Resource-Conscious Optimization of Cryptographic Algorithms on an Embedded Architecture

Wassim Bassalee and David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
{wbassale,kaeli}@ece.neu.edu

Abstract

Cryptographic algorithms are widely used in embedded systems. Applications of cryptographic algorithms include credentials establishment (e.g., authentication), securing contents (e.g., encryption/decryption of communication data or rights-managed objects), IP protection, and tamper-resistance checking. Resource constrained embedded systems can benefit greatly from employing cryptographic algorithms that are tuned to consume as little system resources as possible, while at the same time providing reasonable performance.

In this paper, we study some of the most commonly used cryptographic algorithms. We analyze their performance on an embedded system targeting the Blackfin DSP. We attempt to reduce encryption/decryption time, as well as reduce the energy consumed, by exploiting the architectural features available on the Blackfin. Our results show that by applying a set of optimization steps we can obtain 4x improvement in execution time and as much as 90% reduction in energy consumption.

1 Introduction

Cryptographic algorithms are used extensively in embedded systems. For example, many automated teller machines (ATMs) use cryptographic algorithms in order to communicate securely between the machine and the information processing center. Many smart phones and personal digital assistants (PDAs) use cryptographic algorithms in order to secure their communications. Several portable consumer electronics (e.g., portable media players) employ cryptographic algorithms to insure the copyright protection of the media contents.

As the number of portable devices increases, and as the number of applications which run on these devices and which require security features increases, the need for more efficient cryptographic processing will continue to grow. According to ITU [13], the number of mobile cellular subscribers surpassed the 3 billion mark in 2007. The share of smart phones and other mobile devices that are used for secure mobile commerce and email access is growing rapidly. For example, mobile commerce revenues in the US alone increased from $2.1 billion in 2003 to $58.4 billion in 2007 [4]. This means that cryptographic processing is taking place in more and more portable and mobile devices. Furthermore, as the data rates of these devices increases, the cost, in terms of time and power, of the cryptographic processing will increase significantly. Ravi et al. [20] project that as advances in wireless communication technologies lead to increased data rates, and as stronger cryptographic algo-

rithms are needed to thwart attacks by malicious entities, the amount of software security processing required will increase beyond the capabilities of the embedded processor. Therefore, fast cryptographic processing is necessary to bridge the "security processing gap" [20].

In order to meet the increasing demands for efficient cryptographic embedded processing, various architectural and system-level approaches have been proposed and evaluated. Previous work [7, 3, 20] proposed different architectural and system-level approaches. In this paper, we study software implementations of cryptographic algorithms on an existing embedded architecture. We look at four common cryptographic algorithms, evaluate their performance and power consumption, and explore how to make use of the existing architectural features to better map cryptographic processing to the target embedded system. Our results show that resource-conscious software implementations of asymmetric-key, symmetric-key and hash algorithms on the Blackfin processor can lead to better performance and a reduction in energy consumption.

In the next section, we describe the current state of the art in software implementations of cryptographic algorithms in embedded systems. The following section presents a brief overview of the cryptographic algorithms considered in this paper. In Section 4 we present our optimization approach and experimental setup. In Section 5 we present our runtime improvements and energy savings. Section 6 presents our conclusion.

2 Related Work

An overview of cryptography in embedded systems is presented in [21]. Workload characterization of cryptographic algorithms in constrained environments is presented in [5], where algorithms were coded and optimized in assembly using RISC instructions. Implementations of the five AES finalists and of public key cryptography on a DSP were discussed in [22] and in [12] respectively.

Energy-efficient software implementation of long integer modular arithmetic, which is used in some cryptographic algorithms, has been studied in [6]. Energy evaluation of software implementations of block ciphers were presented in [8].

In our paper, we develop software implementations of multiple cryptographic algorithms targeting an embedded architecture and we optimize our implementations both for execution time and for power consumption.

3 Target Cryptographic Algorithms

Cryptographic algorithms may be divided into public (or asymmetric) key, private (or symmetric) key algorithms, and hash algorithms. Public-key algorithms are typically used to establish a communication session. Once a session has been established, symmetric-key algorithms are used to encrypt/decrypt the information communicated at high speeds. Hash algorithms are used to calculate digests of input messages, often to verify data integrity.

In hash algorithms, an iterative process is applied in order to produce a condensed representation of the input message. An important feature of hash algorithms is their resistance to collisions. A hash algorithm is resistant to collisions if it is very difficult to come up with two different input messages that produce the exact same hash value. Because good hash algorithms produce a different hash (or message digest) from different input messages, this class of algorithms may be used to verify integrity and to implement tamper-resistance. A commonly used hash algorithm is SHA-1 [19]. More secure variants, SHA-256, SHA-384, and

SHA-512 are also described in [19].

Common public-key algorithms include RSA and Elliptic Curve Cryptography (ECC). ECC is gradually becoming dominant because it requires a shorter key length in order to achieve a security level equivalent to that of RSA. The Elliptic Curve Digital Signature Algorithm (ECDSA), a very common ECC protocol specified in [1], is a public-key algorithm used for digital signature authentication. Example applications of ECDSA in embedded systems include boot image verification and device authentication.

In embedded systems, symmetric ciphers are commonly used to decrypt/encrypt communications, media, or other data packet contents. Typically, these operations are performed while other real-time applications are running on the system. For example, a portable media player may decrypt rights-managed media contents, while at the same time decoding and rendering the media contents. Therefore, efficient implementation is paramount in the case of symmetric ciphers. Examples of symmetric ciphers include AES and DES. DES [16] is no longer considered secure because it uses a 56-bit key. This key length is too short to be secure against today's cryptanalysts. TDEA [16] is a serial combination of three DES blocks. With an appropriate choice of keys for each of the three DES blocks, TDEA can become more secure than DES. AES [18] supports 128, 192, and 256-bit keys and is the algorithm recommended for government use. AES is gaining ground over TDEA because, among other reasons, its software implementations execute faster than TDEA on most architectures.

3.1 Secure Hash Algorithm (SHA-1)

We begin our analysis by studying the hot loop present in SHA-1. The loop is shown below:

For t = 0 to 79:
{
$$T = ROTL^{(5)}(a) + f(b,c,d) + e + K_t + W_t$$
$$e = d$$
$$d = c$$
$$c = ROTL^{(30)}(b)$$
$$b = a$$
$$a = T$$
}

Where

$$W_t = \begin{cases} M_t & 0 \le t \le 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus \\ \quad W_{t-14} \oplus W_{t-16}) & 16 \le t \le 79 \end{cases}$$

And

$$f_t(x,y,z) = \begin{cases} (x \wedge y) \oplus (\neg x \vee z) & 0 \le t \le 19 \\ x \oplus y \oplus z & 20 \le t \le 39 \\ (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & 40 \le t \le 59 \\ x \oplus y \oplus z & 60 \le t \le 79 \end{cases}$$

Note the prevalence of logical and rotate operations. Also, note that in the computation of the W_t, we compute one term in the series by XORing four previous terms in the same series. An architecture that supports fast index addressing provides a performance advantage.

3.2 Advanced Encryption Standard (AES)

The AES cipher consists of two stages:

- A key dependant computation called the key schedule. This stage performs a key dependant operation. It takes a 128, 192, or 256-bits key as an input, and produces the rounds keys used in the second stage. This stage is typically performed once per session (the period during which the same key is used).

- The actual cipher operation. This stage performs the actual encryption or decryption of data using the rounds keys generated in the first stage. This operation iteratively executes a combination of functions a specified number of times.

The following is a pseudocode description of the cipher stage:

```
For count = 1 step 1 to N_{r-1}
{
    SubBytes(state)
    ShiftRows(state)
    MixColumns(state)
    AddRoundKey(state, w[round * N_b, (round + 1) * N_{b-1}])
}
```

SubBytes(state) is an operation in which the elements (of size 8 bits or 1 byte) of the state array (an array of 4x4 elements, 16 bytes total) are replaced according to a substitution table as shown in Figure 1. SubBytes(state) is typi-
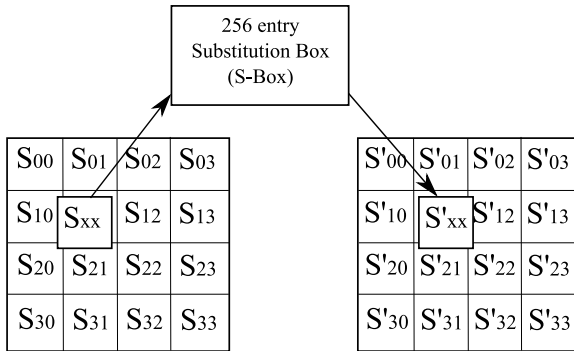

Figure 1: The SubBytes() operation in AES

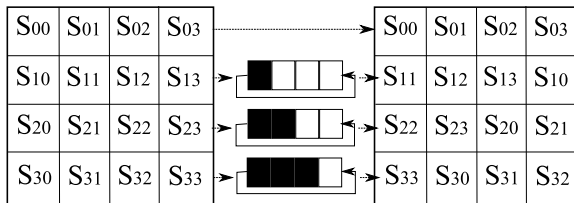cally implemented as a lookup from a 256-entry table. ShiftRows(state) is shown in Figure 2. If

Figure 2: The ShiftRows() operation in AES

every four elements in a row were thought of as a 32-bit word, ShiftRows(state) can be implemented on 32-bit architectures as a word rotate instruction. In MixColumns(state), which is shown in Figure 3, every state column is transformed through a matrix multiplication. The
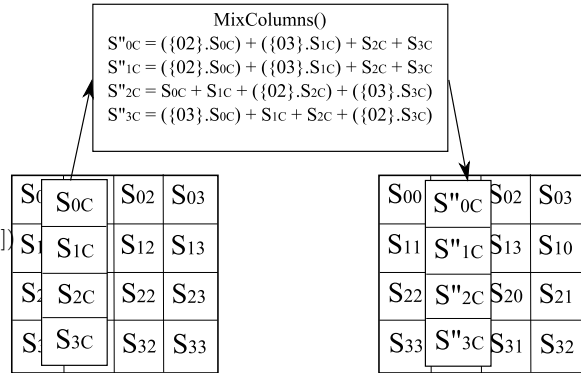

Figure 3: The MixColumns() operation in AES

operations (multiplications and additions) are performed in a finite field. AddRoundKey is typically implemented as bitwise XOR operations.

3.3   DES and TDEA

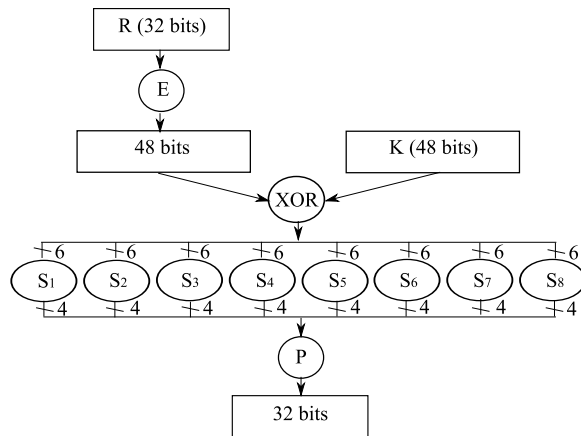At the heart of DES (and TDEA) is the cipher function f() depicted in Figure 4. E() expands

Figure 4: The cipher function f() in DES

the 32-bit input by duplicating some of the bits in the input to produce a 48-bit value that is XORed with a 48-bit round key K. The result of the XOR operation is a 48-bit value that is divided into eight 6-bit elements. Each of the 6-bit elements is transformed into a 4-bit el-

ement through a unique selection function S_x using a lookup table. The outputs of the eight selection functions are then concatenated together into a 32-bit value. Finally, individual bits in the 32-bit value are permuted according to a permutation function P() to produce the output of the cipher function.

3.4 Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA consists of:
- Elliptic curve domain parameter generation and their validation
- Key generation and validation
- Signature generation
- Signature verification

In many cases, the domain parameters and the keys are generated and validated prior to deployment of the embedded system. Therefore, we chose to concentrate on the operations that are more likely to be performed in an embedded system: signature generation and signature verification.

We use the Koblitz curve recommended in [17] for use over the binary field $\mathbb{F}_{2^{163}}$. The field multiplication operation dominates the execution time of ECDSA signature generation and verification. While the percentage of execution time spent in field multiplications to the total time spent in signature generation and verification depends on various factors (including the choice of algorithm), we observed that approximately 70% of execution time is spent in this operation.

4   Optimization Approach

4.1 The Blackfin Embedded Processor

Blackfin Processors include a high performance 16-/32-bit embedded processor core with a 10-stage RISC MCU/DSP pipeline, a variable length ISA for optimal code density, and full SIMD support with instructions for accelerated video and multimedia processing [10]. The Data Arithmetic Unit on the Blackfin contains:
- Two 16-bit MACs
- Two 40-bit ALUs
- Four 8-bit video ALUs
- Single barrel shifter

In a single clock cycle, the Blackfin can read and write up to two 32-bit values. The Blackfin architecture supports a variety of addressing modes, including indirect, auto-increment and decrement, indexed, and bit reversed. In addition, Blackfin supports circular buffering by providing two sets of index, length, and base registers to implement two unique circular buffers.

The Blackfin architecture supports 16 and 32-bit instructions, in addition to a set of multi-issue 64-bit instruction packets. This ensures maximum code density by encoding the most frequently used control instructions as compact 16-bit words and encoding math operations as 32-bit doublewords. The Blackfin architecture supports a number of instruction combinations that can be issued in parallel. Up to three instructions may be combined together in a 64-bit multi-issue instruction.

Blackfin processors are based on a gated clock core design that can selectively power down functional units on an instruction-by-instruction basis. Blackfin processors also support multiple power-down modes for periods where little or no CPU activity is required. Lastly, and probably most importantly, Blackfin processors enable dynamic power management, whereby the operating frequency and voltage can be independently manipulated to meet the performance and power budget required. These transitions may occur continually under the control of an RTOS or user firmware. Most Blackfin processors offer on-chip core voltage regulation circuitry, as well as operation to as low as 0.8V, and are par-

ticularly well-suited for portable applications requiring extended battery life.

4.2 Experimental Setup

To evaluate the performance and power of different cryptography algorithm implementations, we used the VisualDSP++ software development environment as a baseline. This environment includes an optimizing C/C++ compiler, an enhanced user-interface, built-in performance analysis capabilities, and a statistical profiling tool to easily identify programming bottlenecks. VisualDSP++ also provides simulation and profiling utilities that enable the accurate estimation of energy consumption on an instruction level [10, 15].

The instruction-level energy estimation model available in the VisualDSP++ Linear Profiling tool allows for energy-aware programming. This model considers both the energy consumed by individual instructions as well as that caused by interactions among different instructions. This estimation scheme does not capture the differences in energy consumption that result from using different data sets or from using different memory layouts. Due to the variability of absolute energy measurements from processor to processor, the Linear Profiler tool displays the energy consumption as Energy Units. This abstracts the variability that may be caused, for example, from physical differences related to manufacturing. Given a single processor, VisualDSP++'s instruction-level energy model provides accurate estimation of the relative energy consumed by different software.

4.3 Algorithm-Level Optimization

The first step in our optimization approach looked at the cryptographic routines from the algorithmic level. Our aim was to gain as much performance and power improvements through algorithm-level optimizations before diving deeper into low-level target-specific optimizations.

The application of algorithm-level optimization varied among the routines that we studied. While no such optimization was performed in SHA-1, the ones that we applied to the other three algorithms are described next.

4.3.1 AES

Instead of mapping each of SubBytes(), ShiftRows(), and MixColumns() into a different operation, we chose to combine them together into a single lookup table operation similar to that described in [2]. We pre-compute the finite field multiplications performed in MixColumns() and store all the possible resulting terms in an expanded lookup table. Our approach is different from the approach described in [2] in that we only used one table of 256 4-byte word entries instead of 4 tables of 256 4-byte word entries. Therefore, we use 1Kbytes of memory instead of 4Kbytes. Our approach is shown in the Figure 5.
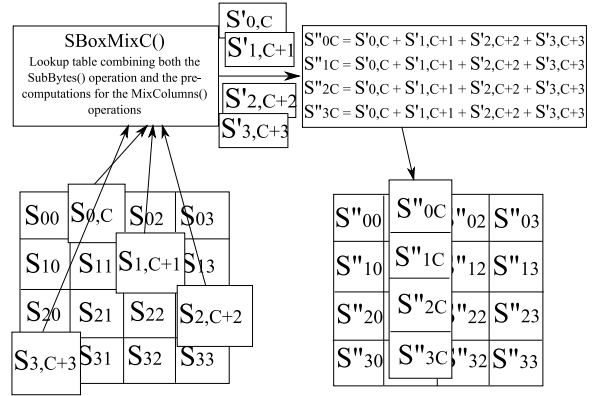

Figure 5: Algorithm-level optimization of AES

4.3.2 DES

In the DES cipher function f(), the 4-bit outputs of the 8 substitution boxes (S_1 - S_8) are

concatenated, and the result is *permuted* according to the bit permutation P(). Since typically concatenations and permutations are not easily implemented in embedded processors and DSPs, a faster approach may be to replace the 4-bit entries of the substitution tables $S_x()$ to produce $P(S_x())$ directly. Hence, the result may be obtained through 8 table lookups and 7 additions. This approach is shown in figure 6. The tradeoff of course is that we now use
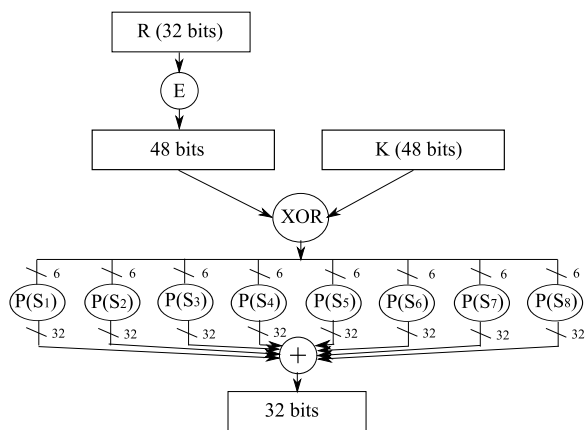

Figure 6: Algorithm-level optimization of DES

8 tables, each consisting of 64 entries of 32-bit words as opposed to 64 entries of 4-bit words.

4.3.3 ECDSA

In ECDSA, we applied high-level optimization to the implementation of the field multiplication operation. As stated earlier, the field multiplication operation makes up more than 70% of the execution time in ECDSA. Therefore, applying optimizations to that operation was expected to be most beneficial. When implementing the field multiplication operation, we used the left-to-right comb method with windows of width w=4, as described in [9, 14].

4.4 Performance and Energy Consumption Heuristics

We start out with two heuristics regarding the impact of instruction selection on final algorithm performance and energy consumption:

The first assumption is that, when all other factors are equal, the choice of Blackfin's multi-issue parallel instructions is typically better in critical loops. The multi-issue constructs are 64-bits in length [11]. When not all parallel issue slots are occupied by instructions, code size may be negatively impacted. However, choosing parallel-issue instructions in hot loops has the potential of improving performance enough to amortize the slight increase in code size.

The second heuristic is that, based on the study performed in [15], the energy consumed by a multi-issue parallel instruction is less than the energy consumed by the individual instructions that make up the multi-issue instruction, and is instead the average of the energy consumed by the two individual instructions. Therefore, by attempting to select instructions that are supported by Blackfin's multi-issue construct, we increase the potential of reducing the overall energy consumed by algorithm execution.

Both of these assumptions lead us to believe that in critical loops, we should try to use parallel instructions as often as possible.

4.5 Low-Level Optimization

We started the low-level optimization out by developing C implementations of all the cryptographic algorithms studied. In order to direct our optimization effort, we used the VisualDSP++ Blackfin simulator to profile the execution of our C implementations to determine the hot code sections that are likely to yield the best performance improvements upon optimization. Once hot code sections were identified, we concentrated our optimization ef-

```
Assembly Sequence 1
R7 = R7 + R1 (ns) ‖ R1 = [I0++];
R7 = R7 + R1 (ns) ‖ R1 = [R1];
R7 = R7 + R1 (ns);
```

```
C Listing 1
sx = rR ≫ 22;
sx &= 0x0000003F;
```

```
Assembly Sequence 2
R3 = 63;
R0 = R7 ≫ 22;
R4 = R0 & R3;
```

```
Assembly Sequence 3
.byte2 extract.params[8] = 0x1206, ...;
I2.H = extract.params;
I2.L = extract.params;

R3.l = W[I2++];
R4 = EXTRACT(R5,R3.L)(z);
```

forts on these sections, and created optimized assembly libraries that capture those critical code sections.

Making use of Blackfin's parallel-issue constructs was sometimes a trivial choice as shown in assembly sequence 1, which performs a portion of the computation of T in the critical loop of SHA-1. The code in assembly sequence 1 performs the computation: $T+ = f(b,c,d) + K_t + W_t$

Here, Blackfin's parallel issue construct allows us to carry out the addition of the various terms in the computation of T in parallel with the fetching of those terms from memory.

In certain cases, using the multi-issue construct did not necessarily seem like the best choice at first. However, upon further analysis, it turned out to improve both performance and energy consumption. For example, in DES/TDEA, we have the source level operation, shown in C listing 1, that performs a portion of the E() expansion.

This operation, repeated 8 times with dif-ferent shift amounts, may be used to extract the 6-bit inputs to the S_x tables in the cipher function f(). The assembly sequence generated by the compiler for this operation is shown as assembly sequence 2.

The first instruction in assembly sequence 2 is simply setting up the mask value of 0x3F. The next two instructions are a straightforward translation of the C operations into assembly. The logical AND instruction in Blackfin cannot be used in a multi-issue parallel instruction. Therefore, the 3rd instruction in assembly sequence 2 cannot be parallelized with other instructions. Since all the shift offsets are known statically, another option for implementing the E() expansion may be to use Blackfin's extract instruction as shown in assembly sequence 3.

Assembly sequence 3 requires two setup instructions to load the address of the extract parameters array into an index register. However, assembly sequence 3 produces faster code in the case of the cipher function f() because the last two instructions in assembly sequence 3 can both be used in a multi-issue construct thus allowing for more parallelism. Since we have eight S_x tables in the cipher function f(), the cost of using two initialization instructions in assembly sequence 3, as opposed to one instruction in assembly sequence 2, is offset by the gain of enabling the parallelization of all the extract instructions.

To perform the lookup operations used by AES and DES/TDEA, the compiler uses approaches that may seem to yield faster execution time. On the Blackfin, load/store instructions use pointer (Px) or index (Ix) registers, whereas compute instructions use the general purpose registers (Rx). However, some limited amount of pointer arithmetic may be performed using the pointer registers.

To implement the table lookups, one intuitive way is to move the table base pointer from a pointer (Px) or index (Ix) register into

```
┌─────────────────────────────────────────┐
│ Assembly Sequence 4                      │
│ Rx = Py;                                 │
│ Ry = Ry ≪ 2;                            │
│ Rx = Rx + Ry;                            │
│ Px = Ry;                                 │
├─────────────────────────────────────────┤
│ Assembly Sequence 5                      │
│ Pz = Ry;                                 │
│ <3 cycles before Pz is available>        │
│ Px = Py + (Pz ≪ 2);                     │
├─────────────────────────────────────────┤
│ Assembly sequence 6                      │
│ r5 = (a1 += r6.h * r1.h), r4 = (a0 += r6.l * r1.l) (IS); │
│ a1 -= r6.h * r1.h, a0 -= r6.l * r1.l (IS) ‖ [p3++] = r4; │
└─────────────────────────────────────────┘
```

a general purpose register (Rx), to add the index into the table using R registers (and possibly multiply by the table element size if it is larger than 1 byte), and then move the final values back to pointer registers to perform the load/store operation. This results in assembly sequence 4. These register move operations are expensive because they cannot be used in a multi-issue construct and because the values moved into a Px or Ix register may not be immediately available for use due to the pipeline structure.

A better implementation of this lookup operation, which was generated by the VisualDSP compiler, is to make use of the pointer arithmetic capabilities available on the Blackfin. Assembly sequence 5 shows the table lookup operation when the pointer arithmetic capabilities of the Blackfin ISA are used.

Here, the multiplication by the table element size and the addition of the index to the table base pointer all occur in a single instruction that utilizes the pointer registers.

It is clear that when the pointer register arithmetic is supported using pointer registers as in assembly sequence 5, the number of instructions is reduced. However, the load of the index into the pointer register Pz has to precede the use of Pz by 4 cycles. If other instructions can be scheduled in those cycles, then assembly sequence 5 may be a better choice than

assembly sequence 4. Otherwise, the reverse is true. In general, these two approaches are useful in control code because they typically use 16-bit instructions; hence, we save memory space. However, in critical loops such as the one we have in AES and DES/TDEA, we may do better by performing the pointer computations in SIMD-type instructions as shown in assembly sequence 6.

Here, the accumulator registers hold the base address. R1 holds the index, R6 holds the index multiplier (the size in bytes of the table elements). The first instruction performs the index calculation, while the second instruction restores the value in the accumulators to the base address again. Using this method, we perform two index calculations in two instructions. In addition, these add/accumulate instructions are parallelizable in multi-issue instructions, thus allowing for the parallel execution of other instructions. As an example, in line 2 of assembly listing 6, we store the result generated in line 1 in parallel. Also, note that by using the accumulators, we approach two lookup index calculations in two assembly instructions. This type of computation is unlikely to be generated by the compiler. However, in the critical loops of our crypto algorithms, we benefit greatly from using this type of parallel-issue constructs.

We followed this approach of generating code sequences that make good use of Blackfin's multi-issue parallel constructs and, when possible, of the SIMD-type commands. We created a library from these optimized versions of the most common and critical operations in the cryptographic algorithms studied. We then reran the same algorithms, but this time using the optimized extensions that we created. The results, which we present in the next section, support our initial assumptions.

Cipher	Base	Optim.	Exten.
AES	58749	13429	3231
3DES	31598	11106	8500
SHA-1	17922	4767	2601
ECDSA	45.4M	8.13M	6.2M

Table 1: Execution time (in cycle) of some cryptographic algorithms

Cipher	Base	Optim.	Exten.
AES	41.5M	8.65M	680K
3DES	42.6M	11.5M	7.6M
SHA-1	13.7M	3.60M	1.91M
ECDSA	59.5B	8.66B	6.56B

Table 2: Energy consumption estimates (in energy units) of some cryptographic algorithms

5   Results

The algorithm-level optimization applied to AES and DES produced more than 3x improvement in execution time and energy consumption. However, we did not apply such optimization to SHA-1, and we did not quantify the gain of the field multiplication optimization of ECDSA. Therefore, we present below results that already include the gains achieved from any algorithm-level optimization applied.

Table 1 shows the cycle count when running the cryptographic algorithms that we studied on the VisualDSP++ Blackfin simulator when no optimization is performed, when the compiler's aggressive optimization is performed, and when we used our optimized cryptographic extensions respectively.

It is clear that our library produced significant improvement in the execution time for the cryptographic algorithms studied. AES gained the most because the compiler generated code for the critical operations was mostly performing byte operations. In the Blackfin architecture, byte operations cannot be parallelized with other operations. Therefore, significant opportunities were lost.

Table 2 shows the energy unit consumption estimates as provided by the VisualDSP++ Blackfin simulator when no optimization is performed, when the compiler's aggressive optimization is performed, and when we used our optimized crypto extensions respectively.

As the study performed in [15] showed, the energy consumed by a multi-issue word is the average of the energy consumption of the individual instructions that make this parallel word. The results presented in the table above, demonstrate a clear advantage, from energy consumption perspective, to using multi-issue friendly crypto primitives. Again, AES exhibited significant improvement because many of the original instructions, which were acting on 8-bit data, were modified to 16-bit data variant. This allowed for increased code parallelism and, as a result, clear improvement in the energy consumption.

6   Conclusion

In this paper, we considered the performance/power tradeoff of running common cryptographic algorithms on a commercial embedded processor. Using C implementations of these algorithms, we profiled their execution to identify critical loops, and we created optimized versions of these critical operations by making use of Blackfin's multi-issue parallel constructs, SIMD-type instructions and other architectural features. We collected these optimized operations into a library, and we compared the performance and the energy consumption of the cryptographic algorithms with and without our optimized extensions. Even when not immediately obvious, the use of multi-issue and SIMD-type operations significantly reduced the execution time and energy consumption of the cryptographic algorithms.

References

[1] ANSI. Publication x9.62 - public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ecdsa), 1999.

[2] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael, 1998.

[3] Adam J. Elbirt and Christof Paar. An instruction-level distributed processor for symmetric-key cryptography. *IEEE Trans. Parallel Distrib. Syst.*, 16(5):468–480, 2005.

[4] ePayNews. http://www.epaynews.com/statistics/mcommstats.html.

[5] Murat Fiskiran and Ruby B. Lee. Workload characterization of elliptic curve cryptography and other network security algorithms for constrained environments. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-5)*, pages 127–137, November 2002.

[6] Johann Großschädl, Roberto Maria Avanzi, Erkay Savas, and Stefan Tillich. Energy-efficient software implementation of long integer modular arithmetic. In *CHES*, pages 75–90, 2005.

[7] Johann Großschädl and Guy-Armand Kamendje. Instruction set extension for fast elliptic curve cryptography over binary finite fields gf(2m). *asap*, 00:455, 2003.

[8] Johann Großschädl, Stefan Tillich, Christian Rechberger, Michael Hofmann, and Marcel Medwed. Energy evaluation of software implementations of block ciphers under memory constraints. In *DATE*, pages 1110–1115, 2007.

[9] Darrel Hankerson, Julio López Hernandez, and Alfred Menezes. Software implementation of elliptic curve cryptography over binary fields. *Lecture Notes in Computer Science*, 1965, 2001.

[10] Analog Devices Inc. http://www.analog.com/processors/blackfin.

[11] Analog Devices Inc. Blackfin processor instruction set reference manual.

[12] Kouichi Itoh, Masahiko Takenaka, Naoya Torii, Syouji Temma, and Yasushi Kurihara. Fast implementation of public-key cryptography on a dsp tms320c6201. In *CHES '99: Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, pages 61–72, 1999.

[13] International Telecommunication Union (ITU), January 2008. http://www.itu.int/ITU-D/ict/statistics/ict/index.html.

[14] Julio López and Ricardo Dahab. High-speed software multiplication in f_2m. In *INDOCRYPT*, pages 203–212, 2000.

[15] Seth Molloy. Energy conservation techniques for the blackfin processor. Master's thesis, Northeastern University, 2007.

[16] NIST. Data encryption standard (des), October 1999. Federal Information Processing Standards Publication 46-3.

[17] NIST. Digital signature standard, February 2000. Federal Information Processing Standards Publication 186-2.

[18] NIST. Advanced encryption standard (aes), 2001. Federal Information Processing Standards Publication 197.

[19] NIST. Secure hash standard (sha), August 2002. Federal Information Processing Standards Publication 180-2.

[20] Srivaths Ravi, Anand Raghunathan, Nachiketh Potlapally, and Murugan Sankaradass. System design methodologies for a wireless security processing platform. In *DAC '02: Proceedings of the 39th Conference on Design Automation*, pages 777–782, 2002.

[21] Thomas Wollinger, Jorge Guajardo, and Christof Paar. Cryptography in embedded systems: An overview. In *Proc. of the Embedded World 2003*.

[22] Thomas J. Wollinger, Min Wang, Jorge Guajardo, and Christof Paar. How well are high-end dsps suited for the aes algorithms? aes algorithms on the tms320c6x dsp. In *AES Candidate Conference*, pages 94–105, 2000.