thus, prefechers can not be relied upon for mitigation of side-channel attacks. The shared cache is main reason for these attacks. Hence, we propose a defensive mechanism which is based on cache-partitioning protocols.

1.3.2 PASS-P: Performance and security sensitive partitioning protocol

Cache partitioning protocols were proposed to avoid interference between multiple threads. One of the cache partitioning mechanism is static cache partitioning [74], which avoids side-channel attacks also. It partitions the cache statically. In such partitioning, no sharing of resources is allowed, so side-channel attacks are mitigated. However, since the partitioning is done statically and processes may need more or less number of lines than allocated, this partitioning comes with performance degradation.

In shared last level cache, to improve the performance, utility-cache based partitioning (UCP) [81] was proposed. In this mechanism, the cache is dynamically partitioned on

the basis of the utility of cache for each application. This improves the performance by 8% over static partitioning. However, we show in this work that UCP is vulnerable to side-channel attacks.

To overcome the security and performance concerns in multi-core architectures, we propose PASS-P (Performance And Security Sensitive Partitioning) mechanism, which gives better security (similar to static partitioning) and performance almost equal to utility-based cache partitioning. We propose to invalidate all the cache lines which are re-allocated among the processes. We are also proposing a modified cache replacement algorithm that reduces the number of write-backs in systems. This helps in improving performance. The proposed work helps in enhancing the hardware security of the last level cache. With PASS-P, we are able to achieve a performance gain of 7.2% relative to static partitioning, which is less then by just 0.35% relative to UCP.

1.4 Contribution of the thesis

In this thesis we are targeting the performance, energy and security aspects of processor. Heterogeneous ISA architectures are utilized for performance improvement. IN these architectures, no one explored the scheduling of a program among different ISAs at run time previously. We have proposed to schedule a program at two different levels: coarse-grained and fine-grained. However, we observe that for single threaded programs, only one of the core is active and remaining cores dissipate power unnecessarily. To overcome this, we propose a new architecture HIDC. HIDC has a dynamic core and it supports multiples ISAs. The migration overhead is reduced in these architectures along with energy consumption. To see the security concerns of multi-core architecture, we first show the feasibility of attack in presence of prefetchers and then propose a new mitigation methodology called PASS_P which is better in terms of security as well as performance.

1.5 Overview of thesis

The rest of thesis is organized as follows. Chapter 2 gives background on multi-core architectures. It discusses dynamic core architectures, scheduling in multi-core architectures. Heterogeneous ISA multi-core architectures have also been detailed. It also briefs about return oriented programming (ROP) attacks. This chapter also discusses on side-channel attacks and mitigation methods for these attacks.

Chapter 3 discusses the proposed scheduling methods in heterogeneous ISA architectures at coarse-grained level. A phase length of 100 million dynamic instruction count has been taken into consideration in this chapter. This has been done by performance estimation method and classification based scheduling mechanism. Migration overhead by [99] has been used in this work. Creation of fat binary by [54] has been used for creating similar fat binary which supports multiple ISAs.

Chapter 4 explains the scheduling in heterogeneous ISA architectures at finer level. We have proposed to the scheduling on each function. We also show that the migration overhead is reduced by 1000 x times in this work. We have also shown this mechanism better in ROP attacks.

Chapter 5 details a new architecture heterogeneous ISA dynamic core (HIDC) with one core for multiple ISAs. It also outlines the changes which need to be done to make HIDC core. We have also discussed a method to reduce the migration overhead in heterogeneous ISAs.

Chapter 6 discusses the potential side channel attacks in multi-core and hence in heterogeneous ISA architectures. This chapter also discusses that prefetchers can not be taken as mitigators in multi-core architectures. We have also shown the attack feasibility in widely used utility based cache partitioning mechanism. We have detailed a security mechanism called PASS-P for access based side channel attacks.

Chapter 7 concludes the thesis with possible future research directions.

– * – * –

Chapter 2

Previous Work

This chapter details the background information related to the thesis. Section 2.1 discusses the multi-core architectures and scheduling in it. Section 2.2 talks about security in multi-core architectures.

2.1 Multi-core architectures

Due to saturation of frequency scaling, improving performance of single-threaded workloads required addition of more and more resources to the core which was infeasible due to the limited power budget. The focus thus shifted from uni-core to multi-core architectures to dynamically migrate workloads between them [99]. Multi-core architectures are classified into homogeneous, heterogeneous, and dynamic cores. The focus of heterogeneous multi-cores (HeMC) [51] [52] architectures has been on exploiting Thread-Level Parallelism (TLP) & Instruction Level Parallelism(ILP), whereas Homogeneous multi-core (HoMC) architectures [91] only focus on TLP. Few proposals on Dynamic core (DC) architectures like Core Fusion [43] focus on improving energy efficiency by dynamically changing core configurations based on requirement.

Computing power demand has been increasing in computers. Moore's law suggests that increased computation in single core machine have problems in terms of thermal constraints, power dissipation etc. Multi-core architectures can exploit the thread level

parallelism. Multi-core architectures have less complex system and hence these architectures provide solutions to these problems. Multi-core architectures turned to be very useful in multimedia applications, high graphic games, embedded systems. A plethora of literature is available regarding multi-core architecture. We attempt to divide this section into three categories. First one is multi-core architectures which is further divided into homogeneous cores with same ISA, heterogeneous cores with same ISA and heterogeneous-ISA multi-core architectures. Second one explores performance modelling in heterogeneous architectures.

2.1.1 Single ISA homogeneous multi-core architectures

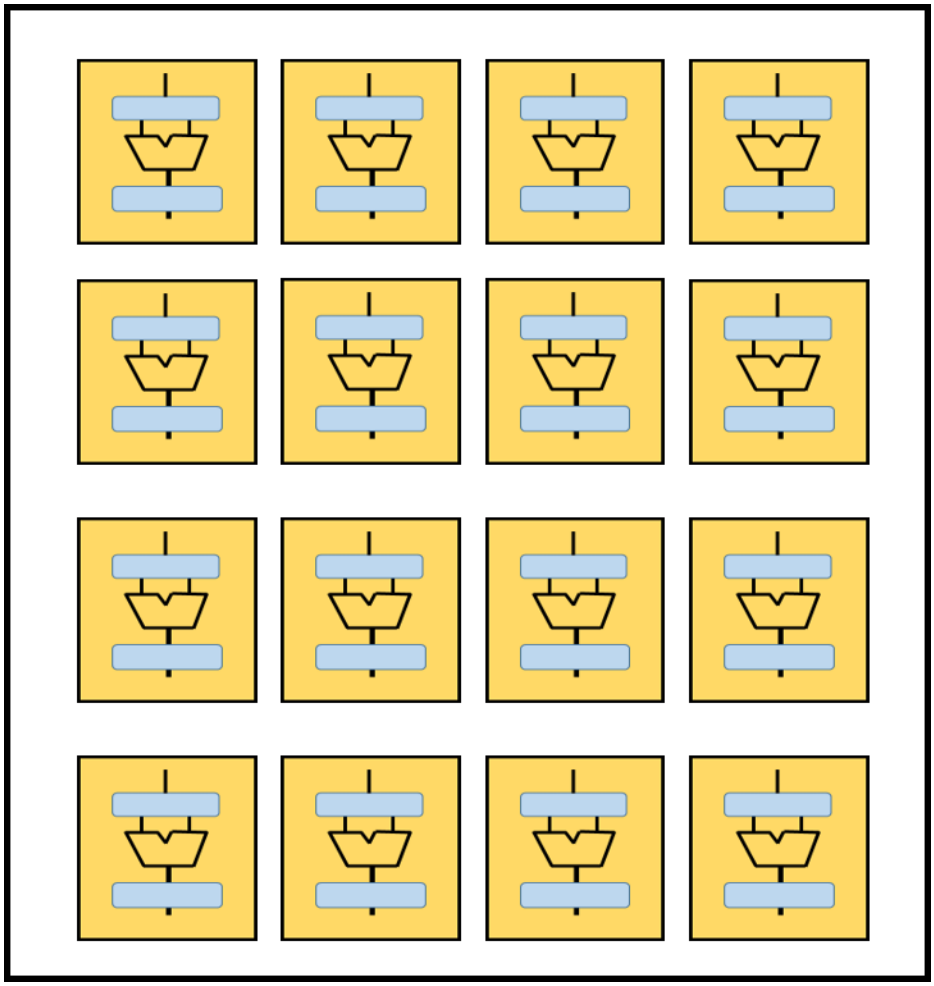


Figure 2.1: Symmetric multi-core with 16 BCE[40].

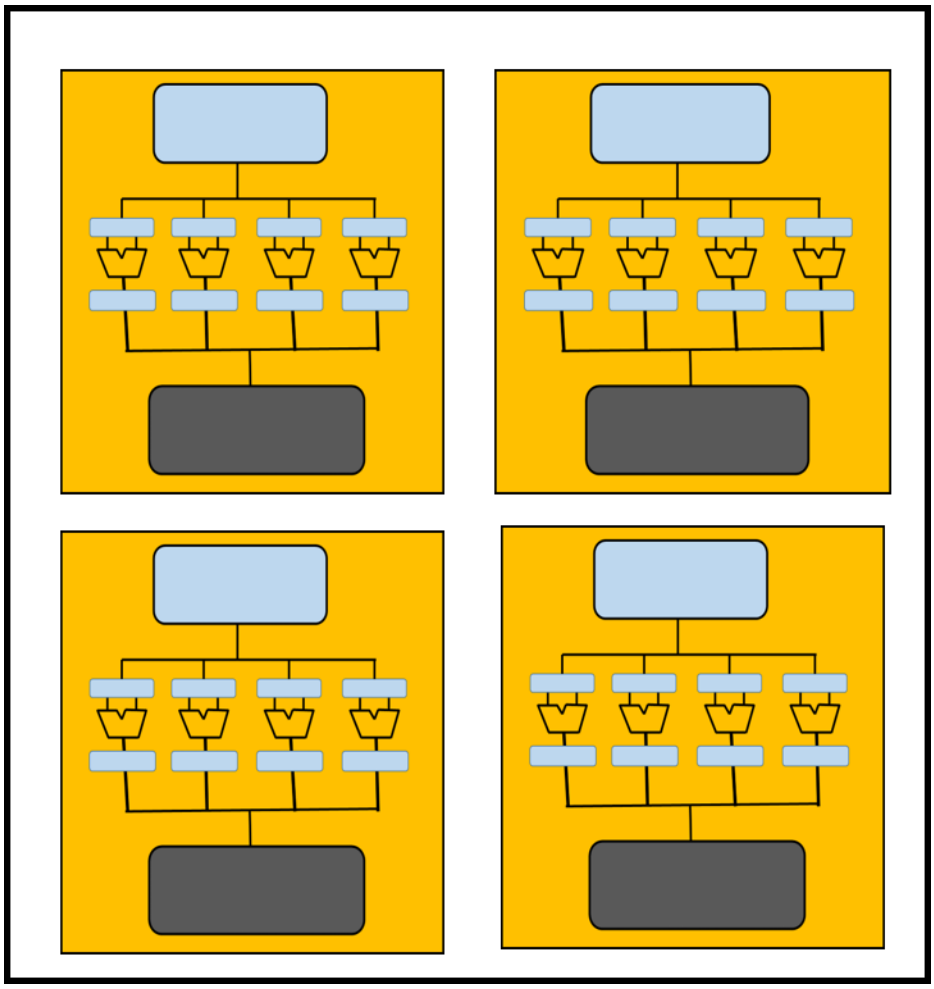


Figure 2.2: Symmetric multi-core with four cores and each core having four-Base core equivalent[40].

In this type of architectures same core is replicated multiple times on die. Architecture and capability of each single core is same. These architectures are also known as CMPs. [35], [66] have highlighted the benefits of multi-core systems. Hammend et al. [66] discuss the differences and benefits of having multi-core over SMT (Simultaneous Multi threading) architectures. Interconnect delays are more in SMT architectures. Gorder et al. [35] have shown that multi-core architectures not only help games but the researchers as well. They discuss of benefits of multi threaded program with each thread is potentially doing same task. Figures 2.1 and 2.2 show different homogeneous multi-core. Figure 2.1 shows a homogeneous multi-core. It has 16 baseline cores in it. Each baseline core is

known as base core equivalent (BCE) [40]. Similarly Figure 2.2 has four core with each core having four BCE. Since, in such architectures, all the cores are identical therefore these are not best in terms of power benefits. [51], [52] have proposed heterogeneous multi-core architectures for power benefits which is discussed in the next section.

2.1.2 Single ISA heterogeneous multi-core architectures

Prior work have shown that multi-core architectures have capability to enhance the program's performance. Various proposals [43], [79], [51], [52], [22], [10], [37], [40], [53], [60], and [72], [90] have been proposed related to the heterogeneity present in the core and ISA. Kumar et al. [51] [52] have shown that single ISA heterogeneous multi-core architectures can greatly enhance power reduction as compared to chip wide voltage/frequency scaling. Figure 2.3 shows heterogeneous multi-core with 13 cores, 12 cores are one BCE and thirteenth core is with four-BCE. This paradigm was widely adopted in commercial designs, most eminently in ARM's big.LITTLE architecture. Yoshimura et al. [108] proposed an SMT processor named OROCHI, where they are having two different pipeline for ARM and VLIW architectures. Ipek et al. proposed core fusion [43] architecture for improving single-threaded program's performance at times by morphing small cores into a big core. This work was further enhanced by Mihai et al. in the Bahurupi [79] architecture which changes the core's structure dynamically to extract maximum TLP and ILP by coalition of two or more simple homogeneous cores. The Bahurupi core increases the performance of single-threaded and multi-threaded applications. This architecture saves area compared to traditional small, big core combination. Lukefahr et al. [59] proposed an improvement on multi-core architectures by using heterogeneous cores that share resources. This sharing of resources enables dynamic migration at much smaller intervals between the heterogeneous cores. These cores however still shared the same ISA, differing only in their micro-architectural implementation. To improve performance and energy efficiency there has been another work DynaMOS [72] where they execute the program on performance efficient big core very first time. The traces of program is memorized and then the memorized trace is executed on In Order core to

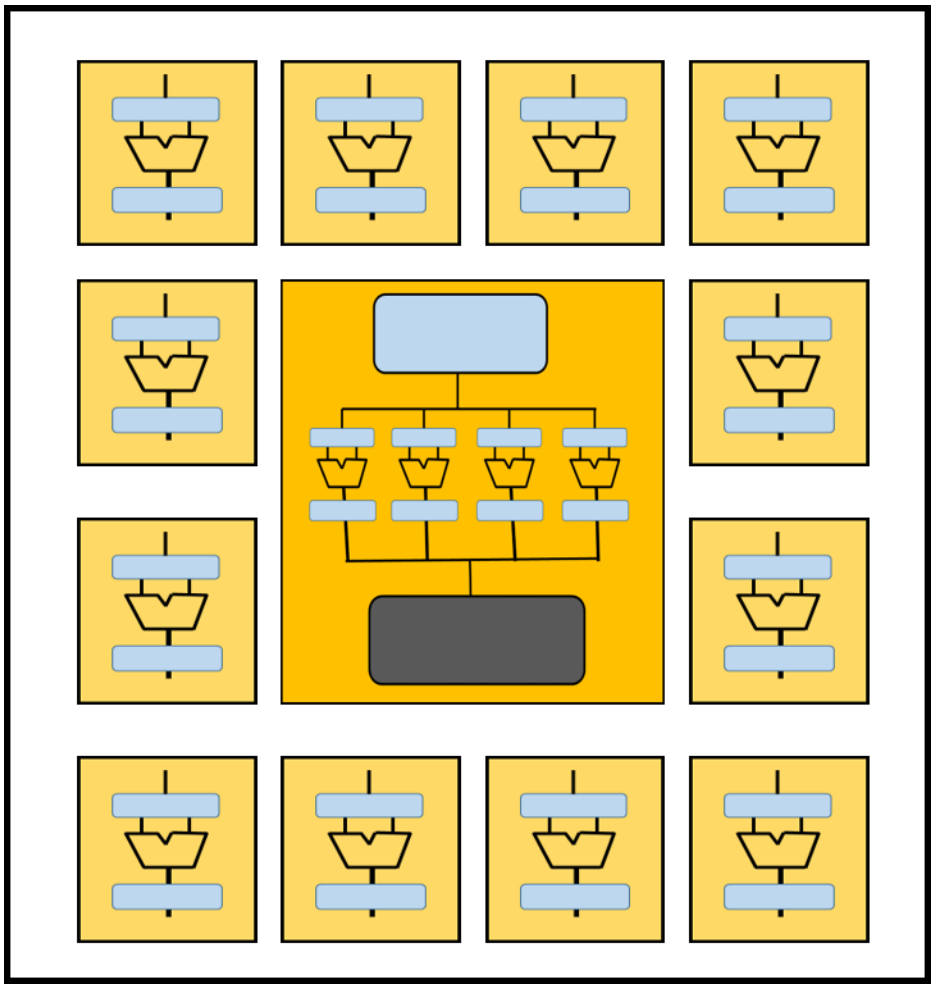


Figure 2.3: Asymmetric multi-core with thirteen cores, one four-Base Core Equivalent core and 12 one-Base Core Equivalent cores [40].

save energy. Bahurupi architecture [79] also improves performance and energy efficiency. Finally, to increase the energy efficiency further, DynamicCore (DC) architectures [43] attempt to modify the micro-architectural parameters of cores on-the-fly, morphing from a single big core into many small cores(or vice-versa) depending upon the program.

2.1.3 Heterogeneous-ISA multi-core architectures

Since the beginning of microprocessor era, a lot of novel ideas have been put forward for improving the performance of single threaded workloads. Several studies [22], [10] have analyzed the role played by ISA in CISC and RISC processors and showed similar

results both in terms of energy efficiency and performance. Lee et al. [55] investigated a subset of instructions in ARM ISA to find complex instructions which could be removed from the ISA without significantly reducing the performance. The reduced ISA has shown great reduction in logical complexity of the hardware. To improve the single-threaded performance, Venkat et al. [99] have observed that most of these CMPs are inefficient as they are unaware about the opportunities of performance improvement by exploiting application's affinity towards a particular ISA. Venkat et al. [99] showed that ISA diversity plays an important role in performance enhancement of heterogeneous ISA and outlined the benefits of such a system in terms of both performance and energy efficiency. The authors have shown that although ISAs seem to have converged over time (RISC ISA has added complex operations and CISC ISA internally translates to RISC micro-ops), these ISAs are different in some key factors. These factors are code density, dynamic instruction count, register pressure, native floating-point arithmetic vs emulation, decode logic and instruction complexity, and SIMD support. Due to such differences in the ISA, it has been shown that different applications have an affinity towards different ISAs and running an application on its affined ISA can improve its performance. Heterogeneity in ISAs in a processor was first introduced by Devuyst et al. [28]. In their proposal, they present that significant performance gain can be achieved if every phase of program runs on its most affine ISA. In our work like Venkat el al. [99], we try to couple Heterogeneous ISAs with heterogeneous hardware.

2.1.4 Performance modelling and scheduling in heterogeneous architectures

To gain maximum benefits from multi-core architectures, an efficient scheduling algorithm is required which predicts the best affine ISA for every phase (phase is defined as 'n' continuous instructions of program) in a program at run-time. Several scheduling methods [95], [97], [4], [64], [62], [63], [67], [86], [3] have been proposed in single-ISA multi-core architectures. The first work on performance modelling in multi-core architectures was by Kumar et al. [52] who used a sampling-based technique to predict migration

of programs on different cores. In this approach, they would run a small section of the code on all available cores and run the remainder of the code on the core that performed best for the small section. This leads to poor resource utilization. This naturally was not scalable and also performed poorly due to dynamic changes in program nature. To overcome this problem, the parameter cycles per instruction (CPI) was also tried to model by various researchers Craeynest et al. [96] used MLP(Memory level parallelism) and ILP parameters and introduced a regression based performance impact estimator. This was used further by Lukefahr et al. [59] and Pricopi et al. [80]. However, all these [27], [61] techniques while being especially suited for the homogeneous-ISA paradigm, provide suboptimal performance in the heterogeneous-ISA paradigm. In composite core [59] work, a thread switches in between two micro-engines, where one micro-engine is more aggressive while the other one is more energy efficient. Only one of the micro-engines remains active at any point in time allowing dynamic switching between the micro-engines so as to attain the best performance. The switching occurs on the basis of the past execution history. Venkat et al. [99] have migrated the thread from one core to another using SimPoint [78] metadata and profiling information from oracle experiments. Barbalace et al. [6] consider the behavior of the application to be unchanging throughout the program. They have calculated the cost (data transfer and migration) offline and found the migration points on the basis of this cost. Execution affinity towards different ISAs is also considered using the mincut algorithm. One another work on estimating performance of single ISA heterogeneous multi-core architecture is [16].

Craeynest et al. [96] introduced a regression-based performance impact estimator that used ILP and MLP (Memory level parallelism) as parameters to predict migration. While techniques work well for Heterogeneous multi-core architectures, they do not make accurate predictions for Heterogeneous ISA multi-core architectures. This is because they are simply not designed to take into account factors that determine performance variation across ISAs, such as code density, register pressure and instruction mix since these factors are identical across cores of the same ISA. To the best of our knowledge, we are proposing the first work attempting the estimation of the performance on one ISA

by executing it on the other. This estimation is the basis of performance modelling of heterogeneous ISAs.

In order to exploit full advantage offered by diversity of ISA, it is essential that the running program is freely able to migrate between different cores. Literature has shown this to be a difficult problem [28],[99].

2.1.5 Migration techniques in multi-core architectures

Once the most affine ISA is determined, the program state has to be migrated with minimum overhead. The Tui system [89] discusses the migration strategy in heterogeneous-ISA systems. In that work, the authors propose to transform the runtime state of program to state of program of required ISA. Some other proposals [7], [70] have explored the optimal state of program which required to be copied during migration.

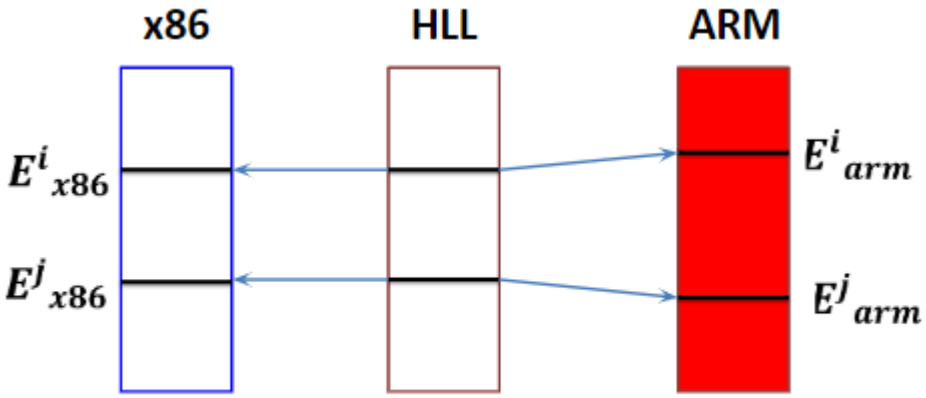


Figure 2.4: The memory state is consistent across different ISAs at Equivalence Points.

Following these approaches, Devusyt et al. [28] described the memory layout of a program executing on two different ISAs. They have shown that dynamic migration from one ISA to another one is possible at minimal performance cost. They observed that most of the memory arrangement could be shared for different ISAs without significantly

affecting the performance of any of ISA. They showed that by making a few modifications to the compiler back end, it is possible to have relatively frequent equivalence points in a program where the memory image of the program would be almost similar across different ISAs as shown in Figure 2.4. This resulted into a great reduction in migration overhead, making heterogeneous-ISA multi-core architectures feasible. A new migration scheme with binary translation was devised by Venkat et al. [99] 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. From 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 during migration. This transformation leads to significant migration overhead, which limits the granularity of migration.

In the proposed work, we are proposing to migrate only on equivalence points to save the cost of binary translation. In the scheduling mechanisms proposed in chapter 3, we are following to migrate in a similar approach by [99]. In chapter 4 and 5, we are proposing new techniques to reduce the migration overhead.

2.1.6 Dynamic core architectures

To increase the energy and performance efficiency further, dynamic core architectures [43] have been proposed, where a single big core is morphed into many small cores (or vice-versa) looking the behaviour of the program. In the proposal [72], authors proposed to remember the trace and then changing the behaviour dynamically of the core either as small or big at run time. Ipek et al. [43] have proposed that energy can be saved if smaller cores are morphed into big cores and vice versa based on the program's behaviour. Mihai et al. [79] have also proposed to make the core dynamic based on ILP and MLP. Another work in this direction was proposed by Padmanabha et al.[72] where they run the program on Big core first time and remember the traces. On the second run of the same program, the core is changed to the smaller core. The traces are run on small core, and hence energy can be saved.

2.1.7 Return oriented attacks

Buffer overflow vulnerabilities have been exploited by injecting malicious codes. To prevent this, modern processors have Executable Space Protection [98], [92] where only one of the operation can be performed on a memory page either writing or execution. Both operations can not be done on a memory page. Hence code reuse attacks have been proposed where the attacker uses the existing code only to mount an attack. Return oriented programming is a type of return-into-libc attacks. Return oriented programming was introduced by Hovav Shacham [88], [83]. In Return-oriented Programming short code snippets are chained together in the program. These short code snippets are called gadgets. Gadgets are chosen with a characteristics that gadgets end with a return or an indirect jump instruction. The stack is overflowed with sequence of return addresses which are very carefully constructed [100]. Once any gadget is executed and the instruction pointer reaches to return instruction, the next gadget is exploited with the help of stack pointer. These attacks have been proven to be turing complete [17], for multiple ISAs. Several variants of return oriented programming have been proposed in literature [11], [20], [44], [23], [18], [18], [41], [87], [84].

Several control flow techniques [1], [24], [25], [75], [109], [110] have been proposed in literature to mitigate these attacks. In such techniques the execution of any program is pre defined by control flow graphs. However, implementing a control flow integrity at run-time is a difficult job. Also some backdoor attacks [30], [33], [34], [19], [26] have been proposed which can by pass these techniques and hence making the system vulnerable. We will show in Chapter 4 that our proposed mechanism reduces the probability of such attacks significantly.

2.2 Security

All these multi-core architectures with shared last level cache are vulnerable towards side channel attacks. Hence, this section is detailed about security aspect of multi-core architectures.

2.2.1 Different security attacks

Multi-core architectures are vulnerable in terms of side-channel attacks. In side-channel attacks, the attacker program makes use of specific unintended effects of the victim program. Prior works in the field of hardware security particularly in the form of side-channel attacks [2], [58], [8], [107], [42], [94], [49], [73], [111], [8], [77], [71] have shown the security vulnerabilities of such systems by extracting the RSA[82] and AES[45], [69] keys. For instance, Bernstein [8] proposed an attack that can be mounted on the AES cryptographic protocol. Using this technique [8], an attacker can exploit the fact that the total execution time of the cryptographic algorithm is input-dependent and this can be used to deduce the private encryption key. The different modes in which an attack can be mounted are referred to as 'channels'. These channels include analysis of execution time, memory accesses, power consumption and electromagnetic radiation of the hardware resources being used by the victim program. The primary assumption in these attacks is the presence of shared hardware resources between the victim and the attacker, mainly caches. Flush+Reload [107], Prime+Probe [58] and Evict+Time [71] are side-channel attacks that analyze the cache access time after modifying the cache lines owned by the victim process. By evicting or appropriately modifying the victim's cache lines, the attacker can thus deduce the addresses accessed by the victim. The Prime + Probe attack [58] consists of three phases - PRIME, IDLE and PROBE. In the prime phase, the attacker fills cache sets with its own data. In the idle phase, it waits for the victim program to finish execution. In the probe phase, it reloads the data that was primed. If the victim program has accessed data corresponding to some of the same cache sets, then the primed data must have been evicted. When it is reloaded, the attacker observes an increase in load time due to higher memory access latency for main memory compared to caches. Yarom et al.[107] proposed another attack called Flush + Reload. This attack also consists of three phases - FLUSH, IDLE and RELOAD. In the flush phase, the attacker flushes a particular cache line from the memory hierarchy. In the idle step, it waits for the victim program to finish execution. In the reload phase, it reloads the same line and measures the time taken to do it. If the victim program accessed

the data that was flushed, then it is brought into the memory hierarchy and is reflected in the reduced load times. Some recent work [105, 106] has shown novel ways for such attacks to be detected by the system at runtime. COTSknight [106] tries to capture the cache occupancy patterns of running processes to identify suspicious applications that could pose a security risk. ReplayConfusion [105] replays a program's execution with a different cache address mapping to discern cache miss patterns.

2.2.2 Mitigation of side channel attacks

Several methods have been explored so far to mitigate cache-based side-channel attacks by sealing the side channel. All of these can be broadly classified into two approaches: cache randomization and cache partitioning [103], [74]. In these proposals, the authors suggest new cache designs that minimize resource sharing across cores. In doing so, they target the fundamental principle underlying side-channel attacks. In the first approach, the address mapping from main memory to the cache subsystem is randomized so that no process can precisely detect the accesses made by any other process. One of the simplest ways to enforce the second approach is through static partitioning. In [74], fixed partitioning of every set in cache is done for all processes. Since no cache resources are shared by processes in this method, it guarantees security against any cache-based side-channel attack. However, static partitioning comes with a heavy performance penalty because many lines in the cache set remain under-utilized. In order to improve performance in a multi-process system, several methods have been proposed to dynamically partition cache lines amongst processes [103], [50] [81] [102] [29]. Utility-based Cache Partitioning [81] is one such method that periodically partitions the cache lines in each set in a way that maximizes the total utility of all running processes. Prior work like COTSknight [106] and DAWG [47] also address similar security concerns. However, both of these require software and OS support and incur higher performance penalties. COTSknight makes novel use of cache monitoring technology (CMT) and cache allocation technology (CAT) features of modern processors to identify and isolate suspiciously behaving processes. However, it does not consider Flush+Reload attacks. Compared to an insecure

LRU baseline, COTSknight shows a slowdown of up to 5%. DAWG proposes a generic mechanism for secure way partitioning to isolate cache accesses and metadata. Compared to an approximate LRU baseline, DAWG exhibits slowdown between 0% and 15% for different experiments. PASS-P, on the other hand, shows an average slowdown of 0.35% and a maximum slowdown of 2.2% compared to insecure UCP baseline. Considering that UCP gives a 10.96% higher performance on average compared to LRU [81], we expect PASS-P to also perform favorably when augmented to UCP.

NoMo [29] is an L1-cache security system which presents a performance-security tradeoff that can be tuned. Like PASS-P, it requires no software support and requires only simple changes to existing cache replacement logic. However, the NoMo configuration which gives complete security is identical to static partitioning and may degrade performance. Compared to an LRU baseline, this configuration gives a performance degradation of up to 5% and 1.2% on average.

2.2.3 Prefetchers

Prefetchers are performance boosting hardware structures that fetch blocks from the main memory before they are required. The idea of prefetchers thwarting side channels was first introduced by Tromer et al. [94]. Due to the speculative nature of it's operation, they sometimes fetch data which end up being unused. An unused prefetched block will be interpreted as a cache hit by the attacker. This leads to false positives in the Flush+Reload attack and false negatives in the Prime+Probe attack. To secure data from cache side channels, Fuchs et al.[32] proposed the idea of a Disruptive Prefetcher. The prefetcher in this mechanism generates spurious memory accesses to pollute the cache and make the victim's accesses indistinguishable to the attacker. Another mitigation mechanism PrODACT has been proposed by Fang et al. [31]. It consists of a detection mechanism and a counterattack mechanism. The defense mechanism identifies the cache sets that are being targeted. The counterattack mechanism obfuscates the cache access patterns by fetching cache blocks not required by the victim program.

Wang et al. [101] proposed PAPP (Prefetcher aware Prime + Probe) that reverse engineers the prefetcher parameters to design a more efficient Prime+Probe that is immune to noise. However, this attack is specific to Prime+Probe and does not work well with other side-channel attacks. The attack that we propose can be mounted on top of any other cache side-channel attack, which makes it more generalized.

– ∗ – ∗ –

Chapter 3

Performance Modelling and Scheduling

Heterogeneous ISA architectures emerged as a good alternative to enhance single-threaded performance (and multi-threaded programs as well). If any program is divided into multiple parts and each part is scheduled on most affined ISA, the significant performance gain is achieved. As discussed in Chapter 1 and 2 the performance modelling and scheduling part of heterogeneous ISA architectures was unexplored to the best of our knowledge. In this chapter, we describe, in detail, our technique for cross-ISA performance modelling and scheduling in heterogeneous ISA architectures. The goal of modelling technique is to utilize micro-architectural and ISA-specific parameters obtained from the core that the program is currently running on, to predict what the execution time of the program would be if run on a different core (and ISA). This prediction will then be fed to the scheduler which dictates when the migration should take place. From here-on, we will use the term *source ISA/core* to refer to the ISA/core that the program is currently running on, and the term *target ISA/core* to refer to the ISA/cores that the program can migrate to. Modelling is done on the basis of few micro-architectural parameters. Given the need for an accurate cross-ISA performance modelling technique, we have solved this problem for coarse grained and finer grained level. We will discuss the finer-grained scheduling in next chapter. Coarse-grained scheduling is discussed in this chapter. We have done all the studies on two commonly used ISAs, RISC type ARM and CISC type x86. However, the proposed work can be as-is on any number of ISAs on chip.

The contributions of this chapter are as follows:

1. Predicting the programs affinity towards an ISA and scheduling the program using performance estimation based scheduler.

2. Predicting the programs affinity towards an ISA scheduling the program using classification based scheduler.

3.1 Performance estimation

Pricopi et al.[80] proposed a software based modelling technique to estimate performance and power of a single ISA asymmetric cores. Basic concept for modelling intra-ISA performance stack is inspired from this work. However only architectural parameters have been utilized unlike[80] which takes support from the compiler. In the beginning an execution time stack model for intra-core for two different ISAs separately is proposed. We refer to this as intra-core execution time stack model. Further inter-core execution time is obtained using intra core execution time model for estimated architectural parameters.

3.1.1 Execution time stack model

We propose a model referred to as execution time stack model for intra-core (same ISA) execution time. If there are no miss events (either branch or memory) and there is no data dependency then execution time is directly proportional to the number of instructions which get executed. Therefore, assuming high availability of ILP and with the knowledge of number of miss events, the proposed execution time stack model includes two kind of miss events (memory and branch) and delay due to stalls because of queuefull events. Our proposed model includes branch misses, L1 cache (D-cache and I-cache) and L2 cache misses. It is generally difficult to capture the impact due to data dependencies on execution time because the effect of data dependency overlaps with the other factors owing to parallelism. We have tried to capture this by considering the count of Instruction Queue (IQ) full events, Store Queue (SQ) full events and Re-order Buffer (ROB) full

events. All queues full events will result into stalls so these may contribute to the execution time of an application. Below equation depicts these relationships in the form of cost estimate in the same ISA for training phase. Here α_1 to α_8 are regressions coefficients and K is a constant.

$$\begin{aligned} C_{ISA_A} = K + \alpha_1.Committed\_Insts + \alpha_2.branch\_miss \\ \_count + \alpha_3.dcache\_miss\_counts + \alpha_4.icache\_miss \\ \_counts + \alpha_5.l_2\_miss\_counts + \alpha_6.IQFullEvents \\ +\alpha_7.SQFullEvents + \alpha_8.ROBFullEvents \end{aligned} \tag{3.1}$$

3.1.2 Inter ISA instruction count estimation

Our model assumes a linear relation between the code generated by the compiler for two different ISA targets. The model to estimate dynamic instruction count is given by the following linear regression equation.

$$Instruction\_count_X = K + \alpha.Instruction\_count_Y \tag{3.2}$$

where X is the core for which we are estimating instruction count using core Y's ISA instruction count.

3.1.3 Inter-ISAs miss events estimation model

Generally, number of branches and their patterns depend on ISAs and compiler. An inherent assumption used for our model is that branch count should follow some relation across ISAs. Thereby we are accounting for difference between branches due to branch predicate support and extra function calls (software emulation, subroutine calls etc.). Further, an additional assumption is that branch predictor accuracy remains the same across ISAs. We estimate branch count across ISAs which is to be multiplied with miss

frequency (= 1 - accuracy) to estimate branch misses count.

$$Branch\_miss\_count\_estimation_X = (1 - Branch\_predictor\_accuracy) * estimated\_branch\_count_X \tag{3.3}$$

$$estimated\_branch\_count_X = K + \alpha * branch\_count_Y^{\beta} \tag{3.4}$$

All 3 types of memory misses (*instruction cache miss*, *L1 D-cache miss* and *L2 cache miss*, considering L2 as LLC) are considered while modelling. We estimate these misses by regression analysis.

Instruction cache misses may happen in two cases, either next fetch address is present in new consecutive block or there is branch, so next fetching address is random. The proposed model for instruction cache misses is given by following regression equation.

$$Icache\_miss\_count\_estimation_X = K + \alpha_1.Icache\_miss\_count_Y \tag{3.5}$$

Data cache misses are highly dependent on memory access pattern of workload. Memory access count is estimated and then it is multiplied with cache accuracy to estimate *Dcache_ miss_ count*. Estimated D-cache access count of one ISA is dependent on *d-cache* access count and the ratio of memory references to total committed micro-ops of the other ISA. Complete model to estimate dcache misses counts is given by the following regression equations.

$$Dcache\_miss\_count\_estimation_X = (1 - Dcache\_accuracy) * estimated\_dcache\_access\_count_X \tag{3.6}$$

$$estimated\_dcache\_access\_count_X = K + \beta_1 * dcache\_access\_count_Y^{\beta_2} + \beta_3 * \left(\frac{dcache\_access_Y}{committedOps_Y}\right)^{\beta_4} \tag{3.7}$$

L2 cache access depends on L1 caches (instruction and data) misses. To estimate *L2 _cache_ miss_ count*, L1 (D-cache) and *L2_ cache_ miss* are considered. The model is given by the following equation.

$$l_2\_cache\_miss\_count\_estimation_X = K + \alpha_1 * l_2\_cache \\ \_miss\_count_Y + \alpha_2.Dcache\_miss\_count_Y \tag{3.8}$$

3.1.4 Inter-ISA queue full event count estimation

To capture data dependence and execution time of an instruction which has many micro-ops, QueueFullEvents has been envisaged.To estimate these events count, a linear regression model is considered.

Estimation of instruction queue full event is dependent on dynamic micro-ops, instruction queue size, and instruction queue full event of the other ISA. The following regression relation gives inter-ISA's IQFullEvents count.

$$IQFullEvents_X = K + \alpha_1 * IQFullEvents_Y + \alpha_2 * \\ \left(\frac{insts\_count_Y}{64}\right)^{\beta_2} \tag{3.9}$$

Store queue full events depend on memory write requests. The following regression relation estimates *SQFullEvents* on inter ISA.

$$SQFullEvents_X = K + \alpha_1 * SQFullEvents_Y + \\ \alpha_2 * mem\_write_Y^{\beta} \tag{3.10}$$

Lastly, model to estimate Instruction ROB full event count is given by the following regression equation.

$$estimated\_ROBFullEvent_X = K + \alpha_1 * ROBFullEvent_Y \tag{3.11}$$

Table 3.1: Core's configuration

Design Parameters	ARM	X86
Architecture Registers	32 GPR	16 GPR
Cache line size	64 byte	
LQ,SQ size	32 entries	
IQ entries	64 entries	
ROB entries	192 entries	
D-Cache, I-Cache size	32KB	
L2 cache	256KB	

3.1.5 Inter-ISA's core execution time estimation

After estimating all the architectural parameters for the other ISA's core, the proposed intra-core execution time stack model can be used to estimate inter-core execution time as follows

$$\begin{aligned} C_{ISA_B} = K + \alpha_1 * estimated\_instruction\_count_B + \alpha_2 * estimated \\ \_branch\_count_B + \alpha_3 * Dcache\_miss\_count\_ \\ estimation_B + \alpha_4 * Icache\_miss\_count\_estimation_B + \\ \alpha_5 * l_2\_cache\_miss\_count\_estimation_B + \alpha_6 * estimated \\ \_IQFullEvent_B + \alpha_7 * estimated\_SQFullEvent_X + \\ \alpha_8 * estimated\_ROBFullEvent_X \end{aligned} \tag{3.12}$$

All the parameters are estimated from ISA 'A' to ISA 'B'. By replacing these values in equation 3.1, the model will estimate inter-ISA execution time.

3.1.6 Results and analysis

The root mean square error values are reported in all the figures for different configuration parameters of simulation framework as listed in Table 3.1. Individual benchmark error is

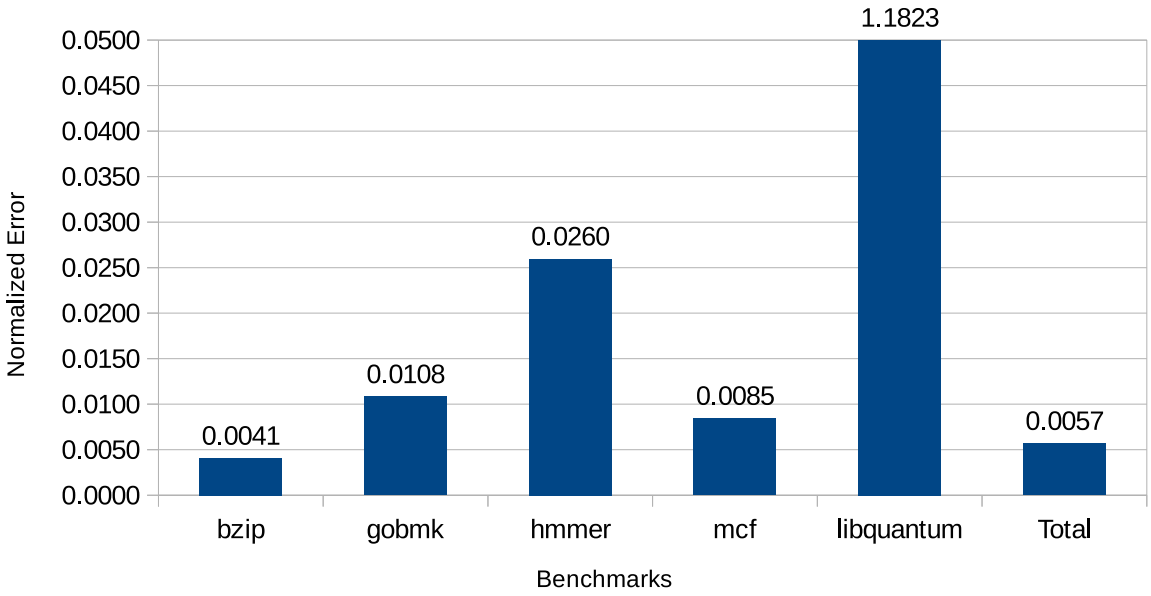


Figure 3.1: ARM execution time stack model error

calculated using corresponding dataset with the proposed model in 3.1 and 3.2 whereas in 3.3 and 3.4 error is shown for each parameter.

Evaluation of execution time stack model

Equation 3.1, is utilized to discover relationships between execution time and the different parameters. Using training data set regression variables and coefficients are computed. The developed model is tested for ARM ISA and the results are shown in Fig. 3.1. For ARM ISA, the proposed model has an approximate accuracy of 99.4%. The same procedure is followed for evaluating the model accuracy for X86 ISA. For this case it is observed that, different delays (data dependency delays of various tasks) get overlapped due to parallelism resulting in reduced execution time. Note that the proposed model is additive in nature which leads to an overly pessimistic computation of execution time. This is reflected from the achieved accuracy (98.9%) for X86 ISA.

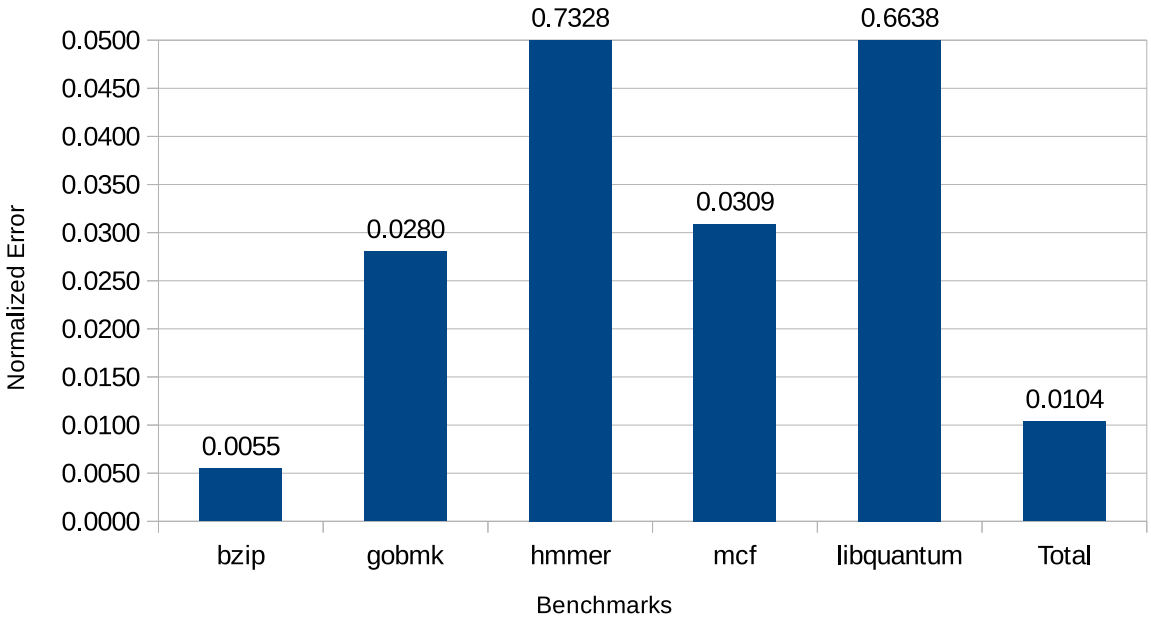


Figure 3.2: X86 execution time stack model error

Inter ISA execution time estimation

After estimation of all online architectural parameters included in the proposed model, the execution time from one ISA to another one is obtained using equations 3.2-3.11. While estimating architectural parameters from ARM to X86 it is observed that L2 miss counts, and ROBFullEvents counts are more error prone as compared to the other parameters. As shown in Fig 3.3, execution time from ARM to X86 is estimated with an overall error of 23% . It is observed that our model gives more error while estimating from X86 ISA to ARM ISA. One probable reason behind this is large number of misses due to queues FullEvents. Execution time from X86 to ARM is estimated with error of 54% as shown in Fig 3.4.

The execution time is estimated with an error of approximately 23% from ARM to x86 and approximately 54% from x86 to ARM. Since the error is huge, this model has not been used for scheduling proposes.

This approach has two main shortcomings:

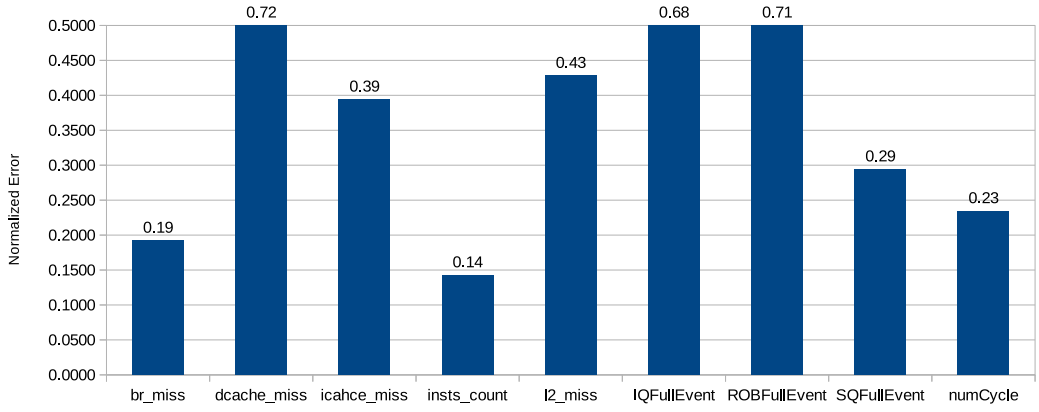


Figure 3.3: Error in estimation from ARM to X86

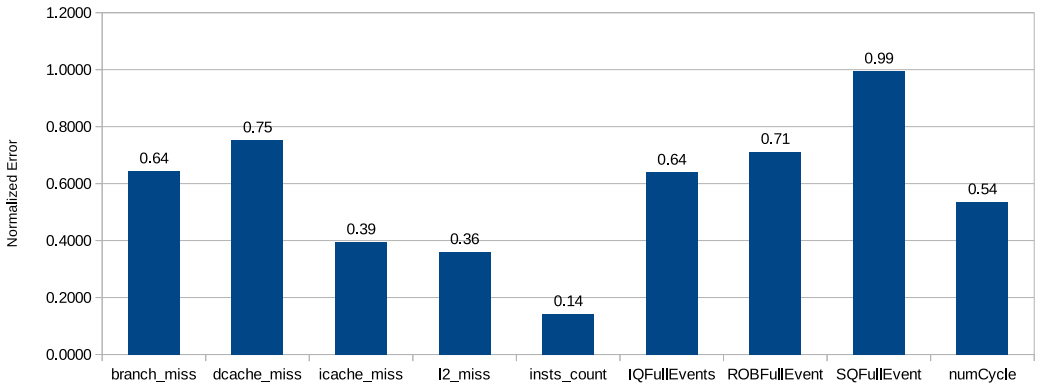


Figure 3.4: Error in estimation from X86 to ARM

- The decoupling of execution time prediction into two independent phases compounds prediction errors. In previous work, execution time predictions can be off by up to 54%.

- In this work, we did not take into account any parameters that characterize the inter-ISA heterogeneity, such as code density, register pressure or instruction mix, leading to high prediction errors.

3.2 Modified performance estimation

In this work, we improve upon the performance model estimation discussed in Section 3.1 in the following ways:

- We eliminate the artificially enforced decoupling in the prediction of execution time. Consequently, we directly predict the execution time on the target ISAs using the parameters of the source ISA.
- We specifically introduce parameters that quantify the inter-ISA heterogeneity, specifically the instruction mix, dynamic instruction count and parallelism (ILP, MLP)
- We replace the regression model with a linear regression model, which performs better for single-level predictions.

3.2.1 Extracting relevant parameters

We now describe the list of parameters we use to predict the execution time on the target ISA and how we extract them. In this work, we use a total of thirteen parameters to predict execution time (number of execution cycles). The parameters include branch miss-predictions, L1-I-cache misses, L1-D-cache misses, L2 cache misses, Reorder Buffer full events, Instruction Queue full events, Store Queue full events, ILP, MLP, MSHR (Miss Status Handling Register) full events, instruction mix (Number of load instructions, Number of floating point instructions) and the dynamic instruction count.

L1-I-cache, L1-D-cache, and L2 cache misses capture the effects of the cache hierarchy on the execution time. Information regarding data-dependency induced stalls has been extracted by the occurrences of the ROB, Instruction Queue and Store Queue being full. ILP and MLP have been considered for determining the available parallelism given the specific ISA and core that the program is running on. Instruction mix and dynamic instruction count are chosen to quantify ISA specificity and finally, the behaviour of the branch predictor is captured using the branch misprediction parameter.

A common feature of all of these parameters is that their extraction is simple and practical. All parameters except ILP and MLP can be directly obtained from hardware performance counters prevalent in every major processor variant today. To calculate ILP and MLP we rely on schemes proposed by previous work [59]. ILP is estimated by using

a hardware counter which maintains a running sum of the instructions in the issue stage whose execution requires the data from the other instructions currently under execution. This counter captures all the instructions which are stalled due to dependencies, thus providing us with an inverse measure. For MLP, we leverage a hardware counter that maintains a running sum of the number of MSHR entries at every cache miss. This gives us an estimate of MLP because during a cache miss, all misses currently being handled in parallel will have an entry in the MSHR. We take individual averages of the running sums maintained by both counters ($running\_sum/total\_instructions$) to estimate the ILP and MLP respectively.

3.2.2 Linear regression model

Given those parameters, we now describe our linear regression based performance model. In our evaluation Section 3.2.4, we show that it outperforms the GRNN model proposed in previous work. Given a source ISA in execution (ISA_A) and a target ISA (ISA_B), our linear regression model for estimation of the number of cycles is given by:

$$\begin{aligned} Cycle_B = & K + a_1.(L1DcacheMiss_A) + a_2.(L1IcacheMiss_A) \\ & + a_3(L2cacheMiss_A) + a_4.(IQFullEvents_A) \\ & + a_5.(SQFullEvents_A) + a_6.(ROBFullEvents_A) \\ & + a_7.(BranchMissPrediction_A) + a_8.(MLP_A) \\ & + a_9.(MSHRFullEvents_A) + a_{10}.(ILP_A) \\ & + a_{11}.(LoadCount_A) + a_{12}.(FloatInstruction_A) \\ & + a_{13}.(DynamicInstructionCount_A) \end{aligned} \tag{3.13}$$

where a_1 to a_{13} are regression coefficients, $Cycle_B$ is number of cycles for ISA_B and K is a constant

In this work, we consider three ISAs namely x86, ARM and Alpha. We build a total of 6 regression models: for each of the three ISAs, we predict the performance of

the other 2. The idea here is that these models are built offline and incorporated into the processor with the coefficients of the model stored in special registers. Then, when programs are executed, these models continuously predict the performance on the other two ISAs and pass these predictions to the scheduler.

3.2.3 Scheduling

In this subsection, we describe the design of our scheduler. We first provide necessary background on how a cross-ISA compiled program looks like, before detailing our scheduling algorithm and the tradeoffs involved.

De Vuyst et al [28] describes compiler modifications that allow a program to efficiently migrate across cores. The generated binary possesses a number of *equivalence points* at which the memory state of the binary is consistent across different ISAs. This identical memory state allows for low-overhead migration of the program at one of these equivalence points. Note, migrating at any other point is expensive and requires dynamic binary translation on the target ISA till the next equivalence point is reached. Given the overhead of migration, these equivalence points are typically kept around 100M instructions apart. We call each such division of 100M instructions a *phase*.

This division of the program into phases poses a significant challenge for the scheduler. Ideally, we would like to use the first few (10-20M) instructions of the phase to predict which ISA it is suited to, much like [52]. Unfortunately, this is infeasible since migration can only occur at the equivalence points. Consequently, all scheduling decisions must be made before the phase begins to execute.

Once we have the predictions for execution time for all the target ISAs, we simply employ greedy scheduling, i.e., at the next equivalence point, the scheduler migrates the program to the ISA with least predicted execution time. Naturally, for all the target ISAs we include the migration overhead into the estimation.

Fig. 3.5 is representing the schematic diagram of the flow of our scheduling model. Each benchmark is divided into training and test data and both are mutually exclusive. Step 1 in Fig. 3.5 represents extraction of parameters to be given to the perceptron

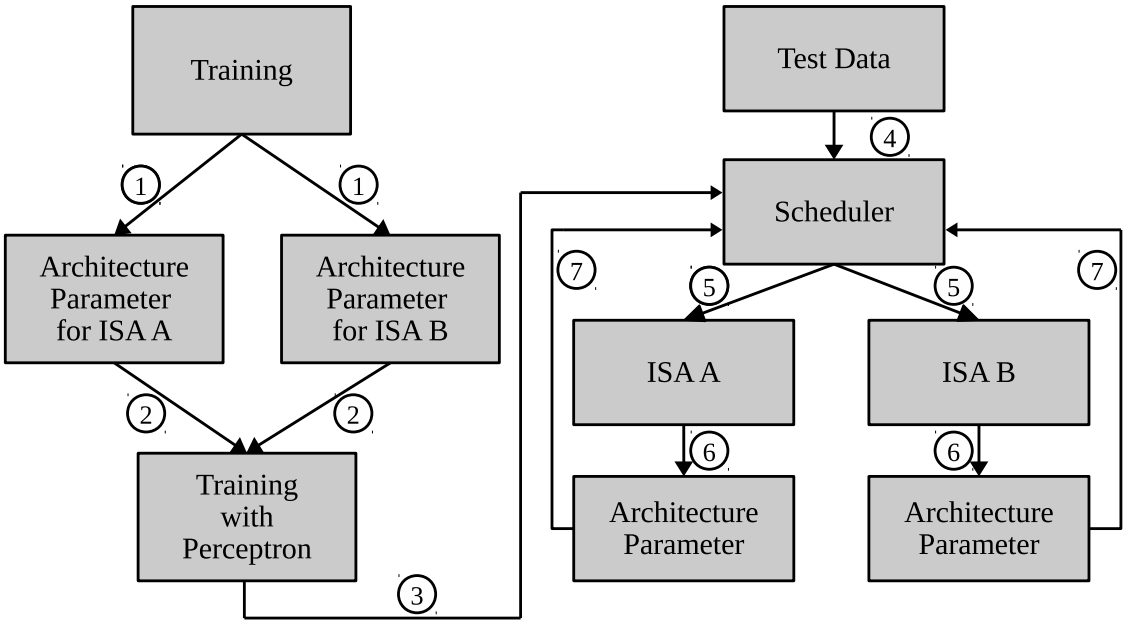


Figure 3.5: A schematic representation of the scheduling model

algorithm for training purpose. Using these parameters, perceptron algorithm tells the affinity of a phase towards any ISA. 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 takes a decision on the basis of few instructions for a phase. We have found experimentally that taking 10 million instructions are sufficient to decide the affinity of a next phase of 100 million instructions. Therefore, on the basis of micro-architectural parameters of last 10 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. 3.5, the steps 1-3 are performed offline whereas steps 4-7 are executed online dynamically. Based on the initial training (step 2), the scheduler decides the migration between ISA 'A' or ISA 'B' for the first phase of 100 million instructions. This is denoted by step 5. Depending on the last 10 million instructions of this phase, the migration decision is taken by the scheduler for the next phase of 100 million instructions. The extraction of architectural

parameters of the last 10 million instructions is depicted by step 6 and the corresponding migration decision is shown by step 7. Once we have the predictions for execution time for all the target ISAs, we simply employ greedy scheduling, i.e., at the next equivalence point, the scheduler migrates the program to the ISA with least predicted execution time. Naturally, for all the target ISAs we include the migration overhead into the estimation.

3.2.4 Evaluation

In this subsection, we describe the results from our evaluation, which answer 2 primary questions: 1) Does the regression-based performance model predict performance accurately across ISAs? and 2) Does the scheduling algorithm correctly migrate the program to deliver overall speedups in single-threaded performance?

3.2.5 Methodology

As mentioned previously, we consider 3 ISAs in this work - x86, ARM and Alpha. Since the focus is on inter-ISA heterogeneity, we keep the configuration for all 3 ISAs identical (Number of GPRs is a property of the ISA). The configuration for all the cores is shown in Table 3.2. The clock speed for all three cores is 2GHz. We use the Gem5 simulator[9] to simulate performance and SPEC CPU2006 [39] benchmark suite as our target programs.

We use the results from [99] to determine the migration overhead. As a simple over-approximation, we consider the maximum migration overhead they report for the SPEC CPU2006 benchmark suite.

3.2.6 Results and analysis

Accuracy of performance modelling technique

To evaluate the prediction accuracy of our regression-based performance model, we compare the Root Mean Squared (RMS) prediction error across several schemes for each of the 6 models that we build. We compare our linear regression-based model against two schemes: 1) A GRNN model from prior work[12] (Previous Model), and 2) A GRNN

Table 3.2: Core configurations for LRNN based scheduler

Design Parameter	ARM	Alpha	x86
Architectural Registers	32 GPR	64GPR	16 GPR
Cache line size(bytes)	64	64	64
LSQ size (bytes)	32	32	32
Fetch width	4	4	4
Instruction Queue entries	64	64	64
ROB entries	192	192	192
DCache,ICache size	32KB	32KB	32KB
L2 Cache size	256KB	256KB	256KB

model using all 13 parameters - including the ones accounting for inter-ISA heterogeneity (GRNN). For (1) we only use values wherever provided. Comparing against (1) illustrates the joint effect of both using linear regression and including parameters that capture the inter-ISA heterogeneity. Comparing against (2) enables us to quantify the improvements due to using linear regression since the parameter set is identical for both schemes.

Figure 3.6 compares the RMS prediction error for all three schemes across predictions for all phases of all the benchmarks in the SPEC CPU2006 benchmark suite. The phase length taken for performance modelling is 10M.

Two clear trends emerge:

- Irrespective of the model, linear regression always outperforms the GRNN. The linear regression based model leads to errors ranging from 1.7% to 5.7%, while the GRNN leads to errors ranging from 7.5% to 9.4%.
- Both schemes outperform prior work considerably. This is due to two factors:
  - The introduction of additional parameters that capture inter-ISA heterogeneity such as the dynamic instruction count and the instruction mix

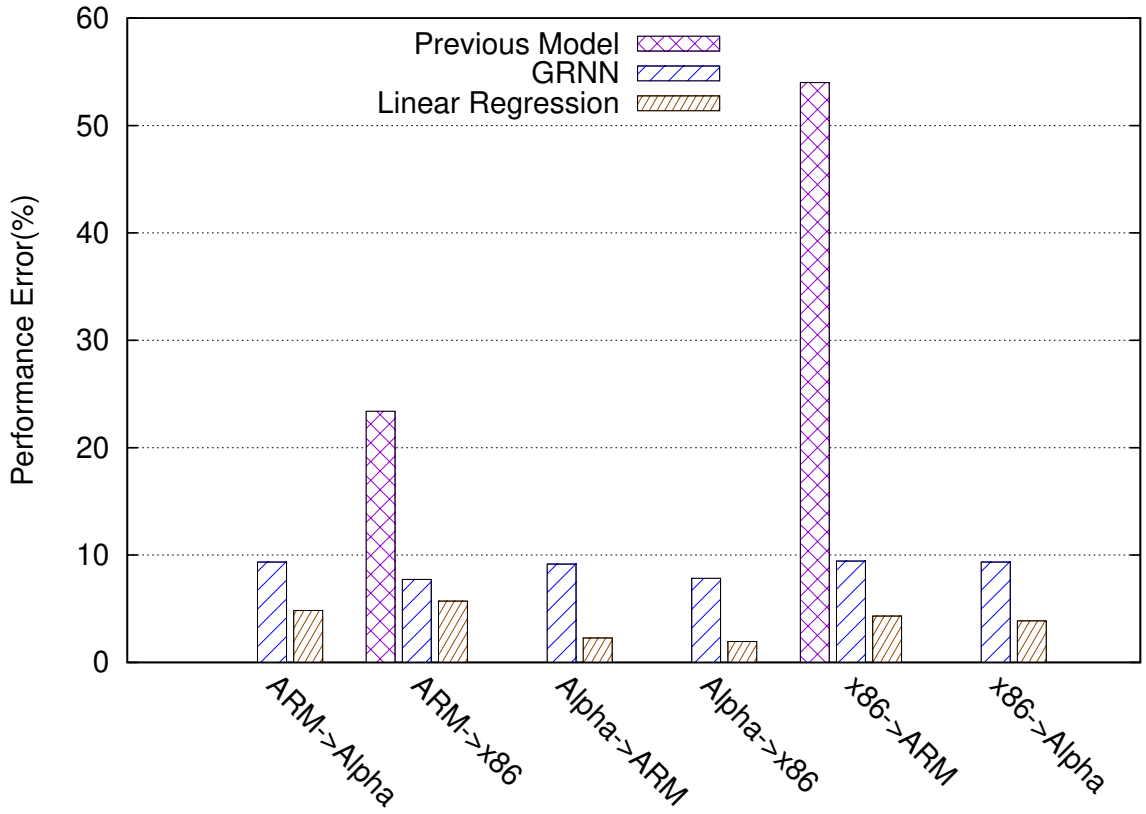


Figure 3.6: Root mean squared error of performance modelling of Previous Model [12], GRNN and Linear Regression. ISA_1 —> ISA_2 denotes estimation of performance for the core with ISA_2 while running the program on the core with ISA_1

– Replacing the two-phase prediction which compounds prediction errors with a simple one-phase prediction scheme.

We also compare the standard deviation of the RMS error for both the GRNN and Linear regression-based models. Linear regression has a 26% lower standard deviation than the GRNN model making it lesser uncertain and more reliable.

Dynamic scheduling

To evaluate whether our scheduling algorithm migrates the program correctly, we compare the speedup obtained in the HeIMC architecture as opposed to the base case of a

single ISA architecture. Figure 3.7 illustrates the result of average speedup for different programs with different scheduling methods. Our regression based scheduler shows a 29.6% increase in mean performance when compared to the x86 baseline and a 19.5% increase in performance on the best performing architecture. Additionally, the regression-based predictor is only 12.2% off the oracle. Please note that the *oracle is a hypothetical case in which each phase runs on the core it is most suited to. Hence the Oracle case represents the maximum possible speedup.* The phase length for dynamic scheduling is taken as 100M. The models were trained using data from all of the SPEC CPU2006 benchmarks (70% training data, 30% testing data).

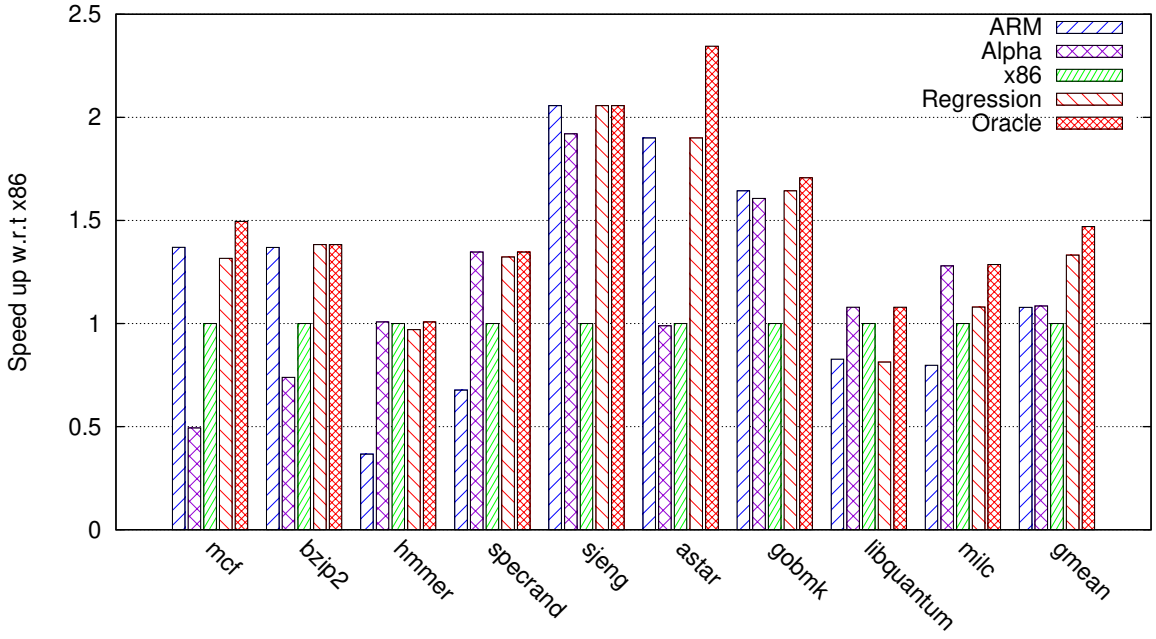


Figure 3.7: Average speedup of different benchmarks when entire program is scheduled on ARM, Alpha or x86 and compared with HeIMC architecture with regression and oracle based scheduling

In the previous experiment, the model was only evaluated on programs it had already been exposed to. When deployed, however, our framework must run accurately even for programs it has not been trained on. To evaluate this generality and resilience of our migration framework, we trained the model using only a subset of the SPEC CPU2006

benchmark suite and tested it on others. Then we measured how accurately the scheduler migrates the program and the speedups achieved on these unseen programs. Figure 3.8 illustrates the results. We see that on average, our system works even for programs it has not seen before, producing an average speedup of 24.4% over x86 and only 16% less than the oracle. Additionally, our system migrates the program to the core it is most suited 82.94% of the time. Given that the SPEC CPU2006 benchmark suite consists of programs that are inherently very different from one another, these results show that our model is resilient and can be deployed without having to be re-trained frequently.

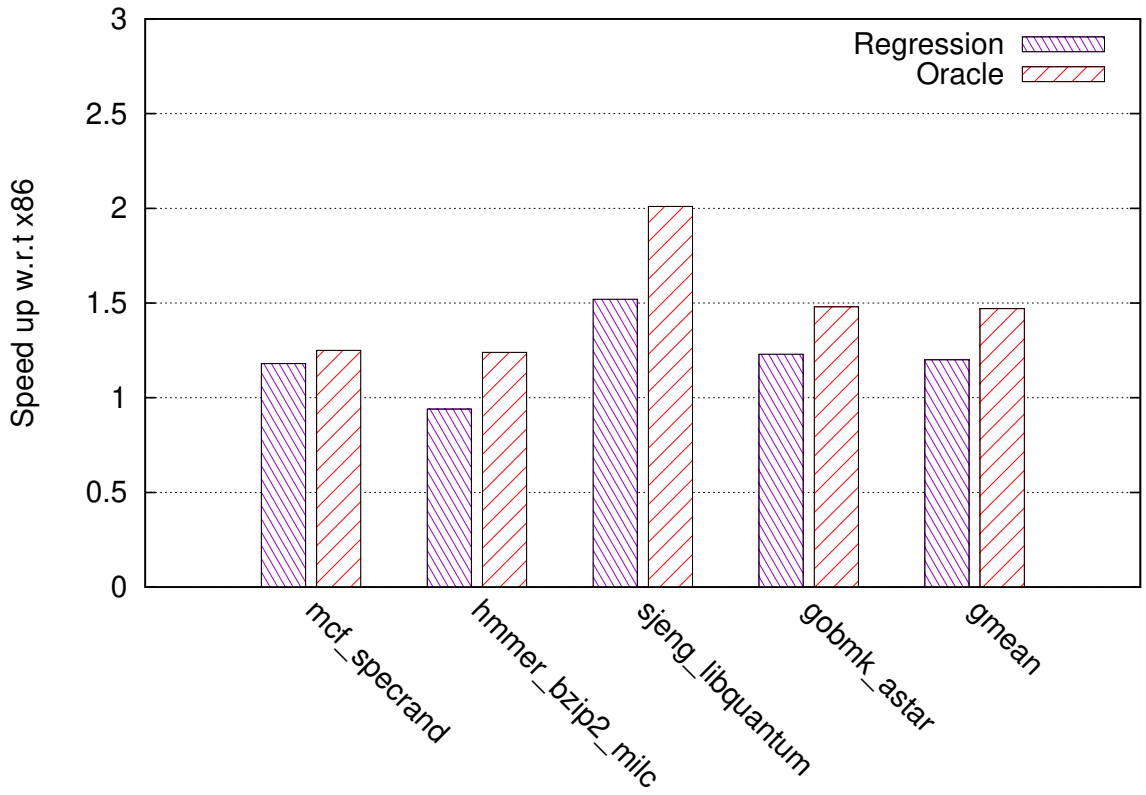


Figure 3.8: Speedup w.r.t x86 of SPEC CPU2006 benchmarks when our system faces programs it has not been trained on. Here the x-axis labels represents benchmarks used only for testing while remaining benchmarks were used for training.

Energy efficiency

We also ran an experiment to see whether the HeIMC architecture would increase the energy efficiency of the system. For this, we used the McPAT[56] simulator to compare the energy consumed. We have calculated the energy consumed while the whole benchmark is run on ARM, alpha, x86 and these energies are compared with energy consumed for the HeIMC architecture which uses our performance model and scheduling algorithm. In our experiments, we find that there is very little change in energy consumption. This is likely because all three cores in our system had an identical configuration the ISA which displays the maximum performance also consumes the maximum power. Additionally, our scheduler is only designed for optimizing performance. In future work, we plan to incorporate the energy consumption also into the decision making process.

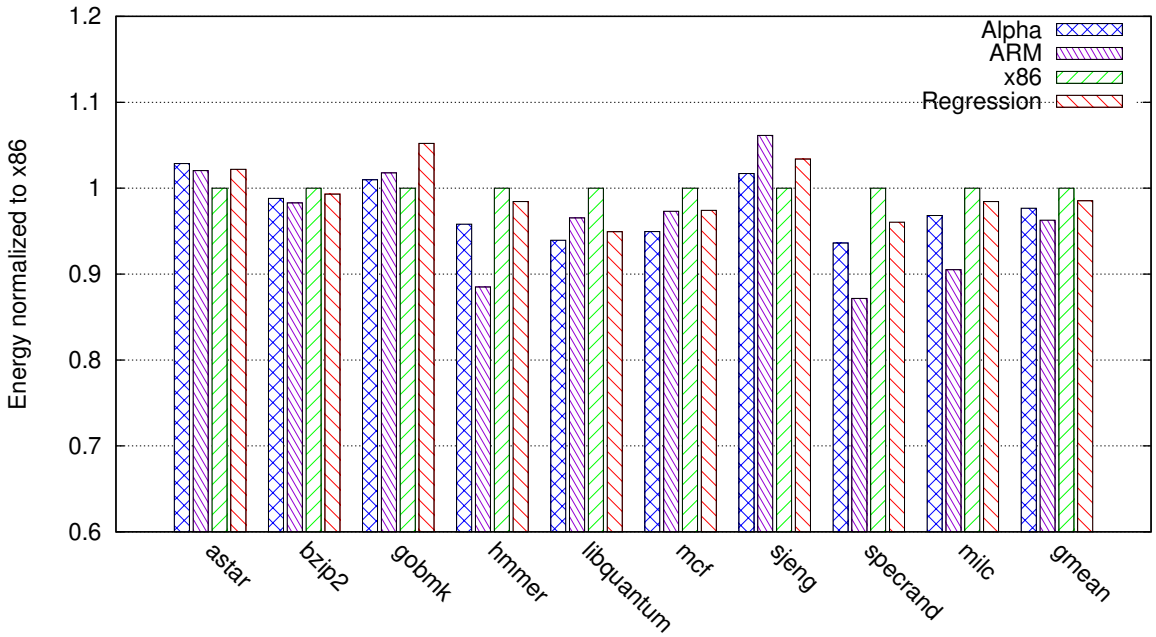


Figure 3.9: Energy comparison

Hardware overhead

The proposed modelling method and scheduling algorithm have minimal hardware overhead. Since we are doing the training offline in software, we only require 14 registers

to store the weights for performance modelling using the linear regression method. The compute scheduler requires thirty-nine 8-bit multipliers for applying weights to the input parameters along with three 32-bit adders. Please note that a GRNN based model has more hardware overhead when compared to a linear regression based one.

In scheduling mechanisms based on performance estimation, the execution time of target ISA is predicted and then the scheduling decision is taken. In next section, we will solve this problem as a classification problem.

3.3 Classification based scheduler

To obtain maximum performance from Heterogeneous architectures, every phase should run on the most affine ISA. Hence scheduler plays a vital role. The work proposed in Section 3.2 uses a linear regression based method which does the performance modelling and then finds the most affined ISA from the modelling. We assume that the sets of phases affine to x86 and ARM ISAs are linearly separable, and tried looking at this as a classification problem. To find the best affine core we are using a single layered neural network (perceptron) based learning algorithm.

3.3.1 Scheduling model

The scheduling model is similar to the model discussed in Section 3.2.3. The flow diagram is also similar to the Figure 3.5. In classification based scheduler, we are employing perceptron classifier as a scheduler.

3.3.2 Perceptron classifier

The diagram for perceptron algorithm is shown in Figure 3.10. The perceptron algorithm initially creates k nodes corresponding to k number of inputs. Then it iteratively takes a training example and in case of a miss-prediction, updates every weight with the given formula

$$w_k(n+1) = w_k(n) - r * (t - c) * x_i$$

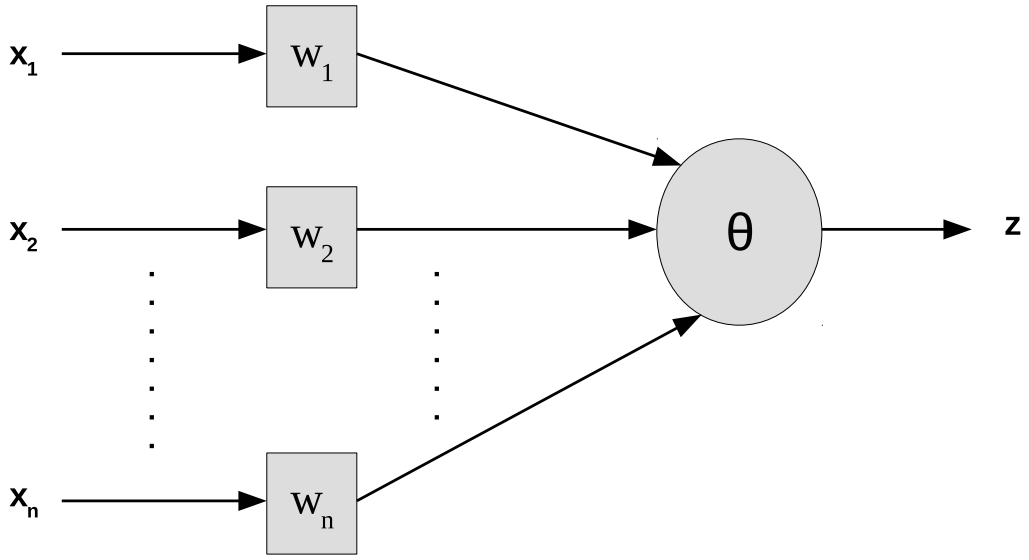


Figure 3.10: Perceptron model

where w_k is weight corresponding to the input x_k, r is the rate of learning, t is predicted output, c is the expected output. Sum of product of weight and micro-architectural parameters is compared with a threshold value θ and the decision is taken on the basis of this comparison. We continue training until the total training set error ceases to improve till the convergence point, the weights and θ values are stored. For the perceptron algorithm which comes under supervised algorithms, we are keeping the most affine core in the labelled data. Data is preprocessed to find the most affine core including migration overhead. This preprocessed labelled data is used to learn the weights for perceptron during the training phase. The model is trained offline. Once the training is completed, it provides two 'weight matrices' and θ value which will be loaded into the hardware. When the program executes, the weight matrix corresponding to the currently executing ISA, is given to the perceptron scheduler. The sum of product of micro-architectural parameters x_i with corresponding weights is compared to the threshold θ. The migration decision for the upcoming phase of the program will be taken on the basis of this comparison.

3.3.3 Evaluation setup

We consider 2 ISAs in this work - ARM and x86. Benchmarks are compiled for x86 and cross compiled for ARM similar to [99]. The configuration for both the cores is shown in Table 3.3. The clock speed for both the cores is 2GHz. SPEC CPU2006 [39] benchmarks are used as our target programs and these are simulated using Gem5 simulator[9]. Phase length and migration overhead is taken similar to [99].

Table 3.3: Core configurations for classification based scheduling

Design Parameter	ARM	x86
Architectural Registers	32 GPR	16 GPR
Cache line size(bytes)	64	64
LSQ size (bytes)	32	32
Fetch width	4	4
Instruction Queue entries	64	64
ROB entries	192	192
DCache,ICache size	32KB	32KB
L2 Cache size	256KB	256KB
SIMD Support	No	Yes

3.3.4 Results and analysis

Program's affinity towards different ISAs

We have analyzed the affinity of the benchmarks to different ISAs. We ran SPEC CPU2006 [39] benchmarks on both ISAs. Fig. 3.11 describes the percentage affinity shown towards these two ISAs. A few of the benchmarks such as *gobmk, h264ref, sjeng* seem to be more biased towards ARM ISA. Still, some of the benchmarks such as *libquantum, specrand_int and specrand_float* show mixed behavior, while *soplex* is biased towards x86. This affinity completely depends on the program behavior such

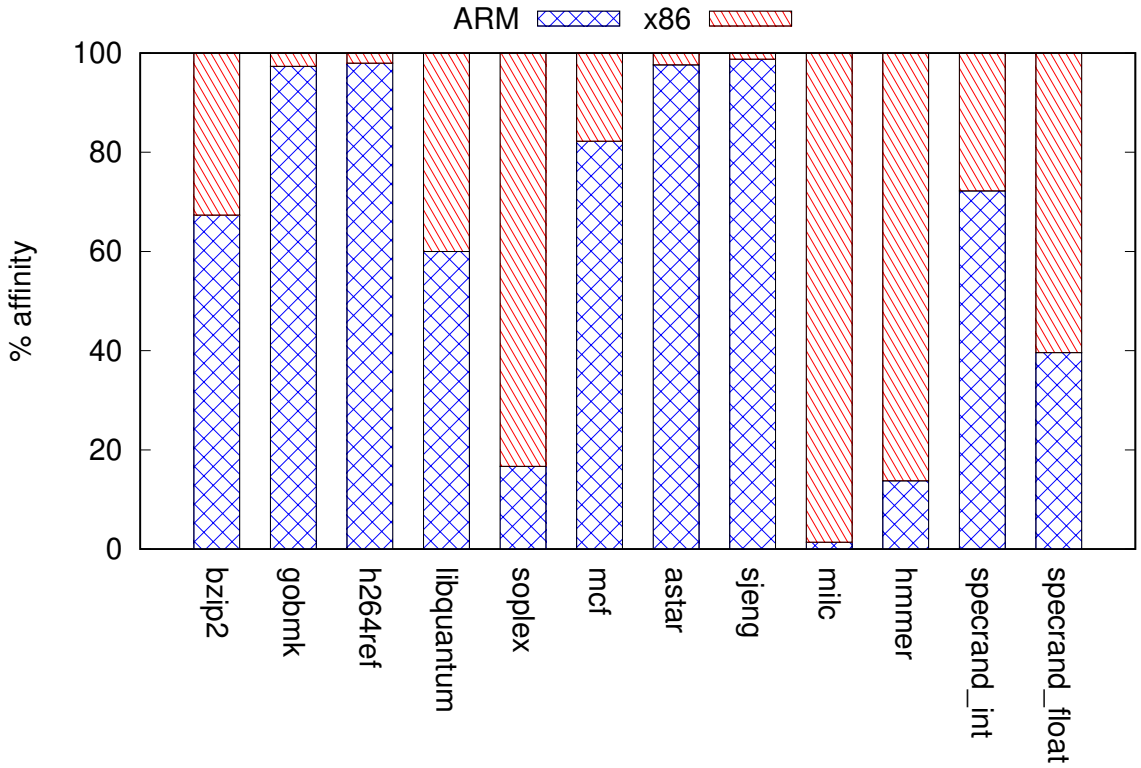


Figure 3.11: Percentage ISA affinity for SPEC2006 benchmarks

as required functional units, memory operations, number of instructions, etc. Floating point benchmarks like *milc, soplex, specrand_float* prefer to run phases with floating point operation on x86 due to x86's floating point operation support. *libqunatum* utilizes SIMD support of x86 and runs those phases with SIMD operations on x86. High ILP phases of *hmmer, bzip2* are run on ARM due to less register pressure in ARM.

3.3.5 Perceptron classifier accuracy

Accuracy is defined as the percentage of time, the most affine ISA is classified by the scheduler. Migration decision accuracy of perceptron scheduler is plotted in Fig. 3.12. Perceptron is classifying the affined ISA with an accuracy of 93.4%, which is better than the linear regression based scheduler, discussed in 3.2.2 where the accuracy was 88.2% and hence there is performance improvement.

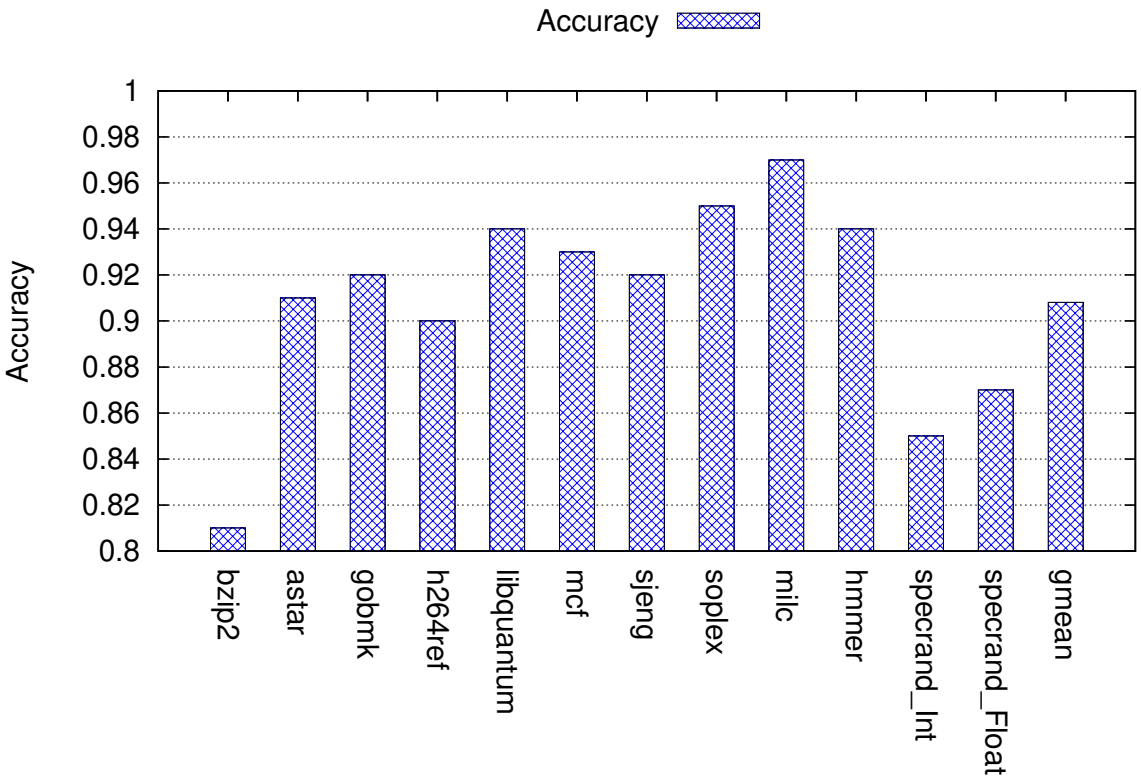


Figure 3.12: Migration decision accuracy for different benchmarks by perceptron scheduler

3.3.6 Performance gain

We have done two different types of experiments for the proposed model. In our first experiment, 70% phases of each benchmark are used for training propose, whereas remaining 30% phases of each benchmark are tested. Perceptron gives a speedup of 35.7% with respect to x86 whereas linear regression based method gives 29.6% gain with respect to x86 as shown in Fig. 3.13. Regression based scheduler classifies the most affined core on the basis of predicted number of cycles whereas perceptron does not require number of cycles to classify the phase to its most affined ISA. Slight error in predicted number of cycles can give the affinity incorrectly which is not the case with perceptron. Hence, classification scheduler turns out to be better than regression.

In the previous experiment, evaluation of the scheduler was done on the programs

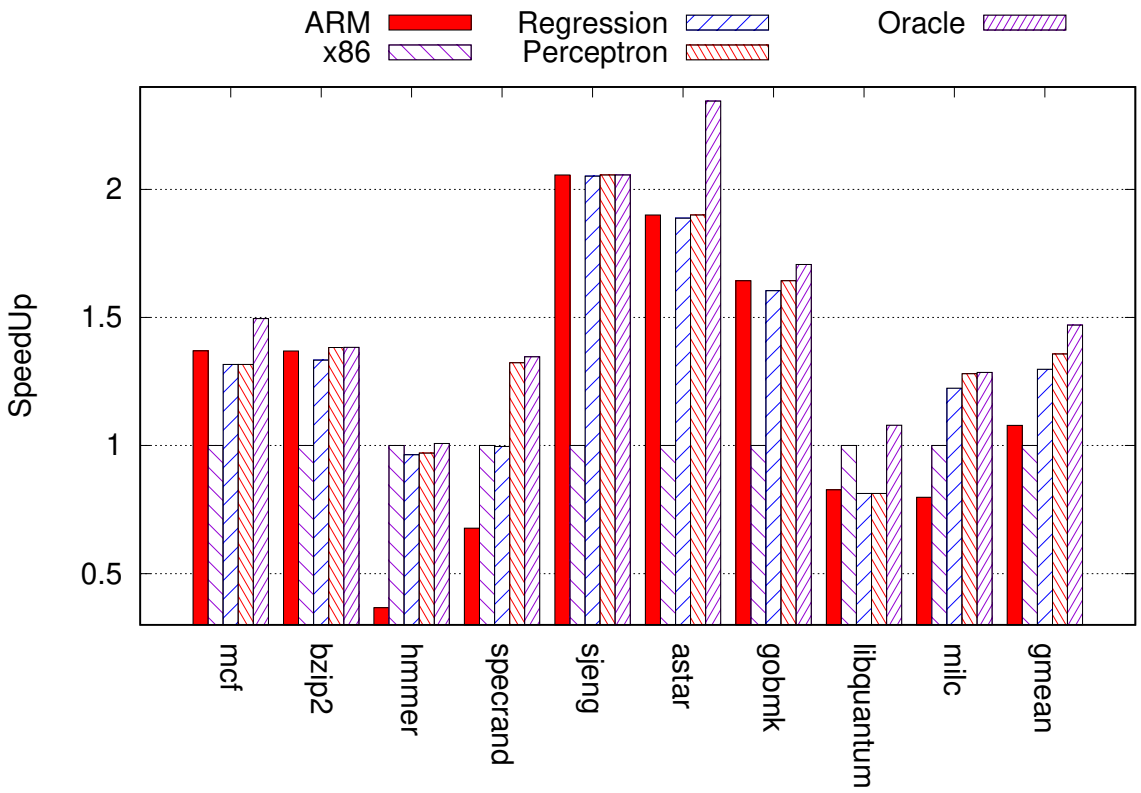


Figure 3.13: Speedup when scheduling is done using perceptron

to which it has already been exposed. Hence, to verify the generality of our scheduler, another experiment is done where the model was tested for the benchmarks which have not been used during training. The model is trained using a subset of the SPEC CPU2006 [39] benchmarks and then tested it for the accuracy on others. The proposed model is tested on the benchmarks on which it has not been trained. The results for the test data are shown in Fig. 3.14. We have achieved perceptron scheduler efficiency of 91.2%. The perceptron scheduler gives a speedup of 28.7% with respect to x86.

3.3.7 Energy efficiency results

We have done experiments to see the energy consumption during execution when perceptron based scheduler is used on HeIMC architectures. To do so, McPAT [56] has been used. The energy consumption has been computed when the benchmarks run on ARM,

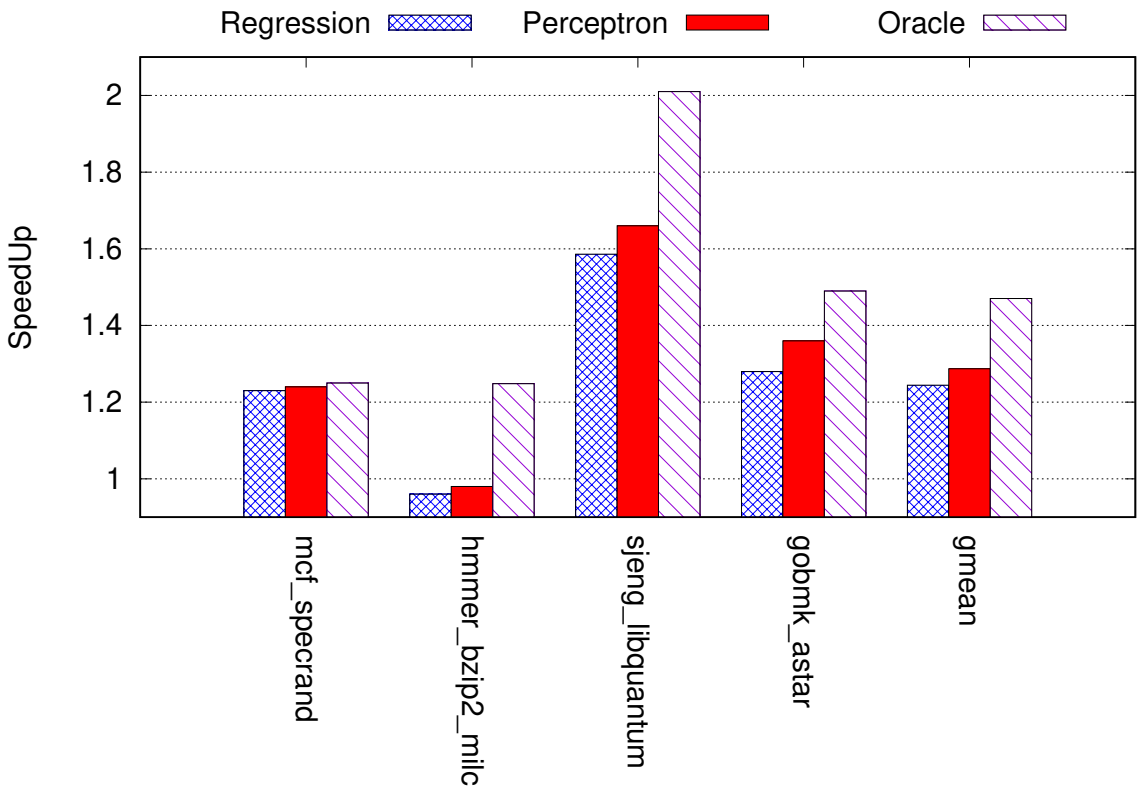


Figure 3.14: Speedup when training and testing is done on different data sets

x86, our perceptron scheduled HeIMC architecture and regression based scheduling detailed in Section 3.2.2. The results are shown in Fig. 3.15. Energy consumed in our proposed scheduler based execution is 1% less than [13]. We do not get much benefits because the scheduler is designed mainly for performance efficient architectures. Energy efficient scheduler is left for future work.

3.3.8 Hardware overhead

The proposed technique has lesser hardware overhead than linear regression based method as we require lesser number of registers. Since the training is done offline therefore in the proposed work, we need twelve registers to store the weight matrix, one register to store the bias and one more register to store the 'threshold value'. Comparator is not required in the proposed work. One 8-bit multiplier and one 32-bit adder are also required to do

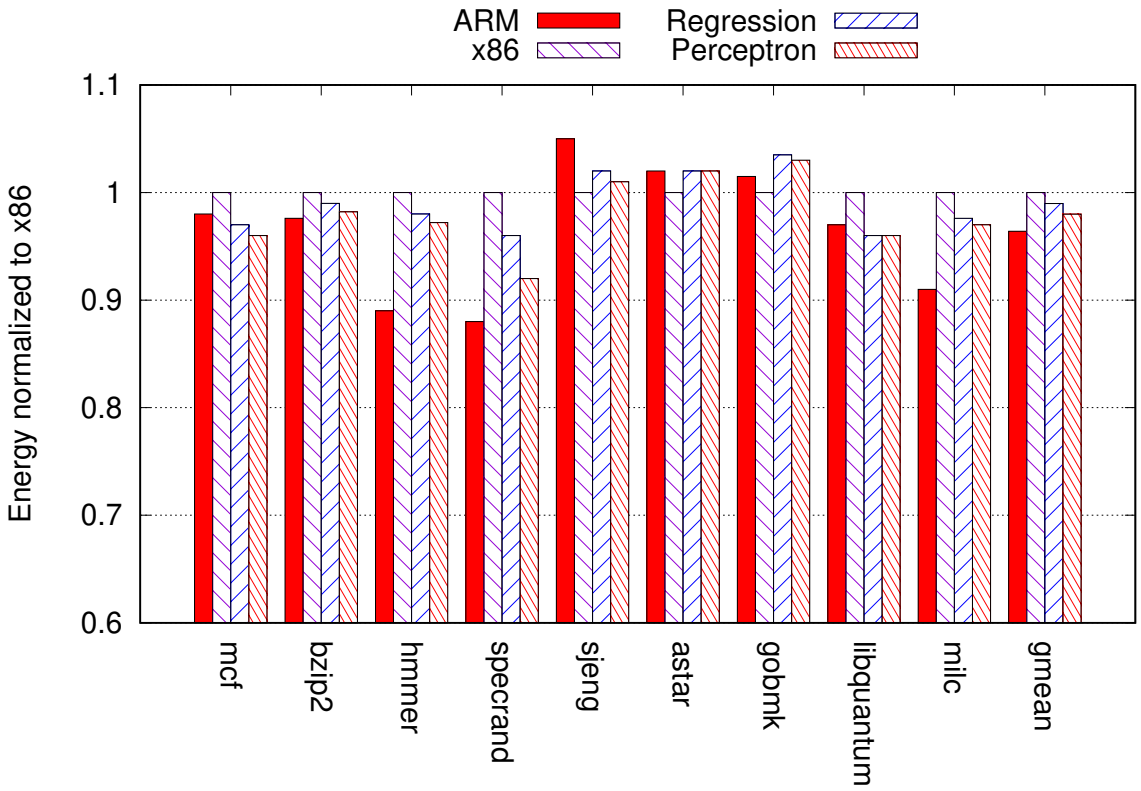


Figure 3.15: Energy when scheduled using perceptron

the computation serially. We would require one another register for temporary storage in this computation.

3.4 Conclusion

With the increase in computing requirements, exploiting the ISA affinity in different phases of a program gives significant benefits. There is very little work on scheduling the phase of program between two ISAs dynamically in heterogeneous-ISA CMP architectures. We proposed regression and classification based schedulers for scheduling in these architectures. We have shown that classification based scheduling mechanism performs better than regression based scheduling mechanisms. The classification accuracy of predicting the most affined ISA is also better for classification based scheduling.

– * – * –

Chapter 4

Fine Grained Scheduling

In the previous chapter 3, we have looked at the problem of predicting the best affined core in heterogeneous ISA architectures. We have developed a general regression based algorithm and perceptron based scheduler for phase-wise affinity prediction, with each phase consisting of 100 million dynamic instructions. *For such a long phase, the heterogeneity can not be exploited effectively, as the program behavior within a phase varies.*

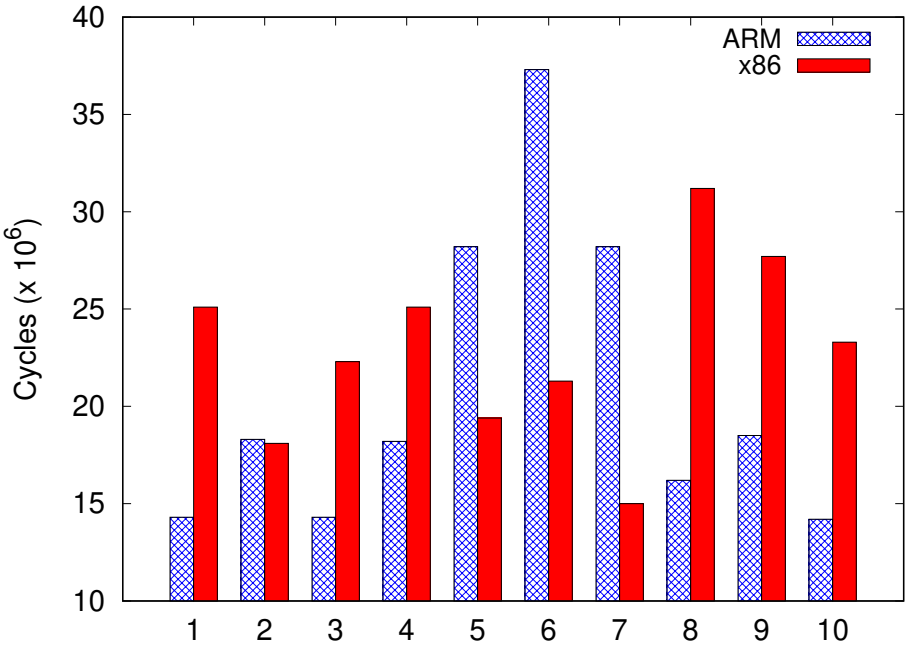


Figure 4.1: Execution time of different phases of benchmark *sjeng*

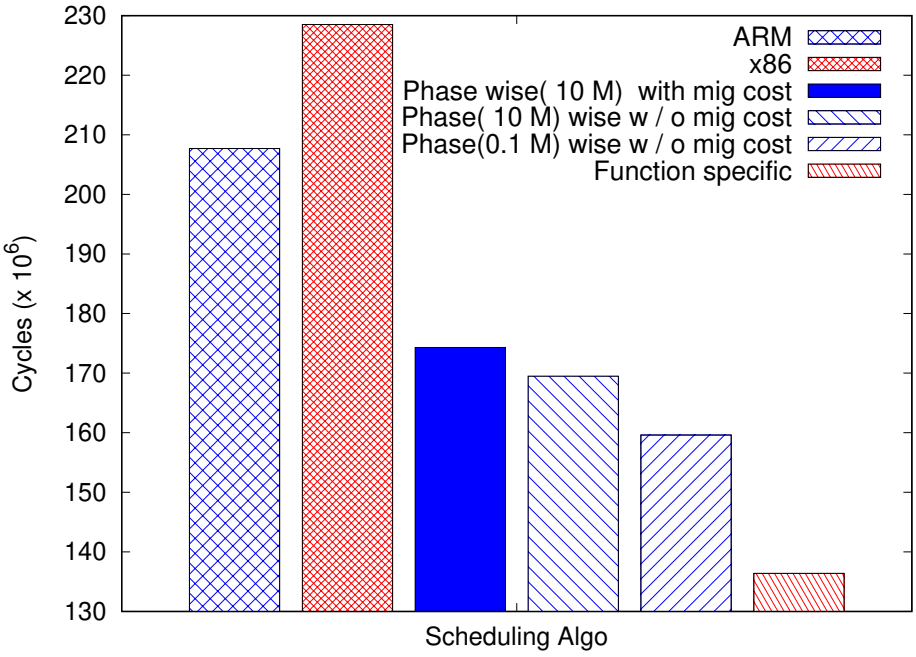


Figure 4.2: Execution time with different scheduling algorithms of benchmark *sjeng*.

The contributions of this chapter are as follows:

1. A novel function-level scheduling technique for heterogeneous-ISA CMP architectures.

2. An algorithm to determine the function's affinity towards an ISA dynamically.

3. Migration overhead reduction by 100× w.r.t [21] and 1000× w.r.t [99] using proposed scheduling method.

4.1 Motivation

To explore the phasic affinity and affinity at finer level, we executed 10 different phases (each of 10 million dynamic instructions) of *sjeng* benchmark from SPEC CPU2006 on a system with cores supporting two most common ISAs, i.e, ARMv7 and x86-64, where x86-64 is CISC with floating point support and ARMv7 is RISC without floating-point

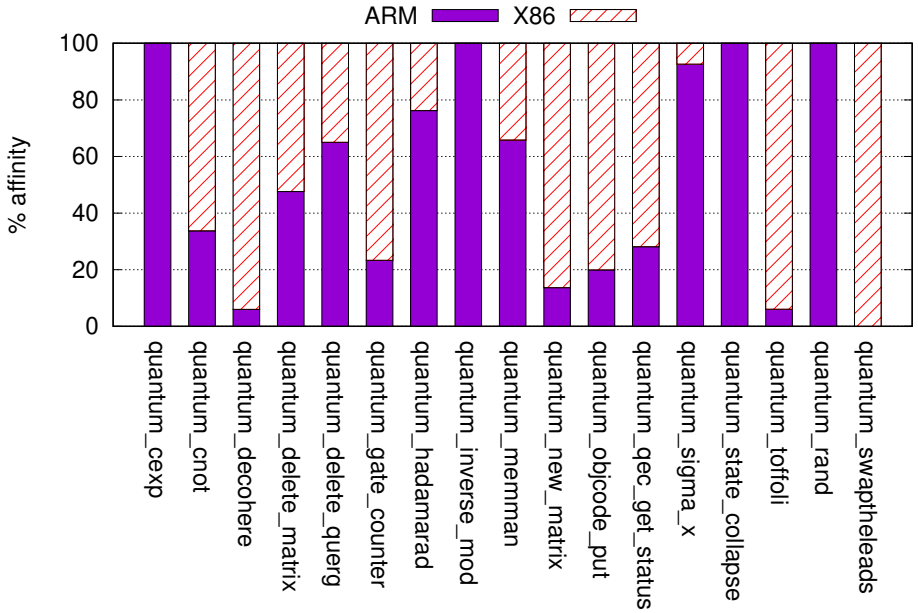


Figure 4.3: *libquantum* function affinity

support. Fig. 4.1 shows the number of cycles taken by each ISA for each phase. Fig. 4.2 (bar 5) demonstrates that shorter phases can give better performance if we ignore migration overhead. Hence, the switching overhead (up to 100 µs in existing schemes) turns out to be the bottleneck in exploiting the fine-grain opportunities. *This chapter focuses on harnessing performance at finer granularity while maintaining equivalence points which is missed in existing state-of-the-art schemes.* Therefore, to exploit program's heterogeneity more effectively, we introduce a fine-grained function-wise scheduling technique, in which every function is scheduled to it's most affined ISA. Fig. 4.2 (bar 6) shows if *sjeng* is scheduled function-wise, a speedup of 27.2% is achieved compared to phase-wise scheduling and 65.8% compared to the case when completely scheduled on x86 ISA. Therefore, *function-wise scheduling is a better candidate* for exploiting program's heterogeneity at a finer granularity than phase-wise scheduling with each phase of 100 million dynamic instructions. Moreover, it also reduces migration overhead as only function arguments have to be transformed from one ISA format to the other in this scheduling technique. This avoids a complete stack transformation which was required in previous

proposals [28, 99]. In principle, function-wise scheduling of a program is a promising technique. However, it raises several issues, such as algorithm to determine affinity, dynamic switching mechanism, and state migration policy.

Fig. 4.2 indicates that different functions have affinity towards different ISAs. This is due to multiple factors, such as *code density, dynamic instruction count, register pressure*, etc. which varies for each function. Functions are individual code snippet which have their own properties and hence affinity. If a particular function is executed multiple times on an ISA with similar data input, the control flow remains same. Therefore, it executes the same set of instructions, hence, it is affined towards the same ISA in each execution iteration. This affinity of a function towards an ISA may only shift by the change in behavior of input data which is passed to the function through parameters or global variables.

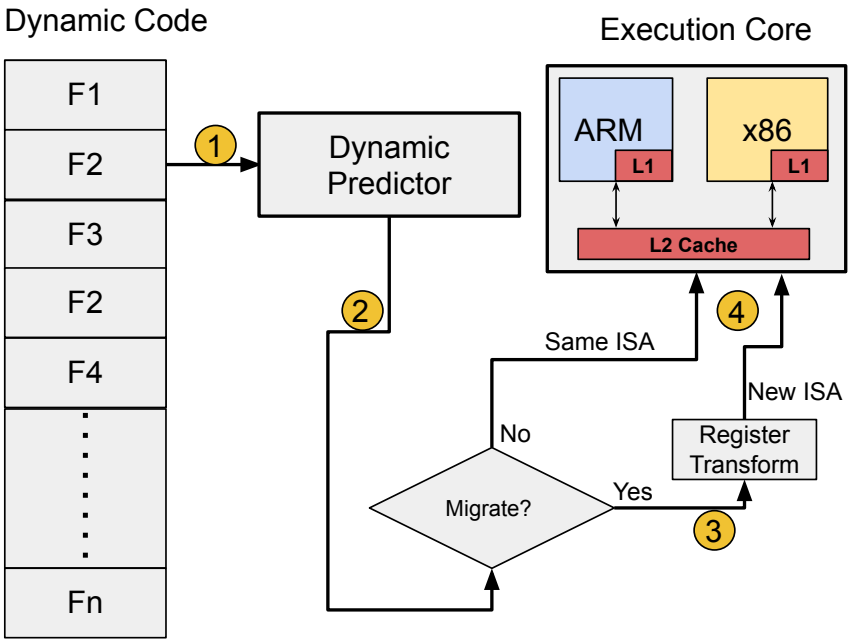


Figure 4.4: Function scheduling flow chart

To show the affinity variation of functions, *libquantum* benchmark was simulated using the same inputs on both ISAs (x86-64, ARMv7) to measure the execution time of

its functions. The execution time measurements were recorded for every call of a function. A function call is said to have an affinity towards an ISA if the execution time of particular call is lesser for that ISA. All the occurrences of a function affined to a particular ISA were counted as shown in Fig. 4.3. Some functions like `quantum_decohere`, `quantum_sigma_x` are biased towards one specific ISA. However, for some functions such as `quantum_delete_matrix`, affinity changes often based on input. Therefore, we have to develop a technique to schedule a function to its best affine core (ISA) dynamically as affinity changes during execution.

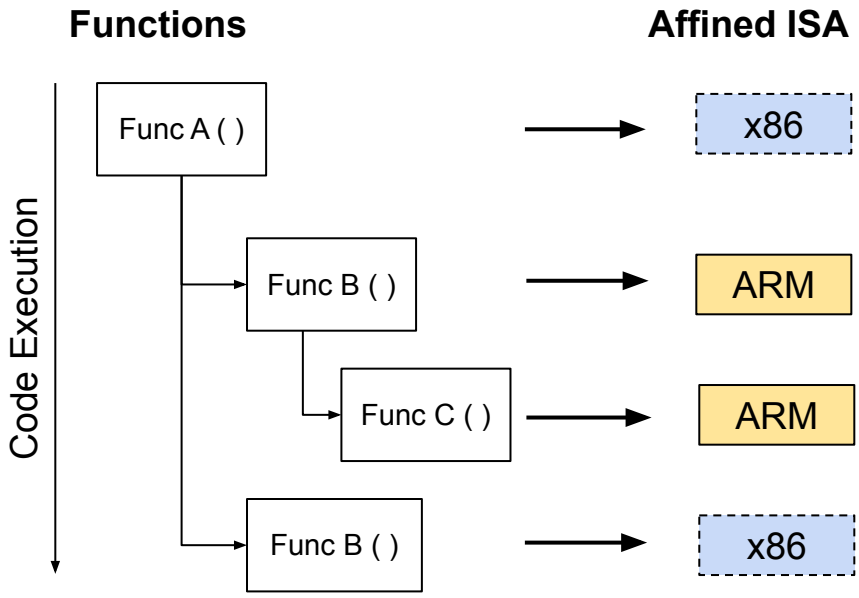


Figure 4.5: Function order

4.2 Fine-grained scheduling

Flow chart for function-wise scheduling is shown in Fig. 4.4. Since the affinity of any function towards an ISA changes during execution, so this affinity has to be determined dynamically. We are proposing to schedule and execute every function on it's best affine ISA. This is done by a dynamic predictor as shown in Step-1 in Fig. 4.4. A heuristics based approach has been developed in this work for demonstrating function wise

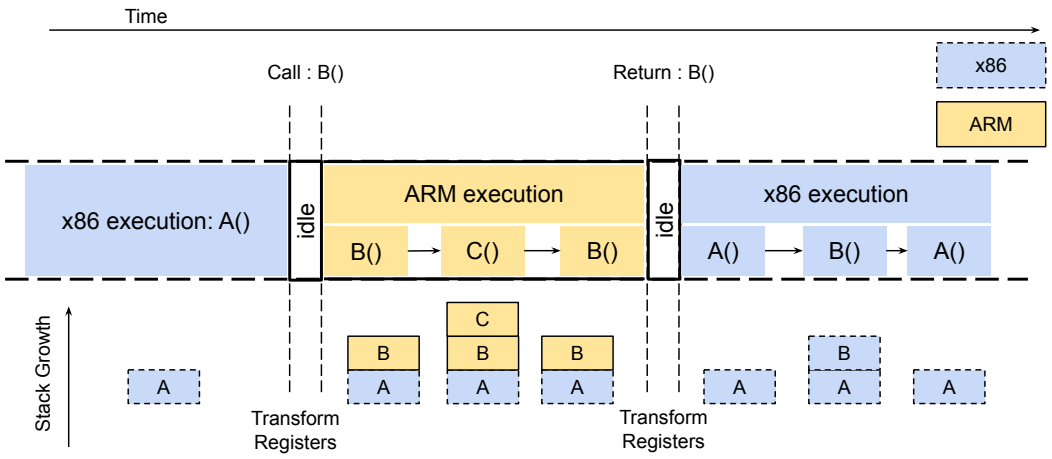


Figure 4.6: Execution flow for example in fig 4.6

scheduling. Further improvement of the heuristic can be explored in future proposals.

The migration decision for a function is taken just before a function is called. Therefore, the function is executed on it's best affine ISA and the stack frame for the function is also formed in it's best affine ISA format. For that we need to transform only local parameters passed by the caller function to callee function. Rest of the memory i.e. global and heap, is common across both ISAs. Hence we do not need to handle heap, pointers and global memory. The memory map is consistent across both the ISAs as in [21].

The migration technique is explained in Fig. 4.5 with a simple example program with three functions A(), B() and C(), called in the same order. The execution flow for the same example is shown in Fig 4.6 along with stack growth, where the idle time depicts migration. Assume initially functions A(), B(), and C() have affinities for x86, ARM, and ARM respectively. In our proposed model, we need to transform only the registers that correspond to the parameters passed to function B(). Function C() has an affinity towards the same ISA as B() has, i.e., ARM ISA. Hence, no transformation is required when function C() is called from function B() and when it is returned back. Later, due to data dependent behavior of function B(), assume its affinity changes to x86 when it is called second time. Therefore, register transformation is needed before function B() is executed. Our study shows that for most of the migrations, the transformation has

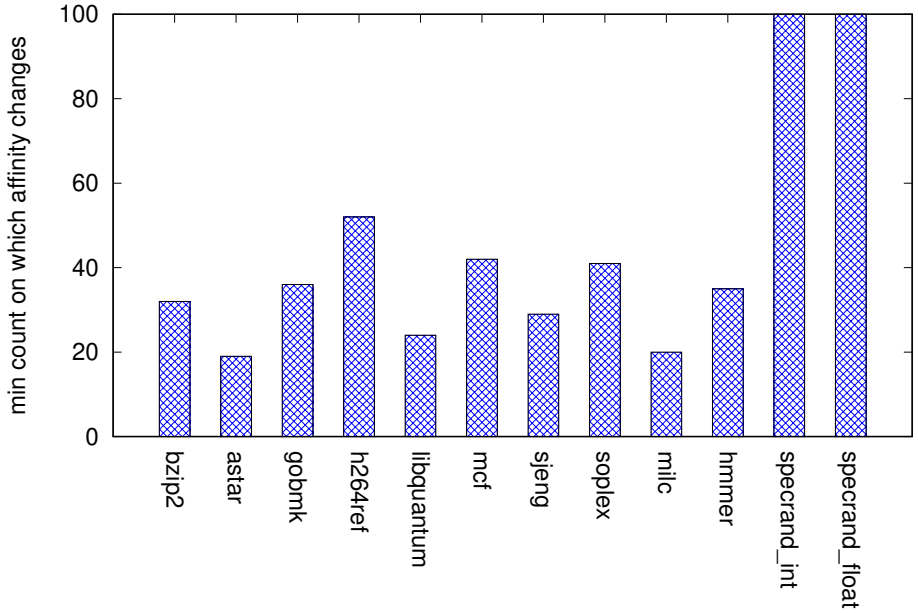


Figure 4.7: Minimum count on which affinity changes

to be performed only for a few registers, leading to minimal migration overhead. Once migration is completed, program is executed on its affined ISA as shown in Step-4 in Fig. 4.4.

4.3 Heuristics based scheduling approach

It has been observed on benchmarks that the affinity of a function does not alter too often for a spell. We have experimentally observed that the spell is approximately 20 consecutive calls as shown in Figure 4.7. We have done statistical analysis of the SPEC benchmarks and found out that a function's behaviour changes mostly after 20 executions of the function as shown in the below Fig 4.7. In Figure 4.7, the Y-axis shows the minimum execution counts for which the affinity of any function does not change. For example in 'bzip2' the affinity of any function does not change before 32 executions. In a few exceptional cases, the behavior changes earlier than this. Looking at past behavior, a sampling-based technique is developed for dynamic prediction, and affinity is recorded

in Affinity table. The affinity of each function is decided for every 20 consecutive calls of the function. Once affined ISA is decided, say x86, the function is made to run on x86 for the following 19 consecutive calls and on the other ISA, ARM, for the 20^{th} call. Execution time is noted for the last two calls, i.e., 19^{th} & 20^{th} of the function, one on x86 and one on ARM. If 20^{th} call (ARM) has lesser execution time compared to the 19^{th} call (x86), then function affinity is changed to ARM, and the next 19 calls are executed on ARM, else affinity stays with x86 and the next 19 calls are executed on x86. This affinity is stored in the Affinity table and used for the next 19 calls. These 19^{th} and 20^{th} execution time are stored for each function in the Affinity table. Please note that, we do not change the affinity of a function if that function is in open state (under execution) to avoid the ambiguous-affinity state caused by recursive calls. For this purpose, we keep an extra 1-bit flag in the Affinity table to store whether a function is opened anywhere in the program. If Affinity table is overflowed with opened functions, we do not calculate affinity for next upcoming function and it is executed on x86 ISA by default.

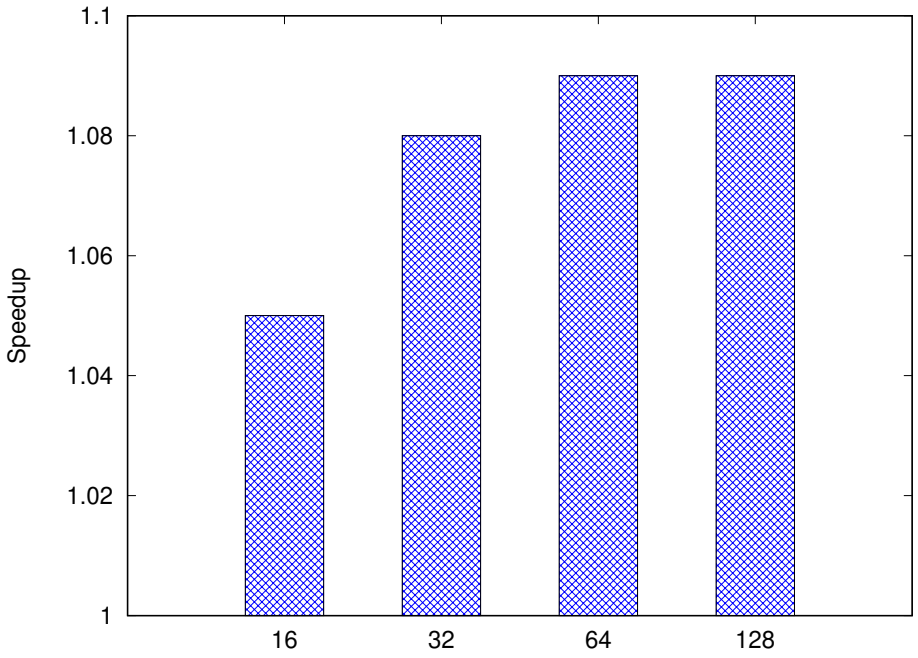


Figure 4.8: Speedup on different size of Affinity table

The size of the Affinity table is 32 entries in our work and it stores the affinity record

for 32 unique functions. When a new function comes, the entries for closed functions in the Affinity table are evicted with LRU policy. In corner cases, if the total number of active-functions in a program exceeds 32, then those new functions (i.e. from active-function number 33 and onwards) will run on x86 ISA by default. We have done a study between performance and size of the Affinity table. The Figure 4.8 shows that we are getting good performance when we take 32 entries in the Affinity table. If the Affinity table size is increased to 64 entries then performance gain is increased by 1.2% only with more than double hardware overhead. Above 64 entries, there is not a significant difference. Hence, 32 entries are taken. Each entry of the Affinity table corresponds to a unique function. Once the affinity decision is taken in Step-2, there may be a need for migration for correct execution, as shown in Step-3 in Fig. 4.4.

Table 4.1: Core configurations for fine-grained scheduling

Design Parameter	ARM	x86
Architectural Registers	32	16
Cache line size(bytes)	64	64
LSQ size	32	32
Fetch width	4	4
Decode, Dispatch, Issue Writeback, Commit Width	4	4
Instruction Queue entries	64	64
ROB entries	192	192
DCache,ICache size	32KB	32KB
L2 Cache size	256KB	256KB
SIMD Support	No	Yes

4.4 Results and analysis

In order to demonstrate the effectiveness of the proposed scheduling algorithm, we have simulated SPEC CPU2006 benchmarks using Gem5 [9] simulator. These benchmarks are compiled using 'O2' and 'O3' optimization in gcc for x86-64 and cross compiler built for ARMv7. The migration results are given with 'O2' optimization so as to fairly compare the migration overhead of our work with the previous work of state-of-the-art DeVuyst et al.[28]. The core configurations used in simulations for both ISAs are mentioned in Table-4.1. Each ISA core has a private L1 cache and a shared L2 cache.

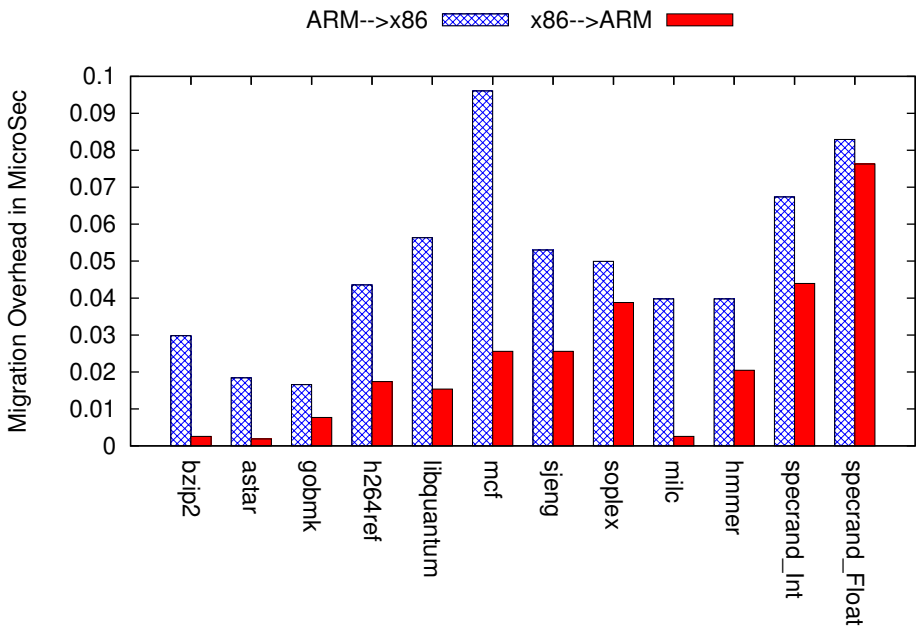


Figure 4.9: Migration overhead for function-wise execution

4.4.1 Migration overhead

The cost of function level migration overhead is less, since only a few register values have to be transformed compared to whole stack transformations required in previous work [99]. The migration time obtained for function level migration is shown in Fig. 4.9 (in the range of 5 ns to 95 ns). Some benchmarks e.g., *mcf, specrand* have higher migration

costs compared to others as they have comparatively more number of parameters in their functions. Due to different register pressure, x86→arm has more movement from stack to register, which causes more load operations. Whereas arm→x86 possesses the opposite behaviour. Hence it has more store operations, resulting in more overhead. A function has been developed to do the job of migration mechanism based on the number of variables that need to be migrated. This function is integrated with Gem5[9] to compute migration overhead.

4.4.2 Performance results

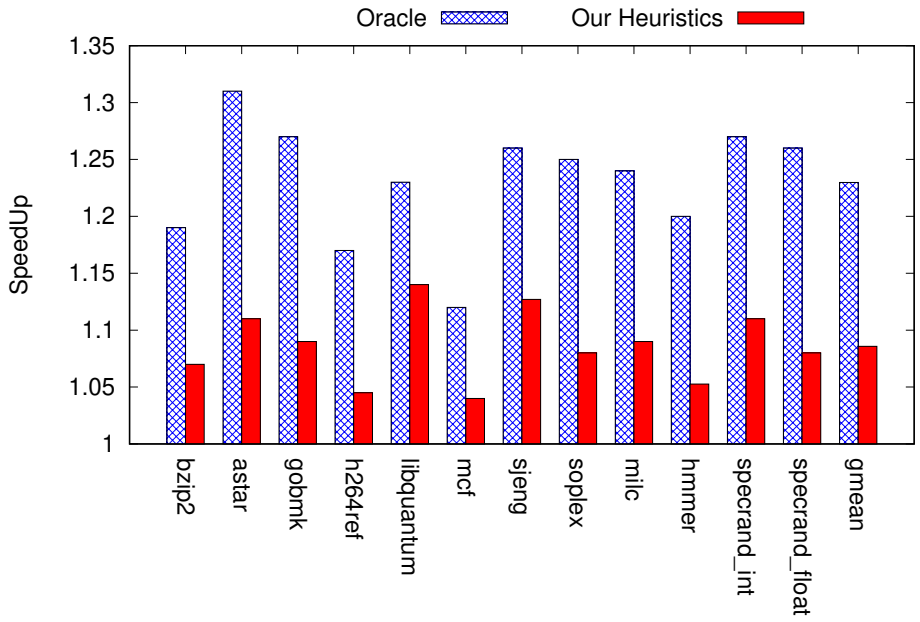


Figure 4.10: Speedup of function-wise migration with respect to heterogeneous-ISA architecture

We have compared the results with the state-of-the-art research by [15] as shown in Fig. 4.10. The proposed approach achieves up to 22.9% speedup using oracle based scheduling mechanism in both cases (ours and in [15]) and the proposed heuristics technique achieves a speedup of 8.4%. The speed up achieved with O3 optimization for oracle based scheduling is 22.1%. Benchmarks like *mcf, h264ref*, and *hmmer* are affined

towards one ISA for the majority of their execution time, hence these benchmarks do not give additional performance gain over heterogeneous-ISA architectures Boran et al. [15]. However, for some benchmarks such as *libquantum, specrand*, the affinity of functions change quite often depending on the input. Therefore, our approach gains significantly by scheduling the functions on the most affined ISA dynamically. Benchmarks such as *specrand-int, specrand-float*, and *milc* have less than 32 unique functions executed multiple times, thereby achieve a good performance gain. *gobmk, astar* have more than 32 functions, hence the storage limit of Affinity table was not enough to store all the affinity. Increasing the size can improve the performance, however, it will result in more hardware overhead.

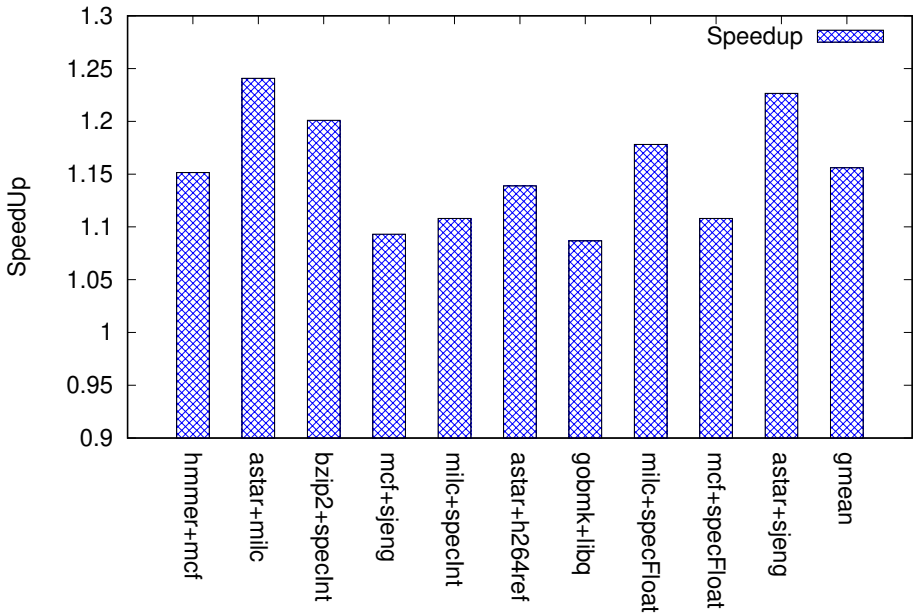


Figure 4.11: Speedup of function-wise migration with respect to heterogeneous-ISA architecture for multi-workloads

4.4.3 Performance results for multi-workload

Although our primary focus is on single threaded performance enhancement, multi-workload benchmarks have also been executed to see the multi-threaded behavior for

the proposed approach. The results are shown in Fig. 4.11. The speedup is with respect to the case when each benchmark is set to run on one of the ISAs (the best performing choice out of two combinations). We achieve an average speedup of 15.6%.

4.4.4 Hardware overhead

Affinity table contains entries for 32 functions. Each entry requires 16-bit space to store the hashed value of PC-address, one-bit flag for affinity, one-bit to store if the function is opened anywhere in the program and 5-bit counter, 5-bit LRU, two registers of 4B each to store execution time of 19^{th} & 20^{th} function call. One 32-bit wide comparator is also required to compare the execution times for different ISAs. Total hardware overhead is 368 Bytes.

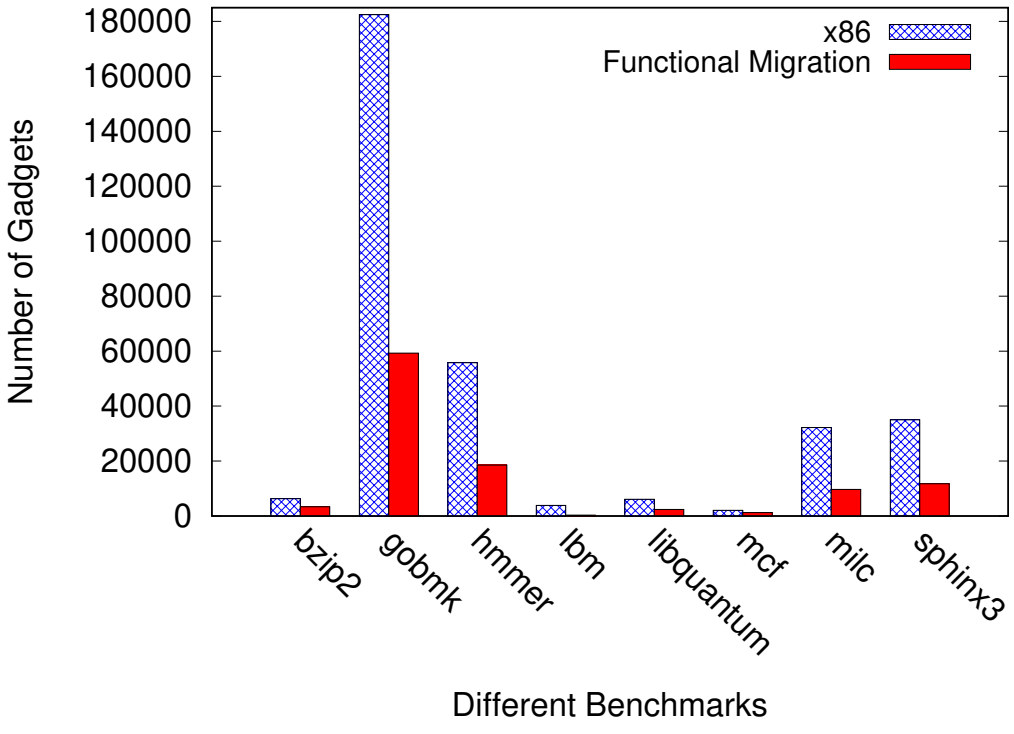


Figure 4.12: Number of gadgets in different benchmarks

4.5 Security analysis

In our our proposed algorithm, the number of ROP gadgets are decreasing by 50% as shown in Fig 4.12. Hence our scheduling algorithm is more secured then all previous given algorithms. In our algorithm, the ISA under execution will keep on changing. Hence, chances of having a successful attack with the use of multiple gadgets decreases. In a heterogeneous ISAs, when the phase is of 10 million length the possibility of attack increases. Our proposed algorithm is having higher probability to switch ISAs more frequent then 10 million.

4.6 Conclusion

The proposal introduced a novel fine grained migration strategy for heterogeneous-ISA CMP, called function-wise scheduling to exploit heterogeneity in ISA more efficiently. Our results show that most of the functions have an affinity to some specific ISA. It requires only the transformation of registers during migration, hence reduces the overhead by more than 1000x (compared to the state-of-the-art [99]). The proposed fine grained function wise scheduling achieved 22.9% in comparison to the state-of-art [13].

– * – * –

Chapter 5

HIDC Architecture

Heterogeneous ISA architectures emerged as a good alternative to enhance single-threaded performance (and multi-threaded programs as well). If any program is divided into multiple parts and each part is scheduled on most affined ISA, the significant performance gain is achieved. The heterogeneous-ISA CMPs could enhance the single-threaded performance by up to 40% [15]. For any single-threaded program, only one core is active in these architectures. Due to this following are the limitations in these architectures.

1. The resources of idle cores are not utilized optimally.
2. All idle cores dissipate static power continuously.
3. On affinity change of a program, the context switching is done from one core to other core. This leads to larger context switching (migration) overhead and results into reduction in performance.

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 [59], [43], [79] 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 [99]. 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.

To reduce the migration cost, 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. [99].

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 this chapter 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.

2. 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. *Simultaneous Transformation* for migration of stack-memory from one ISA to another with reduced migration overhead.

5.1 HIDC architecture

The HIDC consists of a single out-of-order dynamic core which supports multiple ISAs. Fig. 5.1 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. It is independent of the fetch stage, 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. HIDC contains separate decoders for respective ISA, while all of the pipeline stages are shared among all ISAs with appropriate resources as indicated by the CIB. 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.

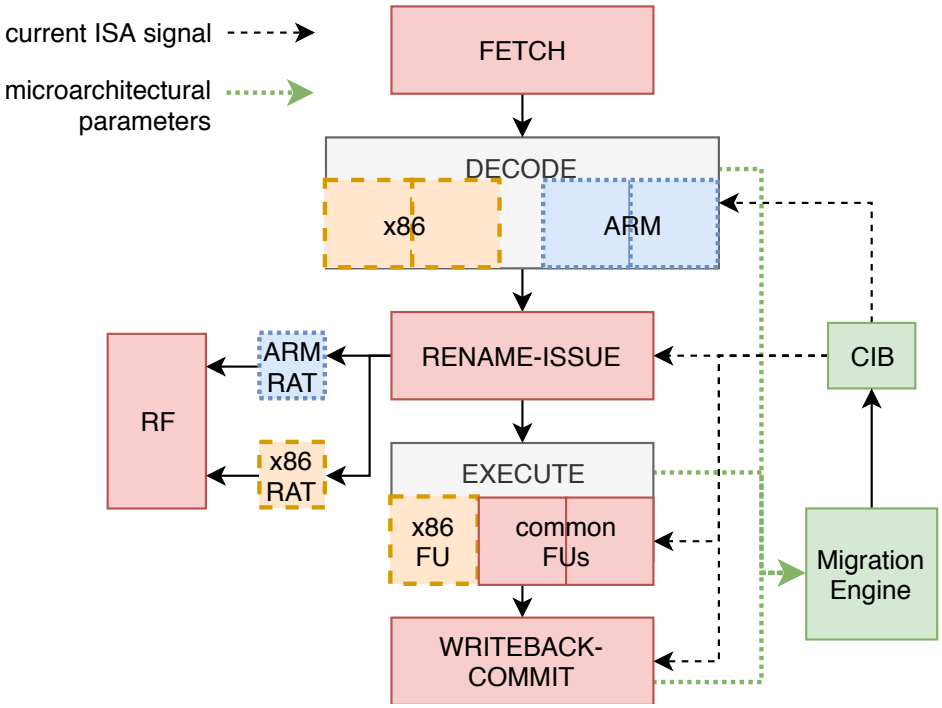


Figure 5.1: High level architecture of HIDC pipeline

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. This 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.

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, as the buffer plays the common role across all 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 present for each ISA, 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 [99] 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.

5.2 Scheduling

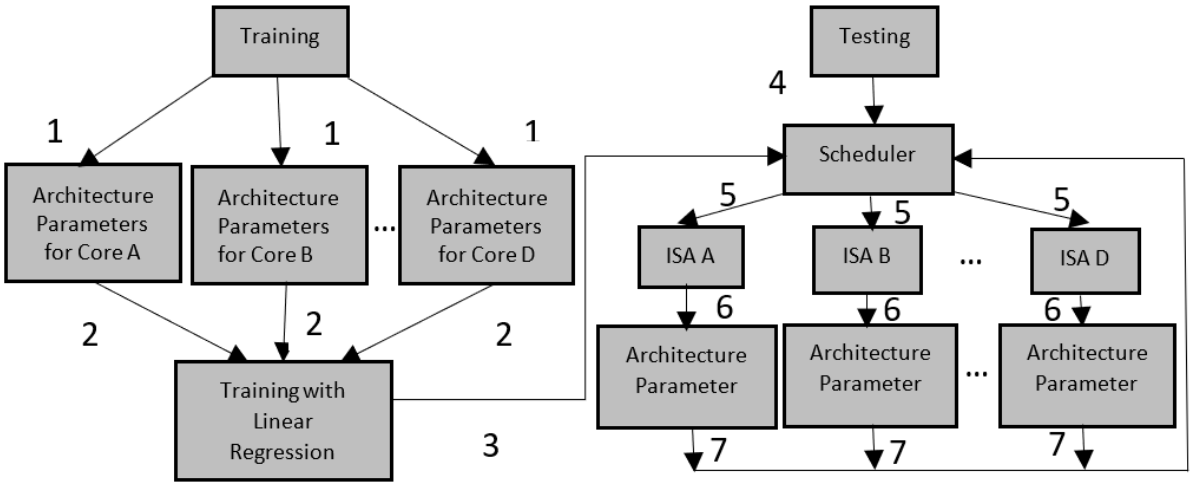


Figure 5.2: A schematic representation of the scheduling model

To get maximum performance gain in HIDC architecture, every phase of the program should be executed on the most affined ISA. Finding most affined ISA for any phase is a classification problem. Therefore, linear regression classifier has been used in our work for this purpose. The scheduling algorithm has to be dynamic, hence is it implemented in hardware. The scheduling is done phase-wise. For that, the program is divided into multiple phases. The length of every phase is kept sufficiently large to minimize performance loss due to the overhead caused by migration. However, the phase length can not be kept very large, since, keeping it large limits the exploitation of ISA diversity. The phase length is taken of approximately 10 million dynamic instructions with respect to x86 ISA. Each phase starts and ends on equivalence points. The corresponding phase is also taken in a similar way for other ISA.

The schematic representation for the scheduling model is shown in Fig. 5.2. For each phase, the scheduler takes the decision on seeing the behaviour of few instructions. We have found that on the basis of 10 million instructions, affinity of the next phase of 10 million instructions can be decided. We extract eleven micro-architecture parameters for each phase to see the behaviour of the phase. This is discussed in Section 5.2.1.

On the basis of micro-architectural parameters of the current phase, the affinity for the next phase is predicted. The migration will take place if it is required. Here, we are assuming the behaviour for the next phase to be similar to the current phase. Wherever the behaviour changes, our scheduler will not be able to give the correct decision. Note that in the Fig. 5.2, the steps 1-3 are for the training of the model and are done offline whereas steps 4-7 are for testing the model at run time and are done online dynamically.

5.2.1 Extraction of micro-architectural parameters

The migration decision is taken on the basis of eleven micro-architectural parameters. Table 5.1 lists these parameters which are measured for every phase similar to [15]. The twelfth parameter 'number of cycles' is extracted as well to use during model training. This parameter is used to label the training data. It is also used to evaluate the scheduler's accuracy. These parameters are recorded by the MEC as shown in Fig. 5.1 from multiple stages of the pipeline and capture the performance of that phase. For most of these parameters, today's CPUs have counters already [93]. ILP and MLP are estimated similar to [59].

5.2.2 Scheduling model

Given those parameters, we now describe our linear regression based scheduling model. Given a source ISA in execution (ISA_A) and a target ISA (ISA_B), our linear regression model for estimation of the number of cycles is given by:

Table 5.1: Micro-architectural parameters used for migration decision

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

$$\begin{aligned} Cycle_B = & K + a_1.(L1DcacheMiss_A) + a_2.(L1IcacheMiss_A) \\ & + a_3(L2cacheMiss_A) + a_4.(SQFullEvents_A) \\ & + a_5.(ROBFullEvents_A) + a_6.(IQFullEvents_A) \\ & + a_7.(BranchMissPrediction_A) + a_8.(MLP_A) \\ & + a_9.(ILP_A) + a_{10}.(FloatInstruction_A) \\ & + a_{11}.(DynamicInstructionCount_A) \end{aligned} \tag{5.1}$$

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

In this work, we consider two ISAs namely x86, and ARM. for both ISAs we have two cores which are small and big, We build a total of 12 regression models: for each of the core of each ISAs, to all other cores and ISAs. We are migrating to small core

if the performance loss is within some specific limits. This is done to save the energy. A limit of performance loss of 5% has is kept in this work. The idea here is that these models are built offline and incorporated into the processor with the coefficients of the model stored in special registers. Then, when programs are executed, these models continuously predict the performance on the other two ISAs and pass these predictions to the scheduler. The migration will take place if it benefits after considering migration overhead.

5.3 Procedure for migration

As the affinity of a program changes, the program state has to be migrated efficiently and correctly from the current ISA to another. The state of all variables and data used by the program have to be kept as expected by the program, to continue smooth execution and avoid runtime errors. Certain parts of the program state can be maintained common between both ISAs at compile time, while the rest have to be transformed at runtime during migration. At compile time, a single *fat* binary is generated for the program common to both the ISAs called the Combined Program Binary (CPB). Instructions are compiled for both ISAs from the source code (including libraries) and stored in the CPB. The CPB also stores migration metadata which is used by the MEC at the time of migration.

Table 5.2: Memory System

Memory	**Part of memory**	**Migration**
Code memory	Code memory	No
Data memory	Global data	Yes
	Heap	No
	Stack	Yes
	Register files	Yes

Note that for all explanations, we assume the case of transforming from x86 to ARM.

When program migrates from x86 execution to ARM execution, the next equivalence point is determined and the HIDC executes using x86 ISA till that point. Then the execution is paused and necessary state transformations are performed to make the memory image consistent with the ARM execution. The HIDC then switches to ARM execution and starts executing from the corresponding equivalent point in ARM-compiled code. DeVuyst et al. [28] have described in detail how the memory is handled for a program switching between ISAs. Code memory is generated by compiling the program for both ISAs and placing the compiled code in adjacent sections. Functions' entry points are considered as equivalent points, hence functions are arranged in the same order and padded at the end for both ISAs. This ensures that each corresponding function entry point is at the same offset from its section and also that both sections are of same size. To minimize the amount of transformations that occur during migration, all data memory (program state) except for stack and register file is stored in the same format for both ISAs.

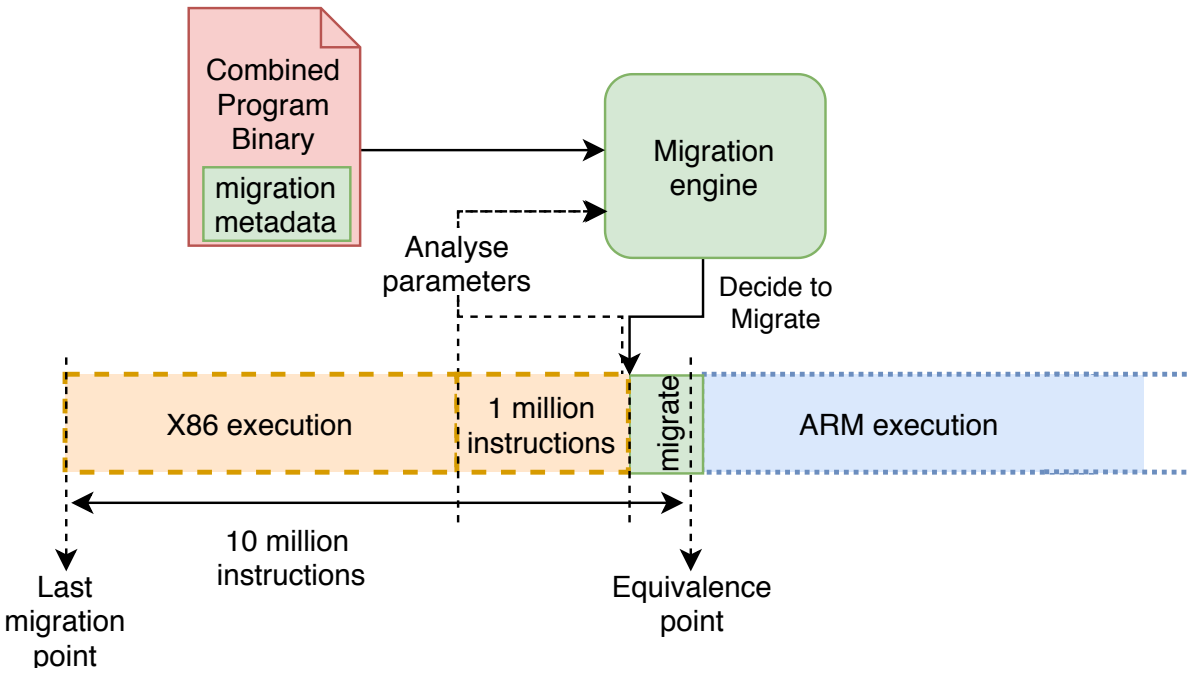


Figure 5.3: Flow Chart for Migration

Different part of memory are shown in the Table 5.2. Program state consists of the

following items:

- **Program Counter:** The PC initially points to x86 instruction and it has to be updated to corresponding ARM instruction. The code generated by the compiler is padded and ordered in such a way that each corresponding function can be found at the same offset for both ISAs. Migration of PC involves a simple offset addition or subtraction.
- **Register File:** The compiler performs live variable analysis which tells us what variables are stored in which registers so that they can be transferred accordingly. To maintain consistent data size of registers both ISAs have to be of same bit-width, i.e., 64-bit x86 with 64-bit ARM. There may be data in callee-saved registers on the stack for one ISA, however the data may have to be present in register file for second ISA so another mapping is generated for them at compile time. The number of such registers is small (<20), so it does not affect migration time significantly.
- **Data memory:** The address for data variables is either hard-coded in `load` instructions or stored in pointers. The correct virtual addresses of each data variable need to be maintained after migration. Data memory can be split into the following sections:
  - **Global variables:** They are placed in a common location in the binary with the same alignment and size for both ISAs.
  - **Heap Memory:** Heap is allocated by functions like `malloc` and using the same implementation in both cases will ensure heap is built in the same way for both ISAs. All heap variables will have the same addresses regardless of the ISA being used at the time of allocation.
  - **Stack:** The compiler optimizes the location (stack-slot) of each variable according to number of registers available and size of data which is highly ISA dependent. If the stack is allocated in same way for both ISAs, it will affect the

performance of the program due to different register pressures. Hence stack locations of each variable are decided by default compiler optimizations and is not interfered with during compile-time. Only the size of a function's stack frame is changed 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 is generated at compile time between the stacks of the two ISAs and a stack transformation is required during migration. The procedure for stack transformation is described in detail in Section 5.3.1.

5.3.1 Inplace stack transformation

To avoid unnecessary migration overhead and simplify the procedure for stack transformation, properties like direction of stack growth and endianness are maintained same across ISAs. x86 stack growth is always downward, because push and pop instructions in ISA are implemented to decrement/increment. In ARM, growth is possible in both directions, but it is generally fixed by the OS to be downward growing stack. 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. Venkat et al.[28] recommend that the two ISAs used for Heterogeneous core should have the same endianness. 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.

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 which stack-frame belongs to which function. Once the stack frame and function is identified, the stack is modified in-place to save on time and memory-accesses during migration. The in-place modification requires going in a sequence in which no data on the stack is overwritten.

Listing 5.1: 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
```

The transformation is conducted in the following order to ensure no data is overwritten.

1. **Stack-Register:** Variables and arguments passed during a function call are not always placed on the stack. Depending on the calling convention, available registers in the ISA and number and size of the variables themselves, they can also be placed in the register file. This placement is decided in the most optimal way by the compiler. Any changes to this placement may lead to less optimal code. This means that some variables may be placed on stack and be expected in the register file, so they need to be copied. The number of loads required is capped at the maximum number of architecture registers among both ISAs. The target register file is in the idle core and contains no data prior to migration, so a copy can be done directly without overwriting any data. This will free up some slots in the stack which can now be safely overwritten in the next steps without having data loss.

2. **Stack-Stack:** One local variable in x86 stack layout may be expected in another

location in ARM stack layout. Again, the stack layout has been decided by the compiler in the most optimal way for code execution and we do not change this expected layout. Hence we need to transform the stack to ARM ISA's expected layout. The variable's size and endianness do not change, so this becomes a simple memory to memory copy. For performing the transformation in place on the stack, it is important to ensure no data is overwritten while doing a memory-to-memory copy. By looking at the stack layout of x86 and ARM, a mapping is created from source address to destination address for each variable. For each variable, we determine which other variable is present in the x86 stack for the destination address of our copy. We add a dependency between these two variables to ensure correct ordering of the copy instructions and in this way, populate a dependency graph. Dependency may be added to any number of variables because the size of our variable can be arbitrary and it may overwrite more than one variable at the destination address. We traverse the graph in topological order to ensure that each copy instruction does not overwrite any data. In case there are cyclic dependencies, we need to resort to mapping at byte level to ensure that each address has only one dependency, or break the cycle by using temporary memory location to do a two-step copy. This dependency and ordering is done at compile time because the mapping is known once compiler generates the stack layouts. Only a sequence of copy instructions needs to be stored in the migration metadata to be accessed at runtime. The transformation sequence can be visualised using Fig. 5.5, which clearly shows how moving one variable will overwrite something else. Algorithm 5.1 lays out a method for migration using the sequence and two temporary variables.

3. **Register-Stack:** Register to Stack copies are done now. The previous stack-to-stack transformation has created empty slots where variables residing in the register file are supposed to be copied. This is because those slots do not come up as a destination address for any of the stack variables. Hence, a copy can be done directly without overwriting any data.

4. **Register-Register:** Variables present in register file of x86 may sometimes be expected in register file of ARM. A register file to register file copy needs to be done in this case. We have ensured that both ISA have same bit-width and endianness so no data manipulation is required in between. Register to Register copies are done in parallel to the above three steps because they will not clash with any of the copies happening above.

5. **Pointer handling:** Functions may be passed 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. Algorithm 5.2 shows a pass-by-reference (`%2`) and a pass-by-value (`%3`) function parameter. `%2` and `%3` are allocated on the stack on line 2, 3. `%3` is loaded on line 6 and passed by value in function call on line 7. This value and other pass-by-value parameters will not need to change when `functionA` is migrated. Pass-by-reference parameters like `%2` are pointers which are locally stored by the function's stack frame. When `functionA` is migrated, the variable is moved to another location on the stack and the value of pass-by-reference pointer stored in stack of `functionB` needs to be updated. 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, so this step is skipped most of the time and does not add to migration time significantly.

Listing 5.2: LLVM-IR example for pointers

```
1. define @functionA
2.   %2 = alloca i32
3.   %3 = alloca i64
4.     .
5.     .
6.   %51 = load i64, %3 i64*
```

```
7.  %52 = call @functionB(%51 i64, %2 i32*)
```

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 seen in Fig. 5.3.

5.3.2 Migration overhead reduction

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 Heterogeneous-ISA approach, implementation requires to wait for all stack frames to transform and then be able to migrate. This leads to a high cost of migration in the range of few 100 microseconds [99].

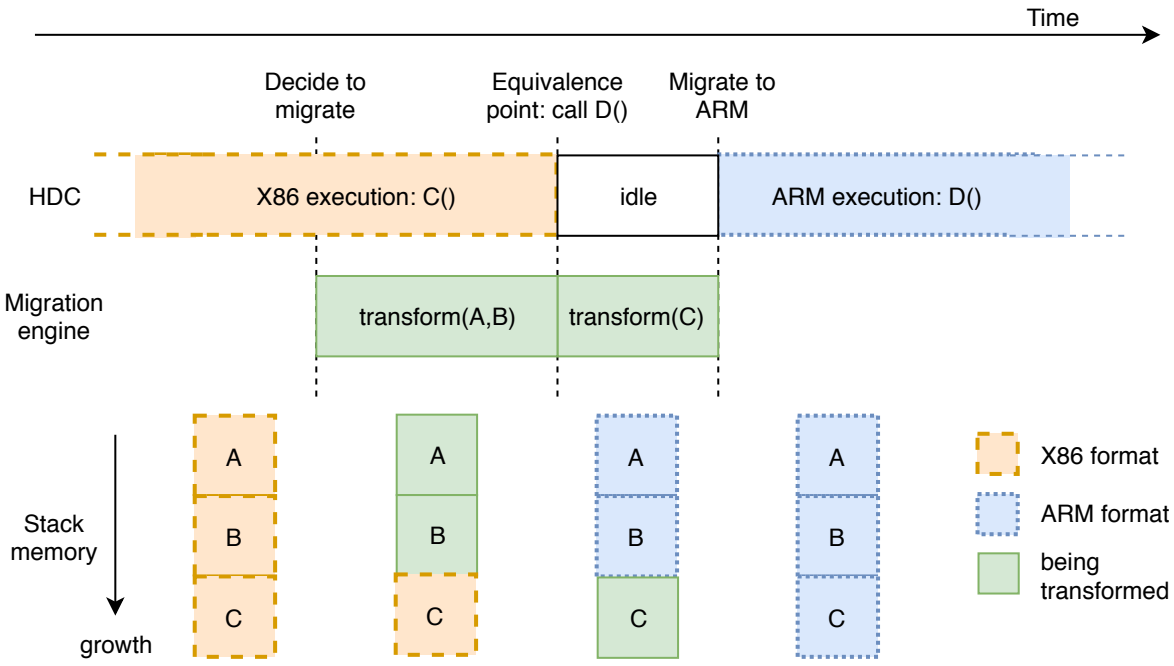


Figure 5.4: Simultaneous Transformation

Simultaneous transformation: In this proposed method, as shown in Fig. 5.4, when executing function `C()`, previous frames in the stack are untouched by the program.

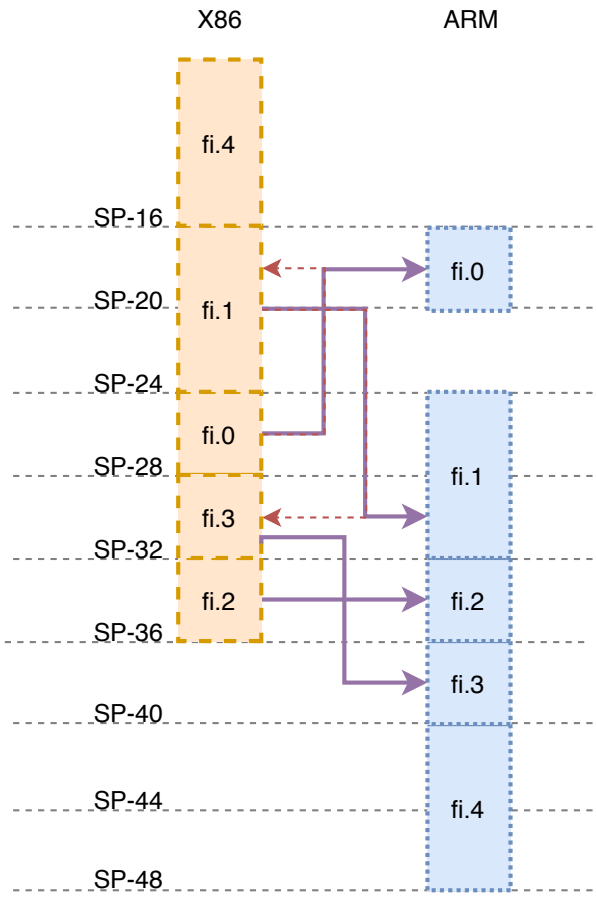


Figure 5.5: Transformation sequence for stack-frame of `BZ2_bzDecompressStream`

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. [28], however majority of that time is masked by execution in parallel to 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 later section and can be appended 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 5.3 shows stack slots for both ISAs for function `BZ2_bzDecompressStream` from *bzip2* benchmark. It shows the locations

Table 5.3: 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]

of different variables of the program in both ISAs. This data is useful in simultaneous in-place migration.

5.4 Hardware implementation

The performance model given in Section 5.2.2 requires an adder, a multiplier and an FSM. Listing 5.1 can be implemented using an FSM and two temporary registers which reads transformation sequence and modify the stack using load/stores. During simultaneous transformation, it is ensured that the data being modified is not in use by the executing 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 data-cache may be inconsistent with the main memory if L1 is non-inclusive or mostly inclusive. Such case will require flushing of entire L1 data-cache, however the performance penalty is not very high because this is done every 10 million instructions.

The classification scheduler is like a fully-connected layer which takes 13 parameters as input and produces output. Thus in total required 8-bit registers are 13 for storing weights and required 32-bit registers are three for bias term per classification scheduler.

Compute scheduler requires twenty one 8-bit multiplier for applying weights to the input parameters along with one 32-bit adder.

5.5 Results and analysis

Experiments are performed using Gem5 [9] simulator with SPEC CPU2006 [39] benchmarks. These benchmarks are compiled using 'O2' optimization in gcc for x86-64 and cross compiler built for ARMv7. Table 5.4 shows the detailed configuration differences for both cores. HIDC is compared with a dual core dual Heterogeneous-ISA architecture with one x86 and one ARM core called 'HetereogeneousISA CMP' [99] for being closest to HIDC. The power and area analysis was done using McPAT [56] tool. The migration code is called at the end of all functions which needs to be migrated. The execution time is calculated for all the memory accesses using the Gem5 simulator, which is similar to the migration cost. As our simulation methodology involves cross-compilation of benchmarks for ARMv7, benchmarks namely *gcc, tonto, sphinx3*, could not be cross-compiled successfully. Few other benchmarks namely *perlbench, gamess, cactusADM* could not run to completion after being cross-compiled. Despite these issues, we are able to study most of the benchmarks reported in [99]. We could study twelve SPEC CPU2006 benchmarks and have reported their results.

5.5.1 Program's affinity towards different ISAs

We have analyzed the affinity of the benchmarks to different ISAs. We simulated SPEC CPU2006 benchmarks on both ISAs. Fig. 5.6 describes the percentage affinity shown towards these two ISAs. Few of the benchmarks used for simulation such as *gobmk, h264ref, sjeng* are more biased towards ARM ISA. Still, some of the benchmarks such as *libquantum, specrand_int and specrand_float* show mixed behavior, while *soplex* is biased towards x86. This affinity completely depends on the program behavior i.e. types of functional units required, memory operations, number of instructions, etc. Floating point benchmarks like *milc, soplex, specrand_float* prefer to run phases with floating

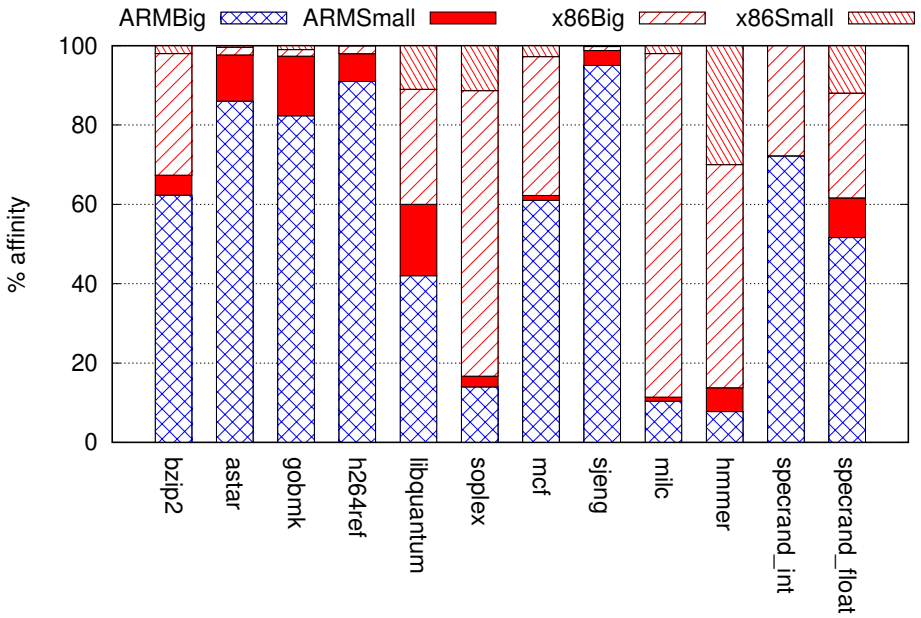


Figure 5.6: Percentage ISA affinity for SPEC2006 benchmarks

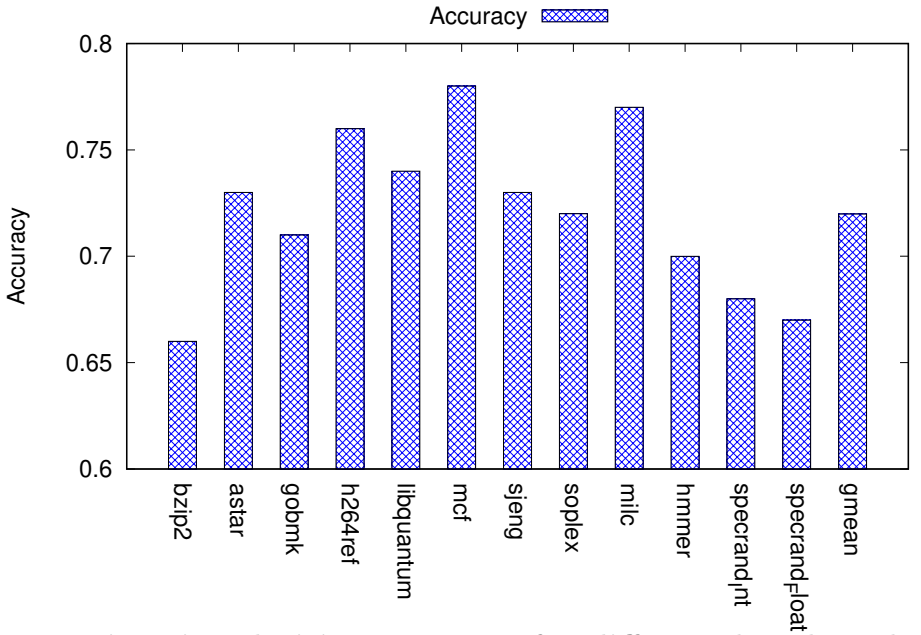


Figure 5.7: Migration decision accuracy for different benchmarks by linear regression scheduler

Table 5.4: Core configurations for HIDC

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

point operation on x86 due to its floating point operation support. *libquantum* utilizes SIMD support of x86 and runs those phases with SIMD operations on x86. High ILP phases of *hmmer, bzip2* are run on ARM due to less register pressure in ARM.

5.5.2 Dynamic scheduling

To evaluate the migration decision efficiency taken by the scheduler, the accuracy of the decision is plotted in Fig. 5.7. The model is trained using only a subset of the SPEC CPU2006 benchmark suite and tested it for the accuracy on others. Model is tested on the benchmarks on which it has not been trained to test the resilience and generality of our migration framework. The scheduler gives affinity prediction accuracy of 71.4%.

5.5.3 Migration overhead

Previously implemented Hetereogeneous-ISA CMP architecture by Venkat et al.[99] shows that migration overhead ranges between 5 microseconds to 150 microseconds. The migration overhead reduces significantly in our proposed HIDC architecture as shown in Fig.

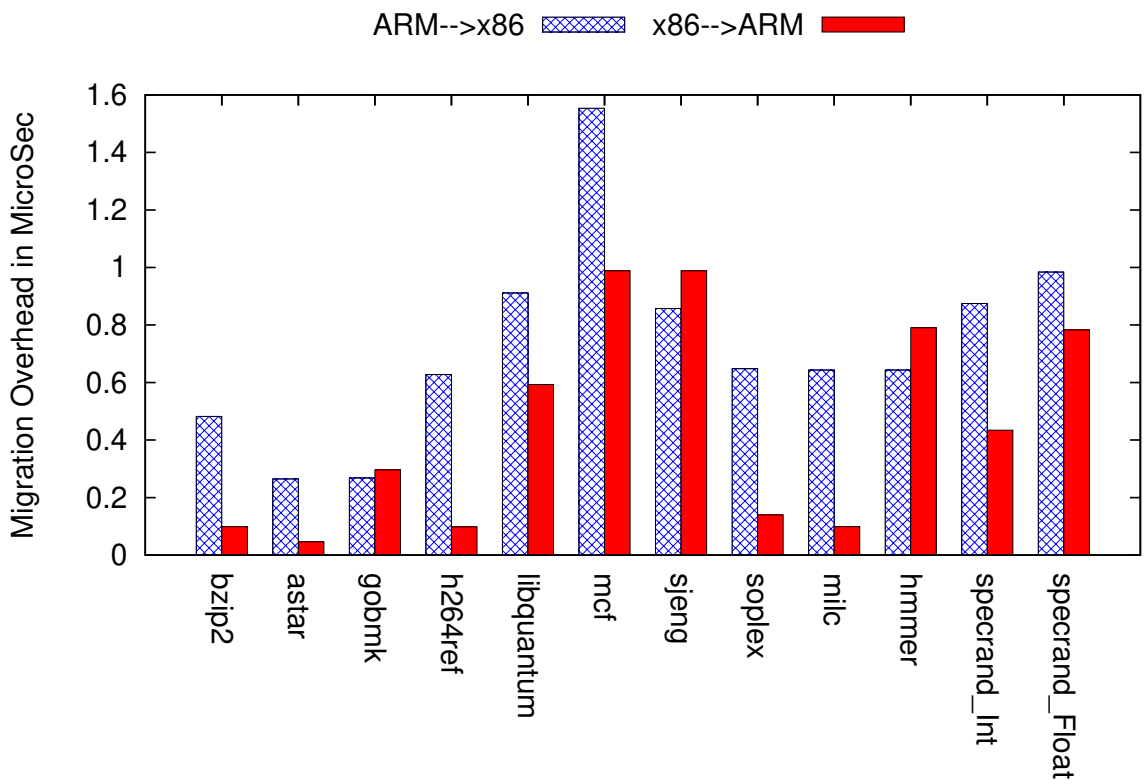


Figure 5.8: Migration overhead for simultaneous transformation

5.8 due to the Simultaneous Stack Transformations. It ranges between 0.1 microseconds and 1.5 microseconds. We are able to transform most of the stack frames in parallel. Only last function is migrated in serial. The migration cost depends on the number of registers in the last function.

5.5.4 Performance and energy results for HIDC

Fig. 5.9 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 in Fig. 5.6, so these benchmarks do not give much performance gain(w.r.t. x86). However, *astar, gobmk, h264ref, sjeng, mcf* have more affinity towards ARM

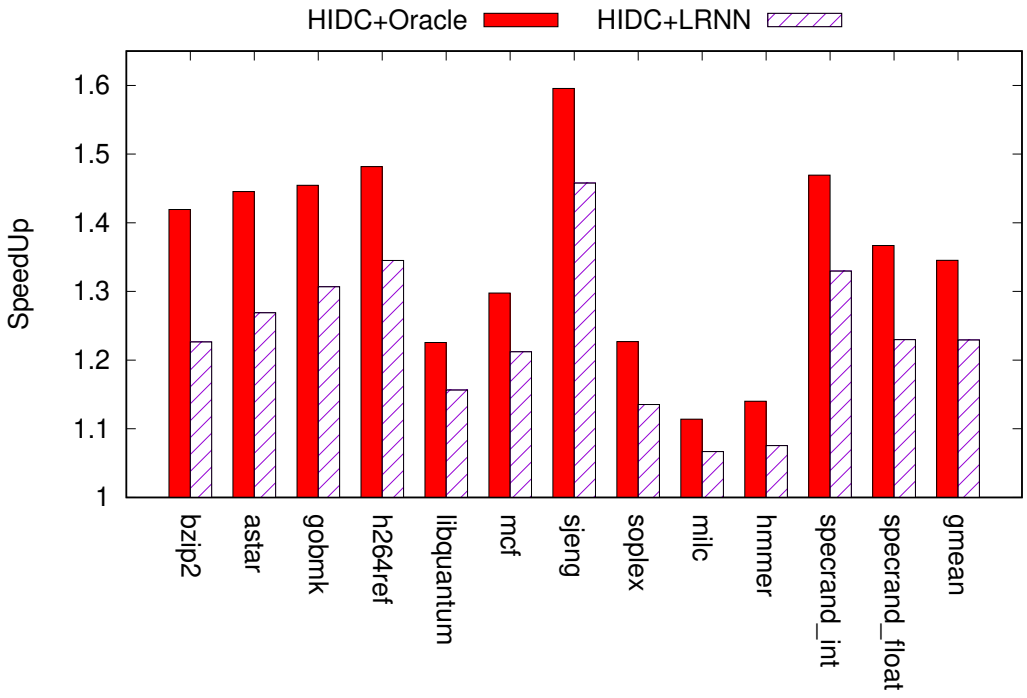


Figure 5.9: Performance of HIDC architecture for benchmarks w.r.t. HISACMP

as shown in Fig. 5.6 and is improved. *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 linear regression scheduling algorithm is applied. In case of oracle, HIDC gains speedup of 34.5% relative to HISACMP and 30.2% relative to x86 core.

HIDC 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

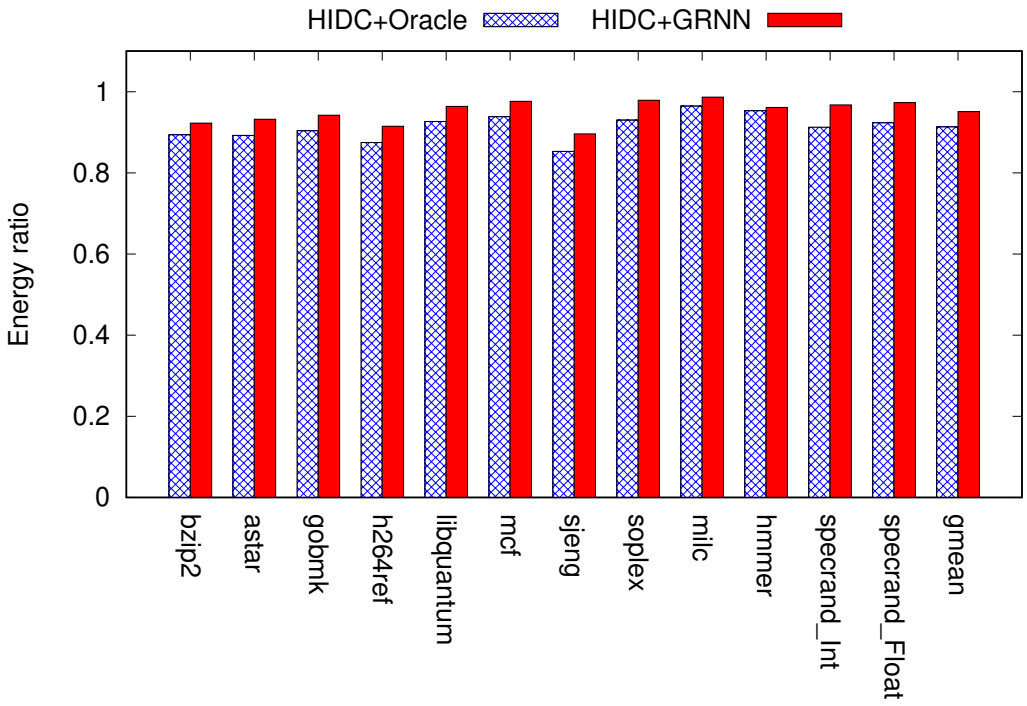


Figure 5.10: Energy consumption ratios of HIDC architecture for benchmarks relative to HISACMP

from the results that programs which are affined towards ARM have shown higher energy efficiency. As shown in Fig. 5.10 about 5.1% reduction is observed in energy consumption by HIDC over HISACMP. In case of oracle 8.9% energy is saved compared HISACMPP.

To see the actual gain from a micro-architectural changes, performance per Joule or performance energy ratio (PER) is plotted in Fig. 5.11. The normalized PER for HIDC is 1.54, which implies that HIDC will give more than 1.5 times performance for every Joule that is consumed by the processor compared to HISACMP.

5.5.5 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

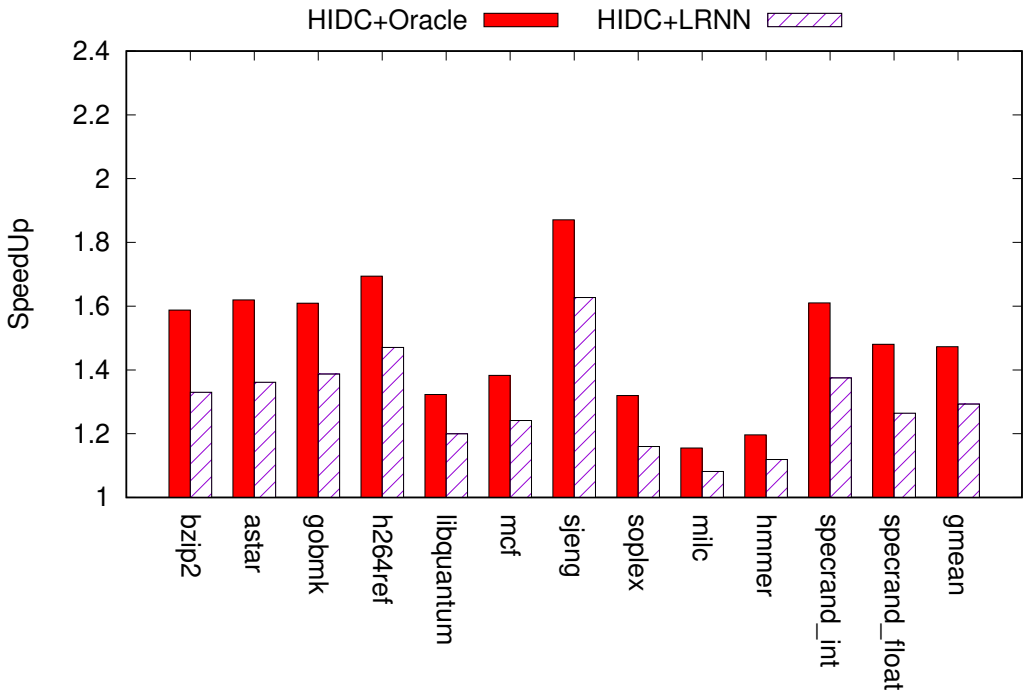


Figure 5.11: Performance per Joule of HIDC for SPEC2006 benchmarks

[14]. 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 [14], 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. 5.12.

5.5.6 Performance results for multi-workload

HIDC is proposed mainly to enhance the performance of single-threaded performance. However, to see the multi-threaded behaviour on HIDC architecture, we have executed multi-workload benchmarks as well. Results are shown in Fig. 5.13. 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.

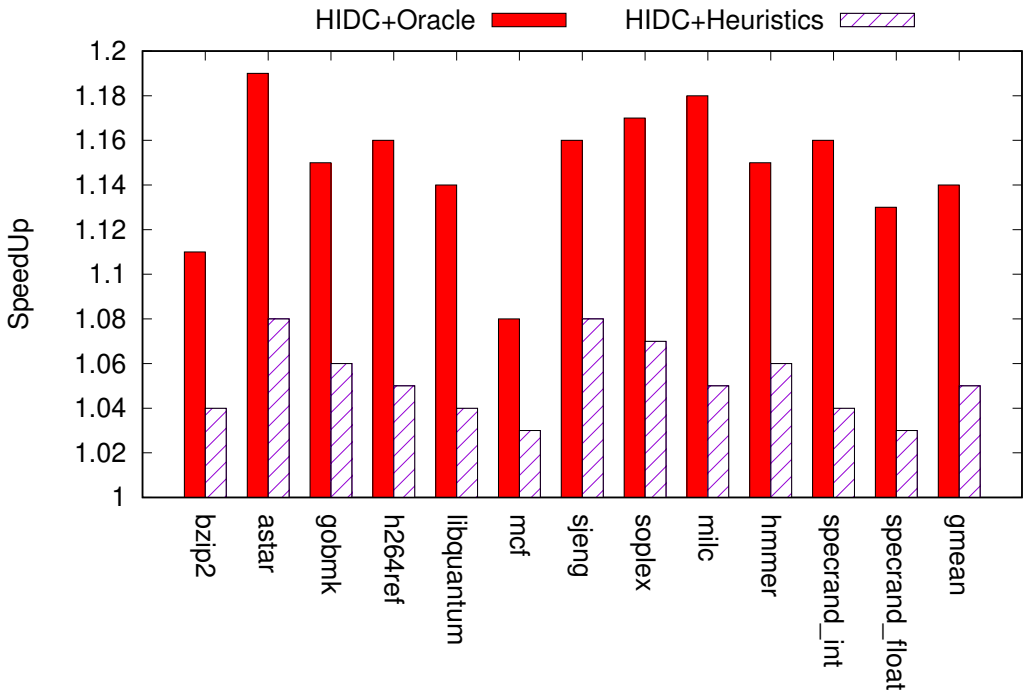


Figure 5.12: Performance of HIDC with fine-grained scheduling relative to coarse-grained scheduling

5.5.7 Area overhead

The area calculation is done using McPAT [56], and the result shows that the area is reduced by 20% in HIDC architecture compared to Heterogeneous-ISA CMP. The area is reduced 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. Clearly from the results, the HIDC comes out to be a better option to improve single thread performance without much change in the architecture.

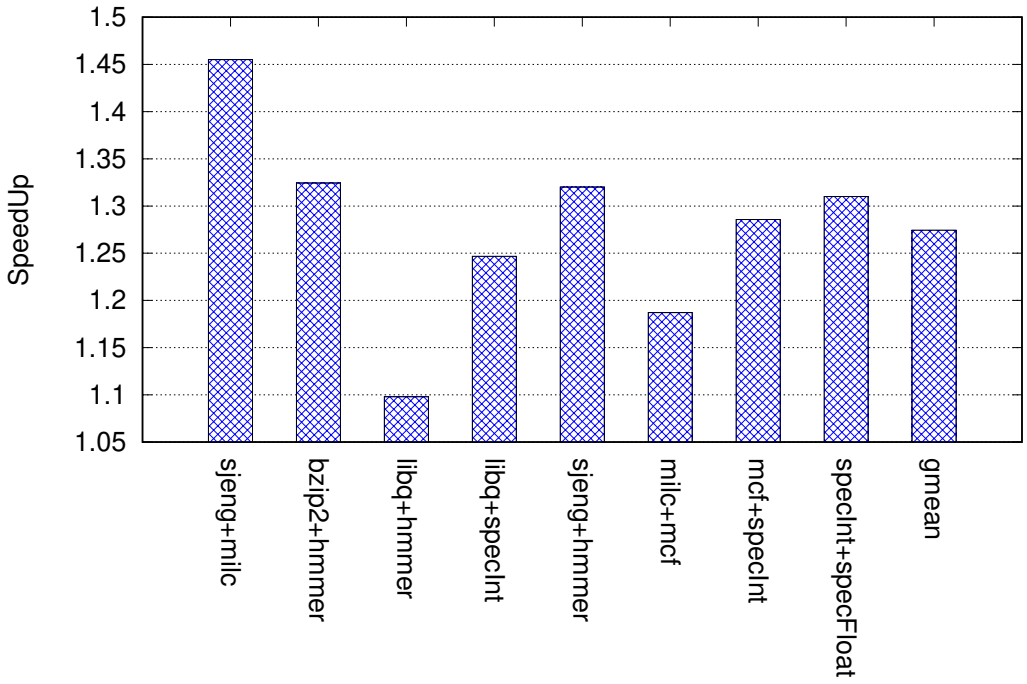


Figure 5.13: Performance of multi workload benchmarks

5.6 Conclusion

The chapter has proposed a novel architecture which supports multiple ISAs in a single dynamic core allowing us to improve single-threaded performance by exploiting the heterogeneity of executing program on different ISAs. 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 shows an increase in single-threaded performance up to 34% along with about 9% energy savings over Heterogeneous-ISA chip multiprocessor.

– * – * –

Chapter 6

Secure Multi-Core Architecture

Almost all modern CPU's implement hierarchical memory architecture that consists of a large (but slow) main memory and multiple levels of smaller (but fast) caches. In all the multi-core architectures, the higher-level caches are private, and the last level cache(LLC) is shared between the cores. This causes a vulnerability in the form of side-channel attacks. The attacker can infer the victim's memory access patterns by monitoring the cache lines in the shared cache. This is done by forcing collisions with the victim and making accesses to that alias with the same lines. This is possible as all the cores can access shared libraries. Since the memory access patterns in most security protocols are dependent on the private key, leaking information about these patterns may compromise the key. Therefore, in any multi-core system, scopes of side-channel attacks always exits when multiple processes are running on different cores.

Researchers have looked into security vulnerabilities in the form of side channel attacks. The most prominent among these are cache-based attacks such as the Prime+Probe attack [58], and the Flush+Reload attack [107]. Researchers [58], [82], [45] have shown that these attacks can decipher the private keys used in security protocols like RSA and AES. However, these attacks are susceptible to noise. As hardware structures are not directly accessible, the attacker relies on timing measurements to make indirect inferences about the victim's memory footprint, which are not always reliable. Data fetched into the cache but not used by the victim will be observed as noise. One of the primary

sources of noise are processes running in the background. They contribute their own cache footprint, distorting the data observed by the attacker. The attacker cannot infer which access corresponds to the relevant process and reaches false conclusions. Another source of noise for an attacker is hardware prefetchers. The memory accesses in most programs follow a pattern. Hardware prefetchers [46], [68], [36] are used to exploit this feature and boost performance. They predict the next access location by taking into consideration the pattern followed by the preceding accesses. Data is fetched from the predicted memory address into the cache before the instruction is encountered, hiding the memory latency needed to access the main memory. One of the most commonly used prefetchers is the stride prefetcher [46]. The stride prefetcher and the attack vectors are discussed in detail in Section 6.1.1. Since the prefetcher brings data into the cache, it interferes adversely with the side-channel. Only the cache accesses corresponding to the victim are of interest to the attacker. However, it observes data fudged by the prefetcher and cannot distinguish the true source of the access between victim memory accesses and prefetched data. Thus, in addition to boosting performance, prefetchers act as good mitigators of cache side channel attacks.

6.1 Prefetchers are also vulnerable

Prior research by [94] [101] [32] on prefetcher noise have looked exclusively on noise contributed by the attacker. In this section, we present an attack that mitigates victim prefetching noise as well.

Our main contributions in this section is :

1. We present a denial of service attack on the prefetcher that prevents the victim from generating prefetches. We exploit the behavior of shared caches and shared prefetchers in designing this attack.

2. We extract the private key in the presence of prefetchers by running the proposed attack on top of Flush + Reload.

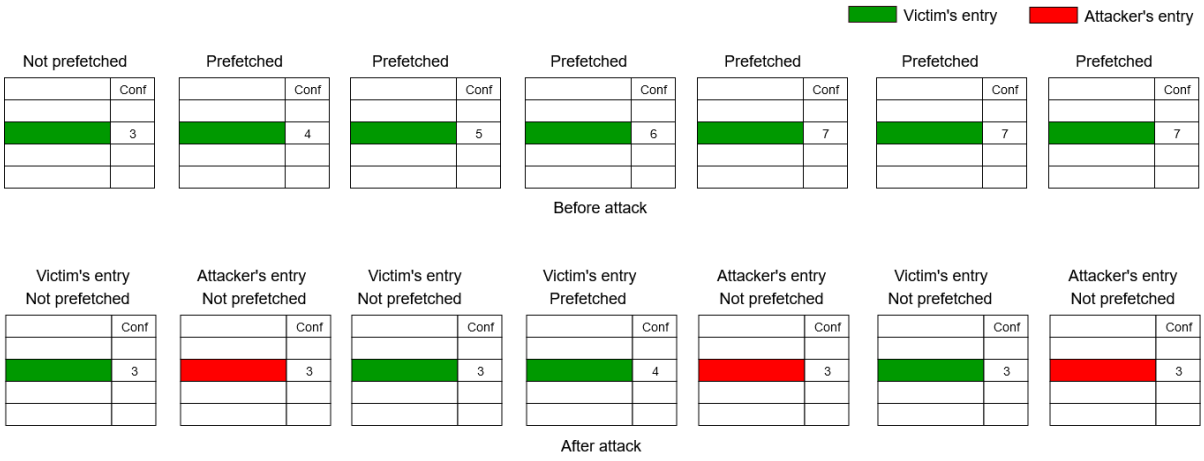


Figure 6.1: Attack methodology

6.1.1 Attack vectors

Stride prefetcher [46] identifies patterns in load instructions having a fixed difference (stride) between the addresses. It consists of a table (prefetch table) indexed by hashing the program counter (PC). Each entry in the table stores the stride value, confidence value and the location of the previous memory address that was accessed. The mode of operation is explained below.

New entries: When a cache miss is encountered, a new entry is created in the prefetch table (if it does not already exist). If the slot is occupied, the old entry is replaced as dictated by the prefetch replacement policy. Entries are initialized with random stride and default confidence value (less than the threshold value for prefetcher).

Prefetch: Prefetching decisions are taken based on the confidence counter. If the confidence of the entry (corresponding to that PC address) is greater than the threshold, a prefetch request is sent for the memory address equal to 'the previous memory address+stride.' The previous address is updated to store the value of the current memory address. This ensures that the required data is present in the cache when a future load request is encountered (provided they follow the same pattern).

Confidence updation: The confidence of an entry is updated whenever the corresponding load instruction is encountered. On a cache hit, the confidence value is

incremented if the new stride (calculated as the difference between the current memory address and the previous address) matches the current stride value. Otherwise, the confidence value is decremented, and the stride is updated to the new value.

The attacker can prevent a particular entry from being prefetched by reducing it's confidence in the prefetch table. However, the confidence increment/decrement depends on the victim program, and the attacker cannot interfere with it directly. Since the prefetcher is shared between the two cores, our aim is to influence the behaviour of victim's entries through the other core. We observe that a new entry is initialized with confidence less than the threshold. This implies that it will take some time to build confidence before successfully contributing to prefetches. We exploit this property to design an attack. The proposed attack program will *evict the victim's entries before they attain the threshold.* This is done by filling up the table with the attacker's entries. Since the prefetcher is shared, the victim and the attacker contend for the same space leading to the eviction of the victim's entries. Also, once an entry is evicted, the prefetcher retains no memory of it and has to rebuild confidence from scratch. By evicting entries before they are prefetched, the prefetching noise for an attacker is eliminated .

6.1.2 Attack methodology

Figure 6.1 explains the mechanism of the proposed attack. It shows the status of the prefetcher before and after attack, for the same sequence of memory access in the victim program. The entries in green correspond to the victim core and the entries in red belong to the attacker core. A new entry is placed in the table with default confidence value (taken as 3 in the example). When the prefetcher is not attacked (upper half of the figure), the victim's entries build confidence for each successful prediction. They reach high confidence values (6-7) that are greater than the threshold confidence (taken as 4 in the example). This will result in a lot of prefetched blocks in the cache. The blocks unused by the victim will be noise to the attacker. However, when the prefetcher is attacked (lower half of the figure), the victim and the attacker fight for the same space in the prefetch table. The victim's entries are evicted continuously and replaced by the

attacker's entries. They do not get enough time to train before they are evicted. When the evicted entries are brought into the table in the future, they have to start from the default confidence all over again. These entries are stuck at low confidence level(s) (3 in our example), and rarely achieve high confidence that qualify them for a prefetch request. Constant eviction results in a net reduction in prefetch count, in turn, leading to low prefetching noise. The attack program is shown in Listing 6.1.

Listing 6.1: Attacker disassembly: loads at aliased PCs

```
00000000000006ca <attack>:
    ...
1   6dc: mov    0x1a020(%rax),%ebx
                <nop slide>
2   6e8: mov    0x1e020(%rax),%ebx
                <nop slide>
3   6fe: mov    0x2e003(%rax),%ebx
                <nop slide>
4   716: mov    0x2300e(%rax),%ebx
                <nop slide>
5   722: mov    0x1f033(%rax),%ebx
                <nop slide>
6   734: mov    0x21005(%rax),%ebx
                <nop slide>
```

The attack program is designed to evict the victim's entries from the prefetch table repeatedly. To do this, the attacker fills up the table with it's own entries. Since the prefetch table is indexed by hashing the PC address, a single load instruction in the attack program will map to only one entry. To evict all entries of the victim, we need to have multiple load instructions at many distinct PC addresses. Further, their load addresses are chosen such that they are mapped to the same cache line (one that the attacker does not monitor). This ensures that the attacker does not contribute to the noise.

Core Type	O3 CPUs 8-wide fetch
Number of Cores	2
L1 Icache	32K 8-way
L1 Dcache	32K 8-way
L2 cache	256K 16-way shared between cores
L2 prefetcher	Stride 64-entry 4-way
Range of confidence values	0-7
Threshold value	4
Default Confidence	3

Table 6.1: Simulation setup for prefetchers

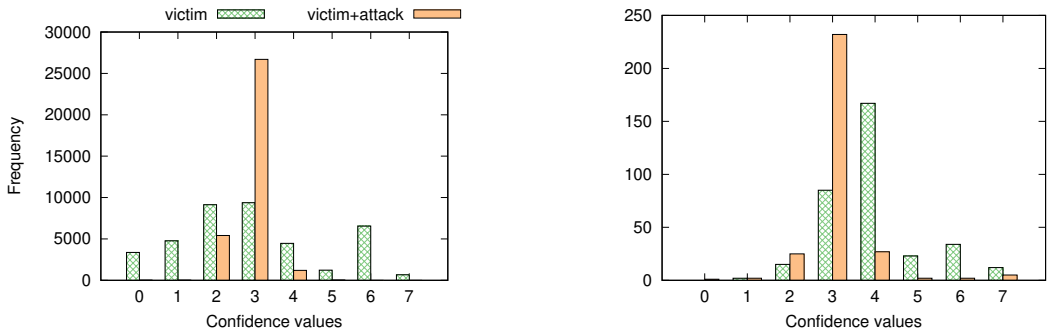
When we execute the attack program in a loop, a sequence of cache misses is encountered (as they map to the same cache line). They evict each other's data in the cache. A cache miss triggers the prefetcher to look for the corresponding entry in the prefetch table. If the entry is not found, the victim's entry is evicted and replaced by the attacker's entry. Thus we use a repeated sequence of cache misses to trigger the prefetcher into kicking out the victim's entry. A string of nop instructions fill the space between the relevant PC addresses and are represented as <nop slide> in Listing 6.1.

6.1.3 Results and analysis

We have simulated a system with two cores, each having a private L1 data and instruction cache. The L2 cache and stride prefetcher are shared between the cores. We have used the Gem5 simulator [9] for our simulations and the configuration is listed in Table 6.1. Simulations have been performed with two different setups:

1. Victim only - A single core system that runs the victim program

2. Victim + Attacker - A system with two cores, where core 1 runs the victim program and core 2 runs the attack program



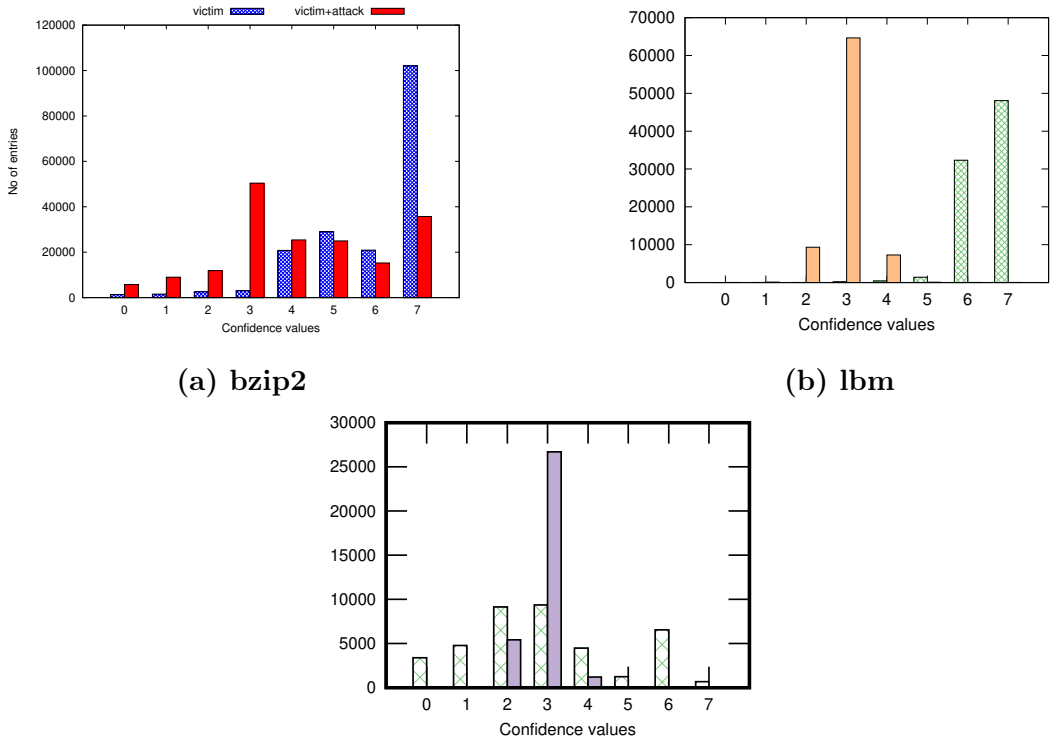
(a) RSA **(b) AES**

Figure 6.2: Confidence distribution in different cryptographic benchmarks

To show the generality of proposed approach, we performed experiments on various SPEC 2006 benchmarks [39] - on applications based on cryptography algorithms shown in Figure 6.2 and non-cryptography algorithms shown in Figure 6.3. On non-cryptography algorithms the experiments were further performed on memory intensive benchmarks (*bzip2* and *lbm*) and compute intensive benchmarks (*sjeng*). The victim executes these programs while the attacker runs the attack based on the pseudo-code given in Listing 6.1.

Confidence reduction

To observe the effectiveness of the proposed attack, we plot the confidence distribution for both setups (Victim only and Victim+Attacker). The effect of proposed attack on RSA can be clearly seen in Figure 6.2a. Without attack (green, hatched line), the entries have very high confidence values (mostly 6 and 7). However, after the attack (orange, filled line), the confidence values are predominantly reduced to 3 and 4. This is because a new entry is initialized with a confidence of 3. Most of the entries are evicted before they can encounter a confidence increment. They do not reach high confidence values that qualify them for prefetch, confirming that the proposed attack works as intended. The same behaviour can be seen in AES 6.2b. Figures 6.3a and 6.3b, and 6.3c show the same nature for the SPEC2006 benchmarks based on non-crptographic algorithms. Hence, the confidence reduction is not limited to cryptographic algorithms, however it



(a) bzip2

(b) lbm

(c) sjeng

Figure 6.3: Confidence distribution in non-cyrptographic benchmarks

is reduced for all type of programs.

Prefetch count reduction

We compare the total number of prefetches for both set-ups. The results for various phases of RSA are shown in Figure 6.4. The blue hatched bar indicates the total number of prefetches before the attack, and the red filled bar indicates the same after the attack. In Figure 6.5, we plot the ratio of prefetches after attack compared to before attack for all the benchmarks mentioned above. We observe a sharp reduction in the number of prefetches ranging from 46% for high prefetch benchmarks like *bzip2* to 78% for benchmarks like RSA. We also observed the execution times for these two setups. The execution time increased by 6.8 % (for Victim+Attacker compared to Victim only). Since this increase is not significant, the proposed attack is difficult to detect.

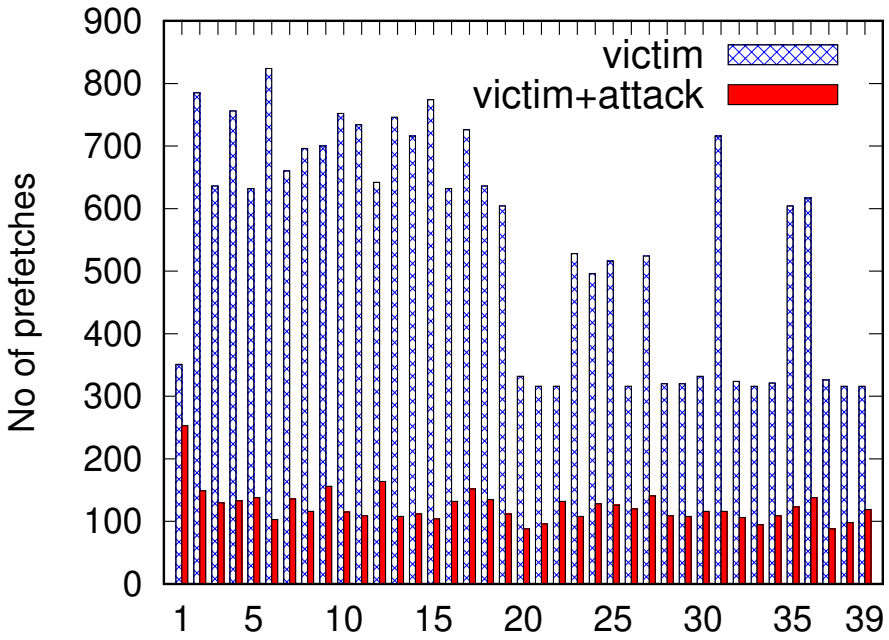


Figure 6.4: Different phases of RSA

Key extraction

We performed an experiment to evaluate the effect of our attack program on key extraction. The proposed attack was mounted on top of the Flush+ Reload attack on RSA [82]. We calibrated the difference in time required to fetch a block from the cache (cache hit) compared to fetching it from the main memory (cache miss) similar to the approach followed in [107]. On the set-up mentioned in Section 6.1.3, we observed the cache hit latency to be approximately 70 cycles, and a cache miss latency of around 170 cycles. We set the threshold as the average of cache hit and cache miss latency, i.e., 120 cycles. If a memory access takes less time than the threshold, it is inferred as a cache hit. Otherwise, it is inferred as a cache miss. Figures 6.6 and 6.7 show the Flush + Reload probe time measurements before and after the attack respectively. The blue dots are cache misses and green dots are cache hits. The red dots correspond to the prefetching noise. Some of these are cache hits (true positives) and some are cache misses (false positives). However the attacker cannot distinguish between the two. When we compare figures 6.6 and 6.7, it is evident that the prefetching noise is lower when the proposed attack program runs.

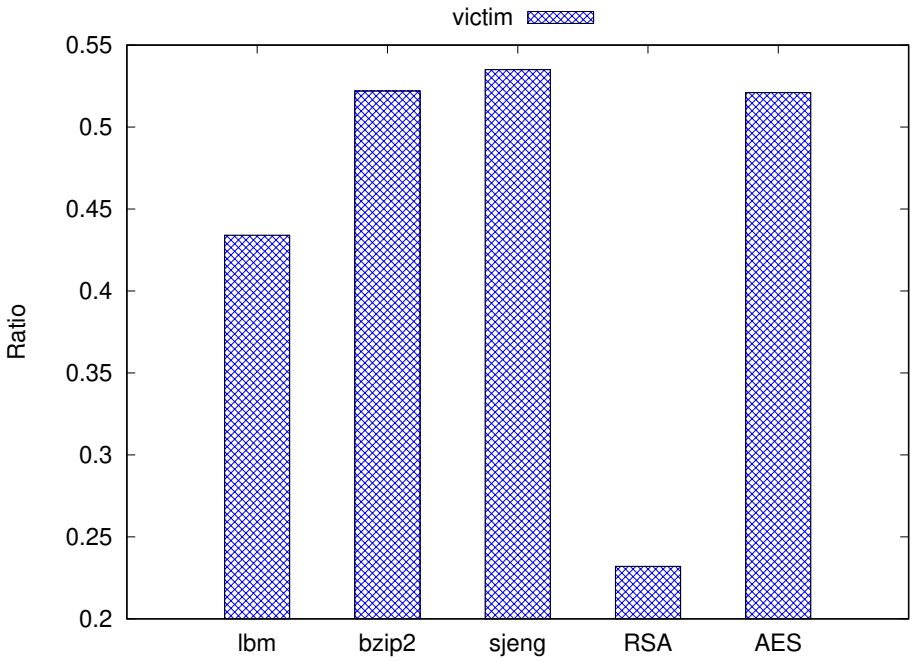


Figure 6.5: Ratio of total prefetches

The noise is significantly reduced by mitigating the victim prefetching noise.

The time required to extract the key is measured for three different cases:

1. Without prefetcher in the system: Completely noiseless environment where one core runs RSA encryption while the other runs Flush+Reload.

2. With prefetcher in the system: Where one core runs RSA encryption while the other runs Flush+Reload.

3. With prefetcher and attacker in the system: Where one core runs RSA encryption while the other runs the proposed attack on top of Flush+Reload.

We have performed key decryption for 2048 distinct keys. In Case 2 when prefetcher was enabled, only 9.8% of the keys could be extracted correctly. Due to high victim prefetching noise, the others keys are inferred incorrectly. However, when our attack was mounted (Case 3) for the same system, the prefetching noise was drastically reduced and all the keys were extracted correctly. Figure 6.9 shows the increase in key extraction time for Case 3 compared to Case 1. We observe an average increase of 21%, whereas the

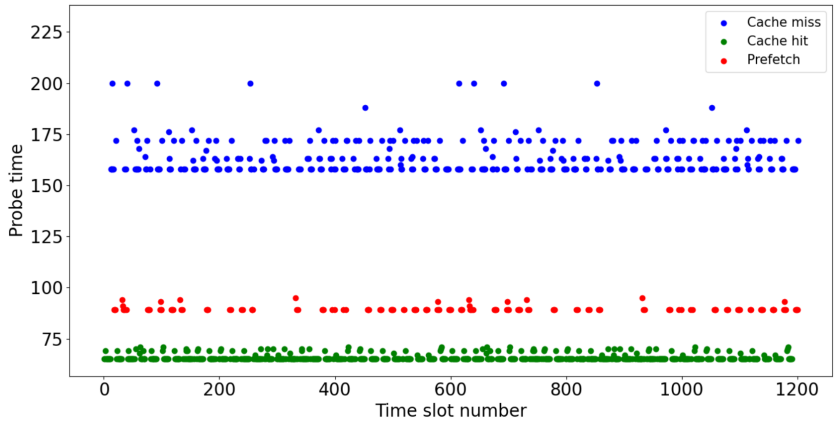


Figure 6.6: Probe time measurement before attack

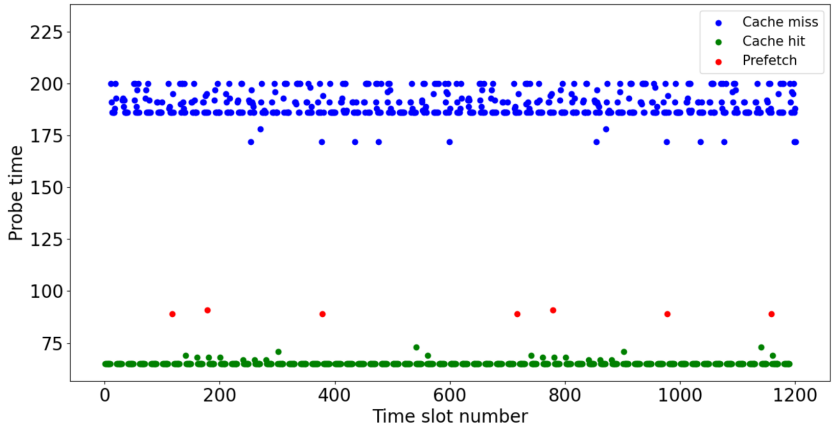


Figure 6.7: Probe time measurement after attack

average increase for Case 2 (when compared to Case 1) was 48%. Figure 6.8 shows the same metric as a comparison between Case 2 (blue, hatched line) and Case 3 (red, filled line). For Case 2, we have plotted only those keys that could be correctly extracted. Even for these keys, we can see that the time taken in Case 2 is significantly higher than Case 3, implying that our attack works better than prefetching noise agnostic Flush + Reload.

The feasibility of attack is there in multi-core architectures because the last level cache is shared. One of mitigation methodology can be "partitioning of last level cache". Various partitioning methods have been proposed which improves the security. There

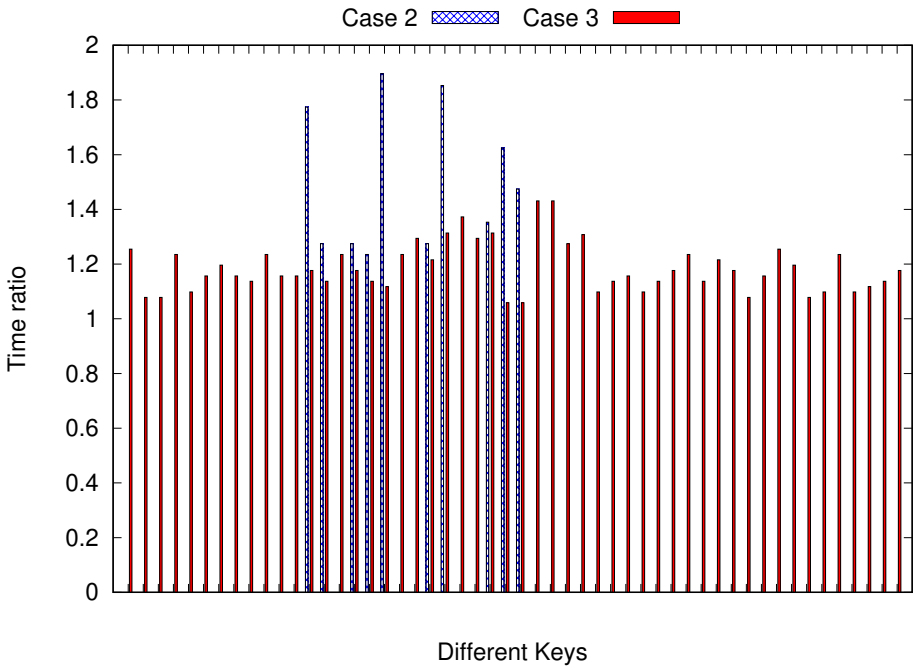


Figure 6.8: Ratio of key extraction times for different conditions

have been few other partitioning methods which helps in improving the performance. The next section details about the cache partitioning methods. We will also show in the next section that partitioned protocol also cause side channel attacks, hence these methods are not safe.

6.2 Cache partitioning

Researchers have proposed various mitigation methods for side-channel attacks. One of the naïve partitioning techniques proposed to do this is static partitioning [74]. Within each cache set, it enforces a fixed partitioning of the lines amongst the simultaneously running processes. Since no cache resources are shared by processes in this technique, it guarantees security against cache-based side-channel attacks. However, static partitioning comes with a heavy performance penalty because many lines in the cache set remain under-utilized [102]. Cache access behavior of a program can change during runtime,

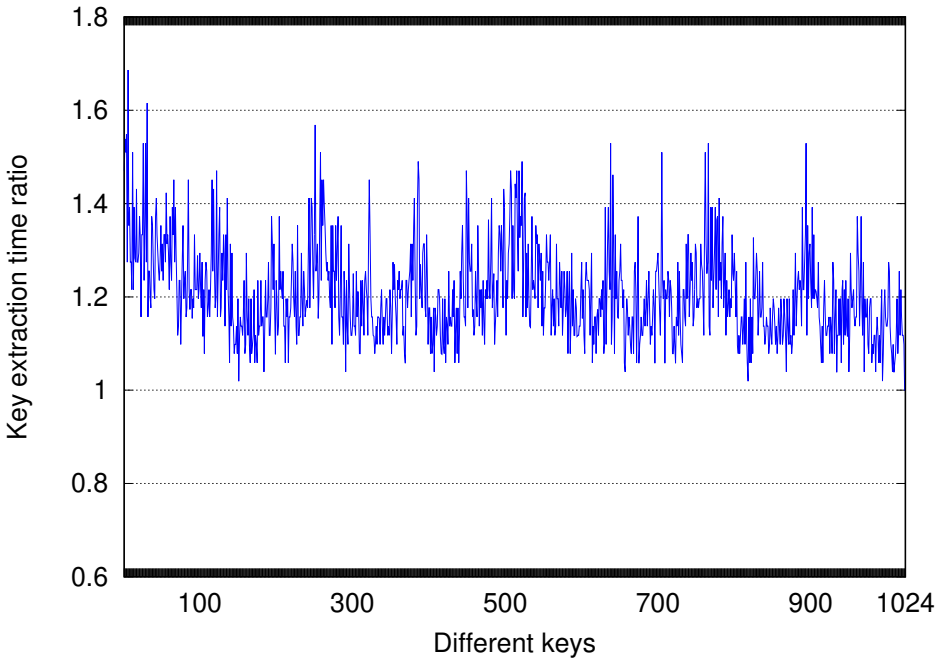


Figure 6.9: Ratio of key extraction times for a system with prefetcher+proposed attack compared to without prefetcher

and static partitioning fails to adapt to this change. To improve performance in a multi-process system, several dynamic cache partitioning (DCP) methods [29, 81, 85, 102, 104] have been proposed. Utility-based Cache Partitioning (UCP) [81] dynamically partitions the LLC in order to maximize the total utility of cache lines for all the running processes. Our evaluation (Figure 6.10) shows that the static partitioning suffers an average performance degradation of 8.3% with respect to UCP for memory intensive benchmark pairs. Figure 6.10 also shows that UCP performs better than static partitioning in all experiments.

While these DCP protocols have significant performance advantages, we will show in Section 6.2.1 that they are susceptible to cache-based side-channel attacks. For mounting any side-channel attack, the following conditions [103] must be satisfied: 1) attacker and victim processes must share a resource; 2) both should be able to change the state of the shared resource; and 3) the attacker should be able to detect the changes made by the victim in the shared resource. The LLC is typically shared amongst all running

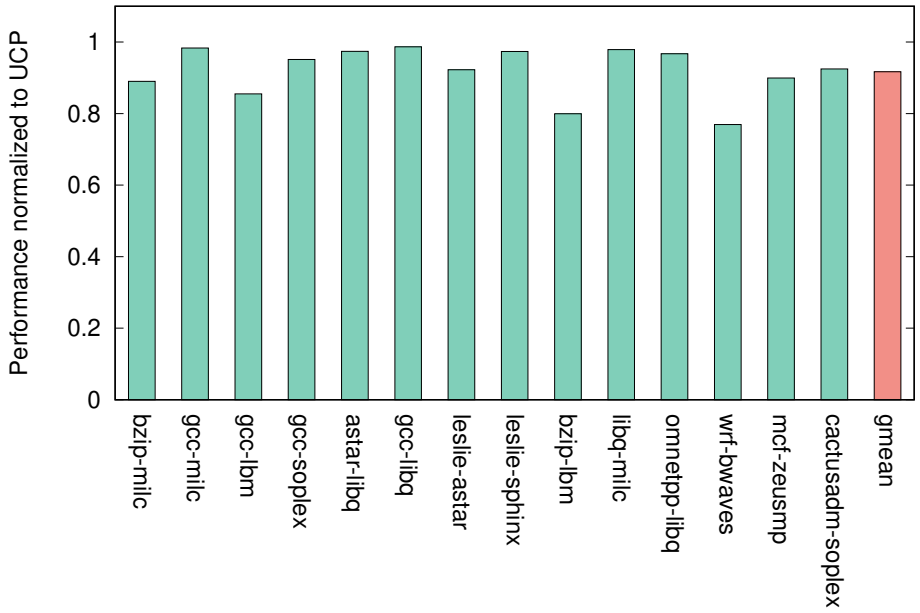


Figure 6.10: Speedup of static partitioning scheme normalized to UCP for memory-intensive benchmark pairs (for Configuration 1 of Table 6.2).

processes. Since LLC lines are reallocated periodically by UCP, all of these conditions are satisfied and an attack can be mounted through the shared LLC. Thus, our goal is to provide security to DCP schemes without incurring a significant performance penalty.

The contributions of this section os chapter are as follows:

- We describe a security vulnerability in DCP protocols and SecDCP [102] via the shared last level cache. The vulnerability allows an attacker to determine memory accesses made by victim process while running simultaneously on the same core or on another core in multi-core system.

- We propose PASS-P, a protocol that mitigates this vulnerability by invalidating cache lines.

- To recover the performance loss due to invalidation, PASS-P uses the novel Modified-LRU reallocation policy. Our detailed evaluation shows that PASS-P regains the performance lost and performs comparably to UCP.

While this work shows how PASS-P makes UCP secure, we believe that our technique works for most DCP schemes.

6.2.1 Vulnerability in dynamic partitioning

Dynamic cache partitioning (DCP) protocols are runtime algorithms that dynamically distribute cache lines amongst running processes. UCP [81], for example, periodically partitions the cache lines in each set in order to maximize the total utility of caches for all running processes. Utility of a cache line for a process is defined as the increase in cache hit-rate if the process was given an additional cache line. At the beginning of each phase (1 million cycles in our study), UCP computes the optimum partitioning based on the utility behaviour of the processes in the previous phase and re-partitions the cache sets. Previous DCP protocols including UCP are designed such that if some lines are reallocated from a process P1 to a process P2, then P1 can *still* access those lines until P2 overwrites them [102]. This was done in order to avoid unnecessary cache misses on reallocated lines in each phase. *The prime reason for a side-channel attack on UCP, or in general on any DCP scheme, is the reallocation of cache lines from one process to another.*

6.2.2 Threat model

Dynamic cache partitioning schemes can be vulnerable to Flush+Reload [107] and Prime+Probe [58] attacks, especially in cases where an attacker application can influence the cache partitioning decisions. In UCP, for example, an attacker program can artificially increase or decrease its utility to cause reallocation of cache lines to and from itself respectively. Moreover, to mount these attacks, the attacker process does not need any elevated privileges. The mechanism of the Flush+Reload attack is described ahead and shown in Figure 6.11.

1. *Flush*: The attacker takes all but one cache lines of every set by increasing its utility and flushes them as shown in step 1 in Figure 6.11.

Figure 6.11: Mechanism for Flush+Reload attack

Figure 6.12: Mechanism for Prime+Probe attack

2. *Execute*: The attacker returns all the flushed lines to the victim by decreasing its utility. It then waits for the victim to execute as shown in steps 2 & 3.

3. *Reload*: Attacker takes all but one lines by increasing its utility again and reloads addresses of interest as shown in step 4. A cache hit or miss on these addresses is indicative of the victim's memory accesses.

Similarly, the following steps show how an attacker could mount the Prime+Probe attack. It is also shown pictorially in Figure 6.12.

1. *Prime*: The attacker takes all but one cache lines of every set by increasing its utility and primes them with its own data as shown in step 1 in Figure 6.12.

2. *Execute*: The attacker returns all the flushed lines to the victim by decreasing its utility. It then waits for the victim to execute as shown in steps 2 & 3.

3. *Probe*: Attacker takes all but one lines by increasing its utility and reloads the addresses that were previously primed as shown in step 4. A cache hit or miss on these addresses is indicative of the victim's memory accesses.

While the attacker and victim must share the code library for Flush+Reload to be mounted, there is no such requirement for Prime+Probe. For Flush+Reload, this ensures that the attacker is able to get a cache hit in the *Reload* step for addresses fetched by the victim in the *Execute* step.

Note that in most dynamic partitioning protocols, no process is permitted to possess all cache lines of a set, in order to prevent starvation of the other processes. Despite this, the attacker can extract critical information from the victim, especially over multiple iterations of the attack. To facilitate a more granular analysis of victim's memory accesses, it is typical for the attacker to use well-known methods to slow down victim's execution considerably For example, as described in [38] an attacker can achieve this by mounting a denial of service (DoS) attack on the completely fair scheduler (CFS) that is used in Linux to divide CPU time amongst running processes.

SecDCP [102], which aims to mitigate such attacks, is vulnerable to the Flush+Reload attack. It only invalidates the lines that are reallocated from a public application to a confidential application, if and only if they were fetched by the public application. The lines which are taken back by the public attacker application in *Reload* step are not invalidated. Therefore, the attacker can infer about the victim's accesses, thus making SecDCP insecure.

6.2.3 Proposed mitigation technique: PASS-P

Performance and security sensitive partitioning (PASS-P), invalidates cache lines to secure the LLC, as described in Section 6.2.4. Section 6.2.5 then describes the Modified-LRU reallocation policy that is adopted by PASS-P for an improvement in performance.

6.2.4 Security through Invalidation

To mitigate the side-channel vulnerability described in Section 6.2.1, the attacker must be prevented from successfully performing differential timing analysis on the reallocated lines. To stop the access of shared resources of other program, PASS-P invalidates all cache lines that are reallocated from one process to another. Because of this preemptive invalidation of lines, no process is able to cause eviction of lines of any other process. Side-channel attacks cannot be mounted in such a system, for the reasons described below.

1. *Flush+Reload*: All lines reallocated to the attacker after the *Execute* step are invalidated and the attacker gets a miss for every targeted address in *Reload* step.

2. *Prime+Probe*: The lines primed by the attacker in the *Prime* step are invalidated when they are reallocated to the victim. Hence, in the *Probe* step the attacker will get a cache miss for all these invalidated lines.

The attacker's differential timing analysis fails because all addresses that the attacker attempts to fetch result in the same cache behavior.

6.2.5 Reallocation policy for PASS-P

The invalidation of the cache lines in PASS-P results in a performance loss. We identify two reasons for this:

1. In UCP, when a process P1 gives up some lines of the shared LLC to another process P2, it can still access the cache lines until P2 overwrites them with its data. However, in PASS-P, due to invalidation of all reallocated lines, P1 will incur additional cache misses. As invalidation is critical for security, the performance drop is inevitable. We propose a modification in the LRU reallocation policy to address the second reason (given below) and regain most of the lost performance.

2. Our experiments show that 32% of all reallocated lines are *dirty* in nature. These must be written to the main memory before their invalidation in the LLC. We

observe that this invalidation can lead to a surge in memory traffic at the start of each UCP phase, as many lines may have to be written back at once. Hence, the running processes face additional delays while handling any new cache misses.

UCP uses the conventional LRU policy to choose the lines belonging to one process that should be reallocated to other processes. PASS-P employs the Modified LRU reallocation policy to adequately address the above causes of poor performance. To ameliorate the effects of the second reason, we can design this policy to prefer reallocation of clean lines over dirty lines, as clean lines do not need an accompanying writeback. However, this may still not give the best performance as it violates the LRU order of line selection.

We seek a balance between these extremes by defining a 'threshold fraction' $f \in [0, 1]$. PASS-P preferentially reallocates *clean* lines over *dirty* lines from a set, only if the *clean* lines are not recently used, so as to still respect the recency order. This reallocation policy reduces the number of writebacks to the main memory. We formally define the policy as: *Reallocate LRU-Clean line from a set if one exists and only if the clean line is in the f fraction of the least recently used lines allocated to the process, else reallocate the (dirty) LRU line.* For instance, consider that a process has to choose a line for reallocation to another process from among its 8 lines in a cache set, given $f = 0.75$. The Modified-LRU policy will inspect the 6 $(= 8 * f)$ least recently used lines and reallocate the least recently used clean line amongst them. If all of these 6 lines are dirty, our policy simply reallocates the least recently used line.

Algorithm 1 describes the selection of lines for reallocation and the entire replacement policy is shown in Figure 6.13. In this figure, 'N' denotes the number of cache lines to be reallocated at the end of the UCP phase, as determined by the partitioning algorithm.

To find the best value of f, we evaluated the performance of a few pairs of memory intensive benchmarks for different values of f. Figure 6.14 compares the geometric mean of speedup obtained with respect to static partitioning for these different values of f. This graph shows that $f = 0.75$ performs the best, giving the highest speedup of 10%. The value of the $f = 0$ corresponds to the standard LRU reallocation policy, resembling the one used by UCP. The graph clearly shows that the LRU policy is not well-suited

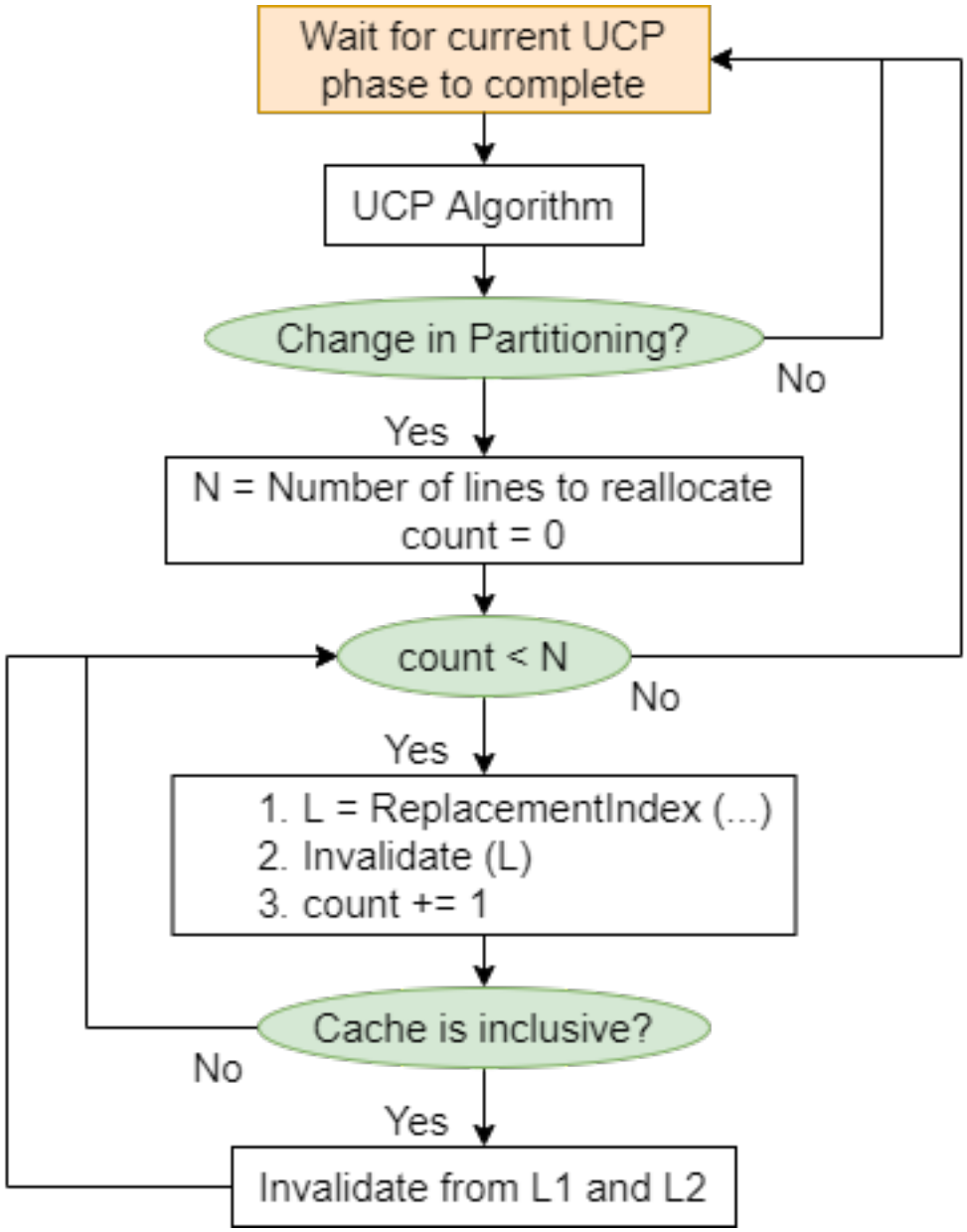


Figure 6.13: Flow Diagram of PASS-P's reallocation Policy

Algorithm 1 Choosing a line for reallocation

Function *ReplacementIndex (**In**: List L < blockIndex, dirtybit >, f, associativity n;*

***Out**: replacementIndex)*

```
l = getIndexLRUCleanLine(L)  if (l != null && l ≤ f * n) then
    return l
else
    return 0; //LRU line
end
```

Note : getIndexLRUCleanLine(L) gives index of LRU clean line

for PASS-P. $f = 1$ indicates a policy that always reallocates clean lines (even if it is at MRU position) whenever such a clean line exists.

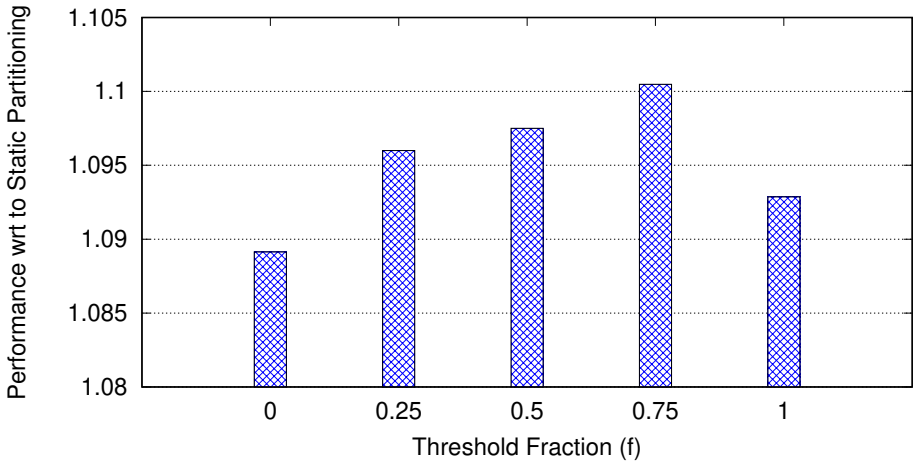


Figure 6.14: Geometric means of speedups of PASS-P for memory-intensive benchmark pairs with respect to static partitioning for different values of f. (For Configuration 1 in Table 1)

6.2.6 Results and analysis

We examine the performance of two benchmarks running simultaneously on two separate cores that share the L3 cache. We evaluate the performance of the two different configurations shown in Table 6.2 on the cycle-accurate Sniper simulator. The first one uses a

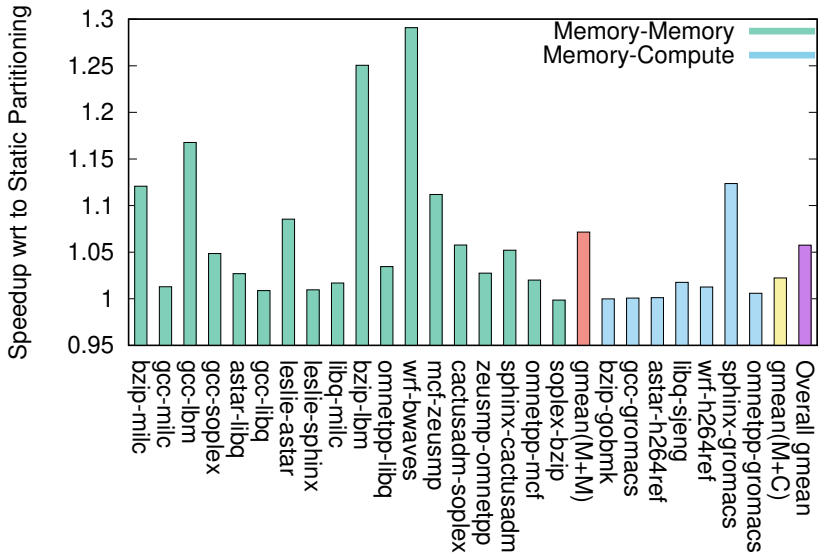


Figure 6.15: Comparison of Speedup of PASS-P (f=0.75) normalized to Speedup of Static partitioning for different benchmark pairs for Configuration 1 of Table 6.2

Table 6.2: Core configurations

	Configuration 1	Configuration 2
LLC	L3	L2
LLC size, associativity	4 MB, 16 way	256 KB, 8 way

16-way associative shared L3 cache of size 4MB. Whereas in the second experiment, the L2 cache is an 8-way associative, 256 KB shared LLC. In each experiment, two benchmarks run simultaneously on two separate cores. The PASS-P and UCP algorithms run with a phase length of 1 million cycles.

We measure performance using 'weighted speedup' metric that is the most appropriate for such multi-core systems [81].

$$WeightedSpeedup = \sum_{i^{th} process} (IPC_i / SingleIPC_i) \tag{6.1}$$

where IPC_i is the IPC of the i^{th} process in the multi-process system, and $SingleIPC_i$ is its IPC when run independently on a single core.

We show the results for twenty-five pairs of benchmarks selected from the diverse set present in the SPEC CPU2006 benchmark suite. In the first eighteen pairs, both benchmarks are memory-intensive [65] ('MM' pairs). In the remaining seven pairs, the first benchmark is memory-intensive, while the second is compute-intensive ('MC' pairs) [65]. MM pairs are of special interest to us, because both benchmarks contend aggressively for cache lines and the optimum partitioning for UCP changes more frequently. Hence, higher number of re-allocations and invalidations take place in the course of their execution, posing a bigger challenge for the performance of PASS-P. Since compute-intensive benchmarks are not sensitive to the cache replacement policies, it is not insightful to study pairs with both compute-intensive benchmarks. In our experiments, we observed that the percentage of reallocated lines which were dirty dropped to around 22% for PASS-P in comparison to UCP's 32%.

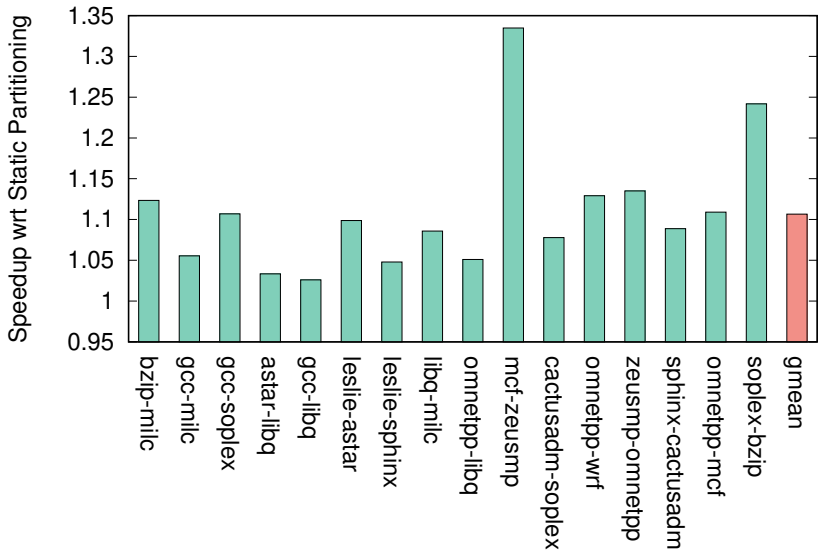


Figure 6.16: Comparison of Speedup of PASS-P (f=0.75) normalized to Speedup of Static partitioning for different benchmark pairs for Configuration 2 of Table 6.2

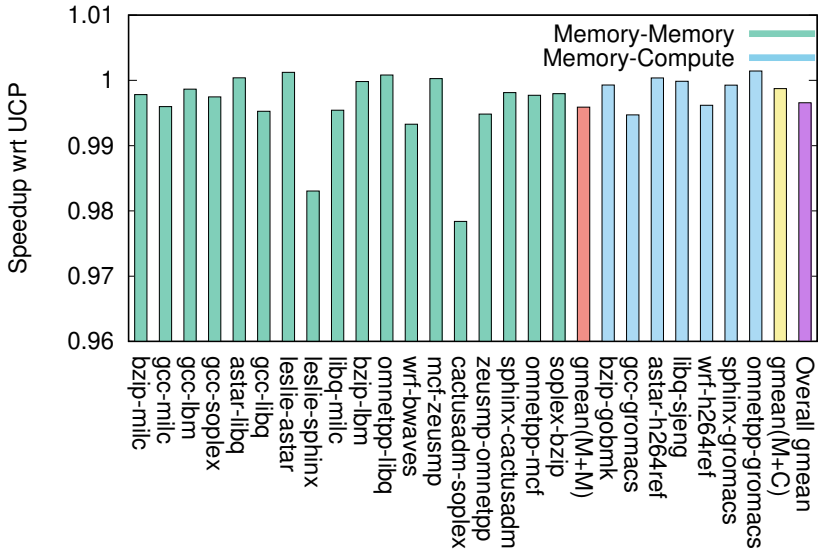


Figure 6.17: Comparison of Speedup of PASS-P ($f = 0.75$) normalized to Speedup of UCP for different benchmark pairs for Configuration 1 of Table 6.2

Figure 6.15 shows the performance gain of PASS-P with respect to static partitioning. Our method with L3 as LLC gives a considerable performance gain of up to 29% and 7.2% on average for MM pairs of benchmarks. The overall average performance gain for all type of combinations is 5.8%. The lower speedup for the combination of MC pairs of benchmarks is expected, because compute-intensive programs do not have a high utility of the cache. The choice of the cache partitioning protocol does not affect compute-intensive programs' performance as much. For 'Configuration 2' mentioned in Table 6.2, the performance gain is up to 33.4% and 10.6% on average as shown in Figure 6.16. The L2 cache, which has lower associativity, benefits more from the proposed policy. The higher gain in this case is because of more efficient utilization of the cache sets.

Figure 6.17 shows the performance of PASS-P (with $f = 0.75$) for all benchmark pairs with respect to UCP. There is a performance drop of 0.50% in case of the eighteen MM pairs, while the value is even lower at 0.12% for the seven MC pairs. Overall geometric mean value for the performance drop is 0.35%. Thus, *PASS-P has only a marginal drop*

in performance with respect to UCP.

6.3 Conclusion

Prefetchers are an important part of modern processors. In addition to improving performance, they help secure the processor from cache based side channel attacks. This paper explored the attack vectors presented by a shared stride prefetcher. The proposed attack was able to significantly reduce the victim prefetching noise and correctly extract the private key with only 21% time overhead when run in conjunction with Flush + Reload. We have also shown that side-channel attacks like Flush+Reload and Prime+Probe can be mounted on dynamic cache partitioning (DCP) protocols. Through cache line invalidation and the Modified-LRU reallocation policy, we are able to overcome the security vulnerability in DCP protocols like UCP, while gaining a speedup of up to 29% and on average 7.12% compared to static partitioning.

– * – * –

Chapter 7

Conclusion and Future Scope

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. The thesis also showed the benefits of having dynamic cores in heterogeneous ISAs. In the security side of thesis, we have shown the possibility of side-channel attacks on dynamic partitioning protocols. We have also shown that prefetchers can not be used as defense mechanism in side channel attacks in multi-core systems. This thesis explored the attack vectors presented by a shared stride prefetcher. We have proposed a new defensive mechanism against side channel attacks which does not compromise with the performance much.

7.1 Summary

In the first proposal of the thesis, we have shown that it is feasible to extract performance benefits from Heterogeneous ISA multi-core architectures by utilizing lightweight and practical techniques for performance modelling and dynamic scheduling. Our first framework combines a regression-based performance model with a greedy scheduling algorithm. Our performance model estimates the performance of a program across ISAs within 6% error-limit and our scheduler migrates the program to the core it is most

suited to 83% of the time for program types it has never seen before. Together, these techniques achieve an average increase of 29.6% in single-threaded performance on the SPEC CPU2006 benchmark suite. Another proposal introduces a novel classification based scheduling mechanism. The scheduler schedules the most affined ISA for each phase with an accuracy of above 93%. We achieved a speedup of 35.7% over x86 ISA.

The second proposal introduced a novel fine grained migration strategy called function-wise scheduling to solve prior challenges of excessive migration overhead and inability to exploit the heterogeneity of a program at finer level. Our results showed that most of the functions have an affinity to some specific ISA. It requires only the transformation of registers during migration, hence reduces the overhead by more than 100x (compared to the state-of-the-art). The proposed fine grained function wise scheduling achieved 22.9% in comparison to perceptron based classifier.

In the third proposal, we have proposed a novel architecture which supports multiple ISAs in a single core allowing us to improve single-threaded performance by exploiting the heterogeneity of executing program on different ISAs. 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 at phase level scheduling. HIDC showed an increase in single threaded performance up to 30.2% along with about 15.2% energy savings over traditional single ISA x86 equivalent core.

Multi-core architectures are vulnerable towards side channel attacks. Static partitioning can mitigate side-channel attacks, however it has a huge performance penalty. In the forth proposal, we propose that through cache line invalidation and the Modified-LRU reallocation policy, we can overcome the security vulnerability, while gaining a speedup of up to 29% and on average 7.12% compared to static partitioning. We also showed that this technique has a marginal performance cost of only 0.35% with respect to UCP on average. Hence, PASS-P can be applied on shared levels of cache for all dynamic partitioning protocols. We have also shown in the forth proposal that prefetchers should not

be trusted upon for side-channel security. Memory accesses generated by the prefetcher pollute the data observed by an attacker. We demonstrated how the contention for space in the prefetch table could be exploited to design an attack. The proposed attack program evicts the victim's entries from the prefetch table by inserting its own entries. It can reduce the prefetcher noise and extract the private key with only 21% time overhead (compared to a noiseless system) when run in conjunction with Flush + Reload.

Through this work, we have shown the possibility of side-channel attacks on dynamic partitioning protocols. Static partitioning can mitigate side-channel attacks, however it has a huge performance penalty. Through cache line invalidation and the Modified-LRU reallocation policy, we are able to overcome the security vulnerability, while gaining a speedup of up to 29% and on average 7.12% compared to static partitioning. We also showed that this technique has a marginal performance cost of only 0.35% with respect to UCP on average. Hence, PASS-P can be applied on shared levels of cache for all dynamic partitioning protocols. Extension of PASS-P to provide security against newer attacks like Meltdown [57] and Spectre [48] is left for future work.

7.2 Conclusion

This thesis focuses on the performance, energy efficiency and security of multi-core architectures. Multi-core architectures can exploit the thread level parallelism present in the applications to improve throughput. However, the single thread performance still remains a bottle-neck. Prior works have shown that the single thread performance can be improved by exploiting the heterogeneity in ISAs. This can be done by dividing a program in multiple phases and then running each phase on its most affined ISA. At the run time, we should have information of the affinity of a phase. We can not give this information at compile-time due to the data dependent behaviour of program. So, the affined ISA to a phase has to be dynamically estimated at run time.

We have shown in this thesis that machine learning based schedulers are effective

in determining ISA-affinity to a phase. This thesis also shows that we can exploit heterogeneity at finer-level to further improve the performance. Function level scheduling is beneficial in terms of migration overhead reduction. Heterogeneous ISAs have multiple cores dedicated to each ISA. For a single-threaded program only one of the core remains active during execution while the remaining cores are idle. To effectively utilize the cores, we have shown that a dynamic core (HIDC) which supports multiples ISAs can not only improve the performance but also improve the energy efficiency. HIDC architecture improves the multi-threaded program's efficiency as well.

Security is another major concern in multi-core architectures. Prefetchers have been considered to be helpful in mitigating side channel attack. However, in this thesis we have shown that, prefetchers can not be relied upon for the security purposes. This is because the attacker can disable the prefetcher behaviour by intelligently bringing data into caches. These attacks can be mitigated by invalidating the cache lines which can cause the vulnerability. However, we found out that this causes huge performance degradation. We purposed a cache reallocation policy (PASS-P) to significantly reduce the performance degradation.

7.3 Future scope

We see several interesting avenues for future work. Exploration of energy efficient schedulers for heterogeneous-ISA architectures is envisaged as a future direction. In fine-grained scheduling [14], there is a scope to develop better techniques and architectures tailored for intra-function-wise migration. One of the limitations in function based technique is that it needs one or two forced migrations for every 20 calls of a function, which introduces unnecessary overhead. Hence, future scope also lies in proposing a predictor for data-dependent behavior of the function and taking the decision in advance for fine-grained scheduling. In future work, studies can be done in exploring scheduling at loop-level. As the binary of program has to kept for all the ISAs, so there can be another direction of future work in doing optimizations at compiler level to decrease FAT binary

size. The PASS-P mechanism mitigates attack possibility on timing side channel attacks. Future Scope also lies in proposing security techniques for remaining side-channel attacks.

– * – * –

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] O. Acıiçmez, B. B. Brumley, and P. Grabher. New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124. Springer, 2010.

[3] A. Adileh, S. Eyerman, A. Jaleel, and L. Eeckhout. Mind the power holes: Sifting operating points in power-limited heterogeneous multicores. *IEEE Computer Architecture Letters*, 16(1):56–59, 2016.

[4] S. Akram, J. B. Sartor, K. V. Craeynest, W. Heirman, and L. Eeckhout. Boosting the priority of garbage: Scheduling collection on heterogeneous multicore processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1): 1–25, 2016.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

[6] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.

[7] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-R. Chuang, V. Legout, and B. Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. *ACM SIGARCH Computer Architecture News*, 45(1):645–659, 2017.

[8] D. J. Bernstein. Cache-timing attacks on aes. 2005.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL http://doi.acm.org/10.1145/2024716.2024718.

[10] E. Blem, J. Menon, and K. Sankaralingam. 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*, pages 1–12. IEEE, 2013.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[12] N. K. Boran, R. P. Meghwal, K. Sharma, B. Kumar, and V. Singh. Performance modelling of heterogeneous ISA multicore architectures. In *East-West Design & Test Symposium (EWDTS), 2016 IEEE*, pages 1–4. IEEE, 2016.

[13] N. K. Boran, D. K. Yadav, and R. Iyer. Performance modelling and dynamic scheduling on heterogeneous-isa multi-core architectures. In *International Symposium on VLSI Design and Test*, pages 702–715. Springer, 2019.

[14] N. K. Boran, S. Rathore, M. Udeshi, and V. Singh. Fine-grained scheduling in heterogeneous-isa architectures. *IEEE Computer Architecture Letters*, 2020.

[15] N. K. Boran, D. K. Yadav, and R. Iyer. Classification based scheduling in heterogeneous isa architectures. In *2020 24th International Symposium on VLSI Design and Test (VDAT)*, pages 1–6. IEEE, 2020.

[16] M. Breughe, S. Eyerman, and L. Eeckhout. A mechanistic performance model for superscalar in-order processors. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 14–24. IEEE, 2012.

[17] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, 2008.

[18] N. Carlini and D. Wagner. {ROP} is still dangerous: Breaking modern defenses. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 385–399, 2014.

[19] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 161–176, 2015.

[20] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.

[21] S. Cho, H. Chen, S. Madaminov, M. Ferdman, and P. Milder. Flick: fast and lightweight isa-crossing call for heterogeneous-isa environments. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 187–198. IEEE, 2020.

[22] R. P. Colwell and C. Y. Hitchcock III. Computers, complexity, and controversy. *Readings in computer architecture*, page 144, 2000.

[23] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963, 2015.

[24] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *Linux Expo*, 1999.

[25] P. Cowan, W. Maier, S. Bakke, W. Grier, and S. Qian Zhang. Automatic adaptive detection and prevention of buffer-overflow attacks, jan. 29, 1998, usenix. In *7th Security Symposium proceedings*.

[26] L. Davi, D. Lehmann, and A.-R. Sadeghi. The beast is in your memory: Return-oriented programming attacks against modern control-flow integrity protection te chniques. *BlackHat USA*, 2014.

[27] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[28] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution migration in a heterogeneous-ISA chip multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 261–272, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151004. URL http://doi.acm.org/10.1145/2150976.2151004.

[29] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21, 2012.

[30] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On

the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy*, pages 781–796. IEEE, 2015.

[31] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. Prodact: Prefetch-obfuscator to defend against cache timing channels. *International Journal of Parallel Programming*, 47(4):571–594, 2019.

[32] A. Fuchs and R. B. Lee. Disruptive prefetching: impact on side-channel attacks and cache designs. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 14. ACM, 2015.

[33] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589. IEEE, 2014.

[34] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 417–432, 2014.

[35] P. F. Gorder. Multicore processors for science and engineering. *Computing in Science Engineering*, 9(2):3–7, 2007. doi: 10.1109/MCSE.2007.35.

[36] M. Grannaes, M. Jahre, and L. Natvig. Storage efficient hardware prefetching using delta-correlating prediction tables. *Journal of Instruction-Level Parallelism*, 13:1–16, 2011.

[37] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 17, 2011.

[38] D. Gullasch, E. Bangerter, and S. Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[39] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, Sept. 2006. doi: 10.1145/1186736.1186737.

[40] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7): 33–38, 2008.

[41] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[42] W.-M. Hu. Lattice scheduling and covert channels. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 52–52. IEEE Computer Society, 1992.

[43] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *ACM SIGARCH computer architecture news*, volume 35, pages 186–197. ACM, 2007.

[44] D. Jang, Z. Tatlock, and S. Lerner. Securing c++ virtual calls from memory corruption attacks. 2016.

[45] D. Joan and R. Vincent. The design of rijndael: Aes-the advanced encryption standard. In *Information Security and Cryptography*. springer, 2002.

[46] S. Kim and A. V. Veidenbaum. Stride-directed prefetching for secondary caches. In *Proceedings of the 1997 International Conference on Parallel Processing (Cat. No. 97TB100162)*, pages 314–321. IEEE, 1997.

[47] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.

[48] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[49] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[50] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM workshop on Computer security architectures*, pages 25–34, 2008.

[51] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. *MICRO-36*, 2003.

[52] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *ISCA*, 2004.

[53] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 23–32. IEEE, 2006.

[54] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[55] W. Lee, D. Sunwoo, C. D. Emmons, A. Gerstlauer, and L. John. Exploring opportunities for heterogeneous-isa core architectures in high-performance mobile socs. Technical report, Technical Report UT-CERC-17-01, The University of Texas At Austin, 2017.

[56] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 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)*, pages 469–480, Dec 2009.

[57] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[58] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[59] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. pages 317–328, 2012.

[60] A. Lukefahr, S. Padmanabha, R. Das, R. Dreslinski Jr, T. F. Wenisch, and S. Mahlke. Heterogeneous microarchitectures trump voltage scaling for low-power cores. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 237–250, 2014.

[61] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. G. Dreslinski, T. F. Wenisch, and S. Mahlke. Exploring fine-grained heterogeneity with composite cores. *IEEE Transactions on Computers*, 65(2):535–547, 2015.

[62] N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Cristal. Thread lock section-aware scheduling on asymmetric single-isa multi-core. *IEEE Computer Architecture Letters*, 14(2):160–163, 2014.

[63] N. Markovic, D. Nemirovsky, V. Milutinovic, O. Unsal, M. Valero, and A. Cristal. Hardware round-robin scheduler for single-isa asymmetric multi-core. In *European Conference on Parallel Processing*, pages 122–134. Springer, 2015.

[64] A. Naithani, S. Eyerman, and L. Eeckhout. Reliability-aware scheduling on heterogeneous multicore processors. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 397–408. Ieee, 2017.

[65] A. Navarro-Torres, J. Alastruey-Benedé, P. Ibáñez-Marín, and V. Viñals-Yúfera. Memory hierarchy characterization of spec cpu2006 and spec cpu2017 on the intel xeon skylake-sp. *Plos one*, 14(8):e0220135, 2019.

[66] B. A. Nayfeh and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9): 79–85, 1997. doi: 10.1109/2.612253.

[67] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, and A. Cristal. A machine learning approach for performance prediction and scheduling on heterogeneous cpus. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 121–128. IEEE, 2017.

[68] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 96–96. IEEE, 2004.

[69] M. Neve and J.-P. Seifert. Advances on access-driven cache attacks on aes. In *International Workshop on Selected Areas in Cryptography*, pages 147–162. Springer, 2006.

[70] J. Nider and M. Rapoport. Cross-isa container migration. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, pages 1–1, 2016.

[71] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[72] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke. Dynamos: dynamic schedule migration for heterogeneous cores. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 322–333. ACM, 2015.

[73] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptol. ePrint Arch.*, 2002(169):1–23, 2002.

[74] D. Page. Partitioned cache architecture as a ėide-channel defence mechanism. 2005.

[75] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent {ROP} exploit mitigation using indirect branch tracing. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 447–462, 2013.

[76] D. Patterson. The trouble with multicore. *Spectrum, IEEE*, 47:28 – 32, 53, 08 2010. doi: 10.1109/MSPEC.2010.5491011.

[77] C. Percival. Cache missing for fun and profit, 2005.

[78] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):318–319, 2003.

[79] M. Pricopi and T. Mitra. Bahurupi: A polymorphic heterogeneous multi-core architecture. *ACM Trans. Archit. Code Optim.*, 8(4):22:1–22:21, Jan. 2012. ISSN 1544-3566. doi: 10.1145/2086696.2086701. URL http://doi.acm.org/10.1145/2086696.2086701.

[80] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin. Power-performance modeling on asymmetric multi-cores. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–10. IEEE, 2013.

[81] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432. IEEE, 2006.

[82] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[83] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[84] C. Rohlf and Y. Ivnitskiy. Attacking clientside jit compilers. *Black Hat USA*, 2011.

[85] D. Sanchez and C. Kozyrakis. Scalable and efficient fine-grained cache partitioning with vantage. *IEEE Micro*, 32(3):26–37, 2012.

[86] L. Sawalha, S. Wolff, M. P. Tull, and R. D. Barnes. Phase-guided scheduling on single-isa heterogeneous multicore processors. In *2011 14th Euromicro Conference on Digital System Design*, pages 736–745. IEEE, 2011.

[87] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, volume 10, 2011.

[88] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[89] P. Smith and N. C. Hutchinson. Heterogeneous process migration: The tui system. *Software: Practice and Experience*, 28(6):611–639, 1998.

[90] S. Srinivasan, N. Kurella, I. Koren, and S. Kundu. Exploring heterogeneity within a core for improved power efficiency. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1057–1069, 2015.

[91] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, et al. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. *ACM SIGARCH Computer Architecture News*, 32(2):2, 2004.

[92] P. Team. Pax non-executable pages design & implementation. *Avaliable: http://pax. grsecurity. net*, 2003.

[93] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[94] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[95] K. Van Craeynest and L. Eeckhout. Understanding fundamental design choices in single-isa heterogeneous multicore architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013.

[96] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). 40(3): 213–224, 2012.

[97] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. Fairness-aware scheduling on single-isa heterogeneous multi-cores. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 177–187. IEEE, 2013.

[98] A. van de Ven. New security enhancements in red hat enterprise linux v. 3, update 3. *Raleigh, North Carolina, USA: Red Hat*, 2004.

[99] A. Venkat and D. M. Tullsen. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 121–132, June 2014. doi: 10.1109/ISCA.2014.6853218.

[100] A. Venkat, S. Shamasunder, H. Shacham, and D. M. Tullsen. Hipstr: Heterogeneous-isa program state relocation. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 727–741, 2016.

[101] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy. Papp: Prefetcher-aware prime and probe side-channel attack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[102] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh. Secdcp: secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[103] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[104] Y. Xie and G. H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. *ACM SIGARCH Computer Architecture News*, 37(3):174–183, 2009.

[105] M. Yan, Y. Shalabi, and J. Torrellas. Replayconfusion: detecting cache-based covert channel attacks using record and replay. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14. IEEE, 2016.

[106] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani. Cotsknight: Practical defense against cache timing channel attacks using cache monitoring and partitioning technologies. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 121–130. IEEE, 2019.

[107] Y. Yarom and K. Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[108] K. Yoshimura, T. Nakada, Y. Nakashima, and T. Kitamura. An energy efficient smt processor with heterogeneous instruction set architectures. In *Proceedings of the 9th IASTED International Conference*, volume 676, page 201.

[109] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573. IEEE, 2013.

[110] M. Zhang and R. Sekar. Control flow integrity for {COTS} binaries. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 337–352, 2013.

[111] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptol. ePrint Arch.*, 2005:388, 2005.