Performance Comparison of 5 AES Candidates with New Performance Evaluation Tool

Masahiko TAKENAKA, Naoya TORII, Kouichi ITOH, Jun YAJIMA
FUJITSU LABORATORIES LTD.
{takenaka, torii, kito, jyajiama}@flab.fujitsu.co.jp

Abstract. We compared the performance of 5 AES candidates, with a new performance evaluation tool that we have developed. This tool automatically evaluates the results of a tune-up implementation without any manual tune-up so that it figures out the lower bounds of performance on real platforms. With this tool, we evaluated the performance of the 5 AES candidates on Pentium II, UrtraSPARC and Itanium systems. Rijndael and Twofish attained the highest performances across all of these platforms, so we consider these two algorithms as good candidates for AES algorithms from the point of view of performance.

1. Introduction

In this paper, we are comparing the performances of 5 AES candidates [1][2][3][4][5]. For a fair comparison, each candidate must be implemented as fast as possible on the intended platform. Fast implementations, however, usually use a heuristic approach, so the resulting performance depends greatly on the know-how of the implementers. To solve this problem, we have developed a performance evaluation tool that enables a fair comparison on real platforms, such as Pentium, UltraSPARC, and Itanium.

A fast implementation has two approaches: one is an improvement of the algorithm, and the other is a tune-up of the implement code. An improvement of the algorithm realizes higher speed processing by changing the structure of the algorithm and the instructions. This improvement is difficult to do work automatically. A tune-up of the implement code enables higher speed processing by choosing the best instruction order for a platform, a suitable usage of the variables that are dependent on the platform, and suitable options for a compiler.

In this paper, we are introducing our tool that can automatically evaluate the results of tune-up implementations without any manual tune-up. In short, our tool can evaluate the lower bounds of the performance on real platforms for any given algorithm.

Our tool evaluates performance under two conditions to achieve the automatic optimization and parallelism. The first condition is that the number of registers is to be unrestricted, and the second condition is that the algorithm should be expressed without branches and loops. We think these conditions are reasonable because of the following factors. About the first condition, the latest processors have a lot of registers. As for the second condition, branches and loops are not used for fast implementation of the symmetric key encryption algorithm.

We evaluated the performance of 5 AES candidates on Pentium II, UltraSPARC, and Itanium system using our tool. The evaluation values on UltraSPARC are in a range from 80% to 90% of the measurement. This is a good index as a lower bound on UltraSPARC because it is RISC processor with a lot of registers. On the other hand, the evaluation values on PentiumII are in a range from 80% to 90% of the measurement of an implementation in assembly language and 60% to 70% in C language. This is a good index in assembly language, but not a good index in C language. Pentium II does not have many registers, so the measurement depends on an efficient usage of registers. We think that our tool can evaluate performance when registers are handled efficiently. Lastly, we also evaluated the performance on Itanium, but we cannot measure the performance of Itanium because it is not available to the general public, and only its architecture has been announced. However, we think our evaluation is good because

Itanium has many registers.

In our comparison of 5 AES candidates using these evaluation criteria, Rijndael and Twofish attained the highest performances in our evaluated platforms. Therefore, we consider these two candidates are good as AES algorithm from the perspective of performance.

2. Design Policy

The purpose of our tool is to evaluate the lower bounds of performance, that is, the minimum number of clock cycles for processing an encryption algorithm. Our tool evaluates the number of clock cycles, but it does not simulate an operation; moreover, our tool does not support automatic optimization for improvement of the algorithm because such support is technically difficult.

The design of our tool is based on the following four objectives.

2.1. Support of multiple platforms

Our tool can evaluate performance on a variety of platforms using information about both the platforms and instruction set. That is, our tool can evaluate performance by defining a target platform.

More precisely, the platforms are defined by two types of files. One type of file is a definition of the number of pipelines provided by a platform. The other type is the definition of the instruction set. The instruction set should be definition based on the assembly instructions of the platform.

2.2. Tiny programming language

Our tool evaluates the algorithm expressed in the tiny programming language, which requires the instruction set to be defined beforehand; furthermore, it does not support loops and branches.

Loop and branch operations are not supported for two reasons. First, these operations are not used for fast implementation of the symmetric key encryption algorithm. Second, these operations cause difficulty in automatic optimizing and parallel processing.

The tiny programming language enables easy expression of the algorithm by a hierarchical subroutine, which is expanded like a macro during evaluation.

2.3. Automatic optimization and parallelism

In general, since the output of automatic optimization and parallelism cannot be expected to easily become an ideal code for use in a the widely used compiler, our tool provides ideally optimization under two limitations.

One limitation is that loops and branches are not supported (see 2.2). Our tool optimizes neither branches nor loops, which are thought to be difficult in compiler design.

The second limitation is that the number of registers on a platform is assumed to be unrestricted, which is a kind of idealized processor. For evaluating the performance, some works have evaluated the ideal performance of an ideal processor [6][7], but no work has been reported about evaluating the ideal performance on an actual platform. With this second limitation, our tool can solve the difficult problems for automatic optimization of register allocation because tune-up implementation are difficult to accomplish by changing the instruction order when the number of registers is limited in an actual platform. Furthermore, all state variables can easily be allocated with the registers. Recent platforms, such as RISC and IA-64, have a lot of registers, so the effect of this idealization is expected to be small, and our tool is expected to provide a good evaluation. On the other hand, CISC processors, such as Pentiums, have fewer registers, so any evaluation depends heavily on the coding technique. In other words, the evaluation will be good when the registers can be used efficiently, satisfying this limitation.

Considering these two limitations, we solved the difficulties of automatic optimization and parallelism so that automatic optimization and parallelism can be ideally performed with the basic optimization techniques of compiler design [8].

Ideal optimization and parallelism give algorithms a performance with lower bounds (i.e., minimum clock cycles). All evaluation values with this tool have lower bounds because our tool provides optimization and parallelism ideally.

2.4. Visualization of evaluation results

This tool includes a simple viewer that shows results of an evaluation. This viewer shows the optimized instruction order, processing clock speed, and processing pipeline based on the expressed algorithm. These results can be used as a reference for implementing real code on the platform.

3. Configuration of Evaluation Tool

Figure 1 shows the construction of the evaluation tool. Our tool consists of a platform definition file, programs in the tiny programming language, an evaluation program, an evaluation results file, and a viewer program.

3.1. Platform definition files

Platform definition files must be prepared for each platform, and they consist of two types of files: function definition files, and a pipeline definition file. Function definition files define the instruction set. The pipeline definition file defines the number of pipelines.

A function definition file is defined for each instruction and contains the following information:

- Instruction name
- Grammar of input/output
- Executable type of pipeline
- Clock cycles and latency

In our tool, the processing of an instruction is not defined because it is not simulated.

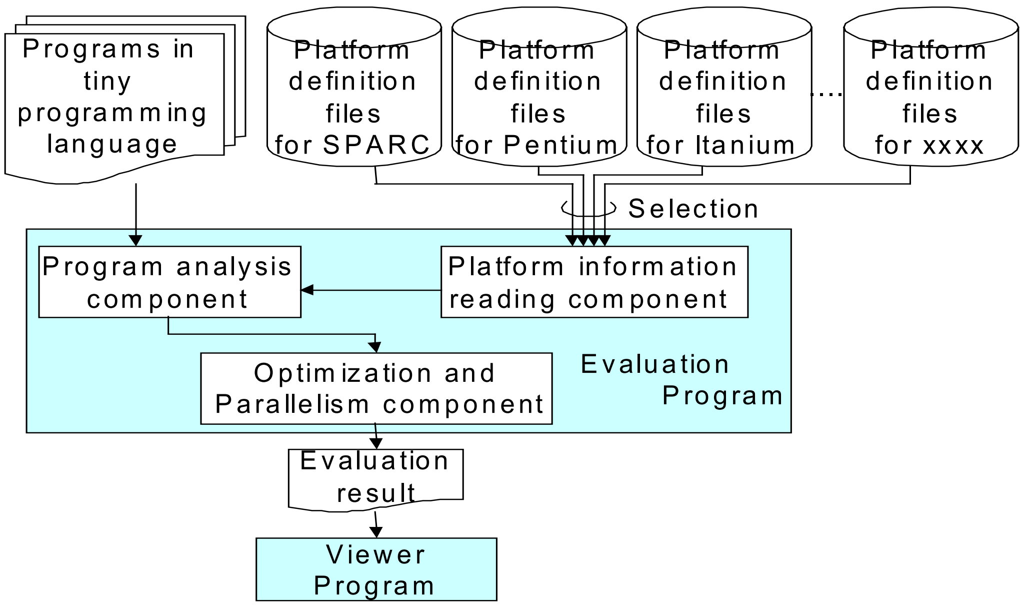

Programs in tiny programming language
Platform definition files for SPARC
Platform definition files for Pentium
Platform definition files for Itanium
Platform definition files for xxxx
Selection
Program analysis component
Platform information reading component
Optimization and Parallelism component
Evaluation Program
Evaluation result
Viewer Program

Figure 1. Structure of the evaluation tool

3.2. Programs in tiny programming language

The tiny programming language consists of definitions of a function, routine, and constant.

· Function definition

Functions are defined, such as

```
(output variable list) = Instruction (input variable list).
```

All variables used for defining functions are temporary. These variables define the data relationship among instructions, and their names are ignored when they are converted to the internal data form.

The grammar of input/output follows the same rules as in the function definition files.

· Routine definition

Routines are defined between

```
 #begin ROUINTE_NAME
```

and

```
 #end ROUTINE_NAME,
```

including one or more functions.

Routines are expanded as macros to be evaluated as sequential functions when they are converted to the internal data form.

· Constant definition

Constants are defined as `#constant VALUE` and processed as an argument of the main routine during the evaluation.

3.3. Evaluation program

The evaluation program consists of a platform information reading component, program analysis component and optimize-and-parallelism component.

· Platform information reading component

Platform definition files are stored in a separate directory for each platform. The files are read as a plug-in when a platform is specified, making the change of a platform easy.

· Program analysis component

This component interprets the algorithm expressed in the tiny programming language and converts it into a sequential program. The component then combines the sequential program with platform information, and it generates an internal data form (function tree). Critical path searches are also processed in this component.

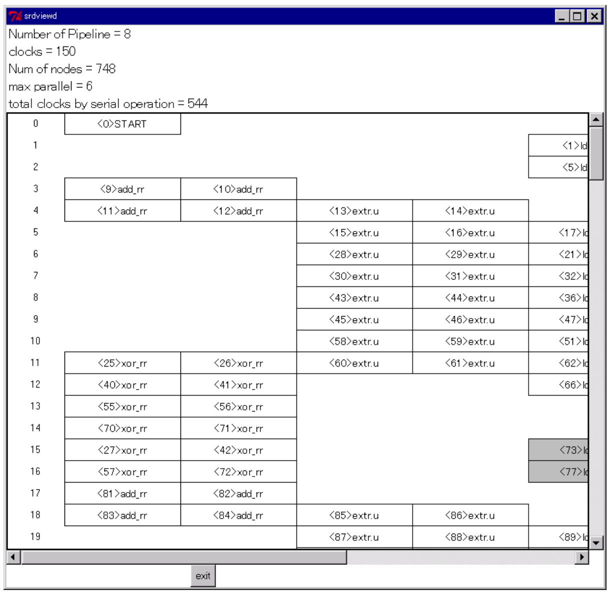
· Optimization and parallelism component.

In this component, the internal data form is scheduled in pipelines using the simplified technique of Gibbons and Muchanic [8].

A function tree is scheduled from the bottom to the top. This scheduling is effective for suppressing the number of registers to their minimum.

3.4. Evaluation results file

An evaluation results file contains the output from the optimization and parallelism. The viewer program displays the contents of this file. The purpose of separating the viewer program and the evaluation program is to quickly provide the results of an evaluation for display by viewer program. In other words, processing to evaluate the algorithm requires more time compared to that required by the viewer program.



```
Number of Pipeline = 8
clocks = 150
Num of nodes = 748
max parallel = 6
total clocks by serial operation = 544
```

0	<0>START				
1					<1>ld
2					<5>ld
3	<9>add_rr	<10>add_rr			
4	<11>add_rr	<12>add_rr	<13>extr.u	<14>extr.u	
5			<15>extr.u	<16>extr.u	<17>ld
6			<28>extr.u	<29>extr.u	<21>ld
7			<30>extr.u	<31>extr.u	<32>ld
8			<43>extr.u	<44>extr.u	<36>ld
9			<45>extr.u	<46>extr.u	<47>ld
10			<58>extr.u	<59>extr.u	<51>ld
11	<25>xor_rr	<26>xor_rr	<60>extr.u	<61>extr.u	<62>ld
12	<40>xor_rr	<41>xor_rr			<66>ld
13	<55>xor_rr	<56>xor_rr			
14	<70>xor_rr	<71>xor_rr			
15	<27>xor_rr	<42>xor_rr			<73>ld
16	<57>xor_rr	<72>xor_rr			<77>ld
17	<81>add_rr	<82>add_rr			
18	<83>add_rr	<84>add_rr	<85>extr.u	<86>extr.u	
19			<87>extr.u	<88>extr.u	<89>ld

Figure 2. Viewer program

3.5. Viewer program

Figure 2 shows the viewer program. This program enables a graphic display of the evaluation results file. The program displays the total number of the clock cycles necessary for the algorithm, and it also displays a matrix that consists processing clock cycle in the rows and scheduled functions in the columns. If a function name in the matrix is clicked, information is displayed about the function and the functions linked to it.

4. Evaluation results

Table 1 lists the 128-bit key encryption performance for 5 AES candidates. These results consists of evaluation results with our tool

- Gladman's results [9],
- Lipmaa's results [10][15],
- Aoki and Lipmaa's result [14],
- Measured results with Gladman's source code compiled in our environment, and
- Measured results with Gladman's source code modified for UltraSPARC.

Table 1. Evaluation results and measured results of 5 AES candidates (clock cycles)

Evaluation	Platform	Evaluated codes	MARS	RC6	Rijndael	Serpent	Twofish
Evaluation using our tool	Pentium II	Original code	249	205	214	605	261
	UltraSPARC	Original code	641	1,006	232	821	294
	Itanium	Original code	326	303	136	602	196
Measured results (in papers)	Pentium Pro / Pentium II	Gladman's C code[9]	376	270	374	992	378
		Assembly[14][15]	306	223	237	-	292
	UltraSPARC	Ported Gladman's C code[10]	840	1,162	334	996	487
Measured results (our environment)	Pentium II	Gladman's C code	367	263	362	985	371
	UltraSPARC	Ported Gladman's C code	794	1,143	334	1,024	485
		Modified Gladman's C code	754	1,142	290	1,021	369

Note that the evaluation results with our tool include evaluations of only the encryption algorithms part, and they do not include evaluations of the clock cycles for memory access, such as subroutine calls and the argument handovers.

On modifying Gladman's source code for UltraSPARC, we changed the table reference method from direct addressing to pointer addressing, which is suitable for UltraSPARC.

4.1. UltraSPARC

Table 2 lists the instructions, clock latency, and executable processing unit names that are used in the evaluation for UltraSPARC [11]. In the evaluation, we presumed that Ultra SPARC has two integer-processing units (IU) and one load-store unit (LSU).

Table 2. Instructions used in evaluation for UltraSPARC

Execution unit	Instruction	Clock cycle
IU0	SLL, SLLX, SRL, SRLX	1
	UMUL	20
IU0, IU1	ADD, SUB, AND, OR, NOT, XOR	1
LSU	LD	2

The evaluation clock cycles were 80% to 90% of the measured clock cycles. We consider these results are good enough as lower bounds of performance because our tool evaluates neither subroutine calls nor argument processing.

4.2. Pentium II

Table 3 lists the instructions, clock latency, and executable processing unit names that are used in the evaluation for Pentium II [12].

Table 3. Instructions used in the evaluation for the PentiumII

Execution unit	Instruction	Clock cycles
port#0	shl, shr, rol, ror, lea	1
	mul	4
port#0, port#1	add, sub, and, or, not, xor	1
port#2	mov(memory read)	3

For the evaluation, we set the number of pipelines for Pentium II based on [12]. Pentium II has two integer operation units (port#0, #1), one memory read units (port#3), and two memory write units (port #3, #4). However, we evaluated the pipeline number of Pentium II as 3 because we assumed all data is on the registers and not in memory; moreover memory write is not used in our tools.

Compared to Gladman's C code, the evaluation clock cycles is about 60% to 70% of the measured clock cycles. This is because registers of Pentium II are not used efficiently in optimization by any widely used compiler. Therefore, the assumption of our tool about an unrestricted number of registers is not satisfied.

On the other hand, compared to the assembly code of Aoki and Lipmaa, the evaluation clock cycles are about 80% to 90% of the measured clock cycles. Aoki and Lipmaa performed manual tune-up and the registers are used efficiently. Therefore, the assumption of our tool is satisfied.

4.3. Itanium

Using the instruction set of a platform, our tool can be applied for any platform. Thus, we ware able to evaluate Itanium, even though it is not available to the public and has only had its architecture announced.

Table 4. Instructions used in evaluation for Itanium

Execution unit	Instruction	Clock cycles
integer- memory (A)	add, sub, and, or, not, xor, shladd	1
integer- memory (I)	extr, mux, shl, shr, shrp	1
integer- memory (M)	ld	2
	getf, setf	1
floating point (F)	xmpy	5

Table 4 lists the instructions, clock latency, and executable processing unit names that are used in the evaluation for Itanium [13].

For this evaluation, we set the number of pipelines based on [13]. Itanium has four integer-memory units and two floating-point units. Instructions of Itanium consist of 6 types, and 4 types of instructions are used in this evaluation: integer ALU (A), non-ALU integer (I), memory (M), and floating point (F). Type (A), (I), and (M) instructions are executed in integer-memory units, and type (F) instructions are executed in floating-point units.

Itanium has 128 general registers, assuming the unrestricted number of registers is acceptable in this platform.

The evaluation codes for Itanium are ported from those for Pentium II. As mentioned above, the evaluation values with our tool depend on the implementation algorithm. Therefore, a higher performance may be obtained when a suitable algorithm for Itanium is selected.

5. Performance comparison of 5 AES candidates

5.1. MARS, RC6

MARS and RC6 use a lot of multiplication and the rotation shift.

Pentium II has high-speed 32-bit multiplication and rotation shift instructions, so good performance is obtained. UltraSPARC does not have rotation shift instructions, and it takes especially many clock cycles to process multiplication instructions, so performance is worse. Itanium has no rotation shift instruction. Furthermore, multiplication instruction is executed with FPU, and data moves between IU and FPU for multiplication. Therefore performance is worse than that of Pentium II.

5.2. Rijndael

Rijndael can be implemented by repeating the table look-up and key XORing. Therefore, Pentium II and UltraSPARC have a lot of memory references, and memory read instructions are critical processing in both platforms, which have only one pipeline for memory access. Itanium archives the highest performance because it has four memory access units.

5.3. Serpent

Serpent mainly consists of logical operations.

Pentium II and UltraSPARC have two pipelines that can execute logical operations. Therefore, performance becomes almost the same in those platforms. Otherwise, the high number of rounds in Serpent is a cause of the performance decrement.

5.4. Twofish

Twofish consists of different operations, such as table look-up, logical operation, and arithmetic operation. These instructions are processed in a good balance. Therefore, Twofish achieves good performance independent of the characteristics of the platforms.

6. Conclusion

We have developed a new tool for evaluating the performance of symmetric encryption algorithms. With information about both a platform and the instruction set, our tool can evaluate the performance on any processor.

We have compared 5 AES candidates with our tool, and Rijndael and Twofish achieved the highest performance on UltraSPARC, Pentium II and Itanium. Therefore, we recommend Rijndael and Twofish as AES algorithms in respect of their performance.

7. Reference

[1] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M.Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, “MARS - a candidate cipher for AES,” NIST AES Proposal, Jun 1998.

[2] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, “The RC6 Block Cipher,” NIST AES Proposal, Jun 1998.

[3] J. Daemen and V. Rijmen, “AES Proposal: Rijndael,” NIST AES Proposal, Jun 1998.

[4] R. Anderson, E. Biham, and L. Knudsen, “Serpent: A Proposal for the Advanced Encryption Standard,” NIST AES Proposal, Jun 1998.

[5] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, “Twofish: A 128-Bit Block Cipher,” NIST AES Proposal, Jun 1998.

[6] J. Nakajima and M. Matsui, “Fast Software Implementations of MISTY (II),” SCIS'98-9.1.B (in Japanese).

[7] C. Clapp, “Instruction-level Parallelism in AES Candidates,” AES2, 1999.
(http://csrc.nist.gov/encryption/aes/round1/conf2/papers/clapp.pdf)

[8] P. B. Gibonns and S. S. Muchnick, “Efficient instruction scheduling for a pipelined architecture,” Symposium on Compiler Construction, SIGPLAN Notices, Vol. 21, No.7, pp. 11-16, June 1986.

[9] B. Gladman, “Implementation Experience with AES Candidate Algorithms,” AES2 conference, 1999.
(http://csrc.nist.gov/encryption/aes/round1/conf2/papers/gladman.pdf)

[10] H. Lipmaa, “AES Candidates: A Survey of Implementations,” AES2 conference, 1999.
(http://csrc.nist.gov/encryption/aes/round1/conf2/papers/lipmaa.pdf)

[11] D. L. Weaver and T. Germond, “The SPARC Architecture Manual Version 9,” Prentice-Hall, Inc., 1994.

[12] Intel, “Intel Architecture Software Developer's Manual”, Intel, 1999
(http://developer.intel.com/design/PentiumII/manuals/)

[13] Intel, “The IA-64 Architecture Software Developer's Manual,” Intel, January 2000.
(http://developer.intel.com/design/ia-64/manuals/index.htm)

[14] K. Aoki and H. Lipmaa, “Fast Implementations of AES Candidates,” AES3 conference preprint, May 2000.

[15] H. Lipmaa, “AES Ciphers: speed.”
(http://home.cyber.ee/helger/aes/table.html)