Fine-Grained Scheduling in Heterogeneous-ISA Architectures

Nirmal Kumar Boran, Shubhankit Rathore, Meet Udeshi, and Virendra Singh

Abstract—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). Programs have affinity towards different ISAs during its execution based on nature of code and data input. To extract maximum performance gain, we need to ensure that at every point in the program's execution, the program runs on its best affine core with minimum migration overhead. In this letter, we propose a function-wise fine grained scheduling algorithm which schedules every function of a program to its most affined ISA. Results show that our function-based scheduler can achieve speedup of up to 22.9 percent over state-of-the-art in heterogeneous-ISA architectures.

Index Terms—Heterogeneous-ISA, dynamic migration, multi-core, scheduling technique

1 Introduction

The history of microprocessor emphasizes the fact that *performance of single-threaded programs* is of prime importance. During the last century, researchers were successful in improving the processor's performance until the frequency scaling was saturated. Further performance enhancement required the addition of other resources like an increase in the size of register file, reorder buffer, instruction width, which ultimately led to an increase in the power budget. However, due to the limited power budget and to exploit the parallelism present in a program, the focus shifted from single-core to multi-core architectures. Multi-core architectures can contain homogeneous or heterogeneous cores. Earlier proposals by Kumar *et al.* [1], [2] explored the use of heterogeneous cores to improve energy efficiency and performance. To increase energy efficiency further, dynamic core architectures were explored by Ipek *et al.* [3], Padmanabha *et al.* [8], and many others. Improving single-threaded program's performance is still of interest to architects.

Venkat *et al.* [4] have shown considerable improvement in performance, power, and the energy-delay product with ISA heterogeneity. They have shown that if a program is divided into multiple phases, each of length 100 million dynamic instructions, then each such phase is found to have an affinity towards a particular ISA. This phasic affinity arises due to multiple factors, namely *code density, dynamic instruction count, register pressure*, etc. of the program. Their work showed that a significant speedup can be achieved by running each phase on its affined ISA, compared to running all the phases on a single ISA. DeVuyst *et al.* [5] developed a technique to migrate a program across different ISA at their equivalence points (points where the memory image of the program is consistent across different ISAs).

To explore the phasic affinity, 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 support. It shows that phases 1, 3, 4, 8, 9, 10 are affined to the ARM ISA, whereas remaining phases are affined to x86 ISA (Fig. 1a). This demonstrates that running the program solely on any of these ISAs is not the best choice, as the behavior of the program in different phases turns out to be different. Fig. 1b (bar 3) shows that when different phases of *sjeng* are executed on their most affined ISA, 31.6 percent speedup can be achieved compared to running all phases on x86 ISA.

Boran *et al.* [6], [7] have looked at the problem of predicting the best affined core. They have developed a general regression based algorithm in [6] and perceptron based algorithm in [7] 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.* Fig. 1b (bar 5) demonstrates that shorter phases can give better performance if we ignore migration overhead. Hence, the switching overhead (up to 100 μs in [4] and up to 18 μs in [9]) turns out to be the bottleneck in exploiting the fine-grain opportunities. To further exploit the ISA-heterogeneity on a finer level, a recent work [9] has proposed to statically partition a program with programmers' help for a master/slave processor architecture. Although their proposal is effective, it is oblivious to the dynamic affinity changes of a program. Hence, it has limited scope to static master/slave processor architectures. To overcome the shortcomings of previous proposals, *this paper focuses on harnessing ISA heterogeneity by scheduling the program dynamically at finer granularity, with minimal migration overhead 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 dynamically to it's most affined ISA. Fig. 1b (bar 6) shows if *sjeng* is scheduled function-wise, a speedup of 27.2 percent is achieved compared to phase-wise scheduling and 65.8 percent 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 [4], [6]. In principle, our proposal of function-wise dynamic scheduling is a promising technique. However, it raises several issues, such as algorithm to determine affinity, dynamic switching mechanism, and state migration policy.

The contributions of this paper 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 [9] and 1000× w.r.t [4] using proposed scheduling method.

The rest of the paper is organized as follows: Sections 2 and 3 presents the proposed function-wise scheduling mechanism and a heuristics. Section 4 demonstrates the effectiveness of the proposal and finally concludes with Section 5.

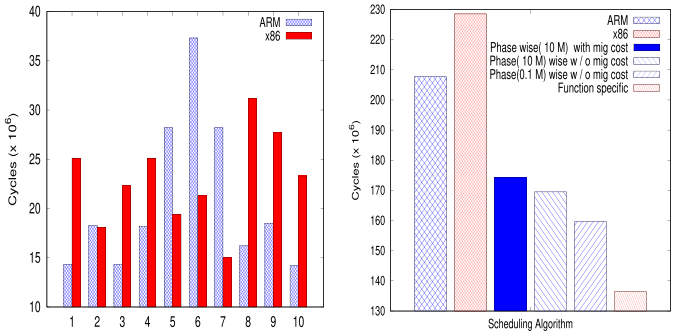
2 Proposed Function-Wise Scheduling

2.1 Function Affinity

Fig. 1b indicates that different functions have affinity towards different ISAs. This is due to multiple factors, such as *code density*,

• *The authors are with the CADSL, IIT Bombay, Mumbai, Maharashtra 400076, India. E-mail: nirmalkboran@iitb.ac.in, {shubhankitrathore, mudeshi1209}@gmail.com, viren@ee.iitb.ac.in.*

Manuscript received 6 Nov. 2020; accepted 30 Nov. 2020. Date of publication 15 Dec. 2020; date of current version 8 Jan. 2021.
(Corresponding author: Nirmal Kumar.)
Digital Object Identifier no. 10.1109/LCA.2020.3045056



(a) Execution time of different phases (b) Phase-wise and function wise execution time

Fig. 1. Execution time of different phases of benchmark *sjeng* and execution time with different scheduling algorithms.

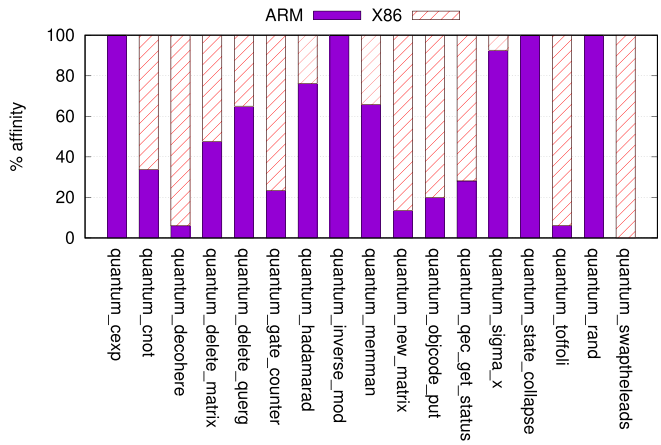


Fig. 2. *Libquantum* function affinity.

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.

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. 2. 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.

2.2 Fine-Grained Scheduling

Flow chart for function-wise scheduling is shown in Fig. 3. 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. 3. A heuristics based approach has been developed in this

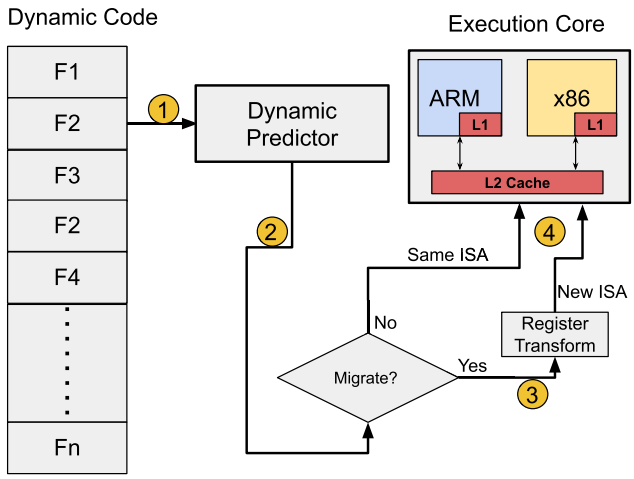


Fig. 3. Function scheduling flow chart.

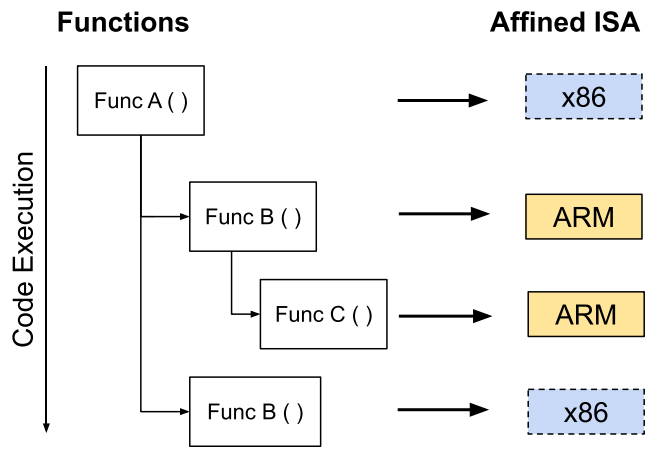


Fig. 4. Function order.

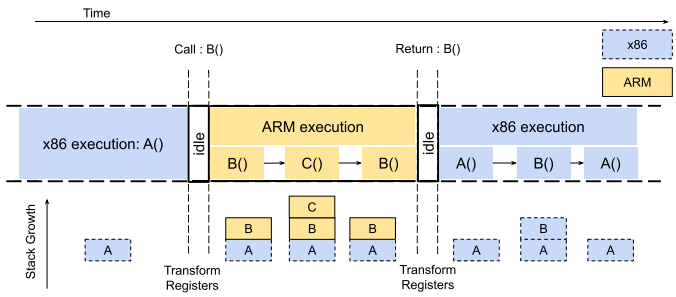


Fig. 5. Function-wise migration.

work for demonstrating function wise 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 [5].

The migration technique is explained in Fig. 4 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

TABLE 1
Core Configurations

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

in Fig. 5 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 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. 3.

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. 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 20th call. Execution time is noted for the last two calls i.e., 19th & 20th of the function, one on x86 and one on ARM. If 20th call (ARM) has lesser execution time compared to the 19th 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 19th and 20th 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.

We have experimentally found that storing 32 functions affinity in the Affinity table using LRU (Least Recently Used) is a balance between hardware overhead and affinity storage trade-off for SPEC workload. If the Affinity table size is increased to 64 entries then performance gain is increased by 1.2 percent only with more than double hardware overhead. 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. 3.

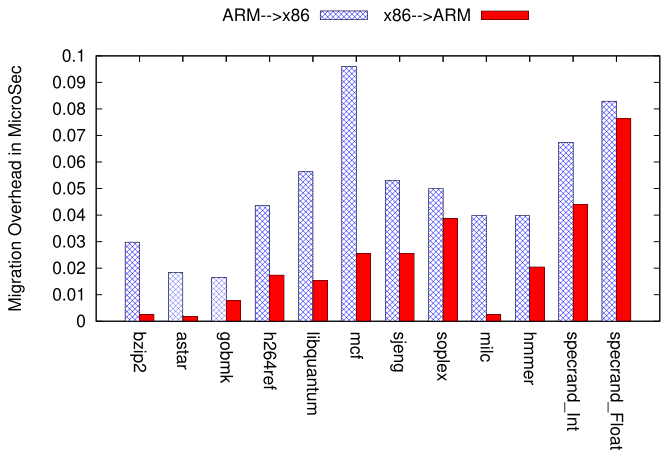


Fig. 6. Migration overhead for function-wise execution.

4 Results

In order to demonstrate the effectiveness of the proposed scheduling algorithm, we have simulated SPEC CPU2006 benchmarks using Gem5 [10] 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 DeVuyst *et al*. [5]. The core configurations used in simulations for both ISAs are mentioned in Table-1 which is similar to state-of-the-art work [6]. Each ISA core has a private L1 cache and a shared L2 cache.

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 [4]. The migration time obtained for function level migration is shown in Fig. 6 (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

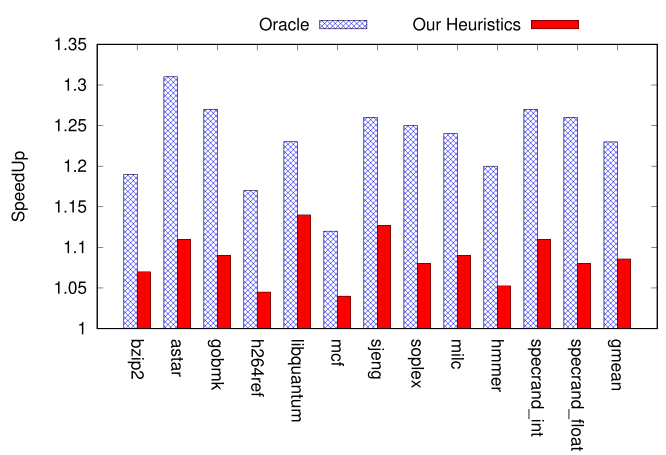


Fig. 7. Speedup of function-wise migration with respect to heterogeneous-ISA architecture.

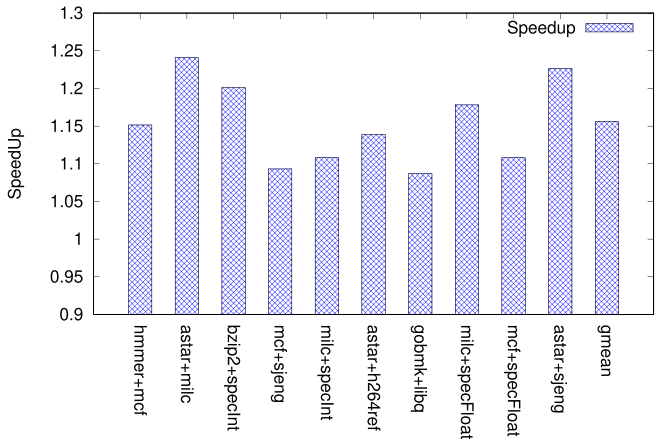


Fig. 8. Speedup of function-wise migration with respect to heterogeneous-ISA architecture for multiworkloads.

to be migrated. This function is integrated with Gem5 [10] to compute migration overhead.

Performance Results: We have compared the results with the state-of-the-art research by [7] as shown in Fig. 7. The proposed approach achieves up to 22.9 percent speedup using oracle based scheduling mechanism in both cases (ours and in [7]) and the proposed heuristics technique achieves a speedup of 8.4 percent. The speed up achieved with O3 optimization for oracle based scheduling is 22.1 percent. 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 [7]. 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.

Performance Results for Multi-Workload: Although our primary focus is on single threaded performance enhancement, multiworkload benchmarks have also been executed to see the multithreaded behavior for the proposed approach. The results are shown in Fig. 8. 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 percent.

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 19th & 20th function call. One 32-bit wide comparator is also required to compare the execution times for different ISAs. Total hardware overhead is 368 Bytes.

5 Conclusion and Future Work

With the increase in computing requirements, processor architecture needs timely improvement. Utilizing the ISA affinity present in different phases of a program can give a significant performance boost. *The proposal introduced a novel fine grained migration strategy for heterogeneous-ISA CMP, called* function-wise scheduling to solve prior challenges. 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 [4]). The proposed fine grained function wise scheduling achieved 22.9 percent in comparison to the state-of-art [7]. Exploration of energy efficient algorithms is envisaged as future direction. There is a scope to develop better techniques and architectures tailored for function-wise migration. Detecting affinity on finer granularity on loop level with the help of compiler can be a future direction. The limitation in this technique is that it needs one or two forced migrations for every 20 calls of a function, which introduces unnecessary overhead. The future scope also lies in proposing a predictor for data behavior and taking the decision in advance.

Acknowledgments

This work was supported in part by Indo Japanese Joint Lab Grant and Visvesvaraya PhD Scheme, Government of India.

References

[1] R. Kumar *et al.*, "Single-ISA heterogeneous multi-coe architectures: The potential for processor power reduction," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2003, pp. 81–92.
[2] R. Kumar *et al.*, "Processor power reduction via single-ISA heterogeneous multi-core architectures," *IEEE Comput. Archit. Lett.*, vol. 2, no. 1, pp. 2–2, Jan.-Dec. 2003.
[3] E. Ipek *et al.*, "Core fusion: Accommodating software diversity in chip multiprocessors," *ACM SIGARCH Comput. Archit. News*, vol. 35, pp. 186–197, 2007.
[4] A. Venkat *et al.*, "Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit.*, 2014, pp. 121–132.
[5] M. DeVuyst *et al.*, "Execution migration in a heterogeneous-ISA chip multiprocessor," in *Proc. Proc. 17th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 261–272.
[6] N. Boran *et al.*, "Performance modelling and dynamic scheduling on heterogeneous-ISA multi-core architectures," in *Proc. Int. Symp. VLSI Des. Test*, 2019, pp. 702–715.
[7] N. Boran *et al.*, "Classification based scheduling in heterogeneous ISA architectures," in *Proc. 24th Int. Symp. VLSI Des. Test*, 2020, pp. 1–6.
[8] S. Padmanabha *et al.*, "DynaMOS: Dynamic schedule migration for heterogeneous cores," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2015, pp. 322–333.
[9] S. Cho *et al.*, "Flick: Fast and lightweight ISA-crossing call for heterogeneous-ISA environments," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 187–198.
[10] N. Binkert *et al.*, "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.