Design Flow for HW / SW Acceleration Transparency in the ThumbPod Secure Embedded System

David Hwang, Patrick Schaumont, Yi Fan, Alireza Hodjat
Bo-Cheng Lai, Kazuo Sakiyama, Shenglin Yang, Ingrid Verbauwhede

UCLA Electrical Engineering Department
{dhwang, schaum, yifan, ahodjat, bclai, kazuo, shengliny, ingrid}@ee.ucla.edu

ABSTRACT
This paper describes a case study and design flow of a secure embedded system called ThumbPod, which uses cryptographic and biometric signal processing acceleration. It presents the concept of HW/SW acceleration transparency, a systematic method to accelerate Java functions in both software and hardware. An example of acceleration transparency for a Rijndael encryption function is presented. The embedded prototype hardware platform is also described. Acceleration transparency yields software and hardware performance gains of 333X.

Categories and Subject Descriptors
E5 [**Case Studies**]; E3 [**HW/SW Co-Design**]: specification, modeling, co-simulation and performance analysis, system-level scheduling and partitioning.

General Terms
Performance, Design, Experimentation.

1. INTRODUCTION
The field of embedded systems is growing at a rapid rate, as evidenced by the burgeoning market for cellular phones, PDAs, digital camcorders, smart cards, and other intelligent portable devices in the last decade or so. Recent interest has piqued in security of embedded systems, as many of these systems contain or transmit sensitive data. Application convergence in embedded systems is also another area of interest, as hybrid solutions of phones and PDAs, GPS receivers and watches, etc. enter the marketplace.

This paper describes the design flow of a secure embedded system called ThumbPod [1]. ThumbPod is a driver for our research in domain-specific processing techniques and design methods. It is embedded system which consists of a 32-b Sparc microcontroller, a fingerprint image sensor, signal processing hardware acceleration, cryptographic hardware acceleration, and a memory module enclosed within a form factor similar to an automobile keychain transmitter. A concept drawing is seen in Figure 1. ThumbPod will offer flexible communication via two ports: 1) an infrared port for wireless communication and 2) a USB port for fast wire-line communication. ThumbPod represents the application convergence of several domains, all of which are biometrically secured. Potential applications include wireless credit card payments, keychain flash memory replacement, universal key functionality (house, car, office), storage of sensitive medical data, and IR secure printing.

DAC 2003, June 2-6, 2003, Anaheim, California, USA.

Figure 1. ThumbPod concept drawing.

1.1 Cryptography & Biometrics
In the design of ThumbPod, two particular domains require implementation consideration. The first domain is the domain of cryptography, in our case secret-key (symmetric-key) cryptography. Secret-key cryptography is based on the premise that two users share a secret key that is known only to them. In order to securely communicate with one another, this key is used both for encryption and decryption purposes.

The most recent standard for symmetric-key cryptography is the Rijndael algorithm, which was made into the NIST AES (Advanced Encryption Standard) in 2001 [2]. For the purposes of ThumbPod, Rijndael is a block cipher which takes in a 128-b block of plaintext and produces a 128-b block of ciphertext based on a 128-b secret key. Arithmetically, the Rijndael algorithm performs computationally-complex functions such as byte substitution, mathematic operations over a Galois field GF(2^8), row shifting, column shifting, and frequent xor operations. Due to this complexity, acceleration of the Rijndael function is desirable.

The other domain that requires implementation consideration is biometric fingerprint signal processing. Fingerprint biometrics can be used as the means to identify a user to a system. In ThumbPod, the entire fingerprint verification process is performed within the embedded system (not on an server). At the computational core of the fingerprint verification process are the minutia detection and matching algorithms. These algorithms require signal processing functionality such as image enhancement, Fourier transforms, edge detection, etc. Due to the complexity of signal processing functionality, acceleration for the fingerprint verification process is also desirable.

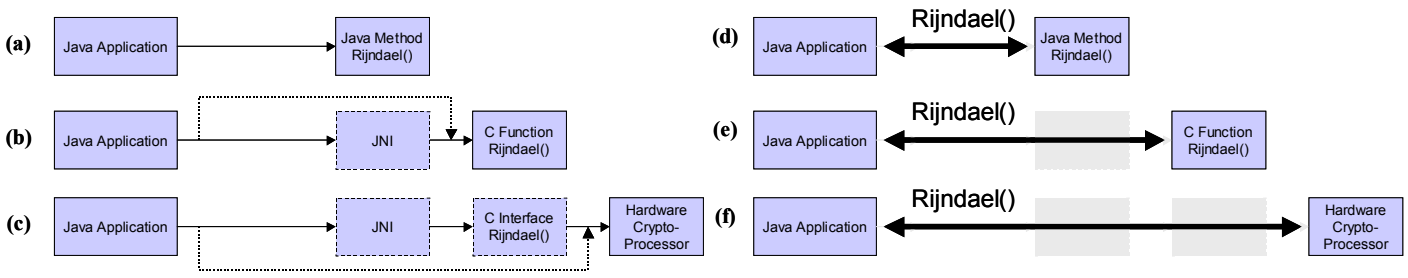

Figure 2. HW/SW acceleration transparency.

1.2 ThumbPod Authentication Protocol

We have developed a protocol for wireless credit card transactions which utilizes ThumbPod and its biometric authentication capabilities. Though the details of the protocol are beyond the scope of this paper, the protocol in essence is based on secret-key cryptography using the Rijndael algorithm. In the protocol, the secret key is a 128-b representation of the user's fingerprint. This secret key is stored at the financial institution's central server and stored (in a different format) on the user's ThumbPod.

In order to make a transaction at a merchant's register, the user uses the ThumbPod's IR port to initiate communication with the register. A series of challenge and response functions is negotiated between the user and the financial institution's central server, all routed through the merchant's register (which cannot interpret the data because it does not posses the key). In the course of the authentication protocol the user places his finger on the fingerprint sensor to ensure the ThumbPod is his (and not stolen or fraudulent). This information is processed within the ThumbPod and, if a match is made, cryptographic hash functions and keys are generated using Rijndael and the protocol continues to its completion.

The protocol is an example of a complex application in which ThumbPod requires both cryptographic and signal processing functionality. There are various other protocols for other ThumbPod applications, all of which share the common denominators or cryptography and biometric signal processing. Some applications, such as encryption/decryption for audio and video systems, require encryption rates beyond the capacity of embedded software implementations and require hardware acceleration.

The rest of the paper describes Java acceleration and a design method called hardware/software acceleration transparency used to simultaneously allow for selective acceleration as well as incremental refinement of the design flow.

2. JAVA ACCELERATION

In our embedded design flow, Java was used for its well-known portability and security advantages [3]. The issue of portability is important in embedded systems because of their high processor heterogeneity. Java's security advantages—such as a safe memory model, byte-code verification, cryptographic interface libraries, and the sandbox model—are important in the design of secure systems.

However, though advantages exist in these domains, Java has one primary drawback: performance. A Java application is slower than its counterpart in C, and much slower than its counterpart in pure hardware. An example of Java's performance drawback can be seen in Table 1, where the 128-b input, 128-b key Rijndael function in Electronic Code Book (ECB) is performed. The Java (KVM) and C figures are on a 1 mW / 1 MHz Sparc processor. This configuration is used to emulate an embedded environment. The ASIC figures are based on a recent implementation performed by our research group [4]. As can be seen in the table, a hardware solution is five orders of magnitude superior in both performance and energy consumption (as measured in Gb/s per Watt). For streaming encryption applications described in the previous section, pure embedded software solutions are inadequate. Hardware acceleration is required.

Table 1. Comparison of 128-b Rijndael on different platforms.

Platform	Throughput	Power	Gb/s / W
Java	450 bits/s	120 mW	0.00042
C	345 Kbits/s	120 mW	0.029
0.18 μm ASIC	2.29 Gb/s	56 mW	35.7

2.1 HW/SW Acceleration Transparency

In order to incorporate software and hardware acceleration and simultaneously allow for incremental refinement in the design flow process, we have used a technique called hardware software acceleration transparency in our design. HW/SW acceleration transparency is described below in further detail and involves three closely related items: 1) incremental acceleration, 2) Java function emulation, and 3) interface transparency.

The goal of acceleration transparency is to begin with Java code on a workstation and to conclude with an embedded prototyped system (with hardware acceleration) whose Java code is identical to the initial code, except for coping with communication issues and processor and hardware support. The main consequence of this method is that code writing and validation becomes easier, providing a gain in terms of development time and quality.

2.2 Incremental Refinement Acceleration

The first principle of acceleration transparency is incremental refinement acceleration. In the example shown in Figure 2a, a Java application calls a Rijndael method. Based upon profiling results, if the performance of the pure Java solution is inadequate, it can be accelerated using a C function, as shown in Figure 2b. Rather than designing a custom interface to the C Rijndael function, as shown in the dotted line in Figure 2b, the application accesses the function through the Java Native Interface (JNI). If profiling and comparison with system specifications determine that hardware acceleration is required, a crypto-processor can be designed and interfaced to the Java application. However, this

crypto-processor does not directly interface with the Java application (as shown in the dotted line in Figure 2c) but is accessed via assembly instructions by a skeletal C function, which itself is accessed by the Java application via the JNI. Though it seems wasteful in terms of overhead to use these interfaces, incremental refinement allows for a smoother design flow than creating custom interfaces at each of the design levels. Methods for the design of domain-specific co-processors can be found in [11].

2.3 Java Function Emulation

HW/SW acceleration transparency also includes Java function emulation, a term used to describe the interface relationship between the Java application and the accelerated function. For example, a Java application wishes to access a Rijndael function via a function call `rijndael()`. From the above discussion, the Java application has one of three alternatives to obtain the implementation: 1) a Java function, 2) a C function, or 3) hardware acceleration.

HW/SW acceleration transparency means that, to the Java application, each of these alternatives is accessed with the same Java function signature. In the pure Java case, this is already apparent: A Java Rijndael function is accessed by the Java application with a simple function call `rijndael()`. For C acceleration, interfaces are constructed such that the Java application can access the C Rijndael function with the *same* function call `rijndael()`. For hardware acceleration, HW/SW interfaces to the crypto-processor are designed such that Rijndael functionality is again accessed by the *same* function call `rijndael()`. In this way, from the Java application vantage point, each of these alternatives "looks" exactly the same. To the application, each of the three alternatives takes in the same input, produces the same output, and is accessed by the same Java function and hence functionally is the same, as seen in Figure 2d, Figure 2e, and Figure 2f.

2.4 Interface Transparency

Implicit to the previously mentioned Java function emulation is the concept of interface transparency. This is also illustrated in Figure 2. Interface transparency (from which we derive the transparency term of HW/SW acceleration transparency) means that to the Java application, all the interfaces in between it and the acceleration implementation are transparent. In other words, the Java application can directly "see" the acceleration implementation (which looks to it like a Java function) regardless of the number of interfaces. Interface transparency essentially raises co-processor control a number of abstraction layers directly to the Java application level.

2.5 Advantages and Potential Drawbacks

Using HW/SW acceleration transparency to facilitate Java acceleration has the following advantages:

- **Smooth interface design flow.** Using HW/SW acceleration transparency, we build our interfaces *incrementally*. Instead of tearing down the previous interface and starting from scratch at each abstraction level, the next interface *incrementally refines* the previously constructed interface. Thus, the interface design flow is smooth and continuous.
- **System performance modeling and verification**. Using HW/SW acceleration transparency allows for system performance modeling at each abstraction level. As each accelerated function is placed into the overall system, the hybrid system can be re-benchmarked and the performance gains ascertained.
- **Smooth Java application design flow**. As the system progresses from software to hardware, the original Java application requires only minor modification. Using HW/SW transparency implies that each of the acceleration modules "looks" like the initial Java function in the original application; hence, the original Java application can remain the same (or relatively unchanged) from the beginning functional simulation to the final HW/SW system implementation.
- **Reconfiguration**. Once the interface hierarchy is constructed, a new acceleration module can be appended to the system through the pre-designed interfaces. A system can thus be reconfigured in a systematic way.

There are a few potential drawbacks to HW/SW transparency acceleration:

- **Interface overhead**. Interface overhead is the cost, in cycle count, of going through the many layers of interface abstractions to reach the acceleration module. Obviously, if the interface overhead outweighs the performance/power gains of acceleration, then the acceleration should not be performed. This topic will be discussed in further detail in a section below.
- **Design time**. It takes a considerable amount of time to construct the interface hierarchy from abstraction level to abstraction level. However, as stated earlier, in the end this may actually save overall design time due to the incremental interface construction.

2.6 Rijndael Example

This section of the paper describes an example of HW/SW acceleration transparency and gives performance measurements for interface overhead. All the advantages of HW/SW acceleration transparency described in earlier sections would be moot if the interface overhead were too large. We are therefore interested in quantifying the overhead of different abstraction levels as we go down from C to hardware. Our simulation environment consists of a cycle-true LEON-Sparc simulator [10]. C code is compiled with the GNU C compiler gcc V3.2 with full optimization (-O2). Java byte-code is interpreted on the KVM embedded virtual machine from the Java2 Micro Edition. Thus, cycle counts for Java are cycles of the target LEON-Sparc which runs KVM that in turn runs the Java program.

We start by choosing the aforementioned interface specification of the Rijndael in Java and C. We use a 128-bit key and 128-bit data block.

```
Java:  int[] rijndael(int[] key, int []din)
C:     void rijndael( int din[4], int key[4],
int dout[4])
```

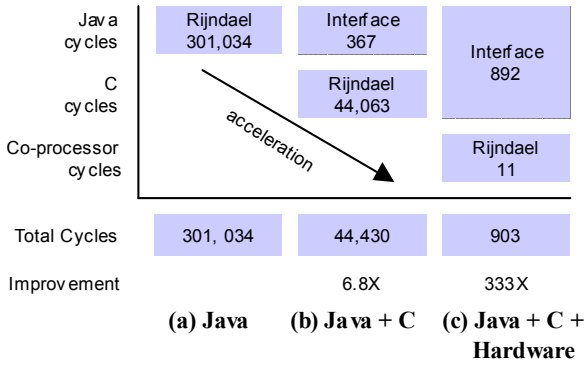

Figure 3. Rijndael acceleration transparency.

A pure Java implementation for Rijndael on top of KVM takes 301,034 cycles, as shown in Figure 3. All numbers in the figure are for one iteration of the Rijndael algorithm, starting from the Java function call. Startup overhead, such as setting up the C or Java runtime environments, is not included.

A first refinement to the pure Java model is to substitute the pure Java implementation with a native implementation in C. A native method in Java is seen Figure 4a. The corresponding C implementation is shown in Figure 4b. A function renaming is required in order to reflect the position of the native method in the Java class hierarchy. The C implementation then can forward control to the implementation of the `rijndael()` function.

The `rijndael()` function of Figure 4b can, at first, call an implementation of the Rijndael algorithm in C. When we use the NIST reference code, we obtain the figures as shown in the second column of Figure 3. We have 44,430 cycles per Rijndael call, of which 367 can be attributed to the interfacing part (Figure 3a and b) and the rest to native implementation. Overall we gain 6.8X performance.

The next step is to substitute the C implementation with a native hardware implementation of the Rijndael algorithm. We use a hardware coprocessor that completes a 128-bit encryption in 11 clock cycles. This hardware processor is interfaced to the co-processor interface of the Sparc, and programmed as shown in Figure 4c. The 128-bit key and data are provided with two double-word move instructions. In this case, the resulting performance was 903 cycles. Here, the interfaces turn out to consume the major part of the cycle budget. The actual encryption takes only 11 cycles; going from Java to hardware consumes 892 cycles. The performance gain in going from Java to hardware is now 333X.

We conclude that, while the performance gain of moving from Java to native implementation is substantial, it is not completely overhead-free. This overhead is primarily caused by moving data across the hierarchy levels in the model. We are refining our method to treat data- and control-flow separately, by which we expect this overhead to substantially decrease. In any case, the incremental refinement of the model is a major advantage from the design-flow point-of-view.

3. DESIGN FLOW ABSTRACTION LEVELS

The design of ThumbPod requires a number of abstraction levels, with each abstraction level requiring design decisions and interface construction. The smooth transition from one model to another allows for successive refinement of the system. This section enumerates the different abstraction levels and their particular characteristics.

```
public final class RijndaelAlgorithm {
    static native int[] rijndael(int[] din, int[] key) ;
    public static void main(String args[]) {
        ...
        dout = rijndael(key, din);
        ...
    }
}
```

(a)

```
void Java_RijndaelAlgorithm_rijndael (void) {
  ARRAY i1, i2;
  ARRAY result = instantiateArray(INT,4);
  i1 = popStackAsType(ARRAY);
  i2 = popStackAsType(ARRAY);
  rijndael(i1->data, i2->data, result->data);
  pushStackAsType(ARRAY, result);
}
```

(b)

```
void rijndael(int din[4], int key[4], int dout[4]) {
  asm(" mov    %0, %%l0" : : "r" (key));
  asm(" ldd    [%l0],    %c0        ! upper double word key
       ldd    [%l0+8],  %c2        ! lower double word key
       cpop1  load_key %c0, %c2    ! load the key");

  asm(" mov    %0, %%l1" : : "r" (din));
  asm(" ldd    [%l1], %c0           ! upper double word data
       ldd    [%l1+8], %c2         ! lower double word data
       cpop1  encrypt_ecb %c0, %c2 ! encrypt AES-ECB
       cpop1  read_output %c4, %c6 ! retrieve output data");

  asm(" mov    %0, %%l2" : : "r" (dout));
  asm(" std    %c4, [%l2]           ! store upper word output
       std    %c6, [%l2+8]         ! store lower word output") ;
}
```

(c)

Figure 4. Rijndael acceleration transparency code. (a) Java interface calling native C (b) Native C forwards call to co-processor interface. (c) Co-processor interface forwards call to co-processor.

- **Functional Model.** The functional model models the entire ThumbPod financial protocol on a PC environment (Pentium processor) in Java. As shown in Figure 5a, this model includes a Rijndael encryption function performed in Java. A C function is also utilized to perform fingerprint verification signal processing. A C function rather than Java is used here in order to incorporate the NIST standard fingerprint detection algorithms given in C code [13]. This function interfaces with the application via JNI. Communication between modules (ThumbPod, register, and authentication server) is performed in a sequential main method.
- **Benchmarking Functional Model.** In this abstraction level in Figure 5b the Rijndael function is accelerated as a C function for benchmarking purposes. An interface is constructed which allows the C Rijndael function to interface with the application via JNI. Rijndael performance measurements are compared with the functional model.
- **Transaction Level Model.** In this abstraction level the communication between modules is modified to allow objects to communicate with one another in a transaction level manner, instead of being controlled by a sequential main method. The transaction-level applications communicate to one another via socket programming models.

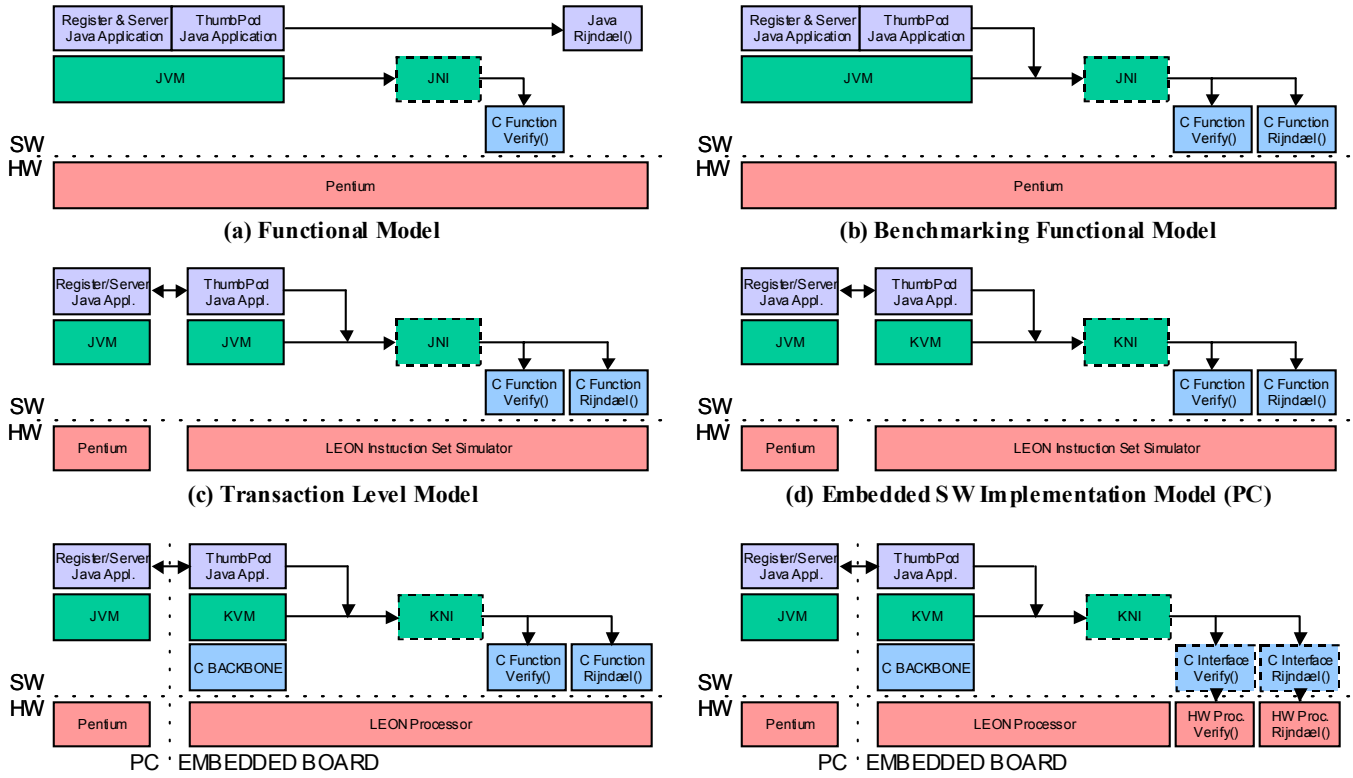

Figure 5. Design flow abstraction levels.

- **Embedded SW Implementation Model (PC).** Since the goal of the project is to implement the ThumbPod on an embedded hardware platform, the next abstraction level is the embedded software implementation model. In this model, the ThumbPod application operates on KVM (an embedded virtual machine) rather than JVM, and communicates with the accelerated C functions through a customized KNI (JNI for KVM) interface, rather than a standard JNI interface. In this model the effects of the constrained embedded environment can be ascertained.
- **Embedded SW Implementation Model (Board).** In this abstraction level, the ThumbPod application is moved entirely onto an embedded hardware platform. At this time the application runs on top of KVM operating on a C backbone on a LEON 32-b Sparc processor (FPGA). The acceleration continues to be performed in C. The FPGA board communicates with the PC via a UART and Java server proxy.
- **Embedded HW/SW Implementation Model.** In this abstraction level, hardware acceleration is introduced both for biometric signal processing and for Rijndael encryption. The hardware co-processors (implemented within an FPGA) interface with the Java application via a C interface and KNI. This abstraction level demonstrates the applicability and performance of HW/SW acceleration transparency.

4. EMBEDDED HARDWARE PLATFORM

Figure 6 illustrates the prototype architecture of our concept demonstrator. The software architecture is built upon an embedded Java virtual machine (KVM) which has been extended with appropriate platform specializations. The KVM executes on top of a LEON Sparc processor [10], which in turn is configured as a soft-core in a Virtex XC2V1000 FPGA. Thus our prototype has three levels of configuration: Java, C and hardware. The prototyping environment is an Insight Electronics development board, which contains besides the FPGA also a 32 MByte DDR RAM.

The LEON/Sparc core provides two interfaces: a high-speed AMBA bus interface (AHB) and a co-processor interface (CPI). Each interface has specific advantages toward domain-specific co-processors. The CPI offers an instruction- and register-set that is visible from within the Sparc instruction set, and allows a close integration of a domain-specific processor and the Sparc. The AMBA bus requires mapping of a co-processor through the abstraction of a memory interface. The CPI provides two 64-bit data ports and a 10-bit opcode port.

The high speed AMBA bus contains a memory interface and a bridge to the peripheral bus interface (APB). The memory interface includes an interface to a 32 MByte DDR RAM memory. The AMBA peripheral bus (APB) contains the fingerprint processor and two UART blocks. One connection is used to attach a fingerprint sensor, while the second one is used to connect an application server. This server is used to download and debug applications, as well as to experiment with the security protocol.

Table 2 indicates relevant gate counts and memory footprints for the application under development.

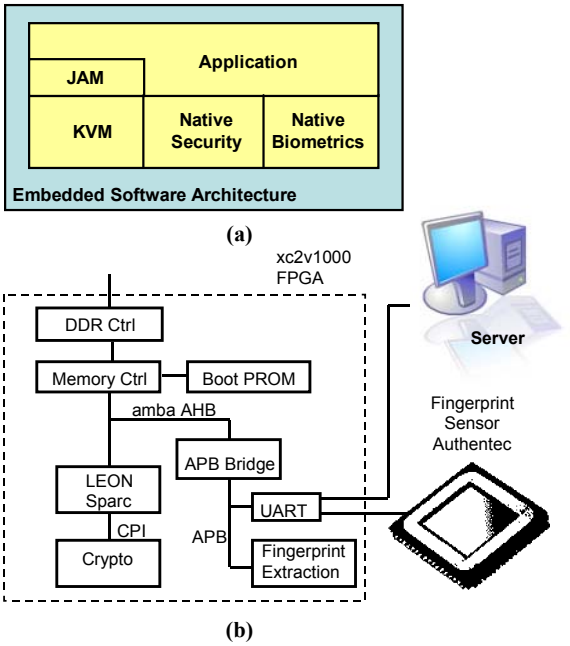

Figure 6. Prototype architecture.

Table 2. Memory Footprints and CLB Counts.

		Unit
Embedded Java VM	339	Kbyte
Pure Java Rijndael	11311	byte
Java Rijndael calling native	530	byte
Pure C Rijndael (Leon)	39488	byte
C Rijndael calling native	8432	byte
LEON processor	4810	Virtex2 LUT
AES co-processor	2197	Virtex2 LUT

5. PRIOR ART

Interfaces have been the cornerstone of several different design methodologies. Interface-based design [6] introduces a clear separation between communication and behavior in a design with the goal of easier design verification and refinement. This idea is also followed by communication-based design [8] where the objective is to create a manageable communications architecture in a system-on-chip. It has also been shown with interface synthesis [5] that interface refinements can be done automatically.

In this work we use interfaces as the driving point of refinement that move an application, which is initially described a high abstraction level, onto an embedded target. In coping with highly complex systems, we prefer to keep as much of the design work as possible in the higher abstraction levels. We refine only the parts which violate system specifications in terms of efficiency. The result is a layer of service interfaces that are specialized to a particular application. TinyOS [7] takes a similar approach to specialization, but works bottom-up. Support for top-down design is important as well.

Accelerator design at the instruction-set architecture level has shown to yield promising results [12]. Our work demonstrates that acceleration has an even wider span, and that it is possible to refine a single specification smoothly into a heterogeneous target architecture.

6. CONCLUSIONS AND ACKNOWLEDGEMENTS

This paper introduced the ThumbPod secure embedded system and described a design flow for the system. The concept of HW/SW acceleration transparency in relation to Java acceleration was introduced. HW/SW acceleration transparency is a systematic method of accelerating Java functions in software and hardware. Its two basic principles are Java function emulation and interface transparency. A design flow example of the HW/SW acceleration process was presented, showing C acceleration yields a 6.8X performance gain, and hardware acceleration yielding a 333X performance gain compared to a pure Java solution. A brief description of the embedded prototype architecture was also presented.

The authors would like to thank the anonymous referees for their comments. This work has been supported by the Fannie and John Hertz Foundation (DH), a DAC 2002 Graduate Scholarship (PS), NSF Grant #0098361, and UC Micro #02-079. We also thank Gaisler Research for providing the LEON-2 Sparc Core and for support in setting up the simulation environment.

7. REFERENCES

[1] http://www.ivgroup.ee.ucla.edu/thumbpod
[2] http://www.nist.gov/aes
[3] http://www-106.ibm.com/developerworks/java
[4] H. Kuo, P. Schaumont, and I. Verbauwhede, “A 2.29 Gbits/sec, 56 mW non-pipelined Rijndael AES encryption IC in a 1.8 V, 0.18 um CMOS technology,” *Proc. 2002 Custom Integrated Circuits Conference*, pp. 147-50, May 2002.
[5] S. Vercauteren, B. Lin, and H. De Man, “Constructing application-specific heterogeneous embedded architectures from custom HW/SW applications,” *Proc. 33rd DAC*, pp. 521-6, June 1996.
[6] J. Rowson and A. Sangiovanni-Vincentelli, “Interface-based design,” *Proc. 34th DAC*, p.178-83, June 1997.
[7] D. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo, “A network-centric approach to embedded software for tiny devices,” *EMSOFT 2001 (First International Workshop on Embedded Software)*, Oct. 2001.
[8] M. Sgroi *et al.*, “Addressing system-on-a-chip interconnect woes through communication-based design,” *Proc. 38th DAC*, pp 667-72, June 2001.
[9] T. Callahan, J.Hauser, and J. Wawrzynek, “The Garp architecture and C compiler,” *IEEE Computer*, April 2000.
[10] J. Gaisler, The LEON2 IEEE-1754 (SPARC V8) Processor, http://www.gaisler.com.
[11] P. Schaumont, I. Verbauwhede, “Domain-specific codesign for embedded security,” *IEEE Computer*, pp. 68-74, April 2003.
[12] S. Ravi, A. Raghunathan, N. Potlapally, M. Sankaradass, “System Design Methodologies for a Wireless Security Processing Platform”, Proc. 2002 Design Automation Conference, pp. 777-782, June 2002.
[13] http://www.itl.nist.gov/iad/894.03/databases/defs/nist_nfis.html