DSPBooster: Offloading Unmodified Mobile Applications to DSPs for Power-performance Optimal Execution

Elliott Wen
The University of Auckland
jwen929@aucklanduni.ac.nz

Jiaxing Shen†
The Hong Kong Polytechnic University
jiaxshen@polyu.edu.hk

2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC) | 978-1-6654-8810-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/COMPSAC54236.2022.00108

***Abstract*—Mobile cloud computing offloads intensive code to remote servers to improve execution performance and battery lifetime. Unfortunately, it is prone to data breaches and dependent on network connectivity. In light of these issues, we explore the potential of an under-utilized local computing resource: Digital Signal Processors (DSPs). Programmable DSPs are widely equipped in mobile devices and can conduct mathematical operations at high speed and low power. However, existing mobile applications rarely offload computation to DSPs due to two reasons. Firstly, conventional DSP development requires high proficiency in low-level programming languages. Secondly, DSP application deployment involves many complex steps such as kernel memory allocation and remote procedure calls. In this paper, we introduce DSPBooster, a framework to facilitate application offloading to DSPs for power-performance optimal execution. DSPBooster supports unmodified applications implemented in various high-level programming languages. It transparently deploys suitable application functions to DSPs based on runtime measurement and prediction. Implementing such a system entails many technical challenges thanks to DSPs' unique micro-architecture and inter-processor communication mechanism. In this paper, we provide workable solutions and a thorough system evaluation. We show that DSPBooster can provide up to 11% performance gain and $3\times$ power reduction.**

I. Introduction

Mobile cloud computing aims to reduce battery consumption and execution time by offloading intensive applications to cloud servers. Unfortunately, this remote computing paradigm is prone to data breaches and highly dependent on network connectivity. In light of these issues, this paper explores the potential of Digital Signal Processors (DSPs), a local computing resource that can be found in nearly all mobile devices.

DSPs can perform mathematical operations at high speed and low power. They are responsible for many essential functionalities of mobile devices such as wireless communication [1] and sensory data processing [2]. Note that early generation DSPs were generally non-programmable and dedicated only to one task. Recent years have witnessed many breakthrough technologies in DSP design such as *Very Long Instruction Word* micro-architecture [3] and System Memory Management Unit [4]. DSPs now can support general-purpose programming languages and handle very versatile workloads.

†Corresponding author

Yet, despite these advances, DSPs are still an under-explored resource in mobile computing. Mobile applications nowadays (excluding those from device vendors) seldom offload computation to DSPs. The reasons are two-folded. Firstly, DSP software development requires a deep knowledge of low-level programming languages (e.g., Assembly). Secondly, DSP application deployment involves many complex procedures such as loading firmware, allocating memory, and issuing remote procedure calls. The high complexity of development and deployment discourages developers from utilizing DSPs. A recent research work [5] proposed an abstraction layer to insulate mobile applications from DSPs-specific details. However, it demands developers to refactor the entire codebase using a particular programming paradigm. This is labor-intensive and would be impossible without access to source code.

In this work, we present DSPBooster, a novel framework to offload mobile applications to DSPs for power-performance optimal execution. DSPBooster supports mobile applications written in various high-level programming languages (e.g., WebAssembly or Kotlin). It transparently deploys suitable application functions to DSPs based on runtime measurement and prediction. This process requires neither access nor modification to the application source code.

Implementing such a system entails many technical challenges. Firstly, before we can offload application code, we first need to translate it into efficient machine instructions understood by DSPs. This is difficult since an application can be implemented in any high-level language and we do not have access to the source code. Secondly, not every function can benefit from offloading. DSPs' unique micro-architecture generally favors tasks with a high level of data parallelism and a high compute-control ratio. Identifying these tasks without manual analysis of source code can be challenging. Thirdly, existing inter-processor communication mechanisms only support function invocation from CPUs to DSPs, but not vice versa. As a result, a considerable number of functions are not directly offloadable since they contain calls to system interfaces running in CPUs.

In this paper, we present practical solutions to cope with the above challenges. We implement a compiler pipeline that can directly ingest high-level language bytecode and generate

978-1-6654-8810-5/22/$31.00 ©2022 IEEE
DOI 10.1109/COMPSAC54236.2022.00108

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.

DSP machine instructions. The pipeline is extensible; new bytecode can be supported by simply transforming it to a language-independent intermediate representation. To identify DSP-friendly tasks, we adopt an reinforcement learning model, which analyses our compiler's internal statistics and predicts the benefits of offloading. The model also accepts the observed performance gain/penalties as a feedback signal to constantly fine-tune itself. To increase the offloading opportunities, we emulate the missing DSP-to-CPU function call mechanism. This is achieved by applying co-routine code transformations on offloaded functions.

We consolidate the above techniques and implement a prototype of DSPBooster on a vanilla Android 10 in a Google Pixel 4 XL device. Our performance evaluation indicates that DSPBooster achieves up to 11% performance speedup and 3× power saving.

The main contributions of this paper can be summarized as follows:

- We propose a framework that allows Android applications to be transparently offloaded into DSPs.
- We present a reinforcement learning model to identify functions that can benefit from DSP offloading.
- We enhance the existing inter-processor communication mechanism to support function calls from DSPs to CPUs.

II. System Overview

Figure 1 demonstrates the main workflow of our framework. The initial stage is called 'profiling', where we start an unmodified application and maintain an invocation counter for each function. If this count exceeds a predefined limit, we mark the function as a potential candidate for DSP offloading. Once a satisfactory amount of profiling information is gathered, our system proceeds into the next phase called 'compilation'. An LLVM-based compiler will take the bytecode of candidate functions as input and emits machine instructions for DSPs. Subsequently, the output binary and the compiler's internal statistics will be fed into a profit prediction component. In this stage, a reinforcement learning model will predict whether it is profitable to offload the function. If yes, the function will be moved to DSPs for execution, otherwise it will continue executing in CPUs. In the meantime, this model can take the runtime performance measurement as a feedback signal to constantly fine-tune itself.

This framework can benefit native or web-based mobile applications implemented in various high-level programming languages such as WebAssembly, Kotlin/Java, Dart and JavaScript. In this paper, we mainly highlight the potential of our framework for WebAssembly applications. Recently, they have been widely adopted by the industry thanks to their high portability and near-native performance [6], [7]. Despite the focus, the following methodology sections are highly generic. Key instructions to adapt our framework to other programming languages are also provided in Section VIII-A. Another thing to note is that our framework is compatible with mobile DSPs from different vendors. In this paper, we use Qualcomm DSPs as the main experimentation platform. Qualcomm is one of the dominant mobile processor producers. In the second quarter of 2021, it leads the smartphone market with 35 percent of revenue share [8].

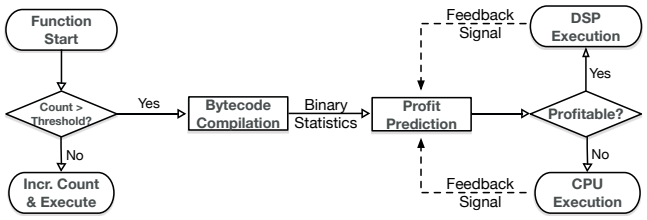

Fig. 1. System workflow

III. Preliminary Profiling

Offloading programs to DSPs requires certain upfront performance costs. The first overhead lies in converting program bytecode to DSP machine code. It is achieved via a compiler with a long optimization pipeline, which is generally memory and computationally intensive. The second overhead derives from CPU-DSP communication latency. CPUs communicate with DSPs using asynchronous interrupt events. A CPU generates an interrupt to a DSP to signal that a task needs to be performed. The DSP also raises an interrupt on the CPU upon task completion. Under ideal conditions where CPU clock rates are set to their maximum and power saving is disabled, the overhead may be as low as 200 microseconds. Unfortunately, these settings are not sustainable for mobile devices due to high battery consumption. In a realistic condition, the CPU clock rates are throttled according to thermal status. CPUs also tend to go into a low-power state while awaiting DSP responses. All these factors increase the round-trip latency up to several milliseconds.

These overheads make us aware that, if a function is rarely executed and each invocation only consumes a limited amount of time, it is not worth offloading the function. Instead, executing it on CPUs may take less energy and fewer CPU cycles. To filter out this type of functions, our framework initially executes applications entirely on CPUs and conducts performance profiling simultaneously. A profiler operates by intercepting calls to functions and injecting additional code to capture performance metrics such as the duration or frequency of a function call. Only if a function's invocation count reaches C or the time to execute a function call exceeds T milliseconds, we consider the function to be 'hot', in other words, a potential candidate for DSP offloading. In this paper, we set the thresholds C and T to be 1000 and 10 respectively, which are heuristic numbers adapted from previous research papers [9], [10]. Noted that our framework only requires fine-grained profiling information in the first few seconds of program startup. Afterward, we can switch our profiler to a low-overhead coarse-grained sampling profiler or just disable profiling. Therefore, performance penalties induced by profiling are nearly negligible.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.

In the context of WebAssembly applications, we start executing them in *Chromium V8*, the default web browser engine in every Android device. V8 provides a built-in profiler, which can collect sophisticated profiling information such as memory footprint and network usage. However, most of the information is unused by our framework and collecting them comes at a price of a significant performance slowdown. To avoid this issue, we implement an in-house lightweight profiler by patching V8's WebAssembly baseline compiler *Liftoff* [11]. Specifically, we generate extra machine codes in each function's prologue and epilogue to collect its invocation duration and count. The information is directly stored in the main memory for post-processing. This technique imposes minor performance penalties to programs because V8 will gradually use its next-tier compiler *Turbofan* to recompile all functions in background [12]. Once it is finished, the executing binary will not contain any profiling instructions and run at full speed.

IV. Bytecode Translation

Once our system gathers sufficient profiling information, it will proceed into the next mission: converting the bytecode of candidate functions into efficient DSP machine code. This is achieved using a three-stage compiler pipeline as shown in Fig 2. In the first stage, the bytecode is translated into an intermediate representation called LLVM IR [13]. The IR is independent of any particular language but still capable of representing the input without loss of information. In the next phase, the LLVM optimizer takes the IR as input, conducts various optimizations, and outputs the efficient IR. Finally, the IR can be dispatched to LLVM backends to generate machine instructions understood by the target DSP architectures.

This pipeline is highly modular and extensible. To support a new programming language, we only need to implement a bytecode translator for the first stage. Specifically, we first use Flex and Bison [14] to generate a lexer and a parser. They then enable us to build abstract syntax trees (ASTs) for the input. By walking the trees, we can visit each bytecode instruction and convert them to LLVM IR. The conversion is generally not difficult since most low-level instructions in the bytecode and LLVM IR are semantically similar. We need to pay special attention to floating-point instructions, which may not be supported by some low-end DSPs and need to be emulated at a considerable performance cost. Even for high-end DSPs, floating-point operations can sometimes degrade performance since they are not likely to be vectorized by compilers. In our implementation, we allow the pipeline to transform floating-point operations into fixed-point operations. Most DSPs possess intrinsic arithmetic instructions for fixed-point data. They are significantly more power-efficient and can be easily vectorized. Sometimes the fixed-point computation may have higher round-off errors. Therefore, we keep the transformation optional for users. Nevertheless, numerical accuracy is not a concern for many real-world applications such as video and audio processing.

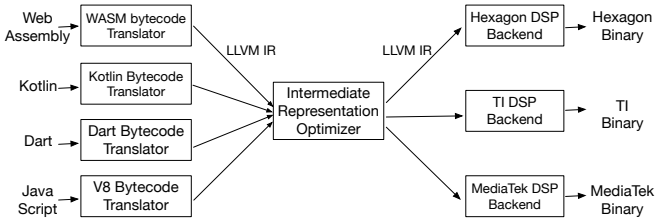

Fig. 2. Three-stage Compiler Pipeline

In the context of WebAssembly, one implementation difficulty is that LLVM IRs are register-based (i.e., operands and results are stored in registers), while WebAssembly instructions are stack-based (i.e., pops operands from and pushes results onto a stack). To conduct the conversion, we need to maintain a stack structure to store the mapping between the WebAssembly operands and LLVM registers. The stack structure is only required during compilation (i.e., no runtime overhead), because WebAssembly's structured control flow allows us to determine operand locations [15] statically. Another tricky issue stems from WebAssembly's linear memory model. Linear memory is a continuous byte-addressable buffer, which spans from address 0 to a mutable amount of memory. In WebAssembly, each memory operation (e.g., 'load' and 'store') expects a linear memory address. For example, a WebAssembly instruction 'i32.load 100' will load a 32-bit integer located in linear memory locations 100-103. This is very different from the memory model of real-world hardware, where applications are randomly allocated in a high virtual memory address space (e.g., 0xffffffff80000000). To bridge the difference, our translator generates necessary instructions to map each WebAssembly memory address to a valid memory address in the OS. In detail, our translator first generates a program initialization routine to obtain a contiguous memory chunk from the underlying OS. The region's address is then recorded in a global variable $linear\_mem$. Afterward, the compiler can offset each WebAssembly memory operation's parameter with the value of $linear\_mem$. For instance, the WebAssembly instruction 'i32.load 100' will be translated into the simplified LLVM IR as shown in Algorithm. 1. We also inject bounds-checking instructions to ensure that all memory operations are well-defined. If out-of-bound access is detected, we can terminate the application immediately to ensure system safety.

Algorithm 1 Transforming a WebAssembly address to a valid OS address

```
1: %1 = load i8*, i8** linear_mem
2: %transformed_addr = getelementptr inbounds i8, i8* %1,
   i64 100
3: call @bounds_check(i8* %transformed_addr)
4: %2 = bitcast i8* %transformed_addr to i32*
5: %result = load i32, i32* %2
```


Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.

V. Profit Prediction

Not every candidate function can benefit from offloading due to potential mismatches between its workload patterns and DSPs' micro-architectures. In detail, modern DSPs adapt a Very Long Instruction Word (VLIW) architecture, which requires a program to explicitly specify multiple instructions (i.e., an instruction group) to execute in each cycle. Any two instructions in the same group are not allowed to have data or control dependency. Usually, the compiler will search for an optimal order of instructions to avoid the dependencies. Sometimes, they are inevitable, especially when scheduling branch control instructions. In that case, the compiler will have to move the offending instruction into a new group and replace the original slot with a NOP instruction. Constant NOP replacement can result in significant performance degradation. Therefore, DSPs generally perform poorly for control-heavy tasks. Instead, DSP-friendly tasks have a high compute-control ratio and a high level of data parallelism.

A. Problem Formulation

Our framework attempts to filter out the ill-suited functions by solving a binary classification problem. Specifically, given a function f, our objective is predicting its likelihood across two labels: beneficial (1) or detrimental (-1). These labels are defined using the following equation:

$$l(f) = \begin{cases} 1 & \text{if } \frac{T_d}{T_c} \leq R \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

Here, T_c and T_d denote a function's execution time on a CPU and a DSP respectively. R is a constant and set to be $\frac{F_c}{F_d}$, where F_c and F_d are clock rates for the CPU and the DSP correspondingly. We can consider R as an approximate energy-efficient ratio, in other words, the DSP is R times more power-efficient than the CPU for the same amount of working time. This is a conservative approximation since DSPs' VLIW design usually provide higher instructions per cycle (IPC) than CPUs. Nevertheless, we ignore the IPC differences for model simplicity. Note that F_c is generally larger than F_d. For example, F_c on a Snapdragon 855 SoC can reach up to 2.8 GHz, while its F_d is only approximately 1.0 GHz. Thus, R is greater than or equal to 1. Intuitively speaking, a function is beneficial if it satisfies one of the following conditions:

1) It takes less execution time on DSPs than on CPUs (i.e., performance-optimal)
2) It consumes less energy on DSPs despite slightly longer execution time (i.e., power-performance optimal).

B. Feature Selection

To solve the above problem, we can build a binary classification model. Before we can proceed, we notice that an input function consists of a sequence of bytecode instructions. Due to its unwieldy high-dimensional spaces, computing on it may require an unreasonable amount of resources. Hence, we must first create low-dimensional representations for the input. Like many other machine learning applications, creating expressive representation is the most challenging step since many domain knowledge factors have to be taken into consideration. In this paper, we address this challenge by exploiting internal statistics from our previous compiler pipeline. The intuition is that many compilers have deep domain knowledge of target processors. They tend to apply as many as hardware-specific transformations on source codes to exploit processor capacities better. If many optimizations are successfully applied, we can infer that the program has a matching workload pattern to the target hardware. In our prototype, we exploit 17 types of compiler statistics. We showcase several potentially expressive features below.

1) **Ratio of the instruction number to the instruction group number**. In a VLIW architecture, the compiler is responsible for bundling instructions to instruction groups. Instructions in the same group can be executed simultaneously in one hardware cycle. Hence, a higher ratio of the instruction number to the instruction group number means better exploitation of instruction-level parallelism in DSPs.
2) **Single Instruction Multiple Data**. DSPs can exploit the Hexagon Vector eXtension (HVX) to accelerate vector operations. Hence, the compiler will attempt to vectorize input functions using various transformations. For example, the SLP vectorizer merges multiple scalar operations into vector instructions, while the Loop vectorizer widens instructions in loops to operate on multiple consecutive iterations. In our prototype, the count of vectorized code segments and the number of HVX instructions are selected as features.
3) **Hardware loop utilization**. DSPs provide hardware loop instructions to perform loop branches with zero overhead. Therefore, the compiler will make the best efforts to transform each software loop into a hardware loop. This transformation will only be carried out if a loop is regular (i.e., countable, not deep-nested, and no function calls inside). Hence, a function with a higher hardware loop count tends to have a high compute-control ratio and can better benefit from DSP execution.
4) **Hardware Threading**. Modern DSPs are multi-threaded. An application's workload can be split and executed in a parallel manner. In our compiler pipeline, we implement an optimization pass to parallelize hot loops automatically. This optimization can only be carried out if a loop is regular and has no dependent iteration. A function with many multi-threaded loops is more likely to benefit from DSP execution.
5) **Peephole optimization**. Peephole optimization is performed on a small sequence of machine codes, which is commonly referred to as peephole or window. It analyses each window and attempts to replace it with shorter and faster code without change in output. For example, LLVM utilizes this optimization to remove redundant sign extends instruction or redundant negation of predicates, which subsequently opens up many dead code eliminations opportunities. Therefore, the usage of peephole

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.

optimization can also serve as a feature.

It should be noted that most selected features are normalized against runtime instruction counts, which can be obtained from the profiler.

C. Online Learning

With the low-dimensional feature vectors, we can proceed to train a classification model. To achieve a good performance, we need to ensure that the training dataset is sufficiently large. Moreover, the dataset should have an unbiased distribution for real-world workload patterns. In practice, preparing such a dataset can be very labor-intensive and complicated. In this paper, we circumvent this issue by exploiting one observation: we can measure the performance benefits or penalties soon after we execute a function on DSPs or CPUs. This measurement can serve as a feedback signal to constantly improve the classification algorithm. Over time, the model will adapt to the real-world data patterns and deliver acceptable performance.

To achieve this, we reformulate our classification problem as a contextual multi-armed bandit problem (CMAB). Specifically, for each round $t \in \{1, ..., T\}$, an agent can observe a multi-dimensional context vector $x_t \in R^d$. It then needs to select an action a_t from a predefined action set $\{1, ..., J\}$. The action will come with a reward r_{t,a_t}, which is unknown until the action is carried out. We will assume that the expected reward is a linear function of the context vector:

$$E[r_{t,a_t}|x_t] = \theta_{a_t}^T x_t, \tag{2}$$

where θ_a is initially unknown but can be gradually learned through action. The goal of our agent is to find a strategy that minimizes the expected regret:

$$R(T) = E\left[\sum_{t=1}^{T}\left(r_{t,a_t^*} - r_{t,a_t}\right)\right], \tag{3}$$

where a_t^* denotes the action with maximum expected payoff at time t.

In the context of our system, the agent needs to select one action between (1) staying at CPUs and (2) offloading to DSPs. The reward for staying at CPUs is 0. When a function f is offloaded to DSPs, the payoff is 3 if it is a beneficial function as explained in Equation 1. Otherwise, a negative payoff of -5 is incurred. Note that the negative payoff governs the trade-off of exploration and exploitation. When the payoff is higher (e.g., -1), the agent is more likely to move unseen functions to DSPs and learn from the observation. In contrast, if the payoff is lower (e.g., -10), the agent may only attempt DSPs offloading if it is confident about getting a positive reward.

This CMAB problem can be solved in an iterative fashion, as shown in Fig 2. The core of the algorithm lies in line 6. The red part estimates the mean reward of each action, while the blue part calculates the upper bound of the confidence interval. Note that α is a hyperparameter and the higher α is, the wider the confidence bound becomes. Intuitively speaking, the algorithm explores actions that we have high uncertainty about while exploiting actions that have superior average returns.

Algorithm 2 Solving CMAB using the LinUCB algorithm [16]

1: $A_1, A_2 \leftarrow \boldsymbol{I}_d$ (d-dimensional identity matrix)
2: $b_1, b_2 \leftarrow \mathbf{0}_{d\times 1}$ (d-dimensional zero vector)
3: **for** $t \in \{1, ..., T\}$ **do**
4: $\quad\theta_1 \leftarrow A_1^{-1} b_1$
5: $\quad\theta_2 \leftarrow A_2^{-1} b_2$
6: $\quad a_t \leftarrow arg\max_{a\in 1,2}\left(\theta_a^T x_t + \alpha\sqrt{x_t^T A_a^{-1} x_t}\right)$
7: $\quad A_{a_t} \leftarrow A_{a_t} + x_t x_t^T$
8: $\quad b_{a_t} \leftarrow b_{a_t} + r_t x_t$
9: **end for**

VI. Execution Offloading.

We now can proceed to the last phase of our framework: execution offloading. On the surface, this may seem to be a straightforward mission because Qualcomm already provides a mechanism called Fast Remote Procedure Call (FastRPC) to enable function calls from CPUs to DSPs. FastRPC features a typical proxy pattern as demonstrated in Fig 3.

1) The CPU process initiates the DSP function invocation using an auto-generated stub.
2) The stub packs the function parameters into an RPC message and sends it to the DSP RPC driver (*/dev/cdsprpc-smd*) using the system call *ioctl*.
3) The kernel driver forwards the RPC message to the DSPs through the Shared Memory Driver (SMD) channel and then waits for the response.
4) The real-time OS running in DSPs dispatches the message to an auto-generated skeleton library for processing.
5) The skeleton unmarshals parameters and calls the target method implementation, i.e., the generated machine code from Section IV.
6) Once the target method is finished, the reply traces the same steps in the reverse direction.

These steps are designed to be synchronous to eliminate the complexity of application implementation. From the application's perspective, a DSP function invocation looks identical to a local call.

However, despite the FastRPC's simplicity, a technical challenge occurs when an offloaded DSP function attempts to invoke a function residing in a CPU. This kind of function call is not yet supported by FastRPC and could happen in two common scenarios. The first situation is that the target function is not sitting on a hot code path. Take Algorithm 3 as an example. The function *foo* is offloaded and consists of a branching instruction. Since the branching condition is satisfied 99% of the time, the function *bar* is therefore executed very often and likely to be offloaded as well. In contrast, the function *baz* is seldom executed and thus not considered for offloading. The second situation is that the target function is a language runtime function (e.g., object allocation in a managed heap) or a system interface function

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.

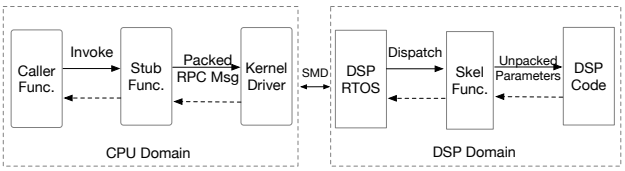

Fig. 3. Function calls from CPUs to DSPs using FastRPC.

(e.g., filesystem access). These functions reside in kernel space and should not be moved out of CPUs for security reasons.

Algorithm 3 An example for DSP to CPU invocation

```
1: function FOO                          ▷ Offloaded
2:     if condition then                 ▷ satisfied in 99% of the time
3:         call bar                      ▷ Hot path and offloaded.
4:     else
5:         call baz                      ▷ Cold path and not offloaded.
6:     end if
7:     ...
8: end function
```

A naive approach to circumvent this challenge is to avoid offloading functions that contain calls to non-offloadable functions. Specifically, we first build a directed graph where each node is a method and each edge is a method call directed from the caller method (parent node) towards the callee method (child node). We then scan each method and mark it as non-offloadable if it contains system calls. Starting from them, we iterate through their parents until we reach the root and set the scanned nodes as non-offloadable. Meanwhile, to prevent the cold path issue in Algorithm 3, when a function is offloaded, all its descendants are forcibly offloaded as well. Unfortunately, this approach incurs considerable CPU overheads. Furthermore, this approach may not maximize the offloading opportunities: a function with any system calls is never offloaded, even if a significant proportion of the function body can benefit from the DSP execution.

In this paper, we present an alternative approach: we implement the missing DSP-to-CPU function call mechanism based on the FastRPC framework. The main workflow is demonstrated in Fig 4. As usual, a CPU utilizes FastRPC to invoke a target DSP function. But this time, the target function is pre-processed by a code transformation called *Coroutine Transformation*. Every invocation to a non-offloadable function is now transformed into a suspend point. When the suspend point is reached, the DSP execution is suspended, and control is returned to the CPU along with a snapshot of stack frames. This snapshot is commonly referred to as *coroutine context*. From the CPU's perspective, this operation is just an ordinary FastRPC function call return. Afterward, the CPU can execute the non-offloadble function on behalf of the DSP function. Once finished, the CPU will again use FastRPC to call the suspended DSP function but with the coroutine context as an additional parameter. This will resume the DSP function

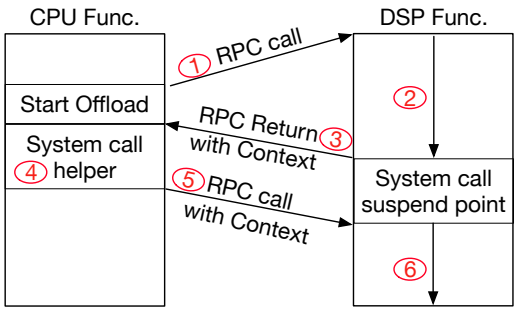

Fig. 4. Flow of execution for a DSP function with suspend points

from the last suspend point.

The key component of this architecture is the coroutine transformation. Behind the scene, it splits a function into a 'ramp function' and an arbitrary number of continuation functions, one for each suspend point. The ramp function serves as an initial entry point and executes until a suspend point is first reached. It then returns a continuation function pointer to indicate where to resume the execution. It is worth mentioning that this code transformation comes at the cost of larger binary sizes and memory footprint. This is mainly attributed to the fact that every continuation function now has a sophisticated prologue and epilogue to recover and generate the coroutine context. Hence, we should avoid offloading a DSP function with many suspend points. To achieve that automatically, we make the number of suspend points as an input feature for our adaptive machine learning model in Section V.

VII. System Evaluation

In this section, we provide a system evaluation of DSP-Booster. We are interested in one key question: how efficient are DSP-offloaded mobile applications in terms of execution time and power consumption? We use the following experiment setups to answer this question.

A. Test Bed Configurations

Hardware. We conduct our experiments on an off-the-shelf Google Pixel 4 smartphone. The device is equipped with a Qualcomm SM8150 Snapdragon 855 chipset, which contains an 8-core 2.8 GHz Kryo 485 CPU and 6 GB of RAM. The chipset also possesses four different types of DSPs, each devoted to a specific application space: sensor (sDSP), modem (mDSP), audio (aDSP), and compute (cDSP). In this paper, we mainly utilize the compute DSP for our experiments. This choice is made to comply with the security policies in DSPs. Specifically, Qualcomm incorporates a proprietary real-time OS named *QuRT* to manage the DSPs. Every executable binary wishing to run on that OS needs to be appropriately signed using a developer certificate, to which only a small number of mobile vendors have access. Our implementation addresses this issue by exploiting a newly introduced sandbox mode called the unsigned protection domain (PD). Unsigned

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.

TABLE I
BENCHMARK PROGRAMS

ID	Benchmark	Description
P1	N-body	Model the orbits of 5000000 particles using a symplectic-integrator.
P2	Merge-sort	Sorts an array of 1000000 integer elements using the merge sort algorithm.
P3	Spectral-norm	Calculate the spectral norm of an 5500×5500 infinite matrix.
P4	Regex	Locate sub-strings from 5000000 string using regular expressions.
P5	Fasta	Genome sequence similarity searching on 2500000 sequences.
P6	Complement	Compute the reverse complement of a DNA sequence with 100000000 genomes.
P7	Integral	Calculates the integral of an image with a resolution of 7680×4320.
P8	Thresholding	Apply the adaptive-level thresholding to each image pixel and transforms it into a binary value.
P9	Colorspace	Convert an image from one color space to another (e.g., HSV to RGB).
P10	Convolution	Convolve a 3×3 kernel with an image. Convolution is the key operation of digital image processing.
P11	MP3 Decoding	Decode a MP3 audio file and output pulse code modulation.
P12	H264 Decoding	Decode H.264 video bitstream and output bitmap for each frame.

PD allows for the execution of signature-free binaries with limited access to underlying hardware resources (e.g., cameras or microphones). This restriction does not impact our framework since we only offload general-purpose computation to DSPs. To enable the unsigned protected domain feature, we need to insert extra system call sequences into stub functions according to the Hexagon DSP documentation [17]. Currently, unsigned PD is only available in Compute DSPs of particular high-tier SoCs. Nevertheless, the support is likely to be extended to all types of SoC shortly, as suggested by Qualcomm.

Software. Our Google Pixel 4 mobile device is running a vanilla Android 11 operating system. To facilitate system implementation and debugging, we obtain the root access by unlocking the bootloader[†] and patching the boot image partition. We also temporarily switch the Security-Enhanced Linux (SELinux) mode from *enforcing* to *permissive*. This step is solely intended for Google Pixel devices since they enforce stringent SELinux policies and disallow third-party applications to access DSP hardware.

B. Performance Benchmarking

We benchmark the programs listed in Table I. Specifically, P1 to P6 are adapted from established CPU benchmark suites such as JetStream [18] and BenchmarkGames [19]. P7 to P10 derive from the open-source computer vision library OpenCV [20]. P11 and P12 are adapted from the open-source multimedia processing library FFmpeg [21].

[†]This can be achieved by accessing on-device developer options in Android.

These programs are written in portable high-level languages (Rust or C++). They consist of an entry-point function and a set of kernel functions for core computational logic. The kernels are optimized to harness the SIMD capability of modern CPUs. For example, they may arrange their internal data structures in a particular alignment such that the compilers' auto-vectorization analysis can automatically convert scalar code into vector code. They can also utilize some C language directives (e.g., 'OMP parallel') to explicitly instruct the compiler to vectorize the chosen code block.

We compile these programs to WebAssembly bytecode using corresponding language toolchains (e.g., Emscripten or RustC). Afterward, we run these executables in V8 WebAssembly runtime 50 times with and without our proposed framework. We measure their average execution time (i.e., T_d and T_c) and battery percentage consumption (i.e., P_d and P_c). To eliminate the potential influence of disk I/O, we preload all input files to the main memory. To obtain more accurate power consumption results, we turn the phone into airplane mode, reduce the backlight brightness to 10%, and shut down other background activities. We also cool down the phones before each test to ensure that the CPUs and DSPs can keep working at a stable frequency during the experiment.

Figure 5 demonstrates the relative execution time T_d/T_c and power consumption P_d/P_c. What stands out are P4 and P7. They run up to 11% faster in DSPs than in CPUs while achieving almost $3\times$ energy reduction. Similarly, P2, P9, and P10 deliver a relative execution time slightly above 1 (i.e., achieve comparable performance in DSPs and CPUs) and conserve 50% battery on average. We pay special attention to P10 since Hexagon DSP SDK provides an alternative implementation in a low-level assembly language. Our preliminary experiment shows that it performs approximately $1.7\times$ faster than our benchmark. This result is not surprising since the assembly version evenly splits the application workload and fully leverages the dynamic hardware threading capacity. In contrast, our compiler pipeline can only parallelize partial workload (i.e., loops without dependent iterations) in the benchmark program. A further refinement of the compiler pipeline is warranted.

We use a horizontal line to reference the beneficial criterion R as discussed in Section V. It can be observed that P1 and P12 are both above the reference line, indicating they may not benefit from DSP offloading due to a potential mismatch between workload pattern and DSPs' micro-architecture. Specifically, P1 runs about $4\times$ slower in DSPs than in CPUs and consumes 36% more energy. P12 performs approximately $3\times$ slower and only reduces 2% power consumption. Note that our profit predictor successfully identifies these two benchmarks as non-beneficial, but we forcibly offload them to DSPs just for performance evaluation. We conduct a preliminary investigation on the two programs and notice that their generated DSP binaries contain limited SIMD instructions. One potential culprit is that their kernel functions contain irregular nested loops, which cannot be automatically unrolled and vectorized by compilers. Another interesting observation is that P12,

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.

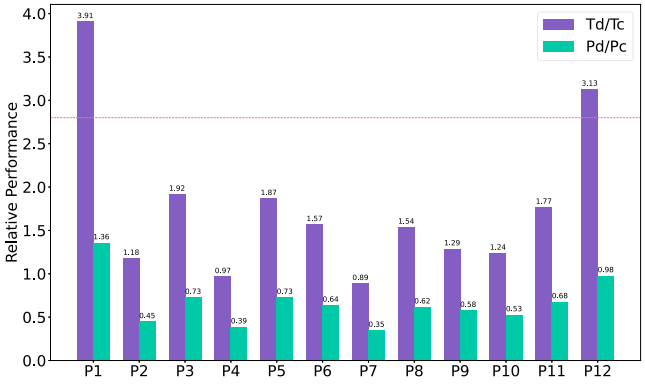

Fig. 5. Comparison of Execution Speed and Power Consumption

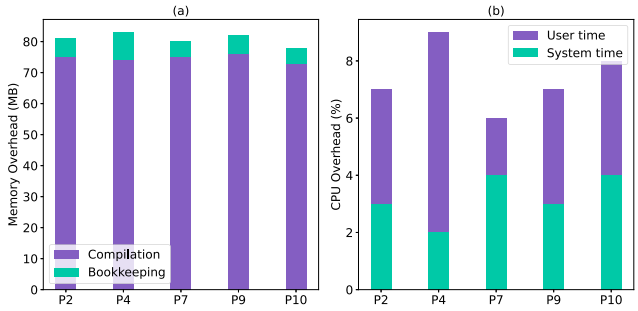

Fig. 6. Memory overhead and CPU overhead

despite being classified as non-beneficial, still manages to conserve a small amount of battery. This indicates that our choice of the beneficial criterion R (i.e., the ratio of CPU frequency to DSP frequency) is conservative.

C. System Overhead

Memory Overhead. Our system requires extra memory space for DSP binary compilation and bookkeeping. In this experiment, we select five benchmark programs from the previous experiment and report their memory overhead in Fig 6 (a). It can be seen that the average memory overhead is approximately 80 MB. This overhead is almost negligible in modern smart devices considering they usually possess gigabytes of memory space. We further examine the programs' memory map information using the *pmap* command. We observe that the compiler component accounts for most of the memory overhead (74 MB). This finding remains somewhat consistent across various programs. Meanwhile, we realize that the remaining overhead lies in storing the compiled binaries and increases proportionally to the number of offloaded functions. To estimate this overhead in a complex program, we duplicate the kennel function in the benchmark program P8 for 10000 times and compile them. As expected, the overhead now reaches 192 MB, which is still moderate.

CPU Overhead. Our system consumes extra CPU resources in several intermediate procedures such as bytecode compilation and profit prediction. In this experiment, we reuse the five benchmark programs above and report the CPU overhead. We conduct the measurement with the help of the *time* command. This command provides three performance metrics for a given task, including elapsed time (i.e., how long the task takes to run), user time (i.e., CPU time spent in user mode), and kernel time (i.e., CPU time spent in kernel mode). For typical CPU programs, the elapsed time approximately equals the sum of user time and system time. However, this is not the case for our benchmark programs since their core computational logic is now done in DSPs. As a result, CPUs will mostly stay at a sleep state and only wake up shortly for the intermediate tasks mentioned above. In other words, the sum of user time and system time now reflects the CPU overhead of our framework. Figure. 6 (b) demonstrates the user time and system time as a percentage of the elapsed time for all the benchmark programs. The average value is merely 7%, indicating that our framework overhead barely impacts the overall system performance.

VIII. System Extensions

DSPBooster is still a research prototype and has room for improvement. In this section, we suggest several potential research directions.

A. Accelerating Different Types of Applications.

In this paper, we demonstrate the potential of DSPs mainly using WebAssembly-based mobile applications. Nevertheless, our methodology can be easily adapted to mobile applications written in various programming languages. We can reuse most system components (e.g., performance profiler, profit predictor, and function offloader) and only need to implement corresponding bytecode translators.

Kotlin: Kotlin is a modern statically typed programming language. It can fully interoperate with Java bytecode and provides a more concise syntax using type inference. Recently, Kotlin has become Google's preferred programming language for developing native Android applications. To implement a translator for Kotlin, we adapt a compiler component from the Android Runtime [22], which provides support for parsing Java bytecode and emitting LLVM IR. We then can directly feed the resulting IR into LLVM backends to generate DSP machine code. Currently, we are still addressing one performance issue. Specifically, a considerable number of Java functions depend on runtime function calls (mainly for managed heap object manipulation). As we have discussed in Section VI, these functions would require sophisticated coroutine transformations. As a result, the overall system performance is negatively impacted.

Dart: Dart is another Google's recommended programming language for native Android applications. Dart supports many modern high-level language features such as object-oriented programing paradigms and automatic garbage collection. To achieve native performance, Dart directly compiles to machine code. Our preliminary experiment found it tricky to translate

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.

the machine code to LLVM IR since it is an open research issue called static binary translation. We need to overcome several technical challenges such as code discovery (i.e., distinguishing data from instructions) and indirect branching (i.e., the jump destination address must be mapped to an address in the translated code). Currently, we are still experimenting with two potential workarounds. Firstly, we are planning to enhance the Dart compiler to embed Dart IR as an auxiliary section in output files. In this case, we only need to implement a Dart IR to LLVM IR translator. Alternatively, we can request the Dart compiler to generate a WebAssembly binary, which can be fed into our WebAssembly bytecode translator. However, this WebAssembly generation feature is still highly experimental and may fail to compile certain source code.

JavaScript: JavaScript is frequently used in Web-based mobile application development. However, JavaScript is generally too slow for compute-intensive tasks [23]. It is mainly used in low-compute scenarios, especially user interface design. As a result, the benefit of DSP offloading may be reduced. Meanwhile, building a bytecode compiler for JavaScript is very challenging due to a lack of typing information. Common runtimes (e.g., V8) essentially bypass this challenge by using interpreters to execute most JavaScript functions. They only compile frequently executed functions using a technique called *Speculative Compilation*. The main idea is to use profiling to infer types dynamically. With the information, the runtimes can generate a statically typed IR (e.g., TurboFan IR in V8) of the dynamically typed program. The generated code needs to place guarding conditions to validate whether the runtime types match the profiling types. If not, it has to throw out the machine code and fall back to interpreter execution. In our prototype, we utilize the result of speculative compilation and implement a translator to convert Turbo Fan IR into LLVM IR. The resulting IR can then be used to generate DSP machine code. Currently, the translator can only process a small subset of Turbo Fan IR. More implementation effort is warranted.

B. Experimenting with Different Types of DSPs

Many premium-tier devices (e.g., Pixel 4XL and XiaoMi 9) can possess different types of DSPs. As we have mentioned in Section VII-A, we only conducted experiments on compute DSPs to comply with the signature verification rule. Recently, we discovered a workaround; if we disable the secure boot mechanism[†], we then can sign executables using a debug certificate to bypass the verification. This trick allows us to explore the potential of other types of DSPs.

Our preliminary study reveals that compute DSPs possess a more advanced MMU compared with other types of DSPs. This leads to differences in the amount of memory data we can share between CPUs and DSPs. Specifically, compute DSPs can access virtually contiguous memory pages, which are not necessarily physically contiguous. Such memory can be easily obtained with the help of the Android generalized memory manager (i.e., ION [24]). Similar to the dynamic memory allocation 'malloc' in C, the allocation will always succeed if there is sufficient physical memory. In contrast, non-compute DSPs demand the memory pages to be physically contiguous. Such memory can only be allocated using the system call 'kzalloc'. When requesting a sizable chunk of memory (e.g., more than 4 MB), this system call is prone to failure due to the memory defragmentation issue [25]. As a result, functions running in non-compute DSPs can only share a limited amount of memory data with CPUs. This limitation significantly lowers the number of functions offloadable to non-compute DSPs. To identify these offloadable functions, we need to adopt a static code analysis technique called 'Pointer Analysis'. It is designed to examine which pointers can point to which variables or storage locations. A function is offloadable to non-compute DSPs only if all pointers insides refer to the small sharable memory or local stack variables.

[†]Secure boot ensures the integrity of firmware and software running on a platform. Disabling secure boot is only possible in a Qualcomm development kit.

C. Tracing Offloading

By default, DSPBooster compiles and offloads code at the function level. Alternatively, DSPBooster can be tuned to work at the tracing level. In this setting, DSPBooster can identify the frequently executed parts inside a function and selectively compiles those structures for the balance of compilation time and memory usage. However, this also significantly increases the implementation complexity of our compiler pipeline. For example, our compiler now has to provide an on-stack replacement (OSR) mechanism, which allows a running function to transfer control to the newly compiled DSP code using the same stack frame. This can be challenging as CPUs and DSPs arrange stack contents in a different layout. We plan to keep improving the stability of tracing offloading.

IX. Related Work

Smartphone vendors have devoted significant research efforts to DSPs. They initially adopted DSPs in the modem, a wireless communication component that continuously processes analog radio signals in real-time. These operations would run notably slower and drain far more energy on CPUs [26], [1], [27]. Later, mobile manufacturers have found DSPs' use in sensory data processing [28], [29], [30]. A typical example is wake-up words for virtual assistants (e.g., *Hey Google* or *Hi Siri*). To implement this functionality, a device needs to monitor a microphone's input continuously. For this task, DSPs are shown to be approximately ten times more energy-efficient than CPUs [2]. Recently, edge artificial intelligence (edge AI) is becoming a hot research topic. A considerable amount of AI computation can now potentially be offloaded from cloud data centers to mobile devices. To prepare for this transition, several mobile manufacturers have proposed AI compute engines specifically optimized for DSPs [31], [32], [33]. Despite the applications outlined above, DSPs are still an under-utilized resource in mobile devices. The high complexity in DSP development and deployment deters developers from utilizing DSPs. Recently, one research

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.

work [5] proposed abstraction layers to conceals most working details of DSPs. However, they demand developers to follow a particular programming paradigm and use a specific set of application program interfaces. As a result, many existing applications have to be refactored, which may be impractical without access to source code.

X. Conclusion

This paper presents DSPBooster, a framework that offloads mobile applications to DSPs to pursue higher execution speed and lower power consumption. DSPBooster can offload applications written in various high-level programming languages. It transparently selects suitable application functions to offload using a reinforcement learning model. DSPBooster requires neither access nor modification to application source code. We conduct a performance evaluation of our prototype. Our experiment results show that DSPBooster achieves up to 11% performance speedup and $3\times$ power saving.

References

[1] H. Yan, S. Zhou, Z. J. Shi, and B. Li, "A dsp implementation of ofdm acoustic modem," in *Proceedings of the second workshop on Underwater networks*, 2007, pp. 89–92.

[2] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "Dsp. ear: Leveraging co-processor support for continuous audio sensing on smartphones," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, 2014, pp. 295–309.

[3] J. A. Fisher, "Very long instruction word architectures and the eli-512," in *Proceedings of the 10th annual international symposium on Computer architecture*, 1983, pp. 140–150.

[4] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, "Hexagon dsp: An architecture optimized for mobile multimedia and communications," *IEEE Micro*, vol. 34, no. 2, pp. 34–43, 2014.

[5] C. Hsieh, A. A. Sani, and N. Dutt, "Surf: Self-aware unified runtime framework for parallel programs on heterogeneous mobile architectures," in *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2019, pp. 136–141.

[6] T. Zohud and S. Zein, "Cross-platform mobile app development in industry: A multiple case-study," *International Journal of Computing*, pp. 46–54, 2021.

[7] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021*, 2021, pp. 2696–2708.

[8] "Global smartphone application processor (ap) market share: By quarter." [Online]. Available: https://www.counterpointresearch.com/global-smartphone-ap-market-share/

[9] P. A. Kulkarni, "Jit compilation policy for modern machines," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 773–788.

[10] X. Xu, K. Cooper, J. Brock, Y. Zhang, and H. Ye, "Sharejit: Jit code cache sharing across processes and its practical implementation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–23, 2018.

[11] "Liftoff: a new baseline compiler for webassembly in v8." [Online]. Available: https://v8.dev/blog/liftoff

[12] "Turbofan: V8's optimizing compiler." [Online]. Available: https://v8.dev/docs/turbofan

[13] 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.* IEEE, 2004, pp. 75–86.

[14] J. Levine, *Flex & Bison: Text Processing Tools.* " O'Reilly Media, Inc.", 2009.

[15] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 185–200.

[16] W. Chu, L. Li, L. Reyzin, and R. Schapire, "Contextual bandits with linear payoff functions," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 2011, pp. 208–214.

[17] "Hexagon dsp documentation." [Online]. Available: https://developer.qualcomm.com/software/hexagon-dsp-sdk/tools

[18] "Jetstream browser benchmark suite." [Online]. Available: https://browserbench.org/JetStream/

[19] "The computer language benchmarks game." [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[20] G. Bradski and A. Kaehler, "Opencv," *Dr. Dobb's journal of software tools*, vol. 3, 2000.

[21] "Ffmpeg: A complete, cross-platform solution to record, convert and stream audio and video." [Online]. Available: https://www.ffmpeg.org/

[22] "Android runtime (art) and dalvik." [Online]. Available: https://bit.ly/3zhs2vH

[23] S. Amatya and A. Kurti, "Cross-platform mobile development: challenges and opportunities," in *International Conference on ICT Innovations*. Springer, 2013, pp. 219–229.

[24] H. Zhang, D. She, and Z. Qian, "Android ion hazard: The curse of customizable memory management system," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1663–1674.

[25] B. Brömstrup and A. Koglin, "Memory subsystem and data types in the linux kernel," 2015.

[26] X. Liu, H. Zeng, N. Chand, and F. Effenberger, "Efficient mobile fronthaul via dsp-based channel aggregation," *Journal of Lightwave Technology*, vol. 34, no. 6, pp. 1556–1564, 2015.

[27] Y.-W. Chen, S. Shen, Q. Zhou, S. Yao, R. Zhang, S. Omar, and G.-K. Chang, "A reliable ofdm-based mmw mobile fronthaul with dsp-aided sub-band spreading and time-confined windowing," *Journal of Lightwave Technology*, vol. 37, no. 13, pp. 3236–3243, 2019.

[28] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, 2016, pp. 320–333.

[29] J.-M. Valin, "A hybrid dsp/deep learning approach to real-time full-band speech enhancement," in *2018 IEEE 20th international workshop on multimedia signal processing (MMSP)*. IEEE, 2018, pp. 1–5.

[30] R. Chen, Y. Chen, L. Pei, W. Chen, J. Liu, H. Kuusniemi, H. Leppäkoski, and J. Takala, "A dsp-based multi-sensor multi-network positioning platform," in *proceedings of the 22nd international technical meeting of the satellite division of the Institute of Navigation (ION GNSS 2009)*, 2009, pp. 615–621.

[31] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 2016, pp. 1–12.

[32] C. Yang, S. Chen, J. Zhang, Z. Lv, and Z. Wang, "A novel dsp architecture for scientific computing and deep learning," *IEEE Access*, vol. 7, pp. 36 413–36 425, 2019.

[33] S. Jagannathan, M. Mody, and M. Mathew, "Optimizing convolutional neural network on dsp," in *2016 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2016, pp. 371–372.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 07,2022 at 02:43:53 UTC from IEEE Xplore. Restrictions apply.