Efficient and scalable bit-matrix multiplication in bit-slice format

Duco van Amstel
ENS Lyon - Département d'Informatique
15 parvis René Descartes
Lyon 69007, France
duco.van.amstel@ens-lyon.org

ABSTRACT

The bit-matrix multiplication (BMM) has until now only been implemented on the Cray supercomputers. Since then multiple publications have proved the usefulness of this instruction in symmetric-key cryptography, linear cryptanalysis and bio-informatics. In the same time the interest for parallel computing has spread to the field of commodity processors and the growth of multimedia extensions has brought the bit-slice data format under attention of many researchers and programmers. The associated Single Instruction Multiple Data programming has proven useful in the efficient implementation of parallel algorithms. However the combination of the BMM instruction with a bit-slice data format remains a challenge on these commodity processors. This investigation shows that general-purpose architectures can also benefit from the advantages of the BMM instruction that were previously only accessible to supercomputers. The proposed method requires some additional hardware support and can be adapted to all matrix-sizes as well as a variable number of parallel bit-slice streams. The investigation analyzes the performances achieved on the AES algorithm. Furthermore implementation details are presented including assembler optimizations and run-time code specialization.

Categories and Subject Descriptors

B.2.4 [**Arithmetic and logic structures**]: High-speed arithmetic; E.1.m [**Data structures**]: Miscellaneous
; D.1.m [**Programming techniques**]: Miscellaneous

General Terms

Performance, Algorithms

Keywords

Bit-slicing, Bit-Matrix Multiply, Bit-Wise Logic,
Advanced Encryption Standard, Hybrid Specialization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SAC 2012 March 25-29, 2012, Riva del Garda, Italy.
Copyright 2011 ACM X-XXXXX-XX-X/XX/XX ...$10.00.

1. INTRODUCTION

A bit-matrix multiply (BMM) instruction is similar to a numerical matrix multiplication while replacing numerical multiplications by bit-wise AND operations and each numerical addition by a bit-wise XOR. The first and only processor architectures to thrust this instruction were the Cray supercomputers [5]. This instruction is used in multiple fields including symmetric-key cryptography, linear cryptanalysis [10] and bio-informatics [13]. More recently a paper by Hilewitz, Lauradoux and Lee [11] emphasized again the usefulness of such an operation. However the hardware synthesis of this facility can be costly. The main reason is that a BMM requires bit-level precision whereas the vast majority of architectures only support byte-level precision [11]. This limits the optimization possibilities for software implementations of BMMs. These software implementations are typical examples of programs where there is need to apply operations to partial data. In other words it uses a processor instruction for only a small part of the operands [2].

At some occasions the lost computing power is even greater as the operation requires to reorganize the data in a different way. An example of this is would be a shift of bits before applying a XOR to a single byte. This contradicts the 32 or 64-bit data-path of most modern processors. As a consequence this can result in an overhead of up to 7 bytes out of 8 of the computational power of the XOR instruction in the case of a non-optimized implementation on a 64-bit processor. The bit-slice data format makes it possible to limit the loss in computational power by lower-than-optimal use of the data-path width. However this data format does not diminish the need for a high number of instructions due to the many XOR operations involved in a BMM as well as the necessity to perform partial bit-wise operations on the bit-matrix by which the data is multiplied.

The contributions of this investigation are as follows :

- It shows that hardware support for a BMM instruction of a size superior to 8×8 is very costly in space whereas modern commodity processors have a 32 or 64-bit data-path
- It proposes a fast BMM of bit-sliced data by a given matrix that exploits a bit-wise logic unit (BWLU) as is presented in [9].
- It presents an use of *hybrid specialization* to accommodate for each use-case as the proposed method for BMMs varies with the bit-matrix.

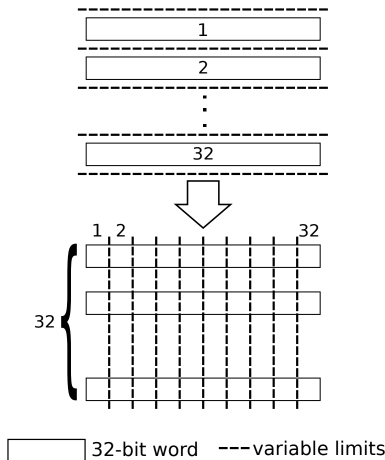

Figure 1: Data reorganisation in bit-slice format

The remaining part of the paper is organized as follows. The bit-slice data format that improves the usage of existing processor instructions is presented in the first section. In the next section potential hardware supports for BMMs are evaluated for their cost in terms of circuitry size. Among them the BWLU instruction then enables an efficient software implementation for a BMM. An example with a bit-sliced version of the Advanced Encryption Standard [14] will then be presented along with the achieved performances based on empirical results. The proposed method has parameters specific to each use-case and thus the code must be dynamically generated and an algorithm for this task is presented in the last section.

2. BIT-SLICE DATA REPRESENTATION

In order to accelerate encryption and decryption with the Data Encryption Standard (DES) algorithm Elie Biham [2] introduced a new data representation. Bit-slice processing exploits a single n-bit processor as n different 1-bit processors that can be used in a Single Instruction Multiple Data (SIMD) fashion. We then speak of an n-bit bit-slice. Data is reorganized so that the i-th n-bit word contains the i-th bit of n different data blocks as in figure 1. Operating on this data format can be as easy as using a n-bit XOR on words k and l to obtain the 32 XOR results of bits k and l of the 32 original data blocks.

This method takes advantage of all the computational power of bit-wise instructions without a single unused result bit. Since its first introduction this technique has found a large domain of application in cryptography as many ciphers are based upon linear and non-linear algebra as well as a wide use of logical bit-wise operations. The more recent Advanced Encryption Standard (AES) has been bit-sliced in different ways for an equal number of different architectures [3, 14].

Bit-slicing has certain limits as only bit-wise instructions can be implemented effectively in this alternative data format. Operations such as adding, multiplying and modular dividing are more complex in this format than in a standard representation. One of the remaining challenges for programmers and for hardware designers is the efficient implementation of non-bit-wise instructions in bit-slice data format. An illustration of this may be found in [9] where the authors evaluate the balance between hardware cryptographic facilities and software implementations. A particular case of a complex operation that raises an issue is the bit-matrix multiplication (BMM). Current implementations range from optimized architecture-specific assembler code to dedicated hardware instructions.

3. THE COST OF HARDWARE SUPPORT

In terms of performance the best implementation for a BMM is a full hardware mapping. Nonetheless this choice can result in prohibitive costs and because of that it is often in software that the differences between the average BMM and the high-speed versions appear. This investigation proposes a trade-off between hardware and software by using a BWLU as presented in [9] capable of efficiently computing complex logical operations by means of a lookup table.

3.1 Costly Bit-Matrix Multiplications

In the linear algebra of the Galois Field of dimension 2^k GF(2^k) the bit-matrix multiplication is a very frequently used operation, specifically in cryptography. The Advanced Encryption Standard (AES) algorithm for example uses multiplications of bit-vectors by bit-matrices at different stages [1] and some optimized implementations use them to change the mathematical base on which the data is represented [4, 14]. The importance of this particular operation is underlined in [11] together with the study of a hardware mapping of bit-matrix-matrix multiplications as well as bit-matrix-vector multiplications.

For an n-bit bit-slice to be processed by a hardware unit this unit should accept a matrix size with at least n columns as each column represents a bit-slice stream. However such a mapping would have a high cost in circuitry. Hardware synthesis with a Virtual Hardware Description Language has been done on different sizes. Table 1 illustrates the growth in circuitry size. For example the switch from 8×8 to 16×16 matrices multiplies the processed data by 4 whereas the necessary circuitry increases by a factor 8. This cost will only grow with the size of the supported bit-matrices. Even a 32×8 unit multiplies the required circuitry by 4. Furthermore a unit of greater size needs an equivalently increased data-feed which in his turn impacts on the required circuitry.

This altogether advocates for a restriction of hardware BMMs to sizes of at most 8×8 bit-matrices on most modern parallel processors as is recommended in [11]. On the other hand the 32 to 64-bit nature of these processors encourages the use of a 32 or 64-bit bit-slice. This causes an incompatibility with the size of the BMM hardware instructions. The solution proposed in section 4.1 exploits the power of a BWLU to take full advantage of the bit-slice data format.

Matrix size	Gates	Area
8×8	850	758 μm
16×16	6656	6023 μm

Table 1: Circuit cost with standard 28nm cells

3.2 Bit-wise logic

3.2.1 Standard architectures

A significant part of linear algebra in GF(2^k) and cryptographic algorithms is the usage of logical operations on data in a bit-wise manner. The bit-slice data format has the advantage of reducing or eliminating computational overhead induced by shift operations on standard data formats. When operating in bit-slice a shift is equivalent to an offset in the memory array representing the data. To illustrate the efficiency of bit-slicing the following example can be taken on 32-bit variables in C :

```
x = ((y<<17)|(y>>15)) & ((z<<5)|(z>>27));   (1)
```

In x86 assembler the fastest way, excluding contextual optimizations, to implement (1) would be :

```
movl %eax $1
movl %edb $2
roll %eax 17
roll %edx 5
andl %eax %edx
movl $3 %eax
```

For a total of 32 x86 DWORDS[1] one should take into account that the latest x86 chips (x86-64) provide sufficient resources to operate upto 3 arithmetical instructions, a load and a store during a single clock-cycle. This gives a theoretical clock time of 128 cycles to process all 32 variable sets.

When taking a 32-bit bit-slice data is represented as arrays where the i-th element represent the 32 i-th bits of all variables. Then the C code becomes :

```
for (i = 0; i < 32; i++) {
   x[i] = y[(i+17)&0x1f] & z[(i+5)&0x1f];
}
```

The assembler code for a bit-sliced version of (1) rends a theoretical clock time here is 100 cycles because of branch prediction. The performance gain is thus approximately 22%.

Going a step further a more complex logical function using more than two operands could be studied. For example :

```
x = ((k>>2) ^ (l<<7)) &
                 (((l>>9)^m) | (l^n));       (2)
```

Bit-slicing the whole operation would again turn out a performance gain because of the absence of rotate operations.

3.2.2 Using a Bit-Wise Logic Unit

An alternative option would be to use a hard-wired bit-wise logical operation as proposed in [9]. Such a hardware instruction is capable of taking several operands and returning the result of a complex logical operation applied to the input in a bit-wise fashion. This is done by means of a lookup table that is also given as operand to the BWLU. The result of the operation is calculated by taking the operand bits as an index value for the lookup. This lookup table can be interpreted as the truth-table of the logical function that is simulated. A diagram of this functioning can be found in figure 2. Another advantage is the circuitry size as a 4-way BWLU can be synthesized with 160 gates on a 370 μm surface[2]. From the programmers point of view the assembler code using a BWLU is significantly shorter.

[1] An x86 DWORD is 32-bit long

From here on and for the rest of this report the low-endian notation is supposed for the given examples and the considered architectures are supposed to be low-endian as well. Assume a processor with a BWLU as in figure 2 counting at least two 64-bit registers named `r1`, `r2` that can be addressed as four 32-bit registers `r1l`, `r1h`, `r2l`, `r2h`. The assembler code for 2 would resemble :

```
load r1l = 0[$1]
load r1h = 0[$2]
load r2l = 0[$3]
load r2h = 0[$4]
bwlu r1l = r1, r2, 0x0770
store 0[$0] = r1l
```

Thus 6 instructions are sufficient for a parallel computing of a single bit for each of the 32 input streams. If the processor supports 64-bit loads and the variables are appropriately stored in memory this can be reduced to four instructions. Even further, if parallel loads and stores are available a continuous throughput of 32 bits every three clock cycles can be sustained. As a result this BWLU can apply an arbitrary logical function to 32 streams of four 32-bit variables in less than 100 clock cycles. This is equivalent to an efficiency of at least 1 byte per cycle.

More specifically in the case of a BMM a BWLU can be used as a 4-way XOR operation. Because the XOR instruction makes up for the largest part of a traditional bit-sliced BMM a significant speed-up can be expected. To support this claim a detailed example of code and performance evaluation will be presented in sections 4.2 and 5 of this report.

3.3 Scalability

This method of implementing a BMM involving a BWLU as a multi-issue XOR does not limit the size of the data-path to which it can be applied. The code that has been presented can be adapted to whatever the size of the data-path of the BWLU. Furthermore it would be possible to enlarge the number of operands enabling lookup tables greater than the presented 4-to-1 format. For bit-slice widths that are larger than the data-path of the BWLU the bit-sliced variables can be processed in multiple passes, hence the scalability.

[2] Using the same 28nm cells technology

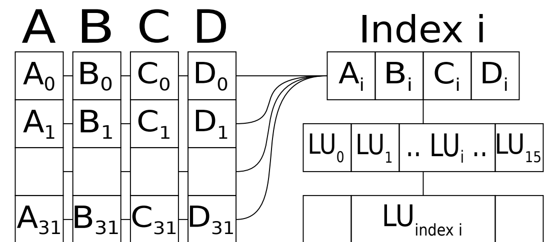

Figure 2: Functional diagram of a 4-way BWLU unit with A, B, C & D as operands and LU a 16-bit lookup table

Another advantage of the BWLU is that such scalability does not imply overhead as the code only depends on the size of the instruction data-path width and not on the bit-slice data-path width.

4. CASE STUDY : BASIS-CHANGE IN THE ADVANCED ENCRYPTION STANDARD

The Advanced Encryption Standard (AES) as defined by the National Institute of Standards and Technology (NIST) in 2001 [1] has been implemented in various ways exploiting a wide range of programming techniques and mathematical interpretations [3]. The bit-slice data representation is one of the techniques that has been used to implement the AES. The implementation of Rebeiro, Selvakumar and Devi in [14] is particularly interesting for their combination of multiple steps of the algorithm into single computational stages. A particular stage of their version of the algorithm involves an algebraic basis-change at byte-level of the data that is being processed. This is equivalent to a BMM by a fixed matrix that is known at the time of compilation. The performed shortcut is represented on figure 3.

4.1 Typical code

From a mathematical point of view the basis-change consists in multiplying each byte b by a matrix X to obtain byte $b' = X \cdot b$. This operation is applied to all bytes of the data. It should be noted that only the positive bit values in the matrix need to result in a XOR operation. In order to operate as a 4-issue XOR the BWLU instruction uses the hexadecimal constant `0x6996` as it is the truth-table of this function. Some XOR operations can be grouped among the lines of the matrix to compute intermediate values. Here we will use the following matrix as an example :

$$X = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

In the case of this particular matrix the 32 initial XOR operations are reduced to 8 BWLU instructions. This results in a very compact code.

The drawback of this method is that it has to be recoded for every matrix with variable costs but the maximum number of XOR operations that are required is limited to 64 and thus, while neglecting register copy costs, gives a rough maximum cost of $64/4 = 16$ BWLU instructions.

4.2 Achieved performances

For experimentation and measure purposes the previously presented software BMM has been implemented on a general purpose Very Large Instruction Word[3] (VLIW) processor [7, 8] with 32 and 64-bit capabilities. The functional core is derived from the Lx Processor Family [6, 16]. This architecture implements a double 4-way BWLU that applies two lookup tables simultaneously to four 16 or 32-bit operands with a one clock cycle latency. Such a feature enables the possibility to execute any pair of 2, 3 or 4-issue XOR on four fixed operands. Another feature is an operand bypass making it possible to feed the result of an instruction directly as operand to an instruction of the next cycle. The VLIW bundles on this architecture can each accommodate up to:

- 1 Branch/Condition instruction
- 2 Arithmetical/Logical instructions
- 1 Load/Store instruction

Furthermore the Branch/Condition and Load/Store units can perform a subset of the Arithmetic/Logical instructions for their normal functioning. The assembler language for this architecture delimits each instruction bundle by a double semi-colon.

Using this architecture the total cost of the optimized code is 10 instruction bundles to process a byte of all 32 bit-slice streams. This count includes the load and store instructions necessary to access the data from memory. Due to cache misses this accounts for a period variable between 13 and 35 clock cycles. These measures were obtained by direct simulation of the architecture by means of hardware description languages.

[3]VLIW processors implement instruction parallelism by performing a bundle of instructions per clock-cycle

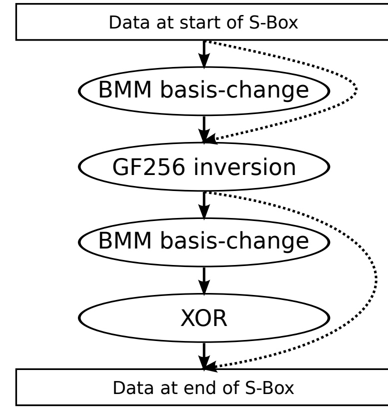

Figure 3: Schematics of the AES S-Box using composite field algebra. The dotted lines represent steps in a bit-sliced implementation.

4.3 Comparing to SSE extensions

The bit-sliced version of AES presented in [14] has been benchmarked on different representatives of the x86 architecture. To increase the performances the authors have efficiently exploited the SIMD capabilities of these processors

Architecture (data-path width)	Cycles/encryption
Pentium 4 (128-bit)	334
Athlon 64 (128-bit)	210
Core 2 (128-bit)	102
VLIW (32-bit)	468
VLIW (128-bit) (estimated)	117

Table 2: Cycles spent on the S-Box per block per encryption

due to their Streaming SIMD Extensions (SSE) [15]. These extensions allow to process data of a size up to 128 bits in a SIMD fashion. The results presented in their paper indicate that the S-Box operations in an AES-256 encryption take 75% of the processing time. Table 2 summarizes the number of cycles per encryption that are spent on the S-box on different architectures including the one used in this investigation as well as the ones in [14].

The higher value on the VLIW architecture can be explained by the data-path that is 4 times smaller. These figures should be compared while keeping in mind the difference in data-path width. Moreover the SIMD structure of the SSE is less flexible than the Multiple Instructions Multiple Data (MIMD) structure of a VLIW architecture.

5. DYNAMIC CODE GENERATION

The remaining part of this investigation supposes the use of the previously described VLIW architecture. There are architectural constraints on the number of simultaneous register file read operations. This means that the lookup tables for the BWLU instruction have to be given as immediate values and should thus be known at compile-time. This could be a problem due to the specificity of the lookup tables to the bit-matrix by which the data is multiplied. To enable a general use of the proposed implementation of BMMs the proposed solution has been sought in the domain of Just-In-Time compilation.

5.1 Fixed versus variable code structure

In a search for an optimal solution regarding the number of required instruction bundles one of the necessities would be to find common sub-vectors to the bytes of the bit-matrix by which will be multiplied. This would allow to regroup a maximum of the XOR instructions within the same BWLU instruction. Such a search can be computationally heavy and the resulting variable code structure would complicate the code generation process as more parameters are introduced. An alternative non-optimal solution would be to have a fixed code template that can accommodate for all possible bit-matrices with a minimal number of parameters accounting for the unique structure of each matrix.

The size of the generated code for an 8×8 BMM on a 32-bit bit-slice can be evaluated for both the optimal and the non-optimal solutions on the previously presented architecture. The 32-bit words that should be multiplied by our fixed bit-matrix can be loaded by 64-bit loads and the result can be stored in a similar manner. This alone accounts for 8 instruction bundles. Supposedly a minimalist bit-matrix will only involve a single BWLU instruction. Thus the minimum of 9 instruction bundles will be required. On the other hand a general purpose template can be coded within 14 bundles by means of pipe-lining an example of which will be shown in the next part of this report.

Moreover it should be considered unlikely for bit-matrices to generate a minimum number of BWLU instructions except for sparse matrices. Empirical examples such as presented in section 4 advocate for an average of 8 BWLU instructions which leads to 12 instruction bundles. This means that there would be a 2 bundle-per-byte gain between the optimal and the non-optimal solution. The ability to absorb the additional overhead for the optimal generation thus depends on the number of reuses of the generated code. However a full optimization can hardly be expected to take up less than 1000 bundles which puts the break-even point between the two solutions at a minimum of 500 code reuses.

```
load64 r1 = 0[r01]
;;
load64 r2 = 8[r01]
;;
load64 r3 = 16[r01]
;;
load64 r4 = 24[r01]
;;
bwlu r5 = r1,r2,lu2_lu0
;;
bwlu r7 = r1,r2,lu6_lu4
;;
bwlu r6 = r3,r4,lu3_lu1
;;
xor r13l = r5l,r6l
xor r13h = r5h,r6h
bwlu r8 = r3,r4,lu7_lu5
;;
xor r14l = r7l,r8l
bwlu r9 = r1,r2,lu10_lu8
store64 0[r01] = r13
;;
xor r14h = r7h,r8h
bwlu r10 = r3,r4,lu11_lu9
;;
xor r15l = r9l,r10l
bwlu r11 = r1,r2,lu14_lu12
store64 8[r01] = r14
;;
xor r15h = r9h,r10h
bwlu r12 = r3,r4,lu15_lu13
;;
xor r16l = r11l,r12l
xor r16h = r11h,r12h
store64 16[r01] = r15
;;
add r01 = r01,32
store64 24[r01] = r16
;;
```

Table 3: ASM template for run-time code specialization on an 8×8 BMM

5.2 Code specialization

The principle of using a fixed code template where only some variables or parameters are changed at run-time is an example of what is called *hybrid specialization* [12], halfway between static and dynamic compilation. To extract a code template that is common to all possible bit-matrices we divide each matrix in 16 nibbles[4]. Each nibble can be interpreted in two ways:

1. The i-th bit indicates the use or not of an additional XOR in the computation of the BMM.
2. The value of the nibble is an index into a table of lookup constants for a BWLU that represent all possible XOR groupings on the four variables.

On the VLIW architecture that is used for testing a fixed code template would look as is shown in table 3. All registers are 64-bits wide with upper and lower-half addressable parts. The `bwlu` instruction computes two lookup tables on the given variables. The 16 nibbles of the fixed bit-matrix are mapped to the 16 lookup tables that are used in the code template as `lu0`, `lu1`, ..., `lu15` by means of an array. The presented code performs in fact a bit-matrix-vector multiplication as only a single byte of each bit-slice stream is computed. By encapsulating this code in an instruction loop one can obtain the desired bit-matrix-matrix multiplication on as many bytes as necessary.

In case one would like to implement a BMM of a different size the code template can be expanded. A 16×16 BMM

[4]A nibble is a 4-bit unit

can for example be seen as 4 seperate 8×8 BMMs with some additional XOR operations to link the different results together.

5.3 Specialization cost

To generate the specialized code from the template for an 8×8 BMM it is necessary to load the 16 possible lookup tables. Then the ones to use should be selected based on the nibbles of the bit-matrix before integrating them into the binary code. The cost of this has been evaluated at 54 bundles including the loading of the bit-matrix and the writing of the specialized code to memory. Considering the speed-up that is gained over a traditional BMM by the usage of the BWLU the cost of code-specialization is absorbed within a single code run. This confirms the earlier statement that the *hybrid specialization* method is of great efficiency for a low to intermediate number of code reuses. For a high number of code reuses the generation of an optimal code may render even better performances.

The hypothesis that the lookup tables for the BWLU instruction may be kept in registers does not eliminate the usefullness of this method. The only consequence would be that the lookup tables are not written to memory as immediate values but are kept in registers instead. This eliminates the initial overhead of the specialization cost.

6. CONCLUSION & FUTURE WORK

This investigation draws a picture of the methods currently used for bit-matrix multiplications and proposes a new way of implementing them in software with limited hardware support. Assuming that the data is stored in the bit-slice parallel format it is possible to make use of a specialized instruction, the Bit-Wise Logic Unit, to efficiently compute a Bit-Matrix Multiplication with a matrix that is known at compiling time. Empirical results have been obtained to support this claim and a case-study has been made to demonstrate the use of the method. This altogether would be an argument in favor of future platforms thrusting such a hardware facility in their instruction set.

Another result is the effective use of dynamic code generation. To overcome the cost of coding a new version of the proposed algorithm for each particular bit-matrix it is possible to use *hybrid specialization* to adapt the binary code to specific bit-matrices with minimal overhead. The code that is obtained is fully scalable and can be adapted to various bit-slice formats and bit-matrix sizes without effort nor computational overhead.

Future improvements would involve an algorithm for the generation of optimal code as well as the possibility to feed the lookup table as register content to the BWLU instruction instead of using immediate values. Moreover this investigation does not analyze the performance of other optimizations on logical operations due to lookup-table use.

7. REFERENCES

[1] Specification for the advanced encryption standard (aes). Federal Information Processing Standards Publication 197, 2001.

[2] E. Biham. A fast new des implementation in software. Technical Report CS0891, Technion, 1997.

[3] J. W. Bos, D. A. Osvik, , and D. Stefan. Fast implementations of aes on various platforms. Cryptology ePrint Archive, Report 2009/501, 2009.

[4] D. Canright. A very compact rijndael s-box. Technical report, Naval Postgraduate School, California, 2005.

[5] Cray. *Cray Assembly Language (CAL) for Cray X1 Systems Reference Manual, version 1.2.* Cray Inc., 2003.

[6] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: a technology platform for customizable vliw embedded processing. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 203–213, New York, NY, USA, 2000. ACM.

[7] J. Fisher, P. Faraboschi, and C. Young. *Embedded computing: a VLIW approach to architecture, compilers and tools.* Morgan Kaufmann, 2005.

[8] J. A. Fisher. Very long instruction word architectures and the eli-512. In *Proceedings of the 10th annual international symposium on Computer architecture*, ISCA '83, pages 140–150, New York, NY, USA, 1983. ACM.

[9] P. Grabher, J. Großschädl, and D. Page. Light-weight instruction set extensions for bit-sliced cryptography. In *Cryptographic Hardware and Embedded Systems - CHES 2008*, pages 331–345. Springer Verlag LNCS 5154, August 2008.

[10] Y. Hilewitz. *Advanced Bit Manipulation Instructions: Architecture, Implementation and Applications.* Phd thesis, Princeton University, 2008.

[11] Y. Hilewitz, C. Lauradoux, and R. B. Lee. Bit matrix multiplication in commodity processors. In *Application-specific Systems, Architectures and Processors - ASAP 2008*, July 2008.

[12] M. A. Khan, H. P. Charles, and D. Barthou. An effective automated approach to specialization of code. In V. Adve, M. J. Garzarán, and P. Petersen, editors, *Languages and Compilers for Parallel Computing*, pages 308–322. Springer-Verlag, Berlin, Heidelberg, 2008.

[13] J. D. Maltby. The cray biolib: A high performance library for bioinformatics applications. In *45th Cray User Group Conference Proceedings.* Cray Inc., 2003.

[14] C. Rebeiro, D. Selvakumar, and A. Devi. Bitslice implementation of aes. In D. Pointcheval, Y. Mu, and K. Chen, editors, *Cryptology and Network Security*, volume 4301 of *Lecture Notes in Computer Science*, pages 203–212. Springer-Verlag, Berlin, Heidelberg, 2006.

[15] S. T. Thakkar and T. Huff. Internet streaming simd extensions. *Computer*, 32:26–34, December 1999.

[16] S. Wong, T. V. As, and G. Brown. ρ-vex: A reconfigurable and extensible softcore vliw processor, 2009.