SOFTENIT: A Methodology for Boosting the Software Content of System-on-Chip Designs

Abhishek Mitra
Dept. of Computer Science
Univ. of California, Riverside
amitra@cs.ucr.edu

Marcello Lajolo
NEC Laboratories America
Princeton, NJ
lajolo@nec-labs.com

Kanishka Lahiri
NEC Laboratories America
Princeton, NJ
klahiri@nec-labs.com

ABSTRACT

Embedded software is a preferred choice for implementing system functionality in modern System-on-Chip (SoC) designs, due to the high flexibility, and lower engineering costs provided by software over hardware. With continuous improvements in embedded processor performance, many system functions, which have traditionally been implemented using dedicated hardware (such as those with real-time performance requirements), are becoming potential candidates for software implementation. For complex SoCs containing many different components, identifying such functions (or hardware blocks), and re-implementing them as embedded software, is a labor-intensive, manual, and error-prone process. In this paper we present techniques for the transformation of behaviors of selected hardware blocks into equivalent software implementations. In particular, we describe SOFTENIT, a methodology for "softening" of SoC hardware, that takes as input, a partitioned and mapped system description, and generates a modified system architecture in which the fraction of system functionality implemented using embedded software is significantly boosted. Application of this methodology to an IEEE 802.11 MAC processor design demonstrates that it can be used to generate new, "softened" system architectures, that yield large reductions in hardware complexity, while satisfying performance requirements, at very low computational cost.

Categories and Subject Descriptors

C.5.4 [**Computer System Implementation**]: [VLSI Systems]

General Terms

Design, Algorithms, Experimentation

Keywords

Embedded Systems, HW/SW Codesign, RTOS, Partitioning

1. INTRODUCTION

In many application domains, especially those affected by international standards, drastic changes to application behavior are relatively infrequent, due to the slow rate at which new standards (or revisions to existing ones) are approved and deployed. For example, the first revision to the encryption algorithms for IEEE 802.11 Wireless LANs occurred more than 5 years after their introduction [1], while for over a decade, MPEG-2 remains the most widely used technique for video compression. On the other hand, hardware implementations of these applications have demonstrated rapid and steady improvements in performance, silicon area, and power consumption. As a result, in such cases, the capabilities of the underlying hardware often exceed the requirements imposed by the applications. In such scenarios, it is crucial to effectively exploit improvements in semiconductor technology to reduce design cost, time-to-market, and improve design flexibility. A natural way to achieve this goal is to, over time, reduce the amount of application-specific hardware used in the system, and realize the same functionality using embedded software.

The gradual migration of application tasks from hardware to software is standard practice for design teams wishing to remain competitive under rapidly advancing technology. In an ideal scenario, this goal can be achieved by periodically revisiting the system specification, and making use of automatic HW/SW codesign tools to arrive at a system architecture that is optimized for the new technology. However, a unified high-level model of the system, and corresponding tool flows for system synthesis are rarely available. Hence, in practice, designers are forced to effectively re-design the system manually, starting with an informal specification of application functionality, leading to high design cost and turn-around-time. With rising system complexity, this task of manually fine-tuning the hardware/software boundary with every advance in semiconductor technology will require significant effort, making it important to identify systematic methodologies to help in part, automate this process. In this work, we consider an approach in which designers take advantage of the *existing design*, *i.e.*, the starting point is a partitioned and mapped system architecture, and *incrementally* modify it to exploit the capabilities of improved technology.

1.1 Paper Contributions and Overview

In this paper, we describe SOFTENIT, a methodology for semi-automatic "softening" of hardware components in System-on-Chip designs. The methodology performs incremental, software-biased, fine-grained *re*-partitioning of an existing design in order to boost the fraction of application functionality implemented using embedded software. The proposed techniques are applicable to system architectures that have been previously partitioned and mapped to a set of components, but deserve re-examination due to availability of either a new fabrication technology (resulting in higher operating clock frequency), or a higher performance embedded processor, both of which result in increased availability of processing headroom. Key to our solution is the concept of "software co-processors", which enables *hardware* components to selectively off-load computation tasks to standard (or customized) embedded processors. To this end, SOFTENIT uses a template architecture to which the new system is mapped. We describe this template architecture, and the various steps involved in the softening process in detail, and illustrate their application to an IEEE 802.11 MAC processor design [1, 2]. A prototype of the SOFTENIT flow has been implemented, and using it, we were able to efficiently modify the MAC processor architecture and achieve substantial reductions in hardware complexity, while being able to satisfy the stringent data-rate requirements imposed by modern Wireless LANs, with low computational effort.

The rest of this paper is organized as follows. In Section 1.2 we discuss related work. Section 2 illustrates, using the MAC processor as an example, the effect of applying the proposed techniques on an existing design. In Section 3, we provide an overview and details of the SOFTENIT methodology. Section 4 presents the results of experiments conducted on the MAC processor to evaluate the proposed flow. Finally, Section 5 concludes the paper with a discussion on areas for future research.

1.2 Related Work

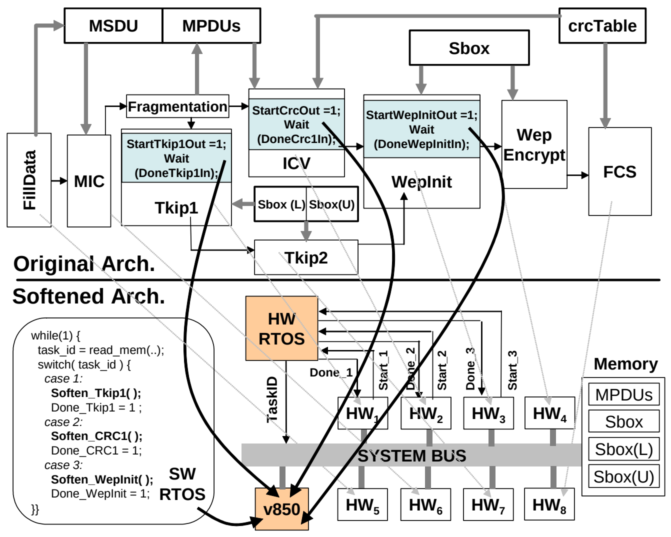
Extensive research has been performed in the past to aid in automatic HW/SW partitioning for application-specific systems. These include those that partition system functionality into hardware and software starting from an implementation independent specification of system behavior (e.g., [3, 4, 5]), as well as those that start with an all software [6], or all hardware [7] description. Most previous work in the area is targeted towards the early stage, first time design of a system architecture, and involves making relatively coarse-grained HW/SW partitioning decisions.

More recently, numerous techniques have been proposed for fine-grained migration of functionality from software to hardware in designs where an initial coarse-grained partitioning has already been performed [8, 9, 10, 11, 12]. Such "performance-driven", fine-grained HW/SW partitioning is achieved via automatic synthesis of customized hardware accelerators (or custom functional units within an embedded processor) to implement performance-critical software loops. Our work is similar in that we consider the migration of functionality *in an existing design*, *i.e.*, in a system that has already been partitioned into hardware and software, and mapped to a set of architectural components. However, we are concerned with a complementary need, that of reducing the complexity of the hardware portion of the application in scenarios where the embedded processor has increased availability of processing resources. In summary, our work targets "cost/flexibility-driven", fine-grained HW/SW *re*-partitioning, based on using *SW* co-processors for increased *flexibility*, in contrast to the notion of using *HW* co-processors for improved *performance*.

2. EXAMPLE: IEEE 802.11 MAC PROCESSOR

Figure 1 illustrates the application of the proposed techniques to a design of an IEEE 802.11 MAC Processor that implements the WPA encryption and data integrity functions of IEEE 802.11 Wireless LANs [1]. While functional details of the system are presented in Section 4, for the purposes of this section, it is sufficient to observe that the original system architecture consists of 8 dedicated hardware components (*FillData, MIC, Tkip1, Frag, ICV, WepInit, WepEncrypt, FCS*) and 6 shared memories (*MSDU, MPDUs, crcTable, Sbox, Sbox(L), Sbox(U)*). Note that, in this example, the original system is entirely mapped to hardware. However, the proposed techniques are equally applicable to systems that contain one or more processors that execute a portion of application functionality.

In Figure 1, portions of modules *Tkip1, ICV, WepInit* have been identified as targets for softening (indicated by the shaded blocks). The lower half of the illustration shows the resulting system architecture after application of the proposed techniques. The following changes to the system architecture should be noted:



```
while(1) {
  task_id = read_mem(..);
  switch( task_id ) {
    case 1:
      Soften_Tkip1( );
      Done_Tkip1 = 1 ;
    case 2:
      Soften_CRC1( );
      Done_CRC1 = 1;
    case 3:
      Soften_WepInit( );
      Done_WepInit = 1;
}}
```

Figure 1: Application of proposed SOFTENIT techniques to an IEEE 802.11 MAC Processor.

- The resulting system is enhanced with a "SW co-processor". In this example, the methodology results in the introduction of a new embedded processor (NEC's v850), since the original system was entirely mapped to hardware. In general, the SW co-processor could be realized by making use of available processing bandwidth on existing processors in the design.
- In the new architecture, the application-specific hardware components of the resulting system are either identical to the original components (*e.g., FillData, MIC, Tkip2, WepEncrypt, FCS*), or simplified, in cases where operations previously implemented in hardware are migrated to software (*e.g., Tkip1, ICV, WepInit*).
- A new hardware component, labeled HW-RTOS, is included in the new system. In general, more than one hardware block may be selected for softening (as in this example). Since hardware blocks execute concurrently, at any given time, the system may potentially have several pending softened tasks to execute [1]. However, dynamic scheduling of the potentially large number of softened tasks could result in the consumption of valuable processor resources that could otherwise be devoted to executing application functionality. To maximize the migration of functionality from hardware to software, it is crucial to minimize the extent to which the processor is involved in performing such operations. For this reason we propose a mixed HW/SW solution for implementing scheduling support. The HW-RTOS component performs scheduling operations and serializes requests to the processor, as illustrated in Figure 1, while the SW-RTOS (illustrated by the code fragment on the lower-left portion of Figure 1) is only responsible for task invocation. It might be argued that the HW-RTOS acts very similarly to a standard interrupt controller. In a certain sense this is true, but the main difference here is that the scheduling policy is not fixed (a standard interrupt controller generally implements a static priority scheduling policy), but instead is customizable.

Note that, in either architecture, data objects that are accessed by the hardware modules selected for softening are mapped to the

[1]We use the term "softened" tasks to refer to software tasks that execute on the SW co-processor (as a result of the migration of functionality from hardware to software), and "softened hardware" to refer to hardware components that have off-loaded part of their functionality to the SW co-processor.

shared on-chip memories. The memory is connected to the bus, and is hence, addressable by the processor. In addition, while in this work we consider single processor based systems, it should be noted that the target architecture could in general, include more than one SW co-processor, where each executes computations off-loaded from a set of softened hardware components.

3. SOFTENIT METHODOLOGY

We next present an overview and details of the proposed SOFTENIT methodology for off-loading selected portions of hardware-mapped system functionality onto software co-processors. The methodology involves two major tasks: (i) migration of system functionality from hardware to software, and (ii) generation of interfaces and handshaking mechanisms between the remaining hardware and the new software. The user, with the help of profiling/analysis tools, selects the softening targets in the original system architecture, and then uses a semi-automated flow to generate the modified system architecture. In the rest of this section, we first present an overview of the various steps in the methodology, and then describe each step in detail.

3.1 Overview

The methodology takes as input a partitioned and mapped system description, which contains at least one hardware-mapped task. The result is a modified system architecture in which the fraction of system functionality implemented in software has been boosted, by "softening" certain hardware modules. The methodology, illustrated in Figure 2, consists of the following steps. Step 1 involves semi-automatic selection of softening targets. Step 2 identifies the set of variables that need to be communicated across the HW/SW boundary as a result of the increase in the system's software content. In Step 3 (Co-processor interface synthesis), identified softening targets are modified to incorporate a communication mechanism that enables the resulting hardware to communicate with the softened tasks (running on the SW co-processor), and the HW-RTOS. In Step 4, the softening targets are converted to functionally equivalent optimized C code (softened tasks) compatible with the target embedded processor. In Step 5, a customized HW-RTOS is generated for run-time arbitration among the set of pending softened tasks. The RTOS can be customized in terms of the scheduling strategy (*e.g.*, round robin, topological sort, earliest deadline first).

In our work, we consider hardware that is modeled at the cycle-accurate functional level, an increasingly popular level of abstraction for design entry [13, 14, 15]. The subject of translating hardware modeled at lower levels of abstraction (*e.g.*, traditional structural RTL descriptions) to software is not dealt with in this work, and could form the basis for future research. We next describe the steps of this methodology in more detail.

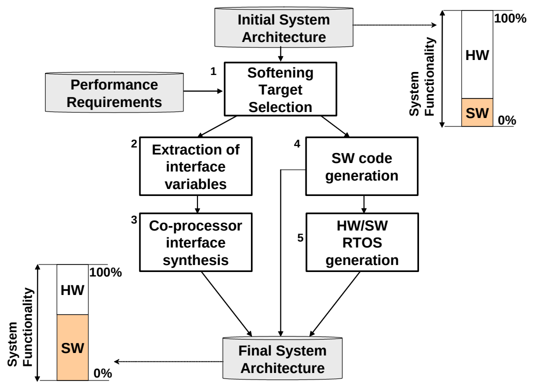
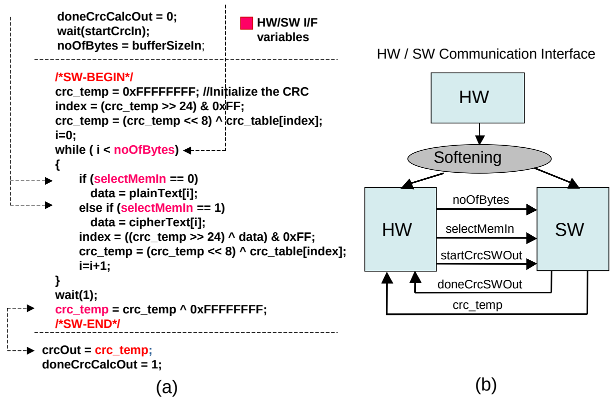

Initial System Architecture
100%
HW
SW
0%
System Functionality
Performance Requirements
1
Softening Target Selection
2
Extraction of interface variables
4
SW code generation
3
Co-processor interface synthesis
5
HW/SW RTOS generation
100%
HW
SW
0%
System Functionality
Final System Architecture

Figure 2: SOFTENIT **methodology: Overview**



```
doneCrcCalcOut = 0;
wait(startCrcIn);
noOfBytes = bufferSizeIn;

/*SW-BEGIN*/
crc_temp = 0xFFFFFFFF; //Initialize the CRC
index = (crc_temp >> 24) & 0xFF;
crc_temp = (crc_temp << 8) ^ crc_table[index];
i=0;
while ( i < noOfBytes)
{
      if (selectMemIn == 0)
         data = plainText[i];
      else if (selectMemIn == 1)
         data = cipherText[i];
      index = ((crc_temp >> 24) ^ data) & 0xFF;
      crc_temp = (crc_temp << 8) ^ crc_table[index];
      i=i+1;
}
wait(1);
crc_temp = crc_temp ^ 0xFFFFFFFF;
/*SW-END*/

crcOut = crc_temp;
doneCrcCalcOut = 1;
```

Figure 3: Softening of CRC hardware: (a) Softening target with interface variables, (b) HW/SW communication interface.

3.2 Softening Target Selection

In this step, each hardware component is analyzed, and specific computation sub-tasks are selected for migration to software. Currently, the methodology does not automate the process of target selection, but provides the designer with automatic tools to analyze the potential advantages of softening different sub-tasks. The designer is empowered with fast architectural simulation [2], and an efficient softening methodology (as will be borne out by the results presented in Section 4) to enable exploration of a large number of softening alternatives with relative ease. This enables the designer to choose potential softening targets based on automatically obtained estimates of hardware savings, and system performance. This selection of softening targets is performed by enclosing between /*SW-BEGIN*/ and /*SW-END*/ tokens, the portions of the original hardware behavior that need to be softened (Figure 3(a). Each hardware module may be instrumented to contain an arbitrary number of non-overlapping pairs of these tokens.

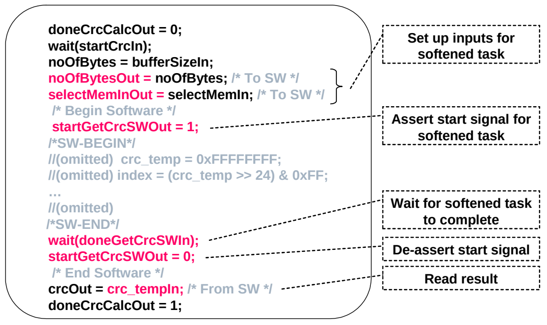
3.3 Extraction of Interface Variables

This step of the methodology involves the identification of variables that need to be communicated across the HW/SW boundary. This is achieved using conventional data-flow analysis techniques. Inputs to the softened task include those variables that are defined outside the softening target, but are read from within it. Similarly, this step also identifies variables that are updated within the softening target, and are used outside it. These variables become the outputs of the softened task. Figure 3 shows a portion of an original hardware behavior (which corresponds to the CRC computation in the MAC processor) in which the softening target has been specified using /*SW-BEGIN*/ and /*SW-END*/ tokens. The dotted lines indicate data dependences, which result in the identification of variables that need to be included in the HW/SW interface. In this example, the variables *noOfBytes* and *selectMemIn* become inputs to the softened task, and *crc_temp* becomes an output of the softened task.

3.4 Co-processor Interface Synthesis

Once the variables that need to be communicated across the HW/SW boundary have been identified, the original hardware description is stripped of the functionality that has been softened, and enhanced with an interface that enables it to communicate with the softened task that executes on the SW co-processor. Figure 3(b) illustrates the communication interface that corresponds to the softening target identified in Figure 3(a). Figure 4 illustrates the modified hardware description that implements the interface behavior.

[2]In our implementation, we used the Classmate simulator, which is part of NEC's C-based design flow [15]



```
doneCrcCalcOut = 0;
wait(startCrcIn);
noOfBytes = bufferSizeIn;
noOfBytesOut = noOfBytes; /* To SW */
selectMemInOut = selectMemIn; /* To SW */
 /* Begin Software */
 startGetCrcSWOut = 1;
/*SW-BEGIN*/
//(omitted)  crc_temp = 0xFFFFFFFF;
//(omitted) index = (crc_temp >> 24) & 0xFF;
…
//(omitted)
/*SW-END*/
wait(doneGetCrcSWIn);
startGetCrcSWOut = 0;
 /* End Software */
crcOut = crc_tempIn; /* From SW */
doneCrcCalcOut = 1;
```

Set up inputs for softened task
Assert start signal for softened task
Wait for softened task to complete
De-assert start signal
Read result

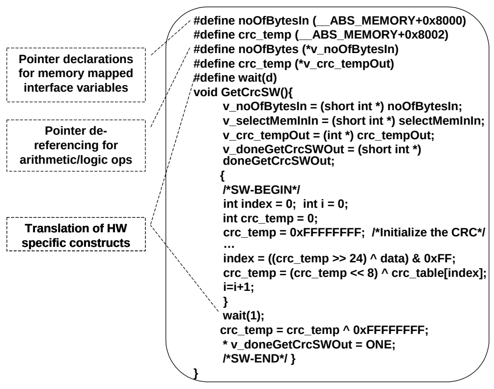
Figure 4: A HW/SW communication interface generated in the CRC module, after the core CRC computation has been softened.

From the figure we observe that the part enclosed between /*SW-BEGIN*/ and /*SW-END*/ has been commented out, and new I/O signals have been generated to (i) exchange data with the softened task, and (ii) handshake with the operating system. Two new output signals are generated at the hardware component interface, namely, *noOfBytesOut* and *selectMemIn*, to pass input parameters to the softened task. The *startGetCrcOut* signal is then asserted to request the HW-RTOS to execute the corresponding softened task. Having asserted this signal, the hardware waits for the completion of the software task's execution, which is indicated by the *doneGetCrcSWIn* signal. At this point, the *startGetCrcOut* signal is reset, and the result of the software computation is returned via the signal *crc_tempIn*, which is then used to update the internal variable *crcOut*. Finally, the signal *doneCrcCalcOut* is set high, which informs the scheduler that the execution of the softened task has been completed.

Once the signals have been generated, addresses are selected for the memory-mapped hardware registers. Since communication between the hardware and the SW co-processor occurs over the system bus, bus interface logic is automatically synthesized to connect the hardware to the bus.

3.5 Software Code Generation

In this step, software code that is compatible with the target SW co-processor is generated, starting from the identified softening target in Step 1. We assume the availability of behavioral, or cycle-accurate, functional descriptions of hardware. Input parameters are passed from hardware to software using memory-mapped registers. Hence, the transformations required to generate corresponding software tasks include declaring (and using) pointers to refer to memory mapped registers and data memory regions. Prior



```
#define noOfBytesIn (__ABS_MEMORY+0x8000)
#define crc_temp (__ABS_MEMORY+0x8002)
#define noOfBytes (*v_noOfBytesIn)
#define crc_temp (*v_crc_tempOut)
#define wait(d)
void GetCrcSW(){
        v_noOfBytesIn = (short int *) noOfBytesIn;
        v_selectMemInIn = (short int *) selectMemInIn;
        v_crc_tempOut = (int *) crc_tempOut;
        v_doneGetCrcSWOut = (short int *)
        doneGetCrcSWOut;
       {
        /*SW-BEGIN*/
        int index = 0;  int i = 0;
        int crc_temp = 0;
        crc_temp = 0xFFFFFFFF;  /*Initialize the CRC*/
        …
        index = ((crc_temp >> 24) ^ data) & 0xFF;
        crc_temp = (crc_temp << 8) ^ crc_table[index];
        i=i+1;
        }
        wait(1);
       crc_temp = crc_temp ^ 0xFFFFFFFF;
        * v_doneGetCrcSWOut = ONE;
        /*SW-END*/ }
}
```

Pointer declarations for memory mapped interface variables
Pointer de-referencing for arithmetic/logic ops
Translation of HW specific constructs

Figure 5: Softened CRC task: generated software code.

to the execution of the softened task, all inputs are fetched from hardware output registers using simple I/O operations. Similarly, outputs generated by the softened task are transfered to hardware by writing the results into memory-mapped registers. Other transformations that need to be applied include removal (or substitution) of hardware-specific constructs, such as synchronization directives and bit-level manipulations, that might appear in the original hardware description and are not supported by a standard C compiler. Figure 5 shows the softened task that is generated for the softening target specified in Figure 3.

3.6 RTOS Generation

In Step 5, a scheduling support is generated, whose purpose is to resolve, at run time, conflicting requests from different hardware blocks, each of which need to execute a certain softened task on a SW co-processor. As described in Section 2, in order to minimize consumption of processor cycles, in our approach, a mixed HW/SW RTOS is generated. The hardware-mapped portion implements the actual scheduling algorithm. The software-mapped portion of the RTOS is only responsible for the computationally inexpensive operation of task invocation.

As illustrated in Figure 1, the original hardware modules that have been softened are augmented with two additional I/O signals (*Start* and *Done*). These signals are used for handshaking with the RTOS, which gathers pending requests and chooses the next software task to run based on a customizable scheduling policy. The selected task is communicated to the CPU using an integer task identifier *task_id* that is sent on the bus. The CPU then calls the appropriate software routine based on the value of *task_id*.

An additional advantage of implementing the scheduler in HW is that the hardware has a complete view of the system. Moreover, since the hardware executes much faster than software, it is possible to implement some scheduling algorithms that are generally not considered in a software based RTOS. One example is topological sorting, which is a classic technique used to schedule event driven descriptions in commercial event driven simulators. Dynamic scheduling policies like Earliest Deadline First (EDF) are rarely implemented in software due to the difficulty involved in computing deadlines. The hardware scheduler is advantageous for this purpose, due to more predictable latency characteristics (as compared to software, where pipeline stalls and cache misses complicate matters significantly).

4. EXPERIMENTAL RESULTS

In this section, we present the results of applying the described softening techniques to an IEEE 802.11 MAC processor design. We first present an overview of the functionality of the system, and the experimental methodology used. Next, we present results that evaluate the impact of the proposed softening techniques on system performance, the corresponding savings in hardware complexity, and the computational efficiency of the proposed techniques.

4.1 IEEE 802.11 MAC Processor

The system we consider implements the integrity and security functionality deemed mandatory for Wi-Fi Protected Access (WPA) compliance in IEEE 802.11 based systems. We provide a brief overview of the functionality of the system. Full details are available in [1, 2]. The system functionality is distributed among seven communicating tasks, illustrated in Figure 6. Data is received from the host in the form of MAC Service Data Units (MSDUs), which may vary in size from 1300 to 2300 bytes. MSDUs are processed by the Message Integrity Check (MIC) task, and may be fragmented into multiple MAC Protocol Data Units (MPDUs), ranging in size from 256 to 1300 bytes. The Temporal Key Integrity Protocol (TKIP) consists of two functions: TKIP Phase 1 executes once every 65536 MPDUs, while TKIP Phase 2 executes

Table 1: Design space of different softening alternatives

Softening Targets	Application-Specific HW (sq. um)	Performance		Throughput (Mbps)
		Hardware Block	Softened Task	
None (all HW)	80308			173.1
MIC	79373	2.61 cycles/B	13.82 cycles/B	173.1
ICV	82278	5.0 cycles/B	20.0 cycles/B	173.1
TKIP Phase 1	60332	264 cycles	8664 cycles	173.1
TKIP Phase 2	64877	69 cycles	1220 cycles	173.1
Wep_Init	64785	1794 cycles	20203 cycles	86.7
Wep_Encrypt	77925	16.62 cycles/B	155.0 cycles/B	20.4
FCS	72278	5.0 cycles/B	20.0 cycles/B	173.1
SA1	**25936**			**163.0**
SA2	**25936**			**86.8**
SA3	**29378**			**84.1**
SA4(all SW)	0			13.9

SA1: MIC, TKIP Phase1, TKIP Phase2, FCS softened, **SA2:** MIC, TKIP Phase1, TKIP Phase2, ICV softened, **SA3:** TKIP Phase1, TKIP Phase2, Wep_Init blocks softened, **SA4:** All blocks softened

once for each MPDU and computes the encryption key. The Integrity Checksum Vector (ICV) task computes a 32-bit CRC on each un-encrypted MPDU. In parallel with the ICV, the encryption tasks (WEPINIT and WEPENCRYPT) encrypt each MPDU using the RC4 ciphering algorithm. Thereafter the Frame Checksum Sequence (FCS) task computes a 32-bit CRC on the encrypted MPDU and MAC header, thereby generating a frame ready to be transmitted over the air. Corresponding operations take place at the receiver, while fragmented MPDUs are aggregated to regenerate the corresponding MSDU.

4.2 Experimental Methodology

To evaluate the proposed SOFTENIT flow, the WPA-based IEEE 802.11 MAC processor was first implemented at the cycle-accurate functional level using a commercial C-based hardware design flow [15], with dedicated hardware units for each of the functions described in the previous subsection. The hardware architecture used as a starting point for our experiments is illustrated in Figure 1. The SOFTENIT methodology was implemented and integrated with this design flow. NEC's v850 [16] embedded processor was used as a SW co-processor. NEC's cycle-accurate architectural simulator Classmate [15] was used to drive selection of the softening targets, and for the large number of experiments that were performed to analyze the resulting systems. In cases where multiple blocks were softened, a round-robin scheduling strategy was used by the HW-RTOS. The system clock frequency for all experiments was set at 300 Mhz, and the data rate requirement of the system is 50 Mbps. To obtain hardware area estimates, the Cyber behavioral synthesis tool [15], and Synopsys Design Compiler [17] were used to generate gate-level implementations using NEC's CB12 0.15 μm standard cell library [18].

4.3 Performance/Area Impact of SOFTENIT

The first set of experiments compares the performance impact

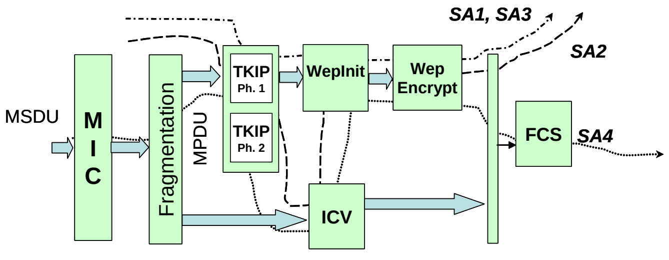


Figure 6: IEEE 802.11 MAC Processor: functional specification and architecture dependent system critical paths.

when different parts of the MAC processor are softened, and the corresponding savings achieved in terms of hardware complexity. Table 1 lists different softening alternatives, including cases where individual as well as a collection of hardware blocks are softened. Columns 3 and 4 indicate the performance impact of softening on individual tasks. Column 5 indicates the performance impact on the overall system. Column 2 indicates the corresponding reductions achieved in application-specific hardware complexity.

From the table, we make several observations. First, softening of system hardware may have significantly variable impact on the performance of individual blocks. For example, softening causes a 33X slowdown for TKIP Phase I, but only a 4X slowdown for ICV. While this is an important metric to consider when selecting which blocks to soften, it is more important to examine the impact of this slowdown from a system-level standpoint. In fact, the results indicate that, in spite of significant deterioration in the performance of specific tasks, overall system performance is often relatively immune to softening. For example, softening the MIC, ICV and TKIP tasks have no impact on system performance. The results also demonstrate the relatively small impact when multiple blocks are judiciously selected (SA-1 through SA-3), and the relatively large impact where a poor selection (SA-4) is made. We observe that significant number of solutions can be found where in spite of softening, the system is capable of satisfying the 50 Mbps requirement. These results can be explained by analyzing the critical paths in the system functionality, which varies, depending on the exact set of tasks that are softened. The critical paths for the four architectures are illustrated in Figure 6. In addition, we observe that softening can achieve substantial savings in terms of hardware complexity. For example, on average, architectures SA-1 through SA-3 achieve 66% savings in the complexity of application-specific hardware used in the design.

For this system, the overall system performance is also determined, to a large extent, by the characteristics of the input traffic. From the above results, since SA-1 through SA-3 appear to be good candidate softened architectures, the next experiment we performed was to evaluate their ability to satisfy performance requirements across varying MPDU sizes. The results of this experiment are presented in Figure 7. From these results, we observe that while SA-1 and SA-2 are capable of satisfying the 50 Mbps requirement at all frame sizes, SA-3 fails to do so for frame sizes below 450 bytes.

These results indicate that significant opportunities exist for softening in complex designs, which, if carefully exploited, can result in substantial savings in hardware complexity, and a boost in the fraction of functionality that is mapped to software, without paying significant penalties in terms of overall performance.

4.4 Computational Efficiency of SOFTENIT

We finally report on the computational complexity of the SOFT-

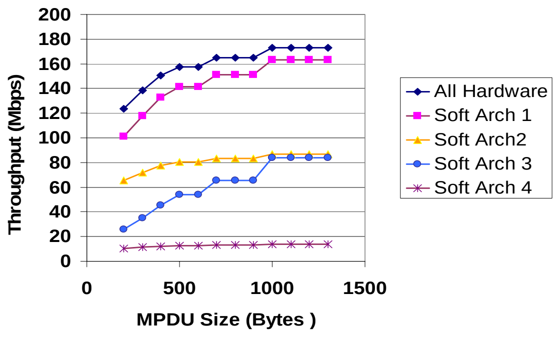


Figure 7: Variation of throughput with MPDU size for candidate softened architectures of the IEEE 802.11 MAC processor.

ENIT methodology. Table 2 reports on the measurements of CPU time consumed in generating 4 candidate softened architectures. All the measurements were performed on a Dell PowerEdge server with a 2.8 Ghz Intel Xeon processor and 4 GB RAM. The column labeled SOFTENIT refers to the time taken to execute our tool (steps 2 through 5 of Figure 2). The total time includes the softening time and the software compile time, and varies, depending on the number of softened tasks. In all cases, the time taken is less than a minute. In comparison, the time taken by an experienced designer to manually construct a new system description from an old one, when provided the selection of softening targets, for the MAC processor system, was in excess of 2 days. From these results we conclude that the proposed methodology is computationally efficient, and thereby facilitates fast exploration of numerous softening alternatives in a short span of time. To put these results into proper perspective, we also report the time consumed in generating hardware for the blocks that were chosen to be softened. The reported time includes the time spent in performing behavioral and logic synthesis. While this in itself is upto 30X slower than the softening process, the actual CPU time spent in hardware design would in practice, be significantly higher, owing to the large computational effort required by physical design tools.

5. CONCLUSIONS

A shift toward software helps reduce time to market, increase the opportunities for design reuse, increase flexibility, and reduce cost. Given a system that was previously designed for certain performance constraints and hardware capabilities, under a revised version of the same design, numerous alternatives for migrating system functionality from hardware to software often emerge, due to increased availability of processing bandwidth. In this paper, we proposed automatic softening of hardware as a technique for efficiently boosting the proportion of system functionality implemented using embedded software over the life time of an application. We presented SOFTENIT, an initial step toward the development of systematic techniques for achieving this goal, and thereby simplify the inevitably error-prone effort of periodically and manually re-designing the entire system, and studied its application to a hardware-dominated design of an IEEE 802.11 MAC processor.

While the results presented in this paper are promising, several research challenges remain to be addressed to make such techniques widely used by modern design teams. The large number of legacy designs will make it important to support transformation of detailed structural (RTL) hardware descriptions into optimized software running on the SW co-processors. The recent emergence of configurable embedded processors raises the possibility of addressing the performance impact issue (when functionality is migrated from hardware to software) through automatic instruction set customization techniques. Increasing system complexity implies that the potential number of softening alternatives may be huge, requiring efficient search techniques for exploring the design space of softening target selection. Finally, for SoCs containing large number of softening targets, and more than one processor, multiple SW co-processor based solutions would need to be developed. In summary, we believe that this work provides a new perspective on system partitioning and a valuable starting point for future research in automatic softening techniques for System-on-Chip designs.

Table 2: Computational effort of softening methodology

Arch.	Computational Effort in Softening			Hardware Synthesis	
	SOFTENIT (sec)	Compilation (sec)	Total (sec)	HLS (sec)	Logic Synthesis (sec)
SA1	14.2	19.9	34.1	2.91	308
SA2	13.8	19.9	33.7	2.91	308
SA3	12.7	14.5	27.2	14.7	282
SA4	16.2	35.7	51.9	15.88	423

6. REFERENCES

[1] J. Edney and W. A. Arbaugh, *Real 802.11 Security: Wi-Fi Protected Access and 802.11i*. Addison Wesley, 2003.

[2] "Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications (Chapter 8)." IEEE Computer Society LAN/MAN Standards Committee, IEEE Std 802.11-1999 Edition.

[3] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[4] F.Balarin, M.Chiodo, P.Giusto, H.Hsieh, A.Jureska, L.Lavagno, C.Passerone, A.Sangiovanni-Vincentelli, E.Sentovich, K.Suzuki, and B.Tabbara, *Hardware-software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, Norwell, MA., 1997.

[5] A. Kalavade and E. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design and Test of Computers*, vol. 10, pp. 16–28, Sep. 1993.

[6] R.Ernst, J.Henkel, and T.Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design & Test of Computers*, pp. 64–75, Dec. 1993.

[7] R.K.Gupta and G. Micheli, "Hardware/Software Co-synthesis for Digital Systems," pp. 29–41, Sep. 1993.

[8] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler," *IEEE Computer*, vol. 33, pp. 62–69, Apr. 2000.

[9] `http://www.tensilica.com`.

[10] `http://www.synfora.com`.

[11] `http://www.criticalblue.com`.

[12] G. Stitt, F. Vahid, and S. Nemetebaksh, "Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems," *IEEE Trans. Embedded Computer Systems*, vol. 03, pp. 218–232, Feb. 2004.

[13] `http://www.systemc.org`.

[14] `http://www.systemverilog.org`.

[15] K. Wakabayashi and T. Okamoto, "C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 1507–1522, Dec. 2000.

[16] `http://www.necel.com/micro/english/v850/product/index.html`.

[17] `http://www.synopsys.com/products/logic/`.

[18] `http://www.necel.com/cbic/en/product/cb12.html`.