Optimizing Parallel Runtime of Cryptanalytic Algorithms by Selecting Between Word-Parallel and Bit-Serial Variants of Program Parts

Patrick Eitschberger and Jörg Keller

Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
<firstname>.<lastname>@fernuni-hagen.de

Abstract: Cryptanalytic algorithms such as dictionary attacks, that test huge numbers of keys to decrypt a ciphertext to a certain plaintext, need lots of computational resources and efficient coding, but allow large scale parallelism such as many-cores plus GPUs. Some attacks have profited from a bit-serial data representation, that allows SIMD-like coding per thread and increases the degree of parallelism. We investigate the question how to decide for distinct parts of such algorithms whether to code them in a bit-serial or normal word-parallel manner. Given bit-serial and word-parallel variants for each part of the cryptographic algorithm, we benchmark the runtime of the variants, and additionally the runtime of the conversion between the different data representations. Then we model the resulting variant selection problem as a direct graph — in the fashion of a global composition optimization problem — and find the optimal runtime by computing the shortest path from source to sink node. We evaluate our approach with the Advanced Encryption Standard (AES) and demonstrate runtime advantages.

Keywords: Cryptanalytic Algorithm; Bit-Serial Computing; Global Optimization; Performance Tuning; Parallel Algorithm

1 Introduction

In recent years, parallel computing often means heterogeneous parallel computing with different processor architectures and accelerators such as GPUs. Finding the optimal allocation of work to different execution units thus becomes a non-trivial optimization problem. This optimization problem is complicated by other aspects as well, such as e.g. the best choice of a (parallel) sorting algorithm, which depends on the number of items to be sorted, their pre-sorting state, and the available implementations of different variants of sorting algorithms on the different execution units. In this manner, performance tuning of parallel applications has become a complex optimization problem that must be solved partly at algorithm design time, partly at compile time, and partly at execution time.

In the present work, we consider cryptographic applications like a dictionary attack (cf. e.g.

[MvOV97]), that execute millions to billions of decrypt operations on the same piece of ciphertext, but with different keys. Hence, these applications exhibit a tremendous amount of independent parallelism. As many encryption algorithms evaluate boolean functions during their execution, bit-serial (sometimes called bit-slice) computing in SIMD fashion (see Sect. 2) has been used for long to increase parallelism, improve speed, and reduce control-flow divergence on GPU architectures.

Our approach to further improve performance is to split a cryptographic algorithm into distinct parts, and provide both a normal (i.e. word-parallel) and a bit-serial implementation variant for each part. Additionally, we employ a known routine for data conversion between the two variants. Now, the application can be modelled in the fashion of a global composition optimization problem [HK14] as a directed graph, where nodes are variants attributed with their runtime, and arcs represent the flow of execution, and might be attributed with the conversion runtime if an arc's head and tail use different variants. By finding the shortest path from source to sink, the best combination of variants is found. To our knowledge, this represents a novel use for global composition of program variants, and has never been used to optimize parallel cryptographic algorithms.

As a case study, we apply our approach to an implementation of the Advanced Encryption Standard (AES) [Nat01], forecast an optimal mix of variants and demonstrate in experiments that runtime advantages over both purely bit-serial and word-parallel implementations are indeed possible.

The remainder of this work is structured as follows. In Sect. 2, we summarize basics about bit-serial computing, while Sect. 3 briefly reviews the global composition of program variants. In Sect. 4, we briefly summarize the AES algorithm as the object of our case study, apply the optimization algorithm from Sect. 3, and report our experimental results. Section 5 concludes and gives an outlook to future work.

2 Bit-serial Computing

Bit-slice processors, i.e. processors with a data width much smaller than a normal word width — in the extreme case called serial or bit-serial processors — have been known for long, e.g. in the Connection Machine [KH89]. Typically, a number of these processors work together in SIMD fashion to operate on data of normal width.

Also, the same concept has been known in software for decades. Biham [Bih97] "view(s) the processor as a SIMD computer, i.e., as 64 parallel one-bit processors computing the same instruction" and sees bit-serial computing mainly as a "non-standard representation" of data. While this may sound strange at first glance, it ensures that all data bits are used by parallelism, which is often not the case in normal computations, e.g. when the instructions operate on bytes or even on single bits while evaluating a logical expression.

We illustrate this concept with three small examples: one that favors bit-serial computing, one that favors normal data representation, and one where it depends on the circumstances which one is better.

Assume that we want to evaluate a boolean expression for many parameter values, e.g. $y_i = a_i \wedge b_i$ for $i = 0, 1, \ldots, 31$. Normally, we code

```
int i;
int y[32],a[32],b[32];
// ... set values in arrays a and b
for(i=0;i<32;i++) y[i] = a[i] & b[i];
```

and apply parallelism in the form of loop parallelization. Yet, if we transfer the lowest bits from each array element `a[i]` into one variable `as` such that the bit from `a[i]` is the ith bit in `as`, then we can simply write

```
int ys,as,bs;
// ... set values in as and bs
ys = as & bs;
```

If 2^{16} evaluations are to be done, then with 32 threads, each thread would have to do $2^{11} = 2048$ evaluations in normal representation, but only $2^5 = 32$ evaluations with bit-serial representation. If surrounding operations are also expressed in this manner, conversion of the representation is not necessary. Thus, bit-serial computing is advantageous in this case.

As a second example, consider that we want to do additions on 32-bit integer variables. The normal code is obvious and similar to the first example:

```
int i;
int y[32],a[32],b[32];
// ... set values in arrays a and b
for(i=0;i<32;i++) y[i] = a[i] + b[i];
```

In bit-serial representation, the data is organized in a manner orthogonal to the normal representation: variable `as[j]` contains bit j of each variable `a[i]` in bit i. This is illustrated in Fig. 1 for $j = 0$. Then, addition is performed bit by bit as in a full adder:

```
int j;
int ys[32],as[32],bs[32],cs[33];
// ... set arrays as and bs, and set array cs to 0
for(j=0;j<32;j++){ ys[j] = as[j] ^ bs[j] ^ cs[j];
cs[j+1] = (as[j] & bs[j]) | ((as[j] ^ bs[j]) & cs[j]); }
```

Please note that the conversion between the normal and the bit-serial data representations is nothing more than the transposition of a bit matrix with the variables `a[i]` and `as[i]` ($i = 0, \ldots, 31$) being the row vectors of the matrix and transposed matrix, respectively (cf. Fig. 2). An efficient algorithm for bit transposition is given in [RSD06], and we will use a variant in the sequel.

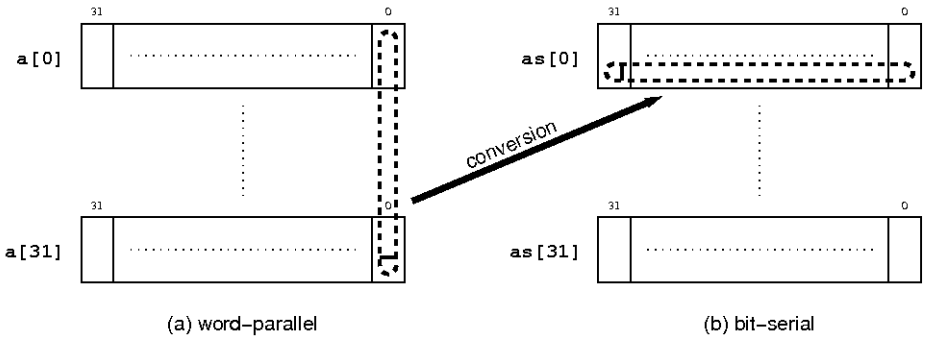

Figure 1: Corresponding word-parallel and bit-serial data representations.

Obviously, bit-serial representation is not advantageous in this second case, as an addition is replaced by 6 logical operations, and two assignments are used instead of one.

Finally, assume that we do a table lookup in a constant array defined over bytes.

```
int i;
uint8 y[32], a[32];
uint8 tab[256];
for(i=0;i<32;i++) y[i] = tab[a[i]];
```

Here, the bit-serial variant is not straightforward. If we use a 32-bit variable `ys[0]`, that contains the bit 0 of each variable `y[i]`, then bit i of `ys[0]` will depend on all bits of `a[i]` and on `tab`. Thus, a table for looking up `ys[0]` would be infeasibly large. However, we can express the dependence of each bit of `y[i]` on the 8 bits of `a[i]` and on `tab` by a boolean function in at most 8 variables, as tab is a constant array.

```
uint32 ys[8], as[8];
ys[0] = some boolean function on as[0] to as[7];
...
ys[7] = another boolean function on as[0] to as[7];
```

Hence, depending on the complexity of these boolean functions, the code might be slower or faster than the original table lookup. For example, if `tab[x]` would give the number of bits set in the binary representation of x (where $0 \leq x \leq 255$), then `ys[7]` to `ys[4]` would be 0, as the maximum number of bits set could be 8 (=00001000 in 8-bit binary), and `ys[3] = as[0] & ... & as[7]`.

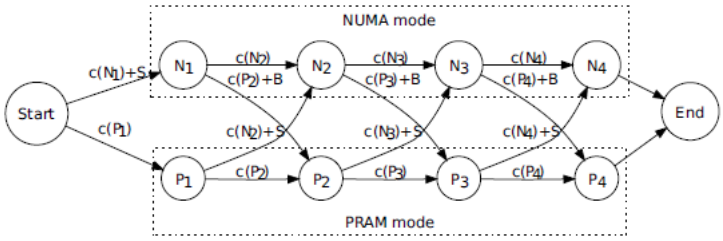

Figure 2: Graph with 4 program parts, each with 2 variants, taken from [HK14].

3 Global Optimization of Variants

To explain the global composition optimization, we use a very simple example. Consider an application that proceeds in rounds, where each round consists of two code parts. Each part is available in two variants A and B. The variants of each part consist of different code, and different data representation, so that combining variant A of part 1 with variant B of part 2 involves a conversion of the data representation between part 1 and 2, and another conversion between part 2 and part 1 of the next round. In the following, a_i and b_i denote the numbers of cycles for variants A and B of part i, respectively. Furthermore, c denotes the number of cycles needed for a conversion in either direction.

If $a_1 < b_1$ and $a_2 < b_2$, then clearly variant A should be chosen. The same holds if variant B is better in both parts. We consider the interesting case where $a_1 < b_1$ but $b_2 < a_2$. Let us assume that the differences are identical, i.e. $d = |a_1 - b_1| = |a_2 - b_2|$. Then the combination of variant A in part 1 and variant B in part 2 should be chosen if $2c < d$, because then

$$a_1 + c + b_2 + c < a_1 + a_2 = b_1 + b_2 \ .$$

Clearly, the same idea can be used if $|a_1 - b_1| \neq |a_2 - b_2|$, but more cases have to be considered to find the optimum. The idea can also be generalized to more than two parts and to more than two variants per part.

This problem has been investigated as *global composition optimization* [HK14], and treats the variant problem formally by modelling with a directed graph, where each variant for each program part is a node, attributed with its runtime, and arcs represent the flow of execution, where each variant of one part is connected to each variant of the following part (i.e. piecewise complete bipartite). If an arc connects different variants, it is attributed with the runtime of the conversion code. An example with 4 parts is depicted in Fig. 2 taken from [HK14]. The best combination of variants is found by computing the shortest path from source to sink. Note that optimization can be done over several rounds [HK16], so that performance improvements might even be possible if $c > d/2$.

4 Case Study

An application domain where mainly boolean operations and table lookups are performed is encryption. Thus, from the above, bit-serial representation looks advantageous. Also fixed bit permutations, which frequently arise in encryption, are easy with bit-serial representation as only the indices of the `as` variables have to be permuted accordingly. Biham [Bih97] already demonstrated that bit-serial representation leads to more efficient implementation of the Data Encryption Standard (DES). Similar approaches has been implemented for its successor Advanced Encryption Standard (AES) [KS09, RSD06].

We therefore illustrate the so far rather abstract idea of mixing word-parallel and bit-serial program parts with Advanced Encryption Standard (AES) as a concrete example. AES is a standard for a symmetric block cipher based on the Rijndael algorithm [DR00], chosen in 2000 by NIST as the successor to DES and published in 2001 as a standard [Nat01]. Encryption of a data block consists of a number of rounds (10 to 14, depending on key size), where each round consists of four steps operating on a 4×4-matrix of bytes: bytewise substitution, rotating the matrix rows by different stepwidths, mix the columns by multiplication with a constant matrix, and bitwise addition of the pre-computed round key to the data matrix. The following pseudo-code illustrates the computations.

```
AES(uint8 w[4][4]){ // input byte matrix
for(rnd=0..9){ // 10 rounds for 128-bit key
for(i,j=0..3) w[i][j] = tab[w[i][j]]; // byte substitution
for(i,j=0..3) wtmp[i][j] = w[i][(j+step[j])%4]; // shift rows
for(i,j=0..3){ w[i][j] = 0;
 for(k=0..3) w[i][j] += mul(cnst[i][k],wtmp[k][j]); } // mixcols
for(i,j=0..3) w[i][j] = w[i][j]^rndkey[rnd][i][j]; // add rndkey
}
return w;  }
```

There have been high-performance AES implementations in software for 8-bit and 32-bit microprocessors (e.g. the add-round-key step greatly profits on a 32-bit architecture), and also implementations in hardware for ASICs and FPGAs. In addition, there have been bit-serial implementations, where all steps have been expressed as evaluation of boolean functions, so that 32 block encryptions can go on in parallel if 32-bit variables are used [KS09, RSD06]. The reader might notice that the steps correspond closely to our code examples from Sect. 2, and the cited implementations proceed like this, in particular they give a formulation of the subbytes and mixcolumns steps expressed as boolean function evaluation.

We have implemented both variants[1] for a block and key size of 16 bytes and measured the runtimes in Tab. 1 on a Lenovo W530 with Intel Core i7-3630QM (Ivybridge) quad-core CPU (up to 2.4 GHz, with 3.4 GHz turbo), 20 GByte of RAM, Windows 7 operating and OpenWatcom C compiler. We encrypt one block of 16 bytes for 10 million times, and compute the resulting runtime. As the computation is independent of the concrete content of the byte matrix `w`, we used the same block in all encryptions. We do not claim to have

[1] For the word-parallel variant, we multiplied the measured times by 32 to get comparable results.

Part	word-parallel	bit-serial
subbytes	14337	38563
shiftrows	5788	0
mixcolumns	27908	5991
addroundkey	6427	10062
conversion	11856	11856

Table 1: Runtimes of bit-serial and word-parallel implementations of AES. Runtimes are given without dimension, as they are computed with `clock()` over 10 million repetitions.

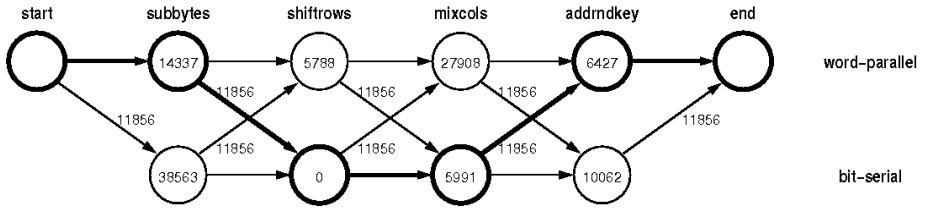

Figure 3: Flow graph of AES variants. Nodes and edges without mark have weight 0.

the fastest implementations for each part, but we strived to have a comparable level of code quality, so that the comparison is fair.

We clearly see that for the first and fourth steps (subbytes and addroundkey) the word-parallel variant is faster, while for the second and third steps (shiftrow and mixcolumns), the bit-serial variant is faster. Figure 3 depicts the flow graph of the variants with the shortest path highlighted[2]. Therefore, a mixed implementation looks like in the following pseudo-code (wp=word-parallel, bs=bit-serial).

```
AES(uint8 w[4][4]){ // input byte matrix
for(rnd=0..9){ // 10 rounds for 128-bit key
for(i,j=0..3) w[i][j] = tab[w[i][j]]; // byte subst wp
bittranspose(w); // convert representation to bs
shiftrow+mixcolumnbitserial(w); // do next two steps bs
bittranspose(w); // convert representation back to wp
for(i,j=0..3) w[i][j]=w[i][j]^rndkey[rnd][i][j]; // add rndk. wp
}
return w;  }
```

We need two conversions per round. We pack steps subbytes and addroundkey as part 1, and shiftrow and mixcolumns as part 2, and see that for part 1, variant A (word-parallel) is faster, while for part 2, variant B (bit-serial) is faster. We get

[2]Note that to get a complete picture, one also has to do the same with start and end in bit-serial representation. This however leads not to a shorter path in this case.

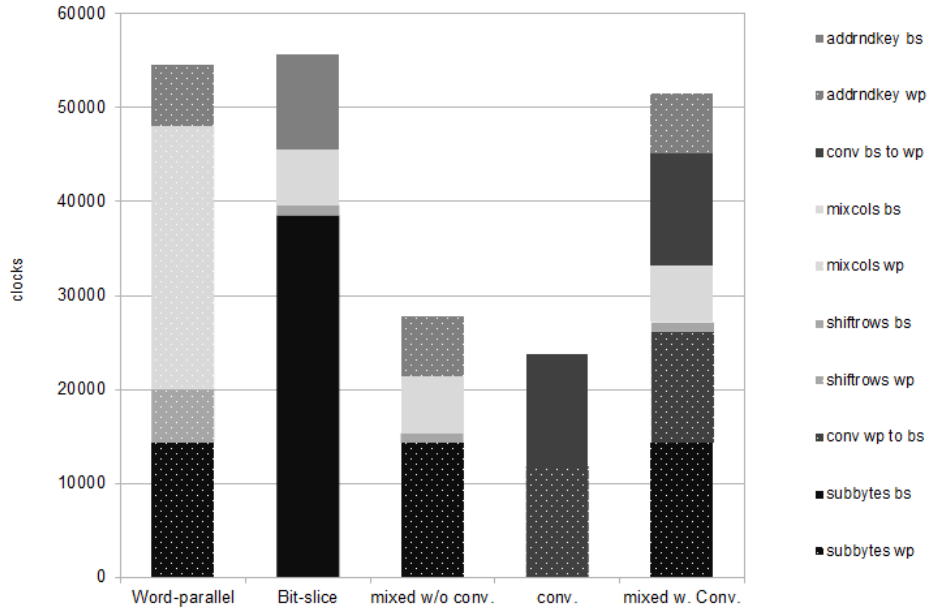

Figure 4: Runtimes of different code variants. Corresponding word-parallel (wp) and bit-serial (bs) code parts have same color but word-parallel variants are dotted. Bit-serial shiftrows really has runtime 0, but has been denoted by small rectangle for clarity.

Variant	Runtime
a_1	20764
a_2	33696
a_{tot}	54460
b_1	48625
b_2	5991
b_{tot}	54616
c	11856

Thus, the bit-serial and word-parallel variants almost have the same runtimes a_{tot} and b_{tot}. The optimal path (without conversion cost) is a_1 and b_2, the add-on for a pure variant is $d = 27861$ (or 27705, i.e. more or less the same for both variants). The double conversion cost is $2c = 23712$ which is lower, so that the runtime advantage of the mixed implementation is about 4000 or 7.4%. Figure 4 illustrates the different runtimes.

While this improvement seems not to be very large, it illustrates that in some applications,

it might be worthwhile to consider this choice. Please note that if conversion would be free (or almost free by providing hardware instructions for bit matrix conversion), then speed could almost be doubled.

5 Conclusions

We have presented a novel application of global composition of program variants: using both bit-serial and word-parallel variants of algorithmic parts in symmetric encryption. The bit-serial variants allow SIMD parallelism even within a single core, while the word-parallel variants serialize the threads of computation but avoid the overhead of bit-serial computation on data items when all bits of a word are needed.

Our case study demonstrates that the AES encryption algorithm can be accelerated compared to purely bit-serial and word-parallel implementations by combining the best parts of both. The massively parallel execution of AES frequently occurs for good and bad: in high-performance environments where multiple communications are encrypted simultaneously, and in dictionary attacks where attackers try to find the password by decrypting a known piece of text with all keys from a dictionary, exploiting the fact that many users still employ existing expressions as a key or password.

Future work will comprise investigation of other use cases, as well as more advanced uses: between computing on single bits (like in boolean function evaluation) and computing on full words (like in ordinary arithmetic computation), there are lots of in-betweens, like e.g. computations on bytes, that still could profit from SIMD parallelism. Such an approach has already been investigated in the frame of multiple executions for fault-tolerance (cf. [EFK09]), but not for performance improvement from parallelism.

Acknowledgements

Our sincerest thanks go to Erik Hansson and Christoph Kessler for long discussions about global composition, that inspired the present work.

References

[Bih97] Eli Biham. A fast new DES implementation in software. In *Proc. 4th International Workshop on Fast Software Encryption (FSE '97)*, pages 260–272. Springer LNCS, 1997.

[DR00] Joan Daemen and Vincent Rijmen. Rijndael for AES. In *AES Candidate Conference*, pages 343–348, 2000.

[EFK09] Klaus Echtle, Bernhard Fechner, and Jörg Keller. PAMOS and PAROS — Parallel Addition of Multiple or Redundant Operands in a Single Word. In *Proc. ESREL 2009*

Conference, 2009.

[HK14] Erik Hansson and Christoph Kessler. Global Optimization of Execution Mode Selection for the Reconfigurable PRAM-NUMA Multicore Architecture REPLICA. In *Proc. 2nd International Symposium on Computing and Networking (CANDAR 2014)*, pages 322–328, 2014.

[HK16] Erik Hansson and Christoph Kessler. Optimized Variant-Selection Code Generation for Loops on Heterogeneous Multicore Systems. In *Proc. International Conference on Parallel Computing (ParCo 2015)*, pages 103–112. IOS Press, 2016.

[KH89] B. A. Kahle and W. Daniel Hillis. The Connection Machine model CM1 architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 19:707–713, 1989.

[KS09] Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In *Proc. 11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009)*, pages 1–17. Springer LNCS, 2009.

[MvOV97] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[Nat01] National Institute of Standards and Technology (NIST). *Federal Information Processing Standards Publication 197: Announcing the Advanced Encryption Standard (AES)*. NIST, 2001.

[RSD06] Chester Rebeiro, David Selvakumar, and A.S.L. Devi. Bitslice Implementation of AES. In *Proc. 5th International Conference on Cryptology and Network Security (CANS 2006)*, pages 203–212. Springer LNCS, 2006.