HARDROID: Transparent Integration of Crypto Accelerators in Android

Luca Piccolboni, Giuseppe Di Guglielmo, Simha Sethumadhavan, Luca P. Carloni
Department of Computer Science, Columbia University, New York, NY, USA
Emails: {piccolboni, giuseppe, simha, luca}@cs.columbia.edu

***Abstract*—Accelerators have become fundamental building blocks of any modern architecture. Accelerators are often deployed on a platform by evaluating performance and energy consumption, while assuming that the software applications can be modified to invoke the accelerators. In some contexts, however, this is impractical. For instance, in an Android-based platform changing the applications to invoke an accelerator can affect their portability. We present *Hardroid*, a heterogeneous platform that allows an Android application to offload tasks to loosely-coupled accelerators on an FPGA in a transparent way, i.e., without modifying the code of the application. To demonstrate the *Hardroid* capabilities, we design four accelerators for cryptography with high-level synthesis (HLS) and we compare their efficiency with two libraries for cryptography, by executing 29 Android applications. While we use FPGAs to implement and evaluate *Hardroid*, our accelerators are designed so that they can be integrated in a system-on-chip (SoC) and we report their energy efficiency also for an ASIC implementation. The experimental results show that *Hardroid* is an effective platform that can be used to evaluate the costs and benefits of integrating accelerators, when these are called by real-world Android applications. We show that invoking accelerators without modifying the code of the applications can affect the energy efficiency of the accelerators.**

***Index Terms*—accelerators, cryptography, embedded systems, high-level synthesis, system-level design, reconfigurable architectures.**

I. Introduction

Modern architectures combine general-purpose CPUs with domain-specific accelerators [11], [19]. Accelerators are hardware computing engines that deliver energy-efficient and high-performance computations for specific tasks within a certain application domain. The adoption of specialized accelerators has been quickly rising in the last decade in several application domains, including graph analytics [27], machine learning [13], database processing [56], brain-computer interfaces [22], genome sequencing [24], video decoding [35], cryptography [6], and many more. This has encouraged the development of heterogeneous platforms that make the accelerators critical components of their architecture, e.g., [5], [21], [33], [37], [46]. For example, the NVIDIA Deep Learning Accelerator (NVDLA), a loosely-coupled accelerator for deep learning [44], has been integrated in various platforms [2], [25]. NVDLA is invoked by a processor with a device driver and it performs its task autonomously, freeing up cycles from the processor execution. In addition to loosely-coupled accelerators, some platforms use tightly-coupled accelerators that are located in the processor and called with custom instructions [5]. In this paper, we focus primarily on loosely-coupled accelerators [14], [16].

The rise of accelerators has been supported by high-level synthesis (HLS) [40], which makes accelerator design accessible to domain experts with little knowledge of hardware. Cycle-accurate specifications at the register-transfer level (RTL) are replaced by untimed specifications in C or SystemC, which are automatically synthesized into corresponding RTL specifications. Hence HLS greatly simplifies the task of accelerator design by abstracting away most of the low-level circuit details that burden the development process [35], [47]. The integration of an accelerator into a platform, however, remains in large part responsibility of the designer, who has to write the software to invoke the accelerator and evaluate performance and programmability. The simplification of accelerator integration through the abstraction of architectural details has been investigated for years [51], [52], [55].

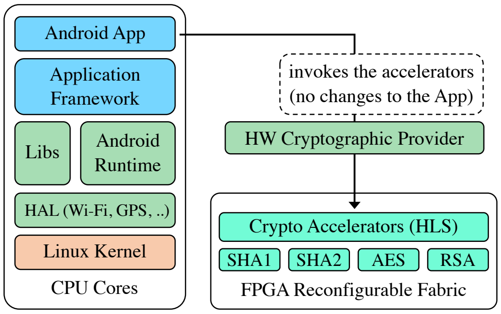

Android App
Application Framework
Libs
Android Runtime
HAL (Wi-Fi, GPS, ..)
Linux Kernel
CPU Cores
invokes the accelerators (no changes to the App)
HW Cryptographic Provider
Crypto Accelerators (HLS)
SHA1
SHA2
AES
RSA
FPGA Reconfigurable Fabric

Fig. 1: *Hardroid*, an FPGA-based heterogeneous platform.

The integration of a loosely-coupled accelerator into a platform typically relies on the assumption that the target software applications can be modified. This requires the replacement of the function call that is computed by the accelerator with an invocation to the driver of the accelerator. In turn, this may require to allocate the memory differently such that it is accessible to both the application and the accelerator. With NVDLA [44], for example, a contiguous region of memory is necessary to move the data from the application to the accelerator and vice versa. Assuming that the software applications can be changed is realistic in many contexts. This is the case, for instance, of non-legacy software invoking accelerators in a system-on-chip (SoC). In other situations, however, it is preferable to avoid modifying the applications when an accelerator is integrated. Android is an example as there is a huge number of applications that we should not modify to preserve their portability across different platforms.

The goal of this paper is to investigate the problem of integrating accelerators in Android-based systems under the assumption that the applications cannot be modified to explicitly call the accelerators. We discuss the extent to which this is possible, the requirements it entails, and its implications on performance and energy consumption. Other works, instead, have focused on enabling Android applications to invoke accelerators directly, e.g., by means of IPC mechanisms [54] or user-level drivers [17]. In pursuing our goal, we focus on cryptographic (crypto) accelerators given their potential for a massive use across Android applications.

To evaluate the integration of accelerators in Android, we need a platform where we can easily combine general-purpose processors and hardware accelerators. Most of the current heterogeneous computing platforms, e.g., [2], [5], [21], [33], [37], [46], do not support Android. A few, however, support Android, e.g., [3], [17], [54], [59] (see Section VI). Among them, we chose Mentor Embedded Android that is an open-source porting of Android 8.1 for Xilinx FPGAs [3]. We extended it by providing support for invoking accelerators from Android applications through a transparent software layer. We developed *Hardroid*, an FPGA-based platform that enables the invocation of loosely-coupled accelerators from Android applications (Fig. 1). *Hardroid* uses a general-purpose processor to run the Android operating system and four accelerators for crypto that we designed and implemented with HLS. *Hardroid* utilizes a Java software layer (*HW Cryptographic Provider* in Fig. 1) to allow the applications to invoke the accelerators in a transparent way, i.e., without requiring changes to the code. We make the following contributions:

(1) We design four accelerators for crypto: *SHA1*, *SHA2*, *AES*, and *RSA*. We synthesized them with commercial HLS tools by targeting both FPGA and ASIC technologies;
(2) We design *Hardroid*, an FPGA-based heterogeneous platform for supporting loosely-coupled accelerators in Android;
(3) We develop the *Hardware Cryptographic Provider*, a library that allows Android applications to call accelerators for crypto without requiring modifications to the code of the applications;
(4) We compare the energy efficiency of our accelerators against two libraries for crypto used in Android: BouncyCastle [9] and AndroidOpenSSL (conscrypt) [15], the Google's optimized porting of OpenSSL [45] for Android systems;
(5) We show that invoking crypto accelerators without changing the applications can affect performance and energy efficiency, because of software overheads for the invocations;
(6) We run 29 Android applications and report results on performance and energy efficiency of the accelerators.

II. Preliminaries

Three critical crypto operations that are usually implemented in all systems are: (1) hashing, (2) symmetric encryption, and (3) asymmetric encryption [32], [38]. We designed accelerators for these operations.

Crypto hash functions take as input an arbitrary amount of data (e.g., a document, an image, etc.) and produce hash values, called message digests [42]. Crypto hash functions are often used to verify data integrity, for instance to check if a document we received has been tampered during its transmission. Some well-known families of hash functions are *SHA1* and *SHA2*. *SHA1* produces digests of 20 bytes, while *SHA2* consists of *SHA224*, *SHA256*, *SHA384*, *SHA512*, *SHA512/224*, and *SHA512/256*, with digests from 28 to 64 bytes.

Symmetric encryption makes data unintelligible. A block cipher, e.g., *AES* [41], takes as input a block of data and a key and it produces the encrypted or decrypted output block. A decrypted block is called plaintext, while an encrypted block is called ciphertext. A padding scheme must be applied when the data size is not a multiple of the block size. Block ciphers support operation modes, which determine how multiple blocks are encrypted or decrypted. Electronic codebook (ECB) encrypts and decrypts each block independently. Counter mode (CTR) uses a counter that must be incremented at each encryption or decryption. Cipher block chaining (CBC) xors each block of plaintext (ciphertext) with the previous block of ciphertext (plaintext) at each encryption (decryption). The initialization vector (IV) is an additional input block that is xored with the very first data block. Galois/counter (GCM) is an operation mode used for authenticated encryption, which provides additional protection against chosen ciphertext attacks.

Asymmetric encryption [49] is used when it is difficult to have a symmetric key shared between two entities. Asymmetric encryption uses two keys, a public key and a private key. The data is encrypted with the public key and can only be decrypted with the corresponding private key. *RSA* is one of the most popular algorithms.

III. Motivation

We focus on crypto accelerators because the tasks they implement are performed by many Android applications directly or with third-party libraries. For the experiments discussed in Section V, we executed 29 applications on *Hardroid*. Fig. 2 shows the percentage of these applications that use each of the four crypto accelerators. Specifically, *SHA1* is used by 76% of the applications, while almost all the applications use *SHA2* (97%). About 45% of the applications use *AES*, while about 14% invoke the *RSA* accelerator.

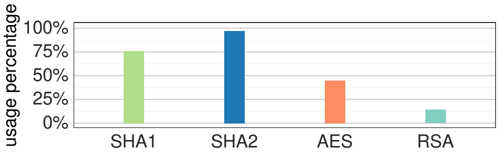


Fig. 2: Usage percentage of *SHA1*, *SHA2*, *AES*, and *RSA*.

Android-based systems support an ecosystem where it is possible to run real-world applications and evaluate accelerators in realistic scenarios. In this context, it is preferable to avoid changing the code of the applications to support accelerator invocation. In this way, we can preserve the portability of the applications across different platforms, where certain hardware accelerators may or may not be available. We use Android as a case study, but in other contexts there are similar requirements for the applications.

Hardroid addresses these challenges by allowing Android applications to invoke an accelerator without requiring modifications to their code. In general, applications must be modified to invoke a loosely-coupled accelerator. The software function that can be performed by means of an accelerator must be replaced with code that performs the accelerator invocation. This requires to write application-specific and error-prone code that configures the accelerator, prepares the input data, calls a device driver, uses a synchronization mechanism to wait for the completion of the accelerator, and obtains the results. Alternatives ways to invoke accelerators, specifically for Android, include using custom libraries, which abstract away the low-level details, and intents [17], [54]. In this work, we exploited the concept of *provider*, which is the basis for the Java crypto library [31]. In Java, the applications perform crypto operations through a common set of application programming interfaces (APIs). These APIs can be implemented by multiple providers and the applications do not need to know which particular provider is used to perform an operation. We develop a new provider (Section IV-B) such that applications call our accelerators on the FPGA. The concept of provider is well-known and we exploited it to invoke accelerators on the FPGA. Other approaches may be used to invoke accelerators transparently.

IV. The HARDROID Platform

A. Hardware Architecture

Hardroid combines general-purpose processors that are responsible for the execution of the Android Software Stack (Android 8.1) and accelerators for crypto (Fig. 3). The processors are hard-core units, while the accelerators are soft-core units deployed on the FPGA. The accelerators are *fixed-function*, i.e., they do not execute instructions. The accelerators are *loosely-coupled* [14], [16]. We offload tasks to an accelerator by means of a device driver. Each accelerator has some memory-mapped configuration registers that are exposed to software. These registers define where the input and the output of the accelerator are located and the values of accelerator-specific parameters, e.g., the number of input bytes and the operation mode used for encryption. Once configured, the accelerator executes the task on behalf of the application, without interrupting the main processors until the task is completed. To communicate with software, the accelerator uses a DMA buffer, which is a contiguous memory region (CMA) that is accessible by both software and hardware (allocated with [30]).

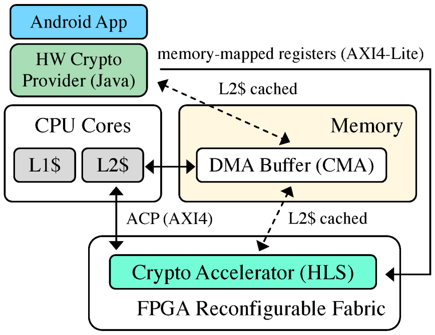


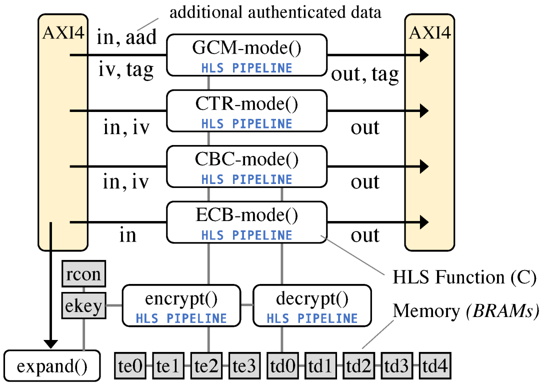
Fig. 3: Architecture of *Hardroid*. The crypto accelerator sits on the reconfigurable fabric and it is invoked by Android applications.

Accelerator Interface. To be integrated in *Hardroid*, the accelerators have to expose two interfaces. The first is an AXI4-Lite interface that is used for the memory-mapped registers. This interface allows the software to read and write single values to configure the accelerator. The second is a standard AXI4 interface that is used by the accelerator to read from and write to the DMA buffer. These interfaces simplify the integration of alternative microarchitectural implementations of a given accelerator according to the principles of latency-insensitive design [10]. We used the Accelerator Coherency Port (ACP) to connect our accelerators to the L2 cache of the processor (Fig. 3). The ACP is recommended for accelerators working on small datasets [39], which is often the case for crypto accelerators in Android (Section V). We used an open-source ACP adapter that converts the AXI4 requests of the accelerators to ACP-compliant requests [29]. There are several other options to handle the coherency of accelerators [12], [26], which we plan to explore in the future. For example, a common choice is to use the High-Performance Port (HP) that is faster for larger datasets.

Accelerator Architecture. We designed four accelerators: *SHA1*, *SHA2*, *AES*, and *RSA*. Fig. 4 shows the architecture of *AES* as an example. The accelerators are implemented in C and can be synthesized to RTL with different commercial HLS tools, thus allowing both their deployment on FPGA and their use in new chip designs[1]. To design *SHA1*, *SHA2*, and *AES*, we started from optimized C implementations in OpenSSL [45]. For *RSA*, we borrowed the ideas of Daly et al. [20]. We patched the unsynthesizable code of OpenSSL and we optimized each function to obtain the hardware implementation:

- *SHA1* and *SHA2*: we designed an accelerator for *SHA1* and an accelerator for *SHA2* that comprises *SHA224*, *SHA256*, *SHA384*, and *SHA512*. We unrolled the computations of each block (the block is 512 bits for *SHA1* and *SHA256*, and 1024 bits for *SHA512*) and we pipelined the fetching of the next block with the computation of the current block.
- *AES*: we developed an implementation based on T-tables [18]. We implemented single-block encryption and decryption functions (*encrypt()* and *decrypt()* in Fig. 4) and we pipelined them. We added support for four operation modes: ECB, CBC, CTR, and GCM. We pipelined the computation of different blocks of data to improve performance except for CBC encryption and GCM, which have some parts that must be performed sequentially. The operation modes share *encrypt()* and *decrypt()*. Alternatively, we can create multiple instances of them to enable more parallelism.
- *RSA*: we implemented a Montgomery exponentiation function, by applying the optimizations discussed in [20]. We keep the computation of $\mathcal{C} = 2^{2n} mod\ \mathcal{M}$ in software. We currently support keys of 256, 512, 1024, and 2048 bits.

[1] We support both Xilinx Vivado HLS and Mentor Graphics Catapult HLS.



ekey = expanded key, *rcon* = round constants,
*te**, *td** = T-tables for encryption/decryption

Fig. 4: Architecture of the *AES* accelerator.

We verified the HLS implementations with the NIST test suite [43]. We decided to implement our own accelerators because we wanted to support many of the configuration options that are used by Android applications. For example, the MachSuite [48] has *AES*, but it supports keys of 256 bits and the ECB operation mode only. CHStone [28] and S2CBench [50] have implementations of *AES* that work with multiple key sizes, but they support ECB only. In contrast, our implementation of *AES* supports multiple key sizes and four operation modes. Xilinx released some accelerators for crypto in the Vitis Libraries [58]. We plan to evaluate the integration of these accelerators in the future.

B. *Software Modifications*

Android (and Java) applications invoke crypto functions through the Java Cryptography Extension (JCE) and the Java Cryptography Architecture (JCA) [31]. These are APIs for crypto that belong to the standard Java Library. JCE/JCA define the important concepts of *service* and *provider*. The service is the task that must be performed, e.g., *SHA2*. The provider supplies implementations of one or more services. Since there are multiple providers for each service, they are given a preference order. An application asks for a service by calling a function called *getInstance()*. This method looks at the available providers to see if there is one that implements the requested service. If there are multiple providers, precedence is given to the one that comes first. The application then receives an instance of a class, e.g., *MessageDigest* for *SHA2*, that can be used to complete the service. The application does not need to know which is the provider; as long as there is at least one implementing that service, the application can get the service seamlessly and transparently. An application can potentially ask for a particular service from a specific provider, but this is not recommended as it limits the portability of the application [31].

In order to provide to any Android application the ability to invoke crypto accelerators transparently, we implemented a provider for JCE/JCA, that we called *Hardware Crypto Provider*. We can install it as default to force all the Android applications to use it. In this way, we obtain transparent invocation of accelerators, without the need of changing the applications. Fig. 5 shows how an Android application can invoke the *SHA2* accelerator by using the Hardware Crypto Provider. After obtaining an instance of *MessageDigest* with *getInstance()*, an application can use some methods to perform hashing. The application specifies the input data with *update()*, and then gets the result of the hashing with *digest()*. When the method *digest()* is called, the Hardware Crypto Provider copies the input data specified by the application to a reserved region of contiguous memory, the DMA buffer. We allocate this region of memory with `u-dma-buf` [30].

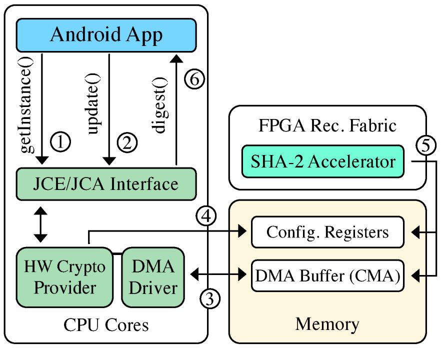


Fig. 5: How we invoke accelerators in *Hardroid*. We show how to invoke *SHA2*; the interface is slightly different for *AES* and *RSA*.

The provider then configures the memory-mapped registers of the accelerator. In particular, it specifies the offset of the input data in the DMA buffer, the offset where the output data should be placed, and other accelerator-specific parameters, such as the number of input bytes. The accelerator is then invoked. As soon as the accelerator completes its execution, the provider copies the output data from the DMA buffer and returns them to the application. This process is transparent to the application.

Programmability & Performance. Thanks to the Hardware Crypto Provider and the decoupling of the concepts of *service* and *provider* of the JCE/JCA, we can run unmodified Android applications and redirect the execution of the tasks to the FPGA accelerators. In Section IV-D, we discuss to what extent this approach is applicable to other accelerators and application domains. Unfortunately, programmability advantages can have performance implications. To invoke an accelerator, the Hardware Crypto Provider must copy the input data specified by the application to the DMA buffer and then copy the output data from the DMA buffer. In fact, we cannot control where the data of the applications are allocated. If we are allowed to change the application, we could allocate the data directly in the region of memory that is accessed by the accelerator. This is the approach taken, for example, by Mantovani et al. [36]. The application can then work with the data allocated in a particular region of memory, avoiding frequent data copies when an accelerator is invoked. In addition to the data copies, there are other overheads for configuring and clearing the DMA buffer and for performing accelerator-specific operations such as handling the padding schemes of *AES*. These overheads depend, of course, on the specific accelerator and application domain.

Future Improvements. Currently, our provider uses a single DMA buffer for the accelerator invocation that is created with `u-dma-buf` and accessed as explained by I. Kawazome [30]. In the future, we plan to support multiple DMA buffers for concurrent accelerator invocations. In addition, for simplicity, currently we grant access to the DMA buffer to the Android applications that can access it through the crypto provider. The buffer should be protected by granting access only to trusted code and by guaranteeing isolation among different applications. With *Hardroid*, our current goal is to provide an evaluation platform for the integration of accelerators in Android. We leave the important issue of investigating their secure invocation to future work.

C. Design Automation

Hardware Flow. *Hardroid* supports the integration of any accelerator that complies with the interfaces we explained in Section IV-A. We automated the entire hardware flow thanks to Xilinx Vivado HLS and Xilinx Vivado. First, we generate the RTL code of the accelerator with Vivado HLS. Then, we integrate the accelerator with the hard-core processor (ARM) on the FPGA board by using (i) the AXI4 interconnects made available by Xilinx and (ii) an open-source ACP adapter [29] that translates the AXI4 requests of the accelerator to ACP-compliant requests. This process terminates with the generation of the bitstream that is used to configure the FPGA at boot time.

	MHz	*FFs*	*LUTs*	*BRAMs*
SHA1	187	13393	28501	1
SHA2	187	73479	104700	4
AES	187	38512	55233	434
RSA	149	87356	92342	4

(a) FPGA target.

Area (mm^2)
0.415
1.997
1.331
0.698

(b) ASIC Target.

TABLE I: Characteristics of the FPGA and ASIC accelerators.

Software Flow. The software flow requires significant effort from the designer if a new accelerator is added to *Hardroid*. The designer must define a Java class that exposes an API to invoke the accelerator. This class manages the DMA buffer, configures the memory-mapped registers of the accelerator, and handles the execution of the accelerator Some parts of this class could be automatically generated, but other parts must be written for the specific accelerator and the target applications that invoke it. The accelerator can then be called from an Android application at runtime.

D. Applicability to other Domains

In the case of crypto accelerators, it is possible to invoke them without changing the applications thanks to the decoupling of the concepts of *service* and *provider* in the JCE/JCA. There are other libraries that have a similar concept. For example, Tensorflow [1] uses *delegates* to run machine learning tasks with different computational engines (e.g., GPUs, CPUs). For some application domains, a similar decoupling might not be available in the software library for which we want to develop hardware accelerators. Therefore, it would be necessary to modify the Android applications to support accelerator invocation. We believe, however, that the concept of provider used in the JCE/JCA has broad applicability and might be adopted in other application domains to simplify accelerator integration and improve portability. Future software libraries can be developed with a similar decoupling mechanism to enable hardware acceleration. In addition, this idea can be exploited in other contexts besides Android and cryptography, e.g., in standard Linux-based systems for machine learning, to support transparent integration of accelerators.

V. Experimental Evaluation

We developed *Hardroid* targeting the Xilinx ZynqMP UltraScale+ ZCU102 [57]. It includes a quad-core ARM Cortex-A53, where we run Android 8.1.0, and a 16-nm programmable logic fabric, where we deployed our accelerators. We configured the FPGA at boot time. We leveraged an open-source porting of Android to the ZCU102 [3]. We added our crypto accelerators (Section IV-A) and the software stack (Section IV-B) to call the accelerators from the Android applications. We designed the accelerators for FPGA by using Xilinx Vivado HLS 2018.2 and performed logic synthesis with Vivado. We ported the accelerators to Mentor Graphics Catapult HLS 10.6*a* to calculate the power for ASIC, using the 45-nm Nangate standard-cell library.

We evaluate *Hardroid* in two ways. In Section V-A, we compare the energy consumption of our crypto accelerators with two popular libraries: BouncyCastle 1.57 [9] and AndroidOpenSSL 1.0.0 (conscrypt), an optimized porting of OpenSSL [45] to Android systems. In Section V-B, we perform an analysis of 29 Android applications.

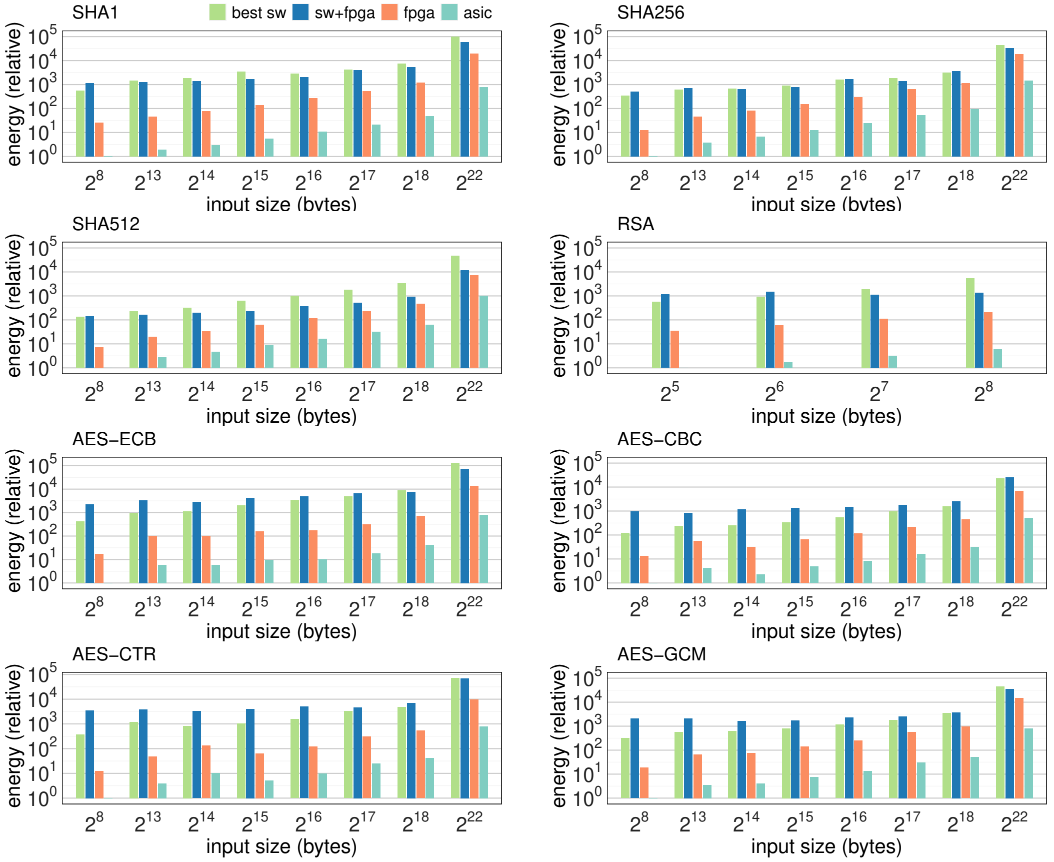


Fig. 6: The figure reports the energy consumption of performing crypto operations in *Hardroid*. We compare (i) the best software implementation (*best sw*), (ii) our accelerators by considering the software overheads for invoking them (*sw+fpga*), (iii) our FPGA accelerators only (*fpga*), and (iv) our ASIC accelerators only (*asic*). The values are relative to the energy consumed by the ASIC accelerators for the smallest inputs.

A. Custom Applications

We designed our crypto accelerators with HLS for FPGA and ASIC. TABLE I (a) reports the results for FPGA in terms of clock frequency (MHz), FFs, $LUTs$, and $BRAMs$. TABLE I (b) shows the results for the ASIC implementations in terms of area (mm^2). We synthesized the accelerators with the same frequency (187 for *SHA1*, *SHA2* and *AES* and 149 for *RSA*) for both ASIC and FPGA and the same architecture[2] (Section IV-A). The RSA code generated with HLS was difficult to close at a higher frequency due to the use of large operators. The goal was to get a conservative estimation of what can be obtained in ASIC. For real ASIC deployment, it is likely that the accelerators are synthesized at the same frequency of the processor (1.2 GHz), thus obtaining an additional speedup of about 6× (8× for *RSA*). For power estimations, we use the dynamic power returned by Vivado and the dynamic power reported by simulations based on switching activity after synthesis with Catapult HLS.

We compared the execution time and power consumption of our accelerators against those of the implementations of the corresponding algorithms in the BouncyCastle and the AndroidOpenSSL libraries, which are executed on the ARM Cortex-A53 processor. We measured the performance on FPGA by executing Android applications that we developed. We installed the Android applications and interacted with them to trigger accelerator invocations. Just-In-Time (JIT) compilation is combined with Ahead-of-Time (AOT) compilation in Android [4]. JIT uses heuristics to cache the translation of methods that are executed at run-time. To obtain better performance for the BouncyCastle library, we warmed up the JIT cache so that the overheads of the JIT caching mechanism are not considered. For the power of the accelerators, we consider the power of the task that they execute in a particular test. For example, for *SHA256* we used the power for *SHA256*, rather than the power of the whole *SHA2* accelerator, which includes *SHA512*. We account for the power spent by the CPU to interact with the accelerator (i.e., configuring it, preparing the input data in memory, and retrieving the output data from memory) but not for the power of the CPU while it is idle during the accelerator execution.

Fig. 6 shows the results obtained from multiple executions on the FPGA. On the x-axis we report the size in bytes of the inputs. On the y-axis we report the energy consumption (in log scale) for the best software implementation (*best sw* in Fig. 6), for the FPGA accelerators including the software overheads due to the invocation (*sw+fpga*), for the FPGA accelerators only (*fpga*), and for the ASIC accelerators only (*asic*). The best software implementation is either AndroidOpenSSL or BouncyCastle depending on the crypto task and the input size. Note that the results are relative to the energy consumed by the ASIC accelerators to perform the smallest tasks (for the smallest task in each graph, the ASIC energy consumption is 1). We can observe that the energy overhead of the software to invoke the accelerators is very high, especially for smaller input sizes. Part of this overhead is due to the data copies of the inputs and outputs of the accelerator. For

[2] The coding style we used for *AES* for Vivado HLS is not fully compatible with Catapult HLS. Thus, to obtain the results for ASIC we used a 32-bit AXI4 interface (instead of 128 bits) and adjusted the size of the internal memories. Similarly, we estimated the power of AES/GCM from the power of AES/CBC by considering the area difference.

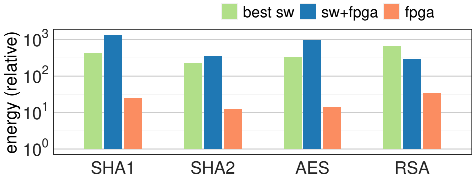


Fig. 7: Energy consumption (geometric mean) of the 29 applications.

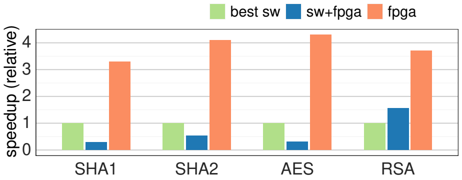


Fig. 8: Performance (geometric mean) of the 29 applications.

SHA1 and *SHA2* the copy overhead is more manageable as the output size has a fixed dimension (20 bytes for *SHA1* and up to 64 bytes for *SHA2*). For *AES*, this becomes a more important issue as the output size scales with the input size. For *RSA*, this is less of an issue as the input sizes are relatively small. If one is allowed to change the applications to invoke the accelerators, part of this overhead could be mitigated by avoiding data copies [36]. Some software overhead is due to the time required to set up the DMA buffer and to handle the different configuration options of the accelerators. If we take a look at the energy consumption of the FPGA accelerators (or the ASIC accelerators), we can see how the software overhead impacts their efficiency. By moving towards larger input sizes, the relative distance in energy consumption between the best software implementation (*best sw*) and the FPGA accelerators (*fpga*) becomes smaller. Switching to HP ports would provide better performance for larger inputs [39]. Note that the energy consumption is sometimes higher for smaller input sizes; although we report the average of multiple runs, short execution times can be affected by other processes and/or Java.

B. Android Applications

We downloaded 29 applications from the Google Play Store. We installed each application in *Hardroid* and we interacted with the graphical user interface (GUI) of the application (with keyboard and mouse). Being already supported by Mentor Embedded Android [3], this is not part of our contributions. A contribution of *Hardroid*, however, is that the actions on the GUI can trigger accelerator invocations. We interacted with the applications and generated about 3000 invocation calls to our crypto accelerators. The invocations are for *SHA1*, *SHA256*, *SHA512*, *AES*-CBC, *AES*-GCM, and *RSA*. As explained in Section IV-B, none of the 29 applications required modifications. In order to evaluate the performance and check the correctness of our accelerators, we modified our provider such that it invokes also BouncyCastle and AndroidOpenSSL. We monitored the logging system of Android to collect the performance measurements.

Fig. 7 reports the estimated energy consumptions as the geometric mean across 29 applications. Each set of three bars report the energy consumption for the best software implementation, the FPGA accelerator with the addition of the invocation overhead in software, and the standalone FPGA accelerator, respectively. The reported values are relative to the energy consumption of the ASIC implementations of these accelerators, which are used as baseline. The results confirm that the invocation cost in software is high, especially considering that the input sizes for many invocations of the accelerators consist of only hundreds of bytes. We could mitigate the energy cost of the software in invoking the accelerators by executing the smaller tasks with the ARM core rather than using the accelerators, while leaving the bigger tasks to the accelerators for better energy efficiency.

Fig. 8 reports the accelerator performance as the geometric mean across 29 applications relative to the best software implementations, which are used as baseline in this case. By considering only the accelerator execution time (*fpga*), we can observe that we obtain better performance than software, thanks to the ACP port that guarantees higher performance for small inputs.

VI. Related Work

The increasing adoption of domain-specific accelerators has fostered the development of several heterogeneous architectures and platforms, for example [2], [5], [7], [21], [33], [37], [46], which have been often made available open-source. All these platforms support Linux-based environments, but they do not support Android applications. There are, however, platforms that can run Android applications. In particular, to develop *Hardroid*, we started from Mentor Embedded Android [3]. This is an open-source porting of Android 8.1 to some Xilinx FPGAs. The same platform has been used to support dynamic reconfiguration of the FPGA [23]. Zedroid, a platform that supports Android on a Zynq SoC [59], has been used to accelerate a kernel for network traffic analysis [8]. Ting et al. [53], [54] show how to provide *accelerator services* for machine learning to Android applications. Their platform supports multiple applications invoking the accelerators as well as multiple accelerators. Similar mechanisms can be added to *Hardroid*. Coughlin et al. [17] described an approach to add reconfigurable hardware into an Android-based system. ‘App Hardware’ are accelerators that can be deployed on the FPGA and called by the traditional applications (‘App Software’). Our goal is different because *Hardroid* is a platform to evaluate the integration of accelerators in Android-based system before these are taped out as components of a new chip.

There are several papers about accelerator integration. For example, Giri et al. [26] describe the advantages of supporting multiple cache-coherence models for different accelerators. They also show how to seamlessly integrate third-party accelerators [25], e.g., NVDLA, into the ESP architecture [37]. Min et al. [39] evaluate approaches that can be used to integrate accelerators on an FPGA-based platform. Lee et al. [34] focus on improving the programmability of data-parallel accelerators. Our paper complements these works by discussing the trade-offs between programmability and performance in integrating accelerators in Android systems.

VII. Conclusions

We presented *Hardroid*, an FPGA-based heterogeneous platform that allows Android applications to call crypto accelerators on an FPGA in a transparent way. The applications do not need to be modified to perform accelerator invocations thanks to the decoupling of providers and services in the JCE/JCA. We believe that the same approach can be used to invoke the accelerators for other important application domains. We explored the trade-offs between programmability, performance and energy efficiency for crypto accelerators, showing that programmability benefits can affect performance and energy efficiency. In the future, we will evaluate more approaches for better accelerator design and integration and we will explore their applicability to other domains.

Acknowledgments

This work was supported in part by DARPA (C#:HR0011-18-C-0122) and the National Science Foundation (A#:1764000). The views and conclusions expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] M. Abadi *et al.*, "Tensorflow: A System for Large-scale Machine Learning," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[2] A. Amid *et al.*, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, vol. 40, no. 4, 2020.

[3] https://github.com/MentorEmbedded/mpsoc-manifest, Android for Xilinx Zynq UltraScale+ MPSoC.

[4] https://source.android.com/devices/tech/dalvik/jit-compiler, ART JIT.

[5] K. Asanović *et al.*, "The Rocket Chip Generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[6] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and Area-efficient FPGA Implementations of Lattice-based Cryptography," in *Proc. of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013.

[7] J. Balkind *et al.*, "OpenPiton at 5: A Nexus For Open And Agile Hardware Design," *IEEE Micro*, vol. 40, no. 4, 2020.

[8] M. Barbareschi, A. Mazzeo, and A. Vespoli, "Network Traffic Analysis Using Android on a Hybrid Computing Architecture," in *Proc. of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2013.

[9] https://bouncycastle.org/, Bouncy Castle.

[10] L. P. Carloni, "From Latency-Insensitive Design to Communication-Based System-Level Design," *Proceedings of the IEEE*, vol. 103, no. 11, 2015.

[11] ——, "The case for Embedded Scalable Platforms," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2016.

[12] M. Cavalcante *et al.*, "Design of an Open-source Bridge between Non-coherent Burst-based and Coherent Cache-line-based Memory Systems," in *Proc. of the ACM Conference on Computing Frontiers (CCF)*, 2020.

[13] T. Chen *et al.*, "Diannao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," in *Proc. of the ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[14] J. Cong *et al.*, "Charm: A Composable Heterogeneous Accelerator-rich Microprocessor," in *Proc. of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISPLED)*, 2012.

[15] https://github.com/google/conscrypt, Conscrypt.

[16] E. G. Cota *et al.*, "An Analysis of Accelerator Coupling in Heterogeneous Architectures," in *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2015.

[17] M. Coughlin, A. Ismail, and E. Keller, "Apps with Hardware: Enabling Run-time Architectural Customization in Smart Phones," in *Proc. of the USENIX Annual Technical Conference (ATC)*, 2016.

[18] J. Daemen and V. Rijmen, *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[19] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific Hardware Accelerators," *Communications of the ACM*, vol. 63, no. 7, 2020.

[20] A. Daly and W. Marnane, "Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic," in *Proc. of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2002.

[21] S. Davidson *et al.*, "The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips," *IEEE Micro*, vol. 38, no. 2, 2018.

[22] G. Eichler *et al.*, "MasterMind: Many-Accelerator SoC Architecture for Real-Time Brain-Computer Interfaces," in *Proc. of the International Conference on Computer Design (ICCD)*, 2021.

[23] https://github.com/1chor/master-thesis, FPGA Reconfiguration Android.

[24] D. Fujiki *et al.*, "Genax: a Genome Sequencing Accelerator," in *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2018.

[25] D. Giri *et al.*, "Accelerator Integration for Open-Source SoC Design," *IEEE Micro*, vol. 41, no. 4, 2021.

[26] D. Giri, P. Mantovani, and L. P. Carloni, "Accelerators and Coherence: An SoC Perspective," *IEEE Micro*, vol. 38, no. 6, 2018.

[27] T. J. Ham *et al.*, "Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics," in *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2016.

[28] Y. Hara *et al.*, "Chstone: A Benchmark Program Suite for Practical C-based High-Level Synthesis," in *Proc. of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2008.

[29] https://github.com/ikwzm/ZynqMP-ACP-Adapter, I. Kawazome, Adapter.

[30] https://github.com/ikwzm/udmabuf, I. Kawazome, U-DMA-BUF.

[31] https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html, JCE/JCA.

[32] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Chapman and Hall, CRC Press, 2014.

[33] A. Kurth *et al.*, "HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA," *Proc. of the Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[34] Y. Lee *et al.*, "Exploring the Tradeoffs between Programmability and Efficiency in Data-Parallel Accelerators," *ACM Transactions on Computer Systems*, vol. 31, no. 3, 2013.

[35] X. Liu *et al.*, "High Level Synthesis of Complex Applications: An H.264 Video Decoder," in *Proc. of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[36] P. Mantovani *et al.*, "Handling Large Data Sets for High-Performance Embedded Applications in Heterogeneous Systems-on-Chip," in *Proc. of the ACM/IEEE International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2016.

[37] ——, "Agile SoC Development with Open ESP," in *Proc. of the International Conference on Computer Aided Design (ICCAD)*, 2020.

[38] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 2001.

[39] S. W. Min *et al.*, "Analysis and Optimization of I/O Cache Coherency Strategies for SoC-FPGA Device," in *Proc. of International Conference on Field Programmable Logic and Applications (FPL)*, 2019.

[40] R. Nane *et al.*, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, 2016.

[41] "Advanced Encryption Standard (AES)," Federal Inf. Process. Stds. (NIST FIPS) - 197, National Institute of Standards and Technology, 2001.

[42] "Secure Hash Standard (SHS)," Federal Inf. Process. Stds. (NIST FIPS) - 180-4, National Institute of Standards and Technology, 2015.

[43] https://csrc.nist.gov/Projects/Cryptographic-Algorithm-Validation-Program, NIST, Cryptographic Algorithm Validation Program.

[44] https://github.com/nvdla/, NVDLA Deep Learning Accelerator.

[45] https://github.com/openssl, OpenSSL.

[46] D. Petrisko *et al.*, "BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs," *IEEE Micro*, vol. 40, no. 4, 2020.

[47] L. Piccolboni *et al.*, "COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, 2017.

[48] B. Reagen *et al.*, "Machsuite: Benchmarks for Accelerator Design and Customized Architectures," in *Proc. of the IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[49] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, 1978.

[50] B. C. Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis," *IEEE Embedded Systems Letters*, vol. 6, no. 3, 2014.

[51] O. Segal *et al.*, "PSparkcl: A Unified Programming Framework for Accelerators on Heterogeneous Clusters," *arXiv:1505.01120*, 2015.

[52] I. Stamelos *et al.*, "A Novel Framework for the Seamless Integration of FPGA Accelerators with Big Data Analytics Frameworks in Heterogeneous Data Centers," in *Proc. of the International Conference on High Performance Computing Simulation (HPCS)*, 2018.

[53] H.-Y. Ting *et al.*, "Dynamic Sharing in Multi-accelerators of Neural Networks on an FPGA Edge Device," in *Proc. of the International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2020.

[54] H.-Y. Ting, A. A. Sani, and E. Bozorgzadeh, "System Services for Reconfigurable Hardware Acceleration in Mobile Devices," in *Proc. of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2018.

[55] M. Vuletid, L. Pozzi, and P. Ienne, "Seamless Hardware-Software Integration in Reconfigurable Computing Systems," *IEEE Design & Test of Computers*, vol. 22, no. 2, 2005.

[56] L. Wu *et al.*, “Q100: The Architecture and Design of a Database Processing Unit,” in *Proc. of the ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[57] “Zynq UltraScale+ MPSoC ZCU102,” https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html, Xilinx.

[58] https://github.com/Xilinx/Vitis_Libraries, Xilinx Vitis.

[59] http://wpage.unina.it/mario.barbareschi/zedroid/index.html, Zedroid.