Software Architecture Exploration for High-Performance Security Processing on a Multiprocessor Mobile SoC *

Divya Arora†, Anand Raghunathan‡, Srivaths Ravi‡,
Murugan Sankaradass‡, Niraj K. Jha† and Srimat T. Chakradhar‡
†Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08544
‡NEC Laboratories America, Princeton, NJ 08540
divya@princeton.edu, anand@nec-labs.com, sravi@nec-labs.com,
murugs@nec-labs.com, jha@princeton.edu, chak@nec-labs.com

ABSTRACT

We present a systematic methodology for exploring the security processing software architecture for a commercial heterogeneous multiprocessor system-on-chip (SoC) for mobile devices. The SoC contains multiple host processors executing applications and a dedicated programmable security processing engine. We developed an exploration methodology to map the code and data of security software libraries onto the platform, with the objective of maximizing the overall application-visible performance. The salient features of the methodology include (i) the use of real performance measurements from a prototyping board that contains the target platform to drive the exploration, (ii) a new data structure access profiling framework that allows us to accurately model the communication overheads involved in offloading a given set of functions to the security processor, and (iii) an exact branch-and-bound based design space exploration algorithm that determines the best mapping of security library functions and data structures to the host and security processors.

We used the proposed framework to map a commercial security library to the target mobile application SoC. The resulting optimized software architecture outperformed several manually-designed software architectures, resulting in upto 12.5X speedup for individual cryptographic operations (encryption, hashing) and 2.2X-6.2X speedup for applications such as a Digital Rights Management (DRM) agent and Secure Sockets Layer (SSL) client. We also demonstrate the applicability of our framework to software architecture exploration in other multiprocessor scenarios.

Categories and Subject Descriptors: D.2.11 [**Software Architectures**]: Domain-specific architectures
General Terms: Security, performance
Keywords: Software partitioning, computation offloading

1. INTRODUCTION

Rapid advances in low-power computing, communications, and storage technologies continue to broaden the horizons of mobile devices, such as cell phones and personal digital assistants (PDAs). As the use of these devices extends into applications that require them to capture, store, access, or communicate sensitive data, (*e.g.*, mobile e-commerce, financial transactions, acquisition and playback of copyrighted content,*etc.*) security becomes an immediate concern. Left unaddressed, security concerns threaten to impede the deployment of new applications and value-added services, which is an important engine of growth for the wireless, mobile appliance, and semiconductor industries. For example, according to a survey of mobile appliance users [1], 52% cited security concerns as the biggest impediment to their adoption of mobile commerce.

A large body of work has addressed the design of specialized hardware for efficient (high performance and/or low power) security processing, through techniques that range from custom hardware accelerators to instruction set extensions for general-purpose microprocessors, and hardware support for security processing is starting to appear in commercial SoCs for mobile appliances [2, 3]. While hardware enhancements go a long way towards addressing efficient security processing, they typically only address the core cryptographic algorithms, leaving a substantial amount of functionality to be implemented in software, *e.g.*, support for various modes of encryption/decryption, protocol processing and packet operations, key generation and management, *etc.* Comprehensive security libraries, which provide support for all the above functions, are quite complex. For example, the commercial library considered in our work contains over 130K lines of code and over 90 application programming interface (API) functions. Moreover, different mappings of the library onto the target platform can give vastly different speed-ups: a test program computing a hash (based on SHA-1) of 1KB of data took 87.8s, 43.4s, and 2.2s, for three different mappings of the library onto the multiprocessor SoC platform used in this work. Therefore, efficient security processing also requires optimized mapping of the security library to the target hardware platform, a problem that has thus far received little attention.

Paper overview: In this paper, we focus on the problem of domain-specific software architecture exploration for security processing on a state-of-the-art heterogeneous multiprocessor SoC for mobile appliances. The SoC contains multiple "host" processors that execute applications, which can "offload" security processing operations onto a dedicated programmable security processing engine. It has been fabricated by NEC, and will be incorporated in mobile phones in 2006. We are given a commercial crypto-processing software library that contains software implementations of various cryptographic algorithms, protocol and packet processing operations, and key management functions. Software architecture exploration in this context refers to the problem of mapping the code and data of the library onto the host processor and security processing engine, with the goal of optimizing the performance of applications that use the library. We present a systematic methodology to perform this task. The methodology is driven by accurate performance measurements from a prototyping platform that give us the execution times of functions in the library when executed on the host processor as well as the security processing engine. Recognizing that interprocessor communication overheads (for synchronization and

*This work was supported by NSF under Grant No. CCR-0326372.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2006, July 24–28, 2006, San Francisco, California, USA.
Copyright 2006 ACM 1-59593-381-6/06/0007 ...$5.00.

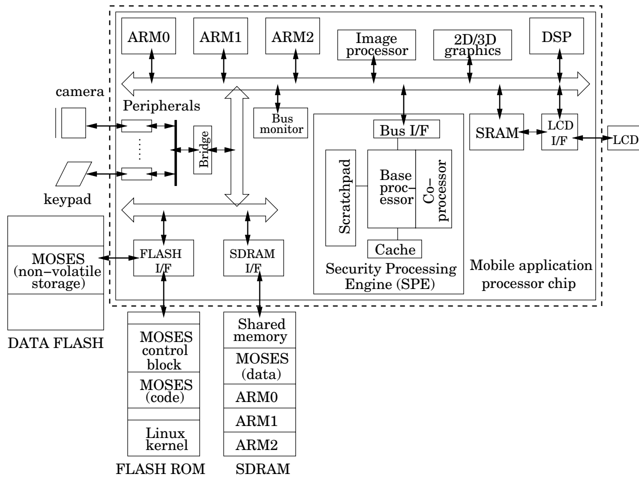

Figure 1: MP211 architecture

data transfer) significantly impact performance, we developed a data structure access profiling framework that allows us to accurately model the communication overheads involved in offloading any given set of functions to the security processor. We formulate software architecture exploration as a partitioning problem and use an exact branch-and-bound algorithm, with suitable branching heuristics and bounding criteria, to determine the best mapping of security library functions to the host and security processors, as well as the best mapping of data structures to the memories of these processors.

Related work: We briefly mention related work in the context of efficient security processing and domain-specific design methodologies for security. Researchers have proposed a wide range of hardware architectures to accelerate security processing. Highly efficient hardwired (custom hardware) implementations of symmetric encryption [4, 5], hash [6], and public-key algorithms [7, 8] have been extensively explored. Recognizing the evolving nature of cryptographic algorithms and security protocols, and the wide range of algorithms that need to be supported in some systems, application-specific programmable processors [9,10] as well as general-purpose instruction set extensions [11,12] have been proposed to accelerate security processing. With security emerging as an important concern in many embedded systems, domain-specific design methodologies that address security in various stages of the design process have also been proposed [13,14].

2. BACKGROUND

This section presents an overview of the mobile application SoC and the crypto-processing library for which this study was conducted.

Target platform: Fig. 1 depicts the high-level block diagram of the MP211 multiprocessor SoC for mobile devices. The SoC contains three "host" (ARM926EJ-S) processors that run the Montavista Linux kernel and execute pre-installed and downloaded applications, a digital signal processor (DSP) for multimedia processing, a graphics engine, and various accelerators and interfaces to peripherals and off-chip memories (FlashROM for code, SDRAM for data, and R/W Flash for non-volatile data storage). In addition, the platform includes MOSES (**MO**bile **SE**curity processing **S**ystem) – a security processing architecture that provides for secure (tamper-resistant) and efficient (high performance, low power) execution of security processing functions.

MOSES was developed to meet the security challenges in emerging mobile appliances, including 3G/4G mobile phones and PDAs. From a hardware perspective, it comprises three key components - a Security Processing Engine (SPE), a hierarchical secure memory sub-system, and a security-enhanced communication architecture. The SPE is a domain-specific programmable processor on which security computations are performed once they are offloaded from one of the host ARM processors onto MOSES. The SPE consists of a 32-bit, 5-stage pipelined RISC core along with a tightly-coupled co-processor that implements custom instructions that can be used to accelerate a wide range of cryptographic algorithms, including symmetric encryption (DES, 3DES, AES, RC4, *etc.*), hashing (MD5, SHA-1), and public-key algorithms (RSA, Diffie-Hellman, DSA, Elliptic Curve, *etc.*). Some custom instructions are dedicated (*i.e.*, used for a specific cryptographic algorithm), while others are shared across multiple algorithms. A programmable processing engine can execute not only core cryptographic algorithms, but also additional functions such as protocol primitives or key management, leading to higher offloading efficiency (lower workload for the host processors) when compared to simple hardware accelerators.

The SPE contains a small amount of on-chip scratchpad memory which includes a non-volatile ROM for the bootloader, device keys, and instruction and data RAMs for cryptographic firmware and critical data, respectively. Portions of off-chip SDRAM and FlashROM are reserved for the SPE and guarded by a security-enhanced communication architecture, which monitors communication traffic on the system bus and ensures that only the SPE is allowed to access the reserved areas. Off-chip non-volatile data storage secured by encryption and hashing is also available to the SPE.

Overview of the crypto-library: A software library, CLib (**C**rypto-**Lib**rary)[1], is installed on the platform for applications requiring security services. Applications can perform various cryptographic operations by invoking the appropriate library functions via the exported interface. CLib is a complex, multi-layered software library with about 90 API functions, over 500 internal functions, and more than 130K lines of C code. The layered architecture enables isolation between data structures accessed by different layers, and also hides unnecessary implementation details from the user program. The topmost layer interfaces with the application while the lowest layer performs core cryptographic operations. The intermediate layers provide support for different operating systems (OSs), parse/decode application commands and pre-process arguments before passing control onto the cryptographic kernel, which then performs the requested operation and returns the results and status of execution to the calling application.

CLib supports a wide range of cryptographic algorithms including symmetric key algorithms (DES, 3DES, AES, RC4), one-way hash functions (MD5, SHA-1), public key algorithms (RSA, Diffie-Hellman, Elliptic curve), and digital signature algorithms (DSA, RSA, Elliptic curve). In addition, the library includes a comprehensive key management infrastructure with primitives for key generation, distribution, storage, use, and destruction. It supports packet processing routines to enable protocols such as IPSec, and also implements routines for large integer arithmetic, modular arithmetic, *etc.*

3. PROBLEM STATEMENT AND MOTIVATION

The objective of this work is to develop a systematic methodology to investigate various possible mappings of the security library CLib onto the heterogeneous multiprocessor platform and find a mapping that optimizes performance. A mapping of the library specifies, for each processing unit (host processor and MOSES [2]), a subset of library functions that it executes and data structures that reside in its memory. As described previously, the host processor is an ARM926EJ-S CPU running at upto 200MHz (operating frequency and voltage are regulated at run-time for power savings) executing the Montavista

[1] name changed to withhold vendor identity

[2] In the rest of the paper, the term MOSES is used to refer to the MOSES SPE.

Linux OS, while MOSES runs at 100MHz (for lower power) and directly executes security firmware without an OS.

Execution model: The software architecture follows a client-server model wherein security processing functions are offloaded from applications running on the host ARM processor onto MOSES. Applications (clients) link to a *stub library* that exposes CLib's API. The CLib library software is divided into two parts – a *front end* that executes on the host processor and interacts with the applications, and a *back end* which comprises all the functions executing on MOSES. To facilitate communication (synchronization and data transfer) between the front-end and back-end, communication drivers are implemented on both the host processor and MOSES. The communication driver on the host processor is implemented as a device driver kernel module that is part of the Linux OS.

The stub library is responsible for disambiguating requests from different applications and isolating their data from one another. Additionally, it copies the arguments of the function to be executed into a global shared memory region. During this process, any pointers embedded in the copied data structures need to be translated to be valid in the memory space of MOSES (note that applications executing on the host CPUs typically use virtual addresses since they execute under an OS, whereas software executing on MOSES uses physical addresses). Finally, the stub library makes a call to a routine in the front-end, which may execute some part of the CLib library, and offloads the remainder onto MOSES. The offloading process is initiated by signaling MOSES (using an interrupt) that it needs to process a request. The call from the stub library into the MOSES front-end has blocking semantics, much like a function call to a software cryptographic library running on the host processor itself. The back-end reads the command to be performed and its operands from the shared memory, performs the appropriate computation, places the result back in the shared memory, and interrupts the host processor to indicate that the computation is complete. The interrupt service routine activates the front-end, which returns control to the application. The host ARM processor simply waits for control to return from MOSES. It may switch to another application in the meantime, or execute an independent thread of the same application that does not depend on the result of the security operation.

A few points bear elaboration at this juncture. Custom instructions of MOSES accelerate parts of crypto-algorithms and are capable of giving speedup at the core cryptographic level, *i.e.*, they accelerate cryptographic subroutines. However, this does not translate directly into application-level speedup because of additional factors such as disparate frequencies of ARM and MOSES (the former runs at roughly twice the speed), and the communication and synchronization costs involved in offloading computations. Depending on these factors, different mappings of CLib onto the given platform can give vastly different speed-ups. The obvious mapping, putting all functions that use custom instructions on MOSES, is not always the best option, as it may involve excessive communication overhead. The size of the CLib library, along with its elaborate data structures and complex interactions between them, makes manual analysis infeasible. The aforementioned factors, in conjunction with the substantial time required to manually implement and verify a sample partition (about 6-9 man-months), present a compelling case for an automated and efficient approach to explore the vast design space, accounting for the platform's characteristics accurately.

Software partitioning for hardware-software systems or multiprocessors has been studied extensively [15]. We do not attempt to propose a general-purpose partitioning methodology here. Rather, we formulate the problem of software architecture exploration for security processing on the MP211 platform as a partitioning problem and propose a domain-specific methodology to solve it. Some of the constraints and trade-offs are specific to this platform and this fact is reflected in the methodology. For example,

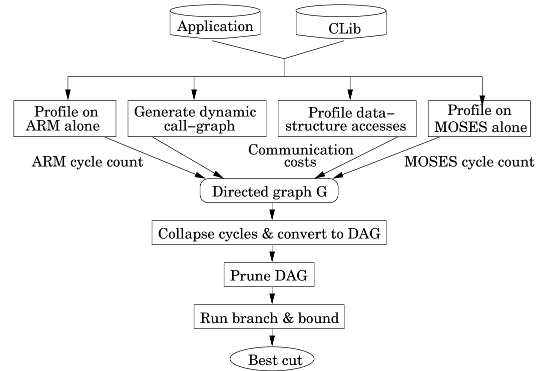

Figure 2: Design flow for mapping CLib onto the given platform

- The target hardware platform and software library are fixed. The number, kind, processing capabilities and interfaces of various processors on the SoC are fixed. This allows us to generate accurate execution time profiles of the security library running on both the host processor and MOSES, for use in partitioning.
- We propose a methodology for accurate communication cost estimation between functions executing on different processors. For this, we have developed a tool for detailed *data-structure access profiling* of programs, that records the variables and their attributes (*e.g.*, type, size, *etc.*) accessed by each function of the program.
- We develop domain-specific techniques that enable us to use an exact branch-and-bound algorithm to solve the partitioning problem. In contrast, using exact algorithms may be infeasible in general, given the NP-Hard nature of the partitioning problem [15].

4. EXPLORATION METHODOLOGY

This section details our methodology to map the given software library onto a combination of one ARM processor and MOSES. The result is in the form of a specification of functions, and global data structures *of the library* that should be placed on the instruction and data RAMs of MOSES. The main idea underlying our technique is the use of dynamic profiling to guide static partitioning. Three kinds of profiles are required for the above:

- Function cycle counts: This profile contains the self cycle counts for each function while running on ARM alone and MOSES alone. It excludes any time spent in the function's descendants.
- Dynamic function call graph: This profile captures the caller-callee relationships between various CLib functions and the number of times a particular call was made during a run.
- Data structure access profile: This profile represents the data memory accesses by each function at the granularity of the data structure accessed. This information is used to estimate data transfer costs in case the function in question is offloaded to MOSES. We developed a tool, DTrack (**D**ata **Track**er) to perform this profiling, which is described in the following subsection.

We formulate the given problem as a modified graph partitioning problem and feed the data extracted from the above profiles to it. Subsequently, a highly optimized branch-and-bound search algorithm is used to step through the search space and determine the optimal mapping.

The design flow described above is depicted in Fig. 2. The figure includes additional steps to detect and remove cycles in the graph extracted from a dynamic call graph profiler and to prune the resulting directed acyclic graph (DAG). The reasons for these steps will become clear in the following sections.

4.1 Data structure access profiling

We developed a tool, `DTrack`, within the Valgrind [16] framework to collect accurate data structure access profiles. Valgrind is an open-source tool suite for profiling and debugging Linux programs. It includes tools for memory error detection, and cache and heap profiling. It provides an extensible framework with an x86 interpreter at its core that permits developers to write their own profiling tools.

Our tool, `DTrack`, performs two mappings – it maps the program counter of the executing instruction to the C subroutine that it belongs to, and the data memory address (if any) to the program variable that it corresponds to. For this, x86 instructions are instrumented to perform appropriate bookkeeping operations. The source (application + `CLib`) is compiled normally, with the `‘‘-gstabs’’` option, producing a binary augmented with debugging information.

Valgrind takes control of the program before it starts, reads in debugging information from the executable and associated libraries, and runs it on a synthetic CPU provided by the Valgrind core. At the heart of Valgrind is a Just-In-Time compiler that compiles and translates each basic block of the x86 code into an internal format, and hands over the translated code to the selected tool (`DTrack`, in this case) which returns the instrumented code that is actually executed.

`DTrack` performs a number of instrumentations. It instruments call and return instructions to map instruction addresses to source function names and maintain a copy of the current call stack. Additionally, it intercepts calls to C library memory management functions, such as *malloc, calloc, realloc, free*, *etc.*, and maintains an internal list of dynamically allocated memory chunks. For each chunk, it stores the address, size, and name of the function that invoked the allocation subroutine. Next, `DTrack` intercepts each load/store instruction and checks if the target address falls within the data range (address, address+size) of a program variable. It adds the variable along with its associated attributes to the *access list* of the currently executing function. The access list is a per function record of data structures accessed by the function along with their access counts. First, the address is matched against the global symbol table (read in the beginning) and the symbol name and size are recorded in case of a match. In case of a failure, the address is compared against addresses in the list of dynamically allocated chunks, and the chunk's information added to the function's access list. If both the above checks fall through, the profiler walks up the program stack to determine the function in whose stack the accessed variable resides. Then, it maps the offset of the accessed variable (address – stack start address) to the stack offsets of variables belonging to the function. A stabs reader extracts the above data from the debugging information present in the binary. The profiler records the variable size and name of the function in which it lies.

The operation of `DTrack` is illustrated in Fig. 3. Note that stack and heap variables embody a notion of an "owner" – the function that is responsible for their allocation. For stack variables, this definition is intuitive as memory for a local variable is allocated on function invocation and will therefore be in the same memory space as the function stack. We apply the same idea to dynamic memory, *i.e.*, memory allocated by a function f is *owned* by f and resides in the memory space of the same processor as the code of f.

The results of the `DTrack` tool can be used to estimate the communication overheads involved in offloading a function from the host processor to `MOSES`. In order to do this, we can consider two possible models for copying data structures – lazy, and eager. In the lazy model, data structures are copied from ARM to `MOSES` the first time they are accessed. On the other hand, in the eager model, all data structures needed for a function and its descendants are copied to `MOSES` before the function is offloaded.

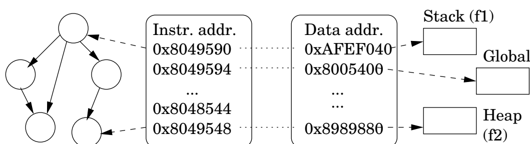

Figure 3: Data structure access profiling

4.2 Formulation

Next, we present the problem formulation.

Execution cost graph: The dynamic function call graph is augmented to generate the *execution cost graph* $G(V+N', E+E'+E'')$, where V is the set of functions in the graph, N' is the set of global variables accessed during the run, E is the set of call graph edges, and E' and E'' are two different sets of "access edges". An edge $e(p,q) \in E \mid p,q \in V$ implies that function q is called from function p. An edge $e(p,q) \in E' \mid p,q \in V$ implies that function q is not a direct successor of p but accesses a data structure *owned* by p. This can happen in several ways – a variable is passed by reference through a sequence of function calls, address of a local/heap variable is stored in a global pointer that is dereferenced later, *etc.* Edge $e(n,q) \in E'' \mid q \in V, n \in N'$ indicates that function q accesses global variable n.

For each $v \in V$, $cyc_a(v)$ and $cyc_m(v)$ denote the execution cycles of the function on ARM and `MOSES`, respectively, and $count(v)$ denotes the number of times v is called during the run. Note that execution cycles on `MOSES` are scaled appropriately to equivalent ARM cycles as the two processors run at different frequencies. Each edge in the graph is associated with one or more parameters that represent data transfer along the edge. The actual communication cost in processor cycles is obtained by scaling the total number of bytes transferred by an empirically measured parameter α. For each $e \in (E \cup E')$, the number of bytes transferred along e is $wt(e) = \sum_{i=0}^{i=e_N} ac_i * sz_i$, where e_N = number of parameter instances copied along e, ac_i = access count of parameter instance i during the run, and sz_i = size of parameter instance i. The size of the passed parameter may vary from one invocation of edge e to another (*e.g.*, arrays of different sizes may be passed for different call instances). This is accounted for by labeling each of these as a different parameter instance in the `DTrack` tool's output. For $e(n,q) \in E''$, the associated data transfer is $wt(e) = ac_e(n) * sz(n)$, where $sz(n)$ is the size of n, and $ac_e(n)$ is the access count for n along e. Fig. 4(a) gives an example of a cost graph.

Cost function: Given the above graph, the objective of the partitioning algorithm is to find a mapping of all nodes to ARM or `MOSES`, minimizing total execution cost. A solution to the problem is an array sol_i, $0 \leq i < |V| + |N'|$. $sol_i = 0$ if i is mapped to ARM else $sol_i = 1$. Let fn_A and fn_M be the sets of functions mapped to ARM and `MOSES`, respectively, and let g_A be the set of global variables mapped to ARM. The execution cost of a solution is given by

$$Sol_{cost} = ARM_{cost} + MOSES_{cost} + E_{cost} + E'_{cost} + E''_{cost}$$

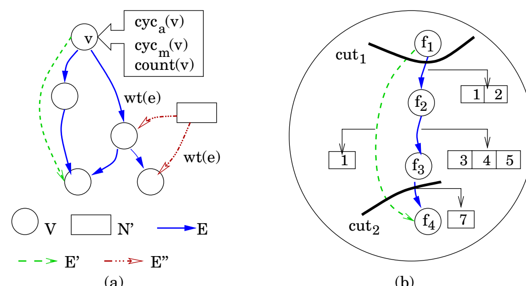

Figure 4: (a) Cost graph $G(V+N', E+E'+E'')$ and (b) E''_{cost} calculation

where, $ARM_{cost} = \sum(cyc_a(v) \times count(v))\ \forall v \in fn_A$,
$MOSES_{cost} = \sum(cyc_m(v) \times count(v))\ \forall v \in fn_M$,
$E_{cost} = \sum wt(e(p,q))\ \forall e \in E, p \in fn_A, q \in fn_M$,
$E'_{cost} = \sum wt(e(q,n))\ \forall e \in E', q \in fn_M, n \in g_A$, and E''_{cost} is computed as follows. G is traversed in topological order. For each function node $v \in fn_M$, the program marks off variables associated with all edges, $e(u,v) \in E\ |u \in fn_A$. Next, the program looks at all edges $e(v,u) \in E''\ |u \in fn_A$ and traverses the parameter list associated with them. If a parameter i is unmarked, it is marked and its cost $ac_i * sz_i$ is added to E''_{cost}. Thus, edges belonging to E'' represent a *conditional copy*, *i.e.*, data are copied along them only if they have not been copied into the MOSES memory by an ascendant of the function which also resides on MOSES. *E.g.*, in Fig. 4(b), function f_4 accesses variable 1 owned by f_1, but is not a direct descendant of f_1. In the evaluation of cost for cut_1, the cost of $e(f_1, f_4) \in E'$ should not be included (since 1 is copied when f_1 calls f_2). However, in the cost for cut_2, the cost of this edge has to be accounted for.

Mapping constraint: Any solution of the above problem must satisfy the following constraint: *If a node $u \in V$ is mapped to* MOSES*, all nodes $v \in V$ in the subgraph rooted at u are also mapped to* MOSES. Due to the above constraint, all function nodes lying on a cycle in the call graph have to be collapsed to a single node. Thus, G is converted to a DAG. This DAG is pruned to eliminate certain functions that have to be on ARM (*e.g.*, routines involved in setting up *main()*) or have to be on both processors (*e.g.*, C library functions).

Procedure 1 Branch and bound algorithm for finding optimal cut

```
1: Inputs: Cost graph G(V + N', E + E' + E'')
2: Output: sol_i, ∀i ∈ [0, |V| + |N'| − 1], sol_i=0 if i is on
   ARM else 1
3: best_cost ← INFINITY, best_solution ← sol_i = X ∀i
4: add_to_list(L_ps, best_solution)
5: repeat
6:    PS ← select_partial_solution(L_ps)
7:    i ← select_unassigned_node()
8:    Generate children P^0 = P^1 = PS, P^0_i = 0, P^1_i = 1
9:    for all children P^j do
10:      LB^j ← lower_bound(P^j)
11:      if P^j is complete then
12:         if is_feasible_solution(P^j) and LB^j < best_cost
            then
13:            best_solution ← P^j
14:            best_cost ← exact_cost(P^j)
15:         return
16:      if LB^j < best_cost then
17:         add_to_list(L_ps, P^j)
18:      else
19:         return
20: until (PS)
```

4.3 Branch-and-bound algorithm

Given the formulation described above, nodes in G are assigned to either ARM or MOSES using an exact branch-and-bound search algorithm. The algorithm is described in Procedure 1. It has been optimized to match problem-specific constraints and enables us to map large graphs (of the order of hundreds of nodes) in reasonable time.

5. EXPERIMENTAL RESULTS

In this section, we describe the application of our methodology and its impact on performance in the context of the Clib cryptographic library and MP211 multiprocessor SoC. We also show the applicability of our framework to other multiprocessor scenarios.

As mentioned before, the mapping algorithm requires three profiles: function cycle counts, dynamic function call graph

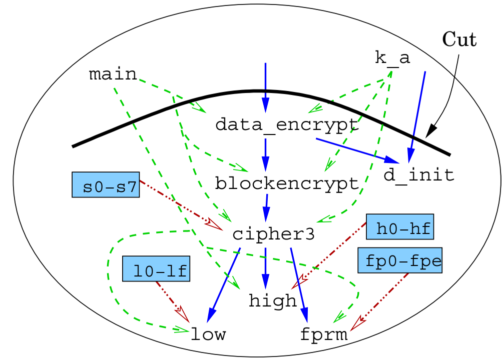

Figure 5: (a) Call graph fragment showing optimal mapping for 3DES

and data structure access profile. Of these, the last profile was generated by our tool DTrack running on Red Hat Linux. The dynamic function call graphs were generated using IBM's performance profiler – Quantify [17]. Quantify reports the exact number of calls for each caller-callee pair, function cycle time, system call time, *etc.*, for a simulated processor.

Function cycle count measurements were performed on an evaluation board that includes the MP211 application SoC and interfaces to PCs/workstations for programming and debugging. The OS running on the ARM processor is Montavista Linux. ARM's RealView Suite running on a PC is used to program the board, as well as to download the OS and MOSES firmware into the FlashROM. ARM profiles were generated with GNU gprof ported to ARM. gprof dumps out a list of functions in decreasing order of time spent by the application in each function. NEC's in-house in-circuit emulator tools are used for monitoring and profiling the execution of software on MOSES.

Library API-level performance: We created testbenches to use different CLib API calls and thus exercise different parts of CLib. These testbenches were run under all the profiling utilities listed above with varying data sizes and data values. Specifically, we included symmetric encryption algorithms 3DES, DES, AES, and cryptographic hash algorithms MD5 and SHA-1 [18]. Each CLib API call was placed in a loop of 50,000 iterations.

Fig. 5 shows a fragment of the call graph for the testbench performing 3DES encryption that corresponds to the optimal mapping (functions and global variables to be placed on MOSES) obtained from our evaluation. There are two degenerate solutions for partitioning – a) running the entire library on ARM alone and b) offloading onto MOSES only the functions that use custom instructions, which also happen to be near leaf-level functions in the dynamic function call graph. We refer to these two solutions as S_{ARM}, and S_{LEAF}, respectively. In the case of 3DES, S_{LEAF} includes functions **low, high, fprm** and **cipher3**. The optimal solution S_{OPT} is not the same as the intuitive solution S_{LEAF} and includes some functions farther away from the leaf nodes. In fact, S_{LEAF} performs worse than S_{ARM} due to the excessive communication costs involved. This corroborates the need for systematically exploring the partition space. Another point to note is that because of our mapping constraint, some functions, such as d_init (see Fig. 5), that actually perform worse on MOSES are still executed on it because of the execution time savings obtained from executing one of their parents in the call graph on MOSES.

Table 1 summarizes the performance of mappings obtained for the benchmarks. Column 2 lists the number of nodes in the DAG. Column 3 refers to the number of feasible cuts in the pruned DAG and column 4 reports the CPU time taken by the branch-and-bound algorithm (running on Red Hat Linux on a 2.6GHz Pentium) to compute the optimal solution. Columns

Table 1: Evaluation of benchmarks on the target platform

Algorithm	# nodes	# cuts	B&B time (s)	S_{OPT}/S_{ARM}	S_{OPT}/S_{LEAF}	Mem. (Bytes)
3DES	107	2.24×10^6	6.45	0.14	0.08	4672
DES	112	3.58×10^7	62.01	0.33	0.08	4680
AES	102	5.92×10^5	2.12	0.48	0.09	8626
MD5	83	2.03×10^4	0.11	0.80	1	0
SHA-1	82	1.62×10^4	0.11	0.42	1	0

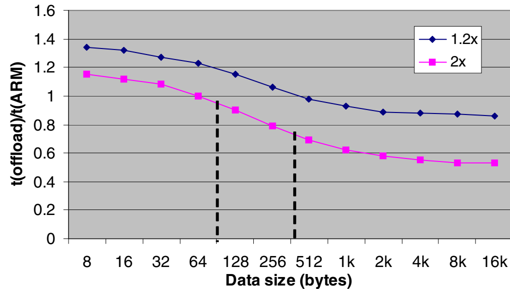

Figure 6: Performance impact of offloading computation with varying data size

5 and 6 report the execution time ratio of S_{OPT} with respect to S_{ARM} and S_{LEAF}, respectively. Column 7 summarizes the scratchpad memory requirements to store the global variables that have been mapped to `MOSES` in S_{OPT}. The performance benefits vary from algorithm to algorithm with a maximum improvement of 7X for 3DES and minimum of 1.25X for MD5. In MD5 and SHA-1, S_{OPT} is found to be the same as S_{LEAF}.

Application-level performance: We evaluated the performance impact of our technique on two applications – an SSL client/server and a DRM agent. The former is based on the OpenSSL (version 0.9.7d) implementation of the SSL protocol [19]. The OpenSSL library was instrumented to make calls to `CLib` and client/server programs were constructed to open an SSL connection over TCP/IP and send an encrypted file through it. The program uses key exchange and public key algorithms during SSL handshake and performs the bulk of data encryption and hashing in the SSL record protocol using 3DES and SHA-1, respectively. We ran the client and server using `CLib` whose functions and data were mapped to the two processors in accordance with results of the proposed exploration methodology, resulting in a 6.2X execution time reduction (*i.e.*, 6.2X data rate improvement) for the SSL client.

The DRM agent is used to download and play back protected video content according to the OMA DRM 2.0 specification [20]. We used a QVGA video file of size 5.31MB (320 × 240 pixels, 16 bits per pixel) in our experiments. The OMA DRM protocol involves RSA-based signature verification and uses AES and SHA-1 for content decryption and hashing, respectively [21]. The performance improvement in this case was found to be 2.2X.

Application to homogeneous multiprocessor systems: Our framework can be applied to other multiprocessor scenarios that involve selective offloading of computation. In this experiment, we simulate two ARM processors running in a master-slave mode, where the master processor offloads computation onto a slave processor when it becomes overloaded. Effectively, the slave runs at a higher speed than the master. Unlike the case of a crypto-processor where a specific set of functions were sped up (through hardware), here the execution time of all functions scales uniformly. Performance benefits obtained from offloading depend on a number of factors such as master processor load, communication costs between the master and slave (which is input data size dependent), *etc.* We ran 3DES on the simulated platform considering two cases where the slave processor runs 1.2X and 2X faster than the master. Accurate communication cost estimation makes it possible to determine the benefit of offloading for different input data sizes. The execution time ratio of offloaded execution to execution on the ARM itself is plotted for varying data sizes in Fig. 6. The figure indicates that the breakeven point for offloading security processing operations is 64B and 512B, respectively, in the two cases. Similarly, it is possible to determine, for a fixed input size, the workload level of the master processor at which offloading results in substantial savings in execution time.

6. CONCLUSIONS

In this paper, we presented a design methodology to map the code and data of a security processing library onto the given heterogeneous multiprocessor SoC. We based it on execution profile measurements of software running on a prototype board combined with accurate estimates of communication costs derived from detailed data structure access profiling. Our framework is fully automated. We demonstrated its efficiency and flexibility by using it to evaluate and optimize a commercial security processing library, resulting in notable performance improvements with respect to manually designed software architectures.

7. REFERENCES

[1] *ePaynews - Mobile Commerce Statistics.* `http://www.epaynews.com/statistics/mcommstats.html`.

[2] *OMAP Platform - Overview.* Texas Instruments Inc. (`http://www.ti.com/sc/omap`).

[3] S. Torii *et al.*, "A 600MIPS 120mW 70uA leakage triple-CPU mobile application processor chip," in *Proc. IEEE Solid-State Circuits Conf.*, Feb. 2005, pp. 136–138.

[4] T. Ichikawa, T. Kasuya, and M. Matsui, "Hardware evaluation of the AES finalists," in *Proc. 3rd AES Candidate Conf.*, Apr. 2000, pp. 279–285.

[5] I. Verbauwhede, P. Schaumont, and H. Kuo, "Design and performance testing of a 2.29 Gb/s Rijndael processor," *IEEE J. Solid-State Circuits*, pp. 569–572, Mar. 2003.

[6] A. Satoh and T. Inoue, "ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS," in *Proc. Int. Conf. Information Technology: Coding and Computing*, Apr. 2005, pp. 532–537.

[7] M. Shand and J. E. Vuillemin, "Fast implementations of RSA cryptography," in *Proc. IEEE Symp. Computer Arithmetic*, June 1993, pp. 252–259.

[8] C. K. Koc, "RSA hardware implementation," RSA Laboratories, Tech. Rep., Apr. 1996.

[9] J. Goodman and A. Chandrakasan, "An energy efficient reconfigurable public-key cryptography processor architecture," in *Proc. Int. Wkshp. Cryptographic Hardware and Embedded Systems*, Aug. 2000, pp. 175–190.

[10] S. Ravi, A. Raghunathan, N. Potlapally, and M. Sankaradass, "System design methodologies for a wireless security processing platform," in *Proc. ACM/IEEE Design Automation Conf.*, June 2002, pp. 777–782.

[11] R. B. Lee, Z. Shi, and X. Yang, "Efficient permutations for fast software cryptography," *IEEE Micro*, vol. 21, no. 6, pp. 56–69, Dec. 2001.

[12] J. Burke, J. McDonald, and T. Austin, "Architectural support for fast symmetric-key cryptography," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Nov. 2000, pp. 178–189.

[13] N. Potlapally, S. Ravi, A. Raghunathan, and G. Lakshminarayana, "Algorithm exploration for efficient public-key security processing on wireless handsets," in *Proc. DATE Designers Forum*, Mar. 2002, pp. 42–46.

[14] P. Schaumont and I. Verbauwhede, "Domain-specific codesign for embedded security," *IEEE Computer*, vol. 36, no. 4, pp. 68–74, Apr. 2003.

[15] G. De Micheli, R. Ernst, and W. Wolf, Eds., *Readings in Hardware/Software Co-design.* Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[16] Valgrind. `http://valgrind.org`

[17] IBM Rational software. `http://www-306.ibm.com/software/rational/`

[18] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C.* John Wiley and Sons, 1996.

[19] *Open SSL Project.* `http://www.openssl.org`.

[20] *DRM Specification V2.0.* `http://www.openmobilealliance.org`.

[21] D. Thull and R. Sannino, "Performance considerations for an embedded implementation of OMA DRM 2," in *Proc. Design Automation & Test in Europe Conf.*, Mar. 2005, pp. 46–51.