Technical Report

Exploring Opportunities for Heterogeneous-ISA Core Architectures in High-Performance Mobile SoCs

Wooseok Lee, Dam Sunwoo, Christopher D. Emmons,
Andreas Gerstlauer, and Lizy John

UT-CERC-17-01

March 10, 2017

Computer Engineering Research Center
Department of Electrical & Computer Engineering
The University of Texas at Austin

201 E. 24th St., Stop C8800
Austin, Texas 78712-1234

Telephone: 512-471-8000
Fax: 512-471-8967

http://www.cerc.utexas.edu

The University of Texas at Austin
Electrical and Computer Engineering
Cockrell School of Engineering

Exploring Opportunities for Heterogeneous-ISA Core Architectures in High-Performance Mobile SoCs

Wooseok Lee[1], Dam Sunwoo[2], Christopher D. Emmons[2],
Andreas Gerstlauer[1], and Lizy K. John[1]
[1]The University of Texas at Austin, [2]ARM Research

ABSTRACT

High-performance processors use more transistors to deliver better performance. This comes at a cost of higher power consumption, often lowering energy efficiency. Conventional approaches mitigate this problem by adding heterogeneous energy-efficient cores for matching tasks. However, reducing high-performance components, such as caches or out-of-order processing capabilities, broadly affects performance. In this paper, we instead explore opportunities to increase energy efficiency by providing cores with restricted functionality, but without necessarily impacting performance. We aim to achieve this by removing support for complex but less frequently executed instructions. Since instruction mixes used by real-world workloads are often heavily biased, the potential to remove unused or less frequently used instructions is relatively high.

To explore such opportunities, we investigate which instructions are worthwhile to remove, how frequently they are used, and how much performance degradation is expected after removing direct hardware support. We analyze a subset of instructions in the ARM ISA and their corresponding logic burden in the microarchitecture. Furthermore, we present instruction profiling results and show performance degradation when candidate instructions are removed. Experimental results show that removing most of the identified complex instructions has negligible impact on performance except for NEON instructions, which result in large performance degradations for floating-point-oriented workloads.

We further propose a heterogeneous-ISA system to achieve energy efficiency without performance degradation using a system architecture that combines both full- and reduced-ISA cores. Results show that by providing the flexibility of heterogeneous-ISA cores, the proposed system can improve energy efficiency by 12% on average and up to 15% for applications that do not require NEON support, all without performance overhead.

1. INTRODUCTION

Current mobile systems-on-chips (SoCs) incorporate numerous heterogeneous computing resources in an effort to improve energy efficiency. Specialized processing elements provide energy-efficient computing by offloading various types of computations onto dedicated accelerators. For general-purpose workloads, architects have traditionally spent available transistors to increase processor performance. However, this also leads to increased power consumption, often deteriorating energy efficiency. The conventional approach to handle this problem is to take advantage of heterogeneity at the system level by switching between high-performance and energy-efficient cores [3]. Such heterogeneous systems match application demands with core types to maximize energy efficiency. In such systems, the heterogeneity lies in the microarchitecture. In bigger cores, more transistors are spent on components that improve performance, such as caches, branch predictors, and out-of-order processing capabilities. In small cores, reducing the amount of performance-relevant resources can, however, be detrimental to some workloads.

An alternate approach is to implement energy-efficient cores by restricting functionality instead of giving up performance. Specifically, by reducing resources that have less impact on performance, power dissipation can be alleviated without losing significant performance. The benefit of implementing certain features in the Instruction Set Architecture (ISA) is highly dependent on workloads. Not all instructions are frequently used by every workload. If a particular workload favors specific instructions that are not directly supported by the hardware, performance dramatically decreases. By contrast, there is little performance degradation if those instructions are not frequently used.

In this paper, we explore opportunities for systems comprised of heterogeneous reduced-ISA cores to improve energy efficiency. We first identify a subset of instructions that are complex to implement, but are less frequently used. We find that ISAs are often broadly inclusive in that they incorporate instructions to support a variety of workloads across processor generations. However, once instructions are defined in an ISA, backwards compatibility requires all future processors to implement them whether they are frequently used or not. These less frequently used components add complexity inside the microarchitecture while contributing little to overall performance.

We identify candidate instructions that are complex to implement and profile how frequently the selected instructions are used in state-of-the-art ARM-based systems. Results show that workloads are biased towards certain instructions depending on the application or program phase. Basic instructions for essential data processing such as add, branch, move, load, and store are frequently used across all benchmarks. However, subsets of instructions are selectively or less frequently used, suggesting opportunities to remove direct hardware support for them. Since instruction selection depends on the compiler, we investigate results from major compilers, where results show minimal variances.

We further evaluate performance degradations if candidate instructions are removed. If performance degradation is large, it is not worthwhile to consider removing them despite the potential power benefit. We evaluate system performance with a reduced instruction set running several benchmarks. Results show that some subsets of instructions are critical to performance while others are not essential since in most cases, their usage frequency is low enough to not impact performance significantly.

Lastly, we propose a heterogeneous-ISA architecture to obtain energy benefits while maintaining performance across a wide range of workloads. In our proposed system ar-

chitecture, reduced-ISA cores remove hardware support for complex instructions, which increases energy efficiency but requires trapping and emulating of non-supported instructions. When unsupported instructions are infrequent, a workload runs on the reduced-ISA core to reduce energy. By contrast, when unsupported instructions are prevalent, the workload is migrated to a traditional full-ISA core. This dynamic core switching allows the system to avoid the performance degradation and energy inefficiency of software-emulated instructions. As long as they are not performance-critical, a compiler can thereby optimize binaries to remove unsupported instructions and thus maximize residency on the reduced-ISA core. Our results show that workloads without performance-critical instructions spend most of their execution time on reduced-ISA cores, achieving energy savings of up to 15%. Workloads with frequent use of unsupported instructions execute exclusively on full-ISA cores with no change in performance or energy consumption. On average, 12% energy savings at little to no performance cost are observed across a variety of benchmarks, where applications migrate between reduced- and full-ISA cores depending on dynamically varying instruction usage.

2. ISA ANALYSIS

In this section, we discuss the relationship between performance and the versatility of an ISA. Furthermore, we analyze the influence of instructions as a source of logic. As a case study, we select the ARM V7 ISA, and one of the performance-oriented ARM processors, Cortex-A15, as a baseline. The general concept and methodology can, however, be applied to any target.

2.1 Instructions and Performance

The performance of a processor can be determined from various factors using the so-called iron law of processor performance as follows:

$$\frac{ExecTime}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle} \quad (1)$$

To reduce execution time, compacting the instruction sequences is helpful for many workloads. Accordingly, instructions are carefully defined to reduce the code required for specific workloads as much as possible. In other words, removing support for specific instructions may cause an increase in instruction count, potentially hurting performance.

However, since they are not completely independent, the performance loss of one factor can be compensated by others. For instance, by removing support for certain instructions, the cycle time, i.e. cycles per instruction (CPI) can be improved. Moreover, as most modern processors have multiple-issue widths, any benefit from reducing a specific data path is multiplied. Even if performance benefits and losses break even, it may still be worthwhile to remove instruction support since the processor has reduced logic, leading to better energy efficiency. Exploring this trade-off is one of the goals of this research.

Minimizing the increase in instruction count due to the removal of instructions is a way to improve the energy efficiency further. Applications often show diverse characteristics. Even a single application can have diverse phases throughout its execution. One characteristic is the instruction utilization pattern. Not all instructions are fully utilized by individual applications or phases. Thus, by meticulously selecting the instructions required to support a range of workloads, we can reduce the number of instructions required in a processor.

2.2 Instructions as a Source of Logic

To maximize the benefit of reducing instructions, identifying instructions that greatly impact logic complexity is essential. Removing instructions that do not result in logic reductions unnecessarily increases the danger of code bloat.

We find that the overall number of instructions in an ISA is not as crucial for logic reduction as we first expected. While macro-operations are directly related to the logic in the decode stage, the rest of the pipeline is more driven by micro-operations. Basic micro-operations share many common datapath resources, such ALUs, that can not be removed. Our study rather focus on the specific semantics required by particular instructions that contribute to large logic within the microarchitecture. Since the processor is unaware of the instruction until decoded, we regard the fetch stage as an irrelevant block to consider for our analysis.

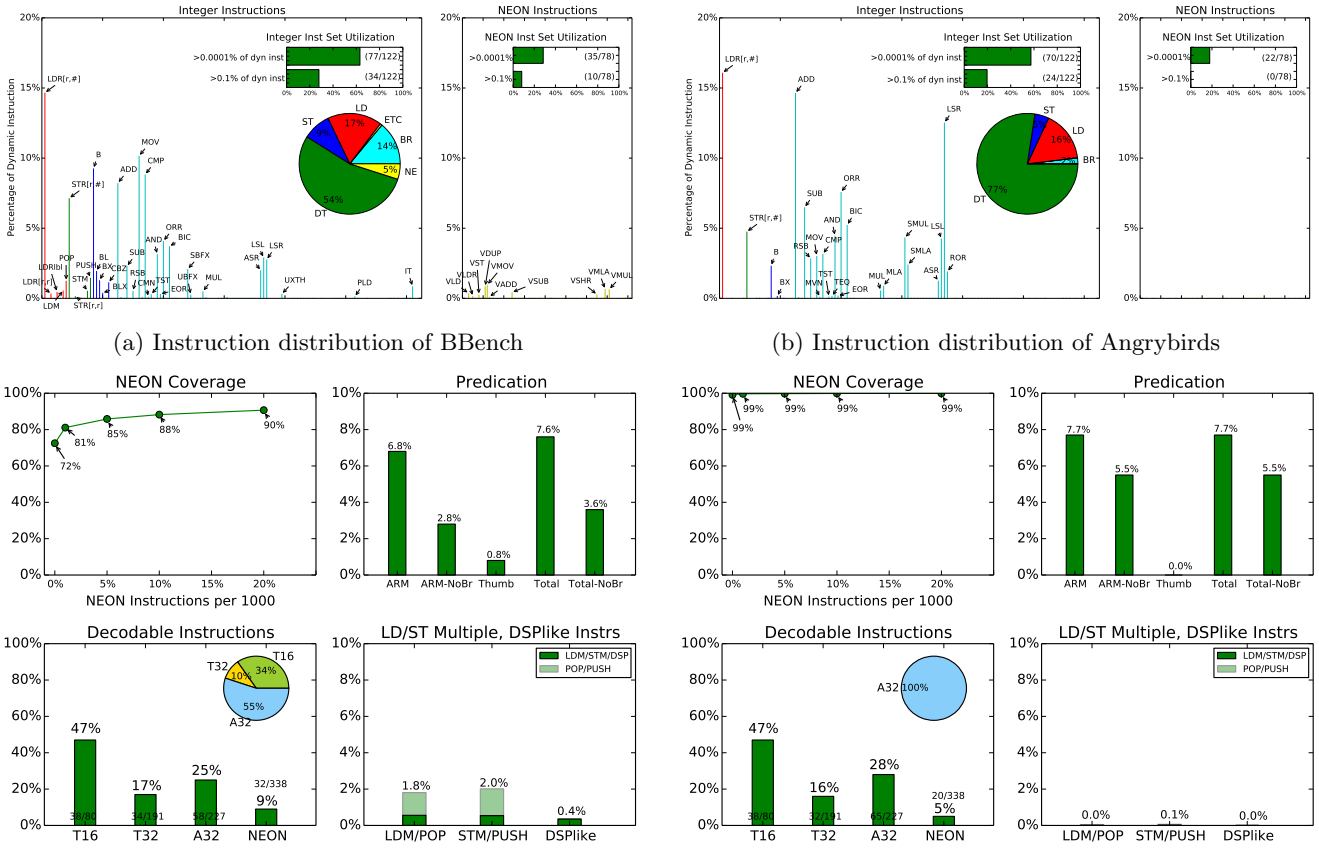
In the decode stage, the number of decodable instructions is a major source of logic. In order to shorten the cycle time, parallel logic is commonly used to determine the fetched instructions. For superscalar machines, the logic reduction effects are multiplied through duplicated decoder blocks. In case of the ARM ISA, multiple instruction sets, ARM and Thumb, mandate separate decoding blocks.

One specific category of instructions that incurs complexity are load and store multiple (LDM/STM) instructions. Determining possible source or destination registers for corresponding micro-operations is difficult to implement within one pipeline stage in a multiple-issue machine. Instead, collecting possible data values is done speculatively for all instructions in parallel with decoding them. Moreover, there is overhead for propagating of decoded instruction semantics through the pipeline and for exception handling of such long-running instructions in the execution back-end.

Another source of logic are predicated instructions [5]. Predicated instructions can be implemented as simple selection logic in the write-back stage. However, to avoid stalls due to data dependencies, modern high-performance processors carry both old and new data values simultaneously through the pipeline. This incurs overhead for pipeline registers and wires. In case of floating point operations, data size is doubled or quadrupled.

DSP-like instructions (QADD, SSAT, etc.) [2] complicate data paths in the integer pipeline. The various types of instructions, which combine consecutive add and multiply operations, help compact the instruction set. However, since normal integer operations are most commonly used, architects tend to keep pipeline stages as short as possible, forcing a parallelized approach of implementing DSP operations. This places additional logic burdens on the pipeline.

The NEON instruction set extensions in ARM support floating-point (FP) and SIMD operations, but are a large source of logic [1, 8]. NEON instructions require extra logic blocks for FP and SIMD functionality. Additionally, predication combined with NEON operations aggravates logic complexity. A register alias table and an extra register file for NEON instructions incurs logic in dispatch stages. Data paths, including data passing, write-back and forwarding of the execution result also make the routing complex. Lastly, a separate decoding block for NEON instructions is required.



(a) Instruction distribution of BBench

(b) Instruction distribution of Angrybirds

(c) Instruction analysis of BBench

(d) Instruction analysis of Angrybirds

Figure 1: Instruction analysis of mobile workloads.

3. PROFILING INSTRUCTIONS

After identifying a few candidate categories of instructions to potentially remove (LDM/STM, predicated instructions, DSP-like instructions, and NEON), we provide detailed profiling results to observe overall instruction usage of our target instructions in this section. For comprehensive exploration, we run mobile Android workloads and SPEC2006 CPU benchmarks for profiling and performance evaluation.

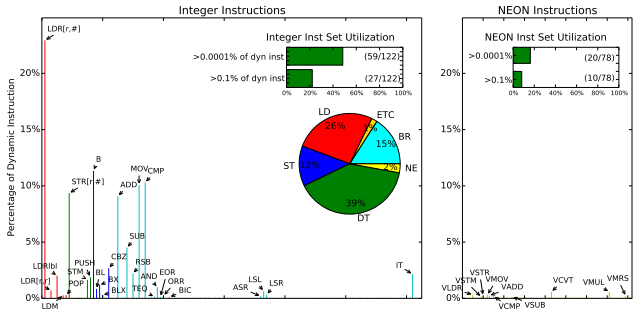
3.1 Instructions in Android

To observe the frequency of our target instructions, we present detailed analyses of instructions for two mobile applications, BBench[15] and Angry Birds, running on Android 4.2 (using the Dalvik JIT compiler). We collect selective instruction information from BBench and Angry Birds, running SimPoints on gem5 with replay of previously recorded user interactions [22]. All of the macro instructions are dumped and parsed with custom Python scripts. Since this approach takes advantage of SimPoints to reduce simulation time, analysis results are appropriately scaled according to SimPoint weights. One minor disadvantage of gem5 is the absence of a GPU model, resulting in software-based openGL support.

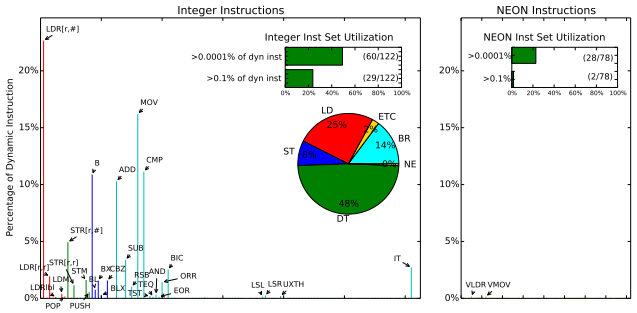
Figures 1(a) and 1(b) show histograms of individual instruction frequencies including overall mix of instruction types separately for integer and NEON instruction categories. To better interpret the numbers, we also provide dynamic instruction counts that account for more than 0.1% and 0.0001% of all instructions. Interestingly, the amount of instructions that show up frequently and thus may affect performance is small. Less than 30% of integer instructions are used for more than 0.1% of dynamic instructions. In case of NEON instructions, Angry Birds uses almost none while BBench takes advantage of limited NEON support. This is attributed to the biased selection of instructions by the compilers depending on the workload.

Figures 1(c) and 1(d) show detailed breakdowns for the four sets of candidate instruction categories. We find that only a limited amount of NEON instructions is used. NEON coverage measures how many NEON instructions are used for each 1000 instruction interval (i.e., 5% means 50 out of 1000 instructions are NEON). For BBench, no NEON instruction exists for 72% of the total execution time while, again, Angry Birds uses almost none. For the predication category, we can see that predicated instructions are prevalent in the ARM ISA while a small number of IT (If-Then) instructions exists for Thumb. Since branches have conditional fields but are not classified as predicated, we excluded branch instructions from these results. Without branches, about 3% to 5% of instructions are predicated in total.

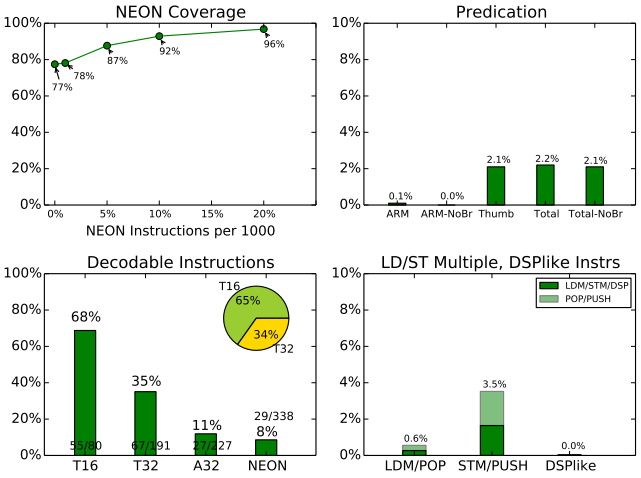
Interestingly, the number of instruction types (opcodes) used in both benchmarks are similar (Decodable Instructions, Figures 1(c) and 1(d)). We counted the instructions that appeared more than once during the whole execution.



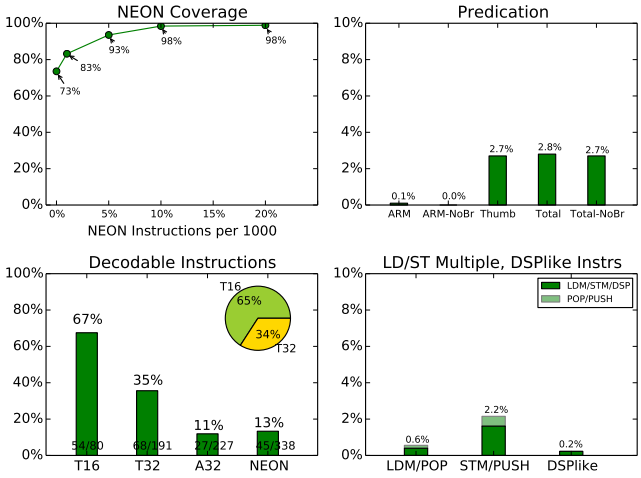
(a) Instruction distribution of SPEC CPU2006 INT (GCC)



(b) Instruction distribution of SPEC CPU2006 INT (LLVM)



(c) Instruction analysis of SPEC CPU2006 INT (GCC)



(d) Instruction analysis of SPEC CPU2006 INT (LLVM)

Figure 2: Instruction analysis of SPEC CPU2006 integer benchmarks.

This allows us to measure how efficiently the instruction set is used. It turns out that only a limited number of instructions are selected by the compiler on Android. Viewed another way, this implies that the logic for unused instructions can be removed. Compilers might generate some instructions very rarely. In theory, the logic for even rarely used instructions must exist to ensure proper functionality of processing those instructions. However, if we control the compiler or modify the OS to not generate or trap and emulate those instructions, the corresponding logic to support them is unnecessary.

Finally, there are a limited number of LDM/STM instructions. In total, less than 4% of instructions are load/store multiples for BBench. For Angry Birds, only a limited amount is used. DSP-like operations are rarely used in either application. Among them, only the sign/zero extension variants and bit manipulation instructions are frequently used. Others are almost never generated by the compiler.

3.2 Instructions in SPEC

We performed a similar profiling and analysis for SPEC CPU2006 [10] benchmarks. Observing the instruction patterns of these benchmarks allow us to learn the general distribution of instructions for the broader spectrum of workloads in SPEC [21]. We ruled out Fortran-based benchmarks because of compiler issues and consequently select 13 benchmarks (9 INT and 4 FP). We use test input sets for instruction and power analysis and the reference inputs for performance evaluation.

Compilers translate high-level source code into various instruction patterns. These instruction patterns can depend on and vary between compilers. To measure the variances in generated patterns, we compare the output of the two most prominent compilers, GCC [7] and LLVM [17]. In our paper, all SPEC binaries are generated using `-O3` and `-mcpu=cortex-a15` flags with static library linking.

Figure 2 shows detailed results of our analysis. We weighted each benchmark equally using arithmetic means. In addition, to avoid interference from different types of workloads, we show analysis results for integer benchmarks only. Due to space limitations, FP benchmark results are excluded. The instructions generated by the two compilers are similar but not identical. Even though instructions accounting for more than 5% are the same in both compilers (Figures 2(a) and 2(b)), the distribution and other minor instructions are different. For instance, while GCC prefers to use more branches, LLVM favors predicated instructions and moves more data. Both compilers use a similar proportion of NEON instructions, where LLVM spreads the NEON instructions more evenly across the execution phases compared to GCC. Interestingly, the number of instructions used more than once are similar, showing that both compilers utilize a limited number of instructions in the end.

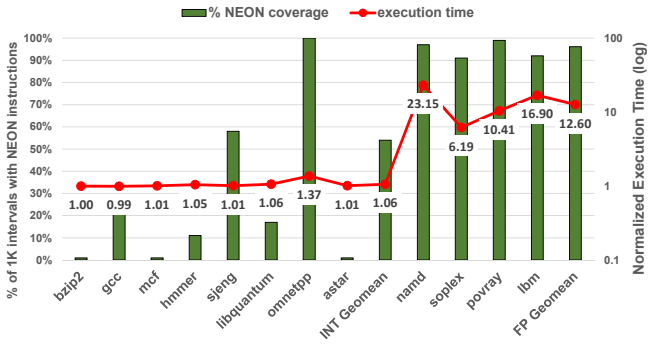


Figure 3: Performance of ISA without NEON instructions.

Figure 4: Performance of ISA without conditional instructions.

4. REDUCED-ISA PERFORMANCE IMPACT

Based on profiling results in Section 3 and the identified relationship between instructions and logic overhead in Section 2, we define a few reduced instruction sets as case studies and observe the performance overhead of each with SPEC CPU2006 benchmarks. For this study, we modify the compiler to generate binaries in which candidate instructions are removed. In general, rather than relying on subsequent instruction emulation, compilers can be controlled to produce desired instruction patterns. By removing unsupported instructions without otherwise affecting the overall program flow, benefits can be maximized. We modified LLVM to generate all binaries with some minor changes to limit the generation of specific instructions. A dynamic instruction trace was generated the same way in Section 3. The execution time was measured on an Arndale board (dual-core Cortex-A15) [6]. We ran the benchmarks as a single thread one at a time on Linux. In addition, to avoid possible CPU throttling, the processor was set to run the applications at 1.2GHz while measuring total execution time.

4.1 Performance Trade-off

Since removing instruction support might be detrimental to performance, care should be taken when refining instructions. The benefit should be larger than the loss. The benefit is in logic reduction and lower power consumption reduction while the loss is in performance.

Measuring the performance loss is a challenging task. Instruction patterns are so delicate that a small change in either the source code or compiler options may result in quite different patterns. Due to the fragility of patterns, the output of binaries can vary a lot. Nevertheless, despite such differences, the number of instructions that are frequently

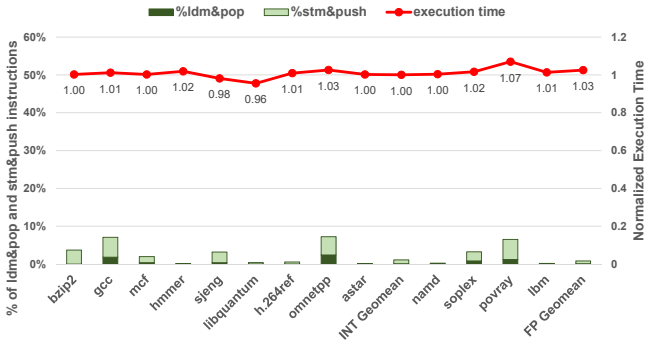


Figure 5: Performance of ISA without Load/Store Multiple instructions.

used is still limited. Hence, with millions of instructions from several benchmarks, we can measure the overall role of certain instructions and the effects of removing them.

NEON instructions are a large source of logic to sustain performance. In other words, the performance loss from removing NEON instructions is also large if they are heavily used. To measure the performance impact of NEON instructions, we compare the execution time of SPEC benchmarks with and without NEON instructions. Note that since we could not fully remove all NEON code from some of the pre-compiled libraries, there still exist a negligible number of NEON instructions. Figure 3 shows the detailed performance loss and original NEON instruction usage for each benchmark. For consistency with Figures 1(c)/(d) and 2(c)/(d), we report NEON coverage here as the fraction of 1K instruction intervals that contain NEON instructions.

For floating-point benchmarks, the performance slowdown is significant, ranging from about 6 to 23 times. When removing floating-point support in the ISA, there is no other choice for the compiler than to resort to soft-float emulation. By contrast, for integer benchmarks, which sporadically use NEON instructions, the performance degradation is less than 7% with the exception of *omnetpp*. Since *omnetpp* contains a large number of NEON instructions, the 37% performance degradation is understandable. These profiling result match prior work [11].

Figure 4 shows results for a reduced ISA that excludes predicated instructions. Avoiding to use of conditional instructions is done by disabling the *SimplifyCFG* optimization pass in LLVM. When removing this optimization pass, the compiler also loses the chance to generate optimal code. Thus, the slowdown is not solely from removing conditional instructions. However, since we observed that only a very small number of instructions are changed compared to full-ISA binaries, we regard such effects as negligible.

Overall, the performance loss is less than 5% except for *hmmer*. Removing conditional instructions results in an increased amount of branches in the code. Results indicate that the branch predictor of a modern high-performance mobile processor is good enough to handle the removal of predicated instructions. In case of *hmmer*, further analysis showed that the single hot loop in which most of the execution time is spent shows significantly worse branch predictor performance, causing the severe performance degradation.

Load and store multiple instructions provide a way to reduce a number of consecutive load/store operations. However, since modern instruction prefetch/fetch and out-of-order logic can hide such an increase in instruction count, the

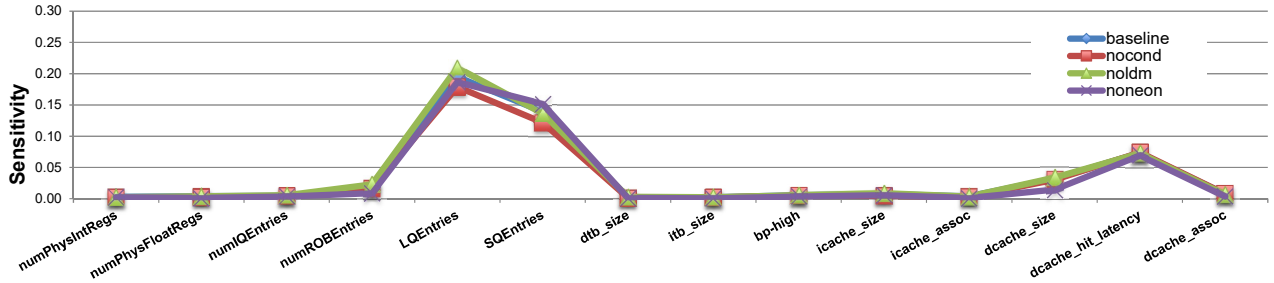


Figure 6: Microarchitectural sensitivity of reduced ISAs to changes in sizes of different microarchitectural components.

penalty of replacing them with separate load/store instructions is somewhat mitigated. Figure 5 shows the detailed performance with and without load/store multiple instructions. The performance loss is negligible across benchmarks.

We find that current compilers hardly use DSP-like instructions for most types of workloads. Some of them are never used by the LLVM compiler back-end in the first place. Consequently, it is meaningless to conduct performance comparison with or without these DSP-like instructions. However, despite the fact that we skipped the performance evaluation, a reduced ISA without DSP-like instructions to reduce the logic burden is still valid.

4.2 Microarchitectural Sensitivity

The role of high-performance microarchitecture components such as caches or branch predictors is to streamline the flow of instructions. Normally, architects design and optimize them under the assumption of implementing all instructions defined in the ISA. However, since we limit and change the patterns of the instruction stream, pre-optimized parameters might not be valid with a reduced ISA. The possibility of suboptimal configurations may affect performance as well as energy efficiency. If the reduced ISA increases pressure on the branch predictor due to the additional branches without conditional instructions, the branch predictor should be reinforced, potentially negating the benefits that we get.

To evaluate these effects, we measure the sensitivity of a reduced ISA to changes in microarchitectural components using fractional factorial experiments [22] on SPEC benchmarks in gem5 (Figure 6). Such experiments have been used in the past for identifying microarchitectural components critical to performance. They measure the normalized performance impact of variations in component parameters over a range of experiments. We use this setup to identify how much each reduced instruction set can potentially benefit from an increase in various microarchitectural resources. Our experiments show that the synthetic change of reduced instruction patterns does not benefit from nor require any additional hardware in the high-performance components.

5. HETEROGENEOUS-ISA SYSTEM

In this paper, we propose adding heterogeneity at the ISA level. Reducing the ISA, instead of high performance components, creates an energy-performance trade-off along a different dimension. Since applications show diverse characteristics, not all instructions in the ISA are required for every application. Some workloads require a very small subset of instructions, while others take advantage of most instructions in the ISA. In particular, frequently executed hot functions often show preference to a small amount of instructions, providing the potential to run them on a reduced-ISA core. Figure 7(a) shows the concept of a heterogeneous system with reduced-ISA cores in comparison to traditional heterogeneous systems. In the remainder of the paper, we define *rISA* as the reduced ISA with all the previously mentioned instructions removed.

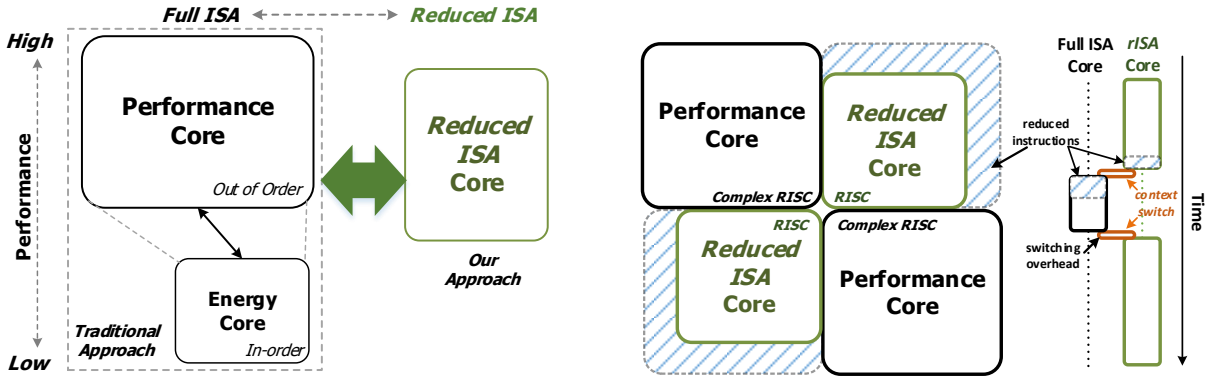
5.1 Full- and Reduced-ISA Cores

We propose a heterogeneous system that contains a combination of both full- and reduced-ISA cores (Figure 7(b)). Traditional full-ISA cores execute applications with no change in performance or energy consumption. By contrast, by cutting down on the instructions supported by the underlying microarchitecture, the logic complexity of reduced-ISA cores is decreased. This allows such cores to achieve lower power consumption. At the same time, unsupported instructions need to be trapped and software-emulated. As discussed in Section 4, a compiler can create binaries that are optimized for execution on the reduced-ISA core. Nevertheless, if removed instructions are critical, as is the case with NEON instructions for floating-point workloads, performance suffers severely.

For workloads with unsupported instructions that are sensitive to performance, a heterogeneous-ISA system can alleviate the performance degradation. When running an application in which performance-sensitive instructions are not prevalent, it is better to run it on reduced-ISA cores. However, when the application is in a phase where it includes performance-sensitive instructions, the application switches to the full-ISA core where it does not lose any performance. The core switching overhead depends on the granularity and subsequent frequency of switching. We present evaluation results for varying core switching granularities in Section 6.

5.2 Dynamic Core Switching

Dynamically monitoring the instruction streams and switching cores based on performance is the best way to maximize energy savings. In this approach, the process scheduler can dynamically map applications to an appropriate ISA core based on the information from hardware performance counters, such as the number of NEON instructions executed during the current scheduling period. As mentioned above, we assume that unsupported instructions are executed using emulation after undefined instruction exceptions in the reduced-ISA core. This will only be acceptable for infrequent use of the software-emulated instructions. Once the number of software-emulated instructions exceeds a certain threshold within a certain time period, the application should be migrated to a full-ISA core to avoid further performance degradation. On a full-ISA core, on the other hand, the same instructions are also monitored. If the instructions are not used during a certain time period, the application is



(a) A heterogeneous system with a reduced-ISA core (b) Full- and reduced-ISA core heterogeneous system

Figure 7: Heterogeneous-ISA system.

Algorithm 1 Dynamic Core Switching

```
function SCHEDULE()
    if neon_cnt ≤ neon_threshold then
        if fISA core then
            if neon_low_ticks ≥ switch_threshold then
                next_core ← rISA
            else
                neon_low_ticks ← neon_low_ticks + 1
            end if
        end if
    else                    ▷ NEON instructions more than threshold
        if rISA core then
            next_core ← fISA
            neon_low_ticks ← 0
        end if
    end if
    neon_cnt ← 0
end function
```

migrated back to the reduced-ISA core.

For optimal energy and performance balance, dynamic core scheduling is imperative. Algorithm 1 shows an example of core scheduling in case of heterogeneous systems with and without NEON support. At the end of each scheduling period, the scheduler determines which core the application should run on based on NEON instruction counts. If there are more NEON instructions than a predetermined threshold, it is better to run the application on a full-ISA core to reduce performance loss. In the opposite case, it is better to run the application on a reduced-ISA core for energy efficiency. Since we are predicting the next period based on the current one, instant switching from full- to reduced-ISA core might induce a ping-pong switching situation. Therefore, we add an optional *switch_threshold* variable. We evaluate how different switch thresholds affect overall performance in our experiments (see Section 6.1). Core switching happens when there are consecutive periods where NEON instructions are above or below the *neon_threshold*. For these experiments, we assume that there are two cores, full- and reduced-ISA, respectively, and no other programs are running.

5.3 Estimation of Power Reduction

Reducing the ISA will reduce the amount of logic. However, the effectiveness varies depending on the target ISA, target instruction, implementation methodology, microarchitecture, etc. In particular, the benefit of reducing logic and routing complexity depends on how aggressive the performance target is. The higher the targeted clock frequency is, the larger the benefit of a reduced-ISA core. Thus, we explore the benefit of a reduced ISA in the context of a high-performance baseline, targeting the reduction of power consumption and proposing a new heterogeneous system that attains the maximum benefit from such reduced-ISA cores.

While implementing a real reduced-ISA core is the best way to prove energy efficiency benefits, designing an actual new processor is extremely challenging. Thus, we use McPAT [18] to estimate power consumption. To quantify the benefits of reduced-ISA cores, we first present implementation details of cells and clock distribution networks.

Cells, either standard or customized, have diverse characteristics depending on the design goal. For less power-hungry processors, high or standard threshold voltage (HVt or SVt) cells can deliver relatively less leaky logic and small area at the expense of higher gate delay. By contrast, low threshold voltage (LVt) cells have relatively faster gates, but require higher leakage power and larger area. Thus, a simple way to achieve both performance and low-power consumption is to implement logic with HVt and SVt cells and later optimize only the timing-critical paths with LVt cells [12]. Unfortunately, LVt cells not only have higher area and power themselves, but also have a negative compounding effect on pre-existing cells. The replaced cells aggravate wire delays since the area expansion of cells in congested areas increases the distance to other cells. Subsequently, with a larger design, disproportionately more LVt cells are needed to compensate for the increased delay, causing significant increases in power during the final stages of physical design.

Clock networks are another source of dynamic power consumption. Since synchronous logic requires a common clock source across the whole design, clock trees must be designed with special care. With larger logic, additional buffers are needed to adjust the jitter for stability and maximum performance. Due to its toggling nature, the clock distribution network is a large source of dynamic power consumption.

In summary, additional logic often comes with a large overhead, requiring additional area, wiring, expensive cell types and larger clock trees. Conversely, a logic reduction can help reduce static and dynamic power significantly.

Unfortunately, there are limitations when estimating power consumption using McPAT. First, it has no notion of implementation details such as cells types. Also, parameter-

Table 1: Estimated rISA logic and area reduction.

	Logic Reduction				Affected Area		
ISA	**ID**	**RE**	**EX**	**CK**	**ID**	**RE**	**EX**
nocond	1%	3%	3%	10%	1	1	0.3
noldm	10%	10%	5%	5%	0.2	0.4	1
nodsp	0%	0%	10%(Integer)	10%(Integer)	1	1	0.5
noneon	40%	50%	10%	10%	0.3	1	1
Clock distribution network area: 0.2							
LVt cells: 0.1(pes)/0.3(normal)/0.5(opt) of total cells with 10% reduction.							
Synergistic combined LVt reduction: 0.02(pes)/0.05(normal)/0.1(opt) of total cells							

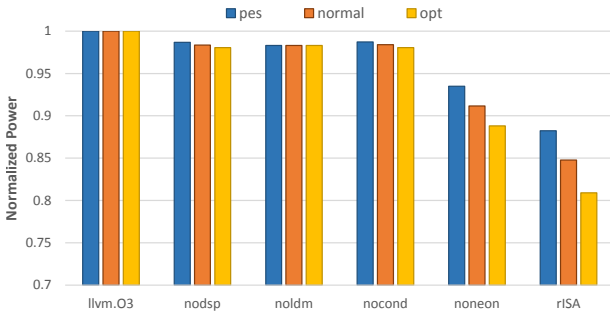


Figure 8: Power estimation of reduced ISA (SPEC CPU2006 INT).

ized models abstract away physical information and effects of logic reductions after actual place and route is done. We estimate such effects by adjusting numbers under certain assumptions about physical and implementation details.

In general, we consider logic, wire and clock reduction effects as well as subsequent replacement of LVt cells for power modeling (Table 1). We use estimates of the affected area and corresponding logic reductions to obtain power reduction factors by which we scale static and dynamic power estimates from McPAT for instruction decoding (ID), rename/dispatch (RE), and execution (EX) blocks. In addition, we apply estimated power reductions in the clock distribution network (CK) of each block using clock tree area and logic estimates. Finally, we consider savings due to wiring and LVt cell reductions, where we account for additional synergistic savings when all rISA variants are combined.

As accurately quantifying power consumption is challenging, we give a range of power estimates: *opt* (optimistic) indicates that the effect of the logic reduction is assumed to be relatively large while *pes* (pessimistic) means the opposite (Table 1 and Figure 8). The runtime power reduction for removing most instruction types is about 2% each, except for NEON instructions, which show about 9% improvement. Due to the orthogonality of removed instructions, we assume that the logic related to each instruction type is independent, and, thus, that power reductions are additive. Our rISA core shows about 15% power reduction compared to the full-ISA one.

6. EXPERIMENTS AND RESULTS

6.1 Performance

To observe the performance degradation based on the core switching scenario, we evaluate SPEC CPU2006 integer benchmarks using QEMU [9] while collecting a trace of all NEON instructions. We focus on NEON instructions as other removed instructions have little impact on performance as previously shown. We simulate core switching using the previously shown scheduling algorithm that accounts for the number of NEON instruction in each scheduling period. Since switching is not free, each core switching incurs a fixed instruction overhead. Furthermore, since emulating NEON instructions leads to additional instructions, every time a NEON instruction is executed, an associated instruction penalty is added to the total instruction counts.

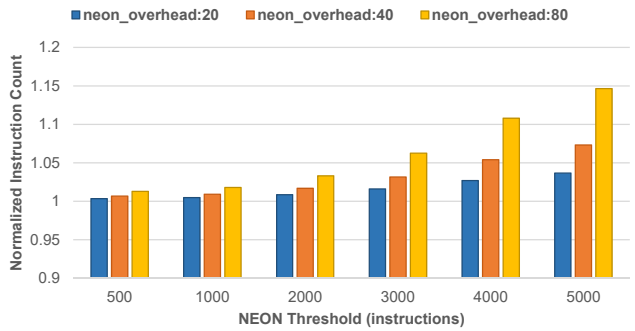


Figure 9: Runlength of SPEC CPU2006 bzip2 with NEON overhead of 20/40/80 instructions.

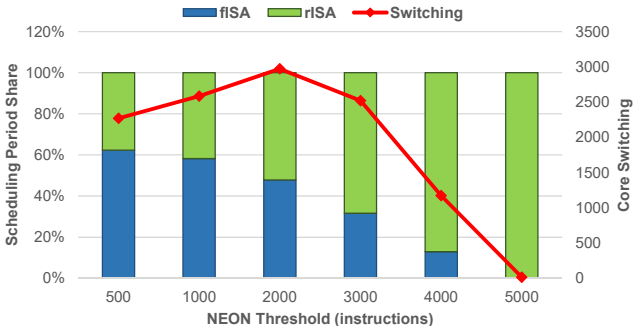
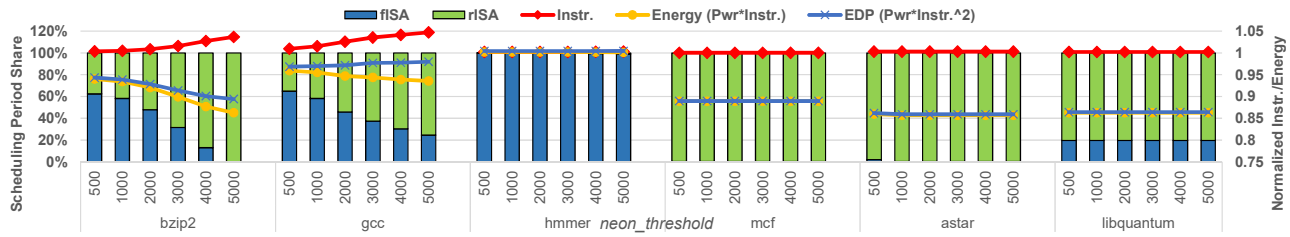


Figure 10: Core residency of SPEC CPU2006 bzip2.

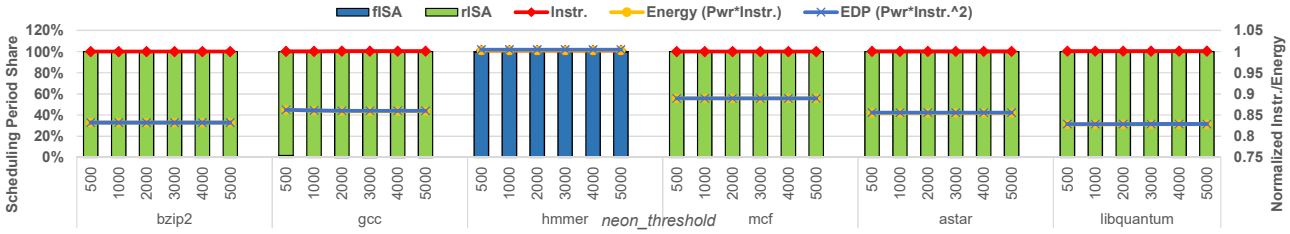
Table 2 shows the detailed parameters used for our evaluation. For optimal scheduling decisions (Algorithm 1), two parameters are important. First, the *neon_threshold* determines the appropriate core for the next scheduling period. To evaluate how it affects energy and performance, we vary the NEON threshold throughout our experiments. By contrast, we observed negligible performance differences as we change the *switch_threshold*. Thus, we fix its value to one. Core switching overhead is estimated from ARM's big-LITTLE scenario, where we account for the amount of task state to be migrated. We observed negligible performance differences under varying switching overheads.

Reducing the emulation penalty is crucial for removing instructions. Ideally, it is desirable for an application to entirely run on the reduced-ISA core. However, we find that the penalty to emulate NEON instructions is critical to performance (Figure 9). We estimate the NEON instruction penalty in increments of 20 instructions corresponding to the maximum 20x performance degradation of FP benchmarks in Section 4. An increase in the penalty per NEON instruction from 20 to 80 directly affects overall performance.

Figure 10 details the fraction of scheduling periods *bzip2* is running on each core and the corresponding number of core switches. The higher the NEON threshold for scheduling, the longer the application can run on the reduced-ISA core. This, however, results in more NEON instructions



(a) SPEC CPU2006 integer benchmarks with NEON instructions. Compiler statically removes LDM/STM, DSP-like, and conditional instructions



(b) SPEC CPU2006 integer benchmarks. Frequently used NEON load/store instructions are replaced to integer load/store instructions

Figure 11: Performance evaluation and energy estimation.

Table 2: Scheduling Parameters

Parameters	Value
Scheduling Unit	1,000,000 instructions
Switch Threshold	1
NEON Instruction Overhead	20/40/80 instructions
Core Switching Overhead	3K instructions

requiring emulation, and, thus, an increase in instruction counts. However, the increase is relatively small. In addition, the number of core switches is low. As such the switching overhead is not crucial for overall performance, and a large switching overhead does not result in significant performance degradation. In other words, lowering the penalty allows more NEON instructions to run on the reduced-ISA core. The performance evaluation of other workloads is discussed in Section 6.2.

6.2 Energy Estimation

Based on results from Sections 6.1 and 5.3, we estimate the energy efficiency of our heterogeneous system. The estimated average power of each benchmark is multiplied by the total number of instructions and the square of total instructions to compute energy and energy delay product (EDP), respectively, assuming a constant CPI per benchmark. Furthermore, the computed energy is normalized to the energy and EDP for the full-ISA core in order to observe how much energy savings our system can achieve. These experiments are conducted with a 20-instruction NEON penalty and a core switching overhead of 3000 instructions.

Figure 11(a) shows the estimated performance and energy for SPEC CPU2006 benchmarks. Among the integer benchmarks, we exclude *omnetpp* and *sjeng*; given their high portion of NEON instructions, it is evident that they will run mostly on the full-ISA core. Since dynamic scheduling determines which core to run on next, other applications show diverse scheduling behavior. *hmmer* runs on the full-ISA core most of the time due to having more NEON instructions than the threshold for each period, resulting in no performance degradation with no energy benefit.

At the opposite end of the spectrum, *mcf*, *astar*, and *libquantum* run most of the time on the reduced-ISA core with negligible performance degradation, but significant energy benefits. Interestingly, *bzip2* and *gcc* show unbiased behavior. In these workloads, due to the broad range of NEON instructions that vary between 500 and 5000 per period, our scheduling algorithm switches cores frequently depending on the NEON threshold. This aggravates the performance loss with higher NEON penalties and thresholds, thus reducing energy efficiency.

6.3 Compiler Optimizations

To optimize energy efficiency, it is better to run applications on the reduced-ISA core as much as possible without losing any performance. This suggests that reducing the amount of NEON instructions is beneficial not only to maximize residency of workloads on reduced-ISA cores, but also to reduce performance degradation. Thus, we further limit the generation of NEON instructions by modifying the LLVM back-end. Since we observe that vector load/store instructions are frequently used in instruction optimizations, blocking such optimizations leads to less NEON instructions. We measure the performance of the two binaries on full-ISA cores on the real board and confirm that performance differences of blocking such optimizations are negligible.

Figure 11(b) shows the performance and estimated energy of the modified binaries. The amount of time *bzip2* and *gcc* run on the full-ISA core is reduced to zero. This leads to a total energy efficiency equivalent to running solely on the reduced-ISA core. *hmmer* still incorporates quite a few NEON instructions, causing it to stay on the full ISA core. Experimental results show that energy savings of up to 15% are achieved for benchmarks that have negligible NEON instructions. On average, about 12% energy savings are achieved across all evaluated benchmarks. Interestingly, with the manipulation of the compiler, core switches are completely avoided, further increasing energy efficiency. This suggests that if a system has control over the compiler, as is the case in Android Dalvik or ART, it is possible to

prefer and only use the limited set of instructions available on the energy-efficient reduced-ISA core.

7. RELATED WORK

Kumar et al. first suggested the possibility of increasing energy efficiency by proposing a single-ISA heterogeneous system [16]. ARM's big.LITTLE architecture [3] subsequently implemented this approach for mobile systems. All of these approaches, however, only focused on the high-performance components in the microarchitecture.

Recently, several notable approaches investigated heterogeneous ISA architectures. Blem et al. revisited the debate about RISC versus CISC, comparing x86 and ARM ISAs [13]. Their research quantifies each ISA and argues that there is not much difference between x86 and ARM ISAs. DeVuyst et al. [14] and Venkat et al. [23] proposed a way of harnessing heterogeneous ISAs. However, their research only focused on the diversities of each ISA and argued the benefits when finding the right ISA depending on the workloads. Several prior works [19, 11] investigated OS and software support for task migration between cores with overlapping ISAs. However, their work does not discuss detailed trade-offs in designing actual architectures for such heterogeneous-ISA systems.

The fundamental idea of having a reduced instruction set is the motivation for RISC processors starting from the 1980's [20]. Simplifying the instruction set in order to obtain the benefit of simplified hardware is the key point. However, it only focuses on single processor architectures.

The ARM ISA inherently possesses heterogeneity in its design [4]. Thumb and ARM ISAs in 16-bit and 32-bit form, respectively, allow for optimal instruction placement targeting either size or performance. This feature is beneficial for the instruction memory footprint both in volatile and non-volatile form. However, it may increase logic requirements since the processor has to handle multiple ISAs even though its internal semantics are similar.

8. CONCLUSIONS

Reducing the instruction set opens up a chance to run high-performance tasks with better efficiency than current systems can provide. By providing heterogeneity in supported instructions and their logic, the energy efficiency of performance-oriented processors can be improved. In this paper, we demonstrated this potential with a case study of a reduced-ISA core and a heterogeneous system that includes both full- and reduced-ISA cores. Our results show that certain complex instructions can be removed with little performance overhead. We argue that in a heterogeneous system that effectively migrates applications to match ISA requirements, significant benefits in energy efficiency can be obtained. By providing heterogeneity in functionality rather than performance as in traditional systems, we can improve energy efficiency with virtually no performance degradation. Results show that our proposed system can improve energy by up to 15% and by 12% on average, all with little to no performance overhead.

9. REFERENCES

[1] ARM Architecture and NEON (page 6). http://xecanson.jp/ARM_PDF/ARM_NEON_STANFORD_lect10arm_soc.pdf.

[2] ARM Architecture Reference Manual ARMv7-A/R Edition. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html.

[3] ARM big.LITTLE Processing. http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[4] ARM Processor Architecture(A32/T32). http://www.arm.com/products/processors/instruction-set-architectures/index.php.

[5] ARMv8 Instruction Set Overview. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.genc010197a/index.html.

[6] Arndale Board. http://www.arndaleboard.org.

[7] Linaro GCC Toolchain 4.7. http://www.linaro.org/downloads/.

[8] Pollack's Rule of Thumb for Microprocessor Performance and Area. http://en.wikipedia.org/wiki/Pollack's_Rule.

[9] Qemu. http://www.qemu.org.

[10] SPEC CPU2006. http://www.spec.org/cpu2006/.

[11] A. Aminot, Y. Lhuillier, A. Castagnetti, and H.-P. Charles. FPU Speedup Estimation for Task Placement Optimization on Asymmetric Multicore Designs. In *Proc of Int. Symp. on Embedded MCSoC*, 2015.

[12] V. K. Balasubramanian, H. Xu, and R. Vemuri. Design automation flow for voltage adaptive optimum granularity LITHE for sequential circuits. In *IEEE 26th International SOC Conference*, 2013.

[13] E. Blem, J. Menon, and K. Sankaralingam. Power Strugles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures. In *Proc. of HPCA*, 2013.

[14] M. DeVuyst, A. Venkat, and D. Tullsen. Execution Migration in a Heterogeneous-ISA Chip Multiprocessor. In *Proc. of ASPLOS*, 2012.

[15] A. Gutierrez, R. G. Dreslinsik, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-System Analysis and Characterization of Interactive Smartphone Applicatoins. In *Proc. of IISWC*, 2011.

[16] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of Micro*, 2003.

[17] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. of CGO*, 2004.

[18] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, and D. M. Tullsen. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc of Micro*, 2009.

[19] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures. In *Proc. of HPCA*, 2010.

[20] D. A. Patterson and D. R. Ditzel. The Case for the Reduced Instruction Set Computer. In *ACM SIGARCH Computer Architecture News*, 1980.

[21] A. Phansalkar, A. Joshi, and L. K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *Proc. of ISCA*, 2007.

[22] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Black, C. D. Emmons, and N. C. Paver. A Structured Approach to the Simulation, Analysis and Characterization of Smartphone Applications. In *Proc. of IISWC*, 2013.

[23] A. Venkat and D. M. Tullsen. Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor. In *Proc. of ISCA*, 2014.