Instruction-level Parallelism in AES Candidates

Craig S.K. Clapp

PictureTel Corporation, 100 Minuteman Rd., Andover, MA 01810, USA
email: craigc@pictel.com

Abstract. We explore the instruction-level parallelism present in a number of candidates for the Advanced Encryption Standard (AES) and demonstrate how their speed in software varies as a function of the execution resources available in the target CPU. An analysis of the critical paths through the algorithms is used to establish theoretical upper limits on their performance, while performance on finite machines is characterized on a family of hypothetical RISC / VLIW CPUs having from one through eight concurrent instruction-issue slots. The algorithms studied are Crypton, E2, Mars, RC6, Rijndael, Serpent, and Twofish.

1 Introduction

Several performance comparisons among AES candidate algorithms have already been published, e.g. [6,7,8,10,16]. However, while such studies do indeed render the valuable service of providing a quantitative rather than qualitative comparison between candidates, and in some cases do so for a number of currently popular processors, they are not necessarily very insightful as to how to expect performance of the algorithms to compare on processors other than those listed, nor in particular on future processors that are likely to replace those in common use today.

One of the lessons of DES [11] is that, apart from having too short a key, the original dominant focus on hardware implementation led to a design which over the years has become progressively less able to take full advantage of each new processor generation. In part this accounts for the ease with which AES candidates have been able to achieve much higher levels of security than DES while running as fast as or substantially faster than it on current generation 32-bit processors.

Since the Advanced Encryption Standard is expected to continue in use for several decades after adoption, it is appropriate to consider how it might perform on future processors, especially since it will take far longer for the AES to need replacement due to inadequate key length than was the case with DES.

Among the characteristics expected of future high performance processors, that which we dwell on here is the anticipated increase in the amount of instruction-level parallelism. Instruction-level parallelism in CPUs can arise both from pipelining of execution units and from provision of multiple concurrent execution paths, as are present in superscalar, SIMD, and VLIW architectures (see [4] for more discussion). Mainstream desktop processors have over the last

few generations seen the addition of increasing SIMD parallelism to existing superscalar architectures, in the guise of Intel's MMX and KNI (Katmai New Instructions) extensions to the Pentium, and Motorola's AltiVec extensions to the PowerPC. Furthermore, Intel's eagerly awaited Merced is expected to have much greater instruction-level parallelism than its predecessors, being the first CPU to employ Intel's highly parallel EPIC (Explicitly Parallel Instruction Computing) architecture. This variation on the VLIW theme promises to further raise public awareness of instruction-level parallelism issues.

In this paper we examine the instruction-level parallelism present in a number of the AES candidates in order to understand how their performance might vary as a function of the execution resources available in the target CPU. By performing an analysis of the critical paths through the algorithms we are able to establish theoretical upper limits on their performance. As a separate exercise we investigate performance on finite machines by using a parameterized C compiler to generate scheduled assembly code for a family of hypothetical RISC / VLIW CPUs having from one through eight concurrent instruction-issue slots.

The algorithms studied are Crypton [9], E2 [12], Mars [3], RC6 [14], Rijndael [5], Serpent [1], and Twofish [15].

The rest of the paper is organized as follows:

Section 2 details the characteristics of the CPU family on which our analysis is performed. This section can be skipped by readers who are willing to take on faith that our architectural assumptions are reasonable. Section 3 describes our analysis of the algorithms' critical path lengths, from which their theoretical performance limits can be inferred, and section 4 presents the results of compiling C-code for machines having a range of instruction-level parallelism.

2 The hypothetical RISC / VLIW CPU

In order to perform our analysis we need to adopt a concrete instruction set and CPU architecture. In particular we need to specify the available instructions together with their respective latencies and issue-rates. By making some simplifying assumptions we can avoid needing to unduly worry about other architectural details such as the register set and memory / cache architecture.

For convenience we take as our hypothetical processor a set of characteristics patterned on a real-life VLIW CPU - the Philips TriMedia TM-1100 multi-media processor [13]. The TriMedia processor is a Very Long Instruction Word (VLIW) CPU containing five 32-bit pipelined execution units sharing a common set of 128 32-bit registers. In the real processor all five execution units can perform arithmetic and logical operations, but loads, stores, and shifts are each supported by only two of them. The two execution units that support shifts are distinct from the two that support loads and stores. Given an appropriate instruction mix the processor can issue up to five instructions per clock cycle.

For our analysis we assume the same instruction set, instruction latencies and issue rates, register set, and cache architecture as the real machine, but we let the number of instruction-issue slots range between one and eight (versus five for the real machine), and adjust the complement of instructions that can be

executed in each slot according to what we believe to be reasonable for machines of a given degree of parallelism.

Despite patterning our architecture on a particular CPU, we believe that the characteristics we have adopted are generic enough that our results should be substantially applicable to other processors having RISC-like instruction sets and family members with varying numbers of execution units.

2.1 The instruction set

Our hypothetical machine has a fairly typical RISC instruction set. It has a load / store architecture and instructions that can specify two source registers independent of the destination register. It has 128 internal registers, which is quite large by current RISC standards, but a large register set is likely to be appropriate for a machine with substantial parallelism, and of the few details so far released on the Merced it too is known to have 128 integer registers. In any case, for the AES algorithms we have coded the register usage is quite modest, so that our results would be little changed if the register set was only one half or one quarter of this size.

We keep our machine definition simple by specifying just two functional unit types; a MEM functional unit performs Load and Store instructions while an ALU functional unit performs all other (register to register) operations. A particular CPU definition is completed by assigning a MEM functional unit, or an ALU, or both, to each of the instruction-issue slots of the machine.

Table 1 lists each of the instruction types together with their latency and recovery times. The latency is the number of cycles between the cycle in which an instruction is issued and the cycle in which its result can be used by another instruction (a latency of 1 means that its result can be used in the very next cycle). The recovery time is the number of cycles between when an instruction is issued to a functional unit and the cycle in which that functional unit is again ready to accept such an instruction. The functional units in our machine are fully pipelined meaning that every instruction can be issued on every cycle.

The only instruction we use which might seem a little uncommon is the Byte Extraction instruction (subsequently referred to as BXT). This instruction allows any byte of a 32-bit register to be extracted and returned as the low-order byte of the result, with the upper three bytes of the result being zero-filled. Some other RISC processors also support this capability (e.g. the `rlwinm` and `rlwnm` instructions on the PowerPC). Without this instruction, extraction of either of the two middle bytes of the word would take two operations (a shift and a masking operation) and two cycles. However, even without this specialization, the outer bytes of the word can be accessed with just one instruction on most processors (a masking operation for the low byte and a logical-shift-right for the high byte).

In our hypothetical machine we do not distinguish between its capability to perform rotations to the left and to the right. In the real TriMedia CPU only a rotate-left instruction is actually native to the instruction set. This lack of a rotate-right instruction has no impact on rotations by a *fixed* amount, since the conversion from a fixed rotate-right to the corresponding fixed rotate-left can be done at compile time. However, right-rotations by a *variable* (data dependent)

Instruction type	*Functional unit*	*Recovery time (cycles)*	*Latency (cycles)*
LOAD	MEM	1	3
STORE	MEM	1	1†
ADD, SUB	ALU	1	1
AND, OR, XOR, NOT	ALU	1	1
BYTE EXTRACT	ALU	1	1
SHIFT, ROTATE	ALU	1	1
MULTIPLY (32 x 32)	ALU	1	3
CONSTANT	ALU	1	1
BRANCH	ALU	1	3 delay slots

†. A Store latency of 1 means that a Load from the same address can be issued in the very next cycle, not that the write necessarily completes in one cycle

Table 1. Instruction characteristics

amount translate into two instructions: subtraction of the rotation amount from 32 followed by leftward rotation by the result of the subtraction. Thus, a variable rotate-right uses two instructions and takes two cycles, while a variable rotate-left takes just one of each. We mention this architectural asymmetry because it is not restricted to just the TriMedia chip: the PowerPC instruction set also only natively supports rotations to the left, so similar considerations will apply there. Some other RISC instruction sets have no native support for rotations. On such processors a rotation must be synthesized by two complementary shifts and a merge of the results, which for a *variable* rotation will use four instructions and have a minimum latency of three cycles after the rotation amount becomes known. For these machines left- and right-rotations will have identical performance but variable rotations will be more expensive than fixed ones.

Of the algorithms studied here, only RC6 makes use of *variable* rotations to the *right*, and then only for decryption.

One further restriction we place on our hypothetical machine is that just like in the real TriMedia processor, no more results can be written to registers in any one cycle than the number of instruction issue slots that the machine has. One consequence of this is that the compiler cannot schedule single-cycle instructions for issue in all slots in the cycle immediately preceding the result of an instruction having multiple cycles of latency.

2.2 Instruction-level parallelism in the family members

Our family of CPUs is constructed by letting the number of instruction-issue slots range between one and eight, and assigning a MEM functional unit, or an ALU, or both, to each of the instruction-issue slots of the machine.

In every cycle the machine can issue one (or zero) instructions in each instruction issue slot. The instructions that can be issued in a given slot are restricted to those corresponding to the attached functional units.

Our simplest machine has just one issues slot (a scalar machine) which performs all instruction types. The next larger machine (which we refer to as configuration ***1+1***) can perform a memory operation concurrently with one ALU operation. Our two-slot machine allows any combination of two instructions per cycle: two ALU operations, one ALU operation and one memory reference, or two memory references. Configuration ***2+1*** allows up to three instructions per cycle so long as not more than two of them are ALU operations, and not more than two of them are memory references. For the larger machines we continue adding ALU functionality in each additional instruction issue slot, while holding the memory accesses to a limit of two per cycle. We place this limit on memory accesses because adding more read or write ports to cache memory is generally a much more expensive proposition than adding computational units. Today it is quite common for CPUs to have two concurrent ports into cache memory and we have yet to see evidence of this increasing.

Our set of CPU configurations is summarized below in table 2.

Instruction issue slots	***Disposition of functional units***							
	Slot 1	***Slot 2***	***Slot 3***	***Slot 4***	***Slot 5***	***Slot 6***	***Slot 7***	***Slot 8***
1	ALU, MEM							
1+1	MEM	ALU						
2	ALU, MEM	ALU, MEM						
2+1	ALU, MEM	MEM	ALU					
3	ALU, MEM	ALU, MEM	ALU					
4	ALU, MEM	ALU, MEM	ALU	ALU				
5	ALU, MEM	ALU, MEM	ALU	ALU	ALU			
6	ALU, MEM	ALU, MEM	ALU	ALU	ALU	ALU		
7	ALU, MEM	ALU, MEM	ALU	ALU	ALU	ALU	ALU	
8	ALU, MEM	ALU, MEM	ALU	ALU	ALU	ALU	ALU	ALU

Key: ALU = Arithmetic / Logic Unit, MEM = Load / Store Unit

Table 2. CPU configurations

2.3 Compiling C-code for the hypothetical machines

Of particular interest for our purposes is that the TriMedia code generation tools include an efficient *parametrized* C compiler in which the machine definition - the number of registers, the number of parallel instruction units and their capabilities, instruction latencies, etc. - can all be specified to the compiler through a configuration file. By varying the machine definition from that of the real-life device we can explore the performance of candidate algorithms on hypothetical machines having varying CPU resources, while hopefully eliminating the compiler as a variable in our comparisons.†

---

†. This feature is not officially supported, but is a vestige of the compiler's heritage as a research vehicle for exploring VLIW architectures. We take care to modify the machine definition within bounds that empirically have been found not to break the tool chain.

For most processors, an assembly listing tells relatively little about the run-time performance of the code. This is because processor pipeline stalls arising from instruction dependencies, memory latencies, cache-bank conflicts, or other effects are not directly apparent from the assembly listing. For our VLIW processor it is the nature of the compiler that instruction dependencies and cache memory latency issues are resolved *at compile time*. Thus the assembly listing becomes a cycle-by-cycle representation of how the CPU will perform except for stall conditions which cannot be known until run-time. These include instruction-cache misses, data-cache misses, cache-bank conflicts, and interrupts. For the purposes of our analysis we ignore interrupts. We can also ignore instruction-cache misses if we are only interested in the asymptotic performance of the algorithms (i.e. their performance on large blocks of data). For the algorithms considered, the fact that all sub-key arrays and look-up-tables are static (for a given key) means that we can always avoid cache-bank conflicts by replicating any necessary sub-keys or look-up-tables in multiple cache banks (outside of the inner processing loop) and by then arranging for simultaneous accesses to refer to distinct copies. The only remaining uncertainty is the impact of data cache misses. However, for all of the algorithms studied the only memory references not yet accounted for are those that access the input data buffer and write back to the output data buffer (provided that our coding style allows all intermediate results to remain in registers, which it does in practice). As a practical matter we can reasonably expect the occasional cache miss arising from these accesses to contribute very few additional cycles per block on average, so that real-life performance would not degrade by more than a few percent from that implied by the assembly code's static schedule.

From the foregoing discussion it can now be seen why some architectural details such as the register set and memory / cache architecture are, within bounds, largely irrelevant to our analysis, as was asserted earlier.

3 Critical-path analysis of the AES candidates

To determine the upper limit of performance of each candidate on our processor architecture we attempt to identify the software *critical path* through the algorithm. This is the path through the algorithm, from plaintext to ciphertext, that has the largest *weighted* instruction count, the weighting being the number of cycles of latency associated with each instruction in the path. This sets the theoretical upper limit on performance (within the framework of the specified instruction set) since *no amount of added parallelism can let us evaluate this path any faster*.

In some cases an algorithm can be evaluated in several different ways, and there might be a trade-off between arranging its computational flowgraph to minimize the number of instructions used or to minimize the critical path length. We note this to be the case for E2, and expect the same of Serpent. For all the other algorithms studied we have not found a case where the flowgraph which minimizes the critical path length is not one which also minimizes the number of instructions used.

For this portion of our evaluation we comment that neither the C compiler nor a particular piece of C-code plays any part in this analysis - we are simply analyzing the computational flowgraphs of the algorithms.

The results of our paper analysis of the algorithms are summarized in table 3. For more detail behind these results refer to Appendix A.

Algorithm	*Number of rounds*	*Number of instructions per 128-bit block*	*Cycles in critical path (per 128-bit block)*	*Effective parallelism*	*Loads per critical-path cycle*
Crypton	12	632	85	7.44x	2.9
E2 *NIST 32-bit*	12	844	210	4.02x	1.4
8-bit w/32-bit sub-keys		1096	150	7.31x	1.7
Mars	16	504	214	2.36x	0.6
RC6 *Encrypt*	20	328	181	1.81x	0.24
Decrypt		328	161	2.04x	0.27
Rijndael	10	528	71	7.44x	2.9
Serpent *Encrypt*	32	1336	≤526	≥2.54x	0.25
Decrypt		1332	≤436	≥3.06x	0.3
Twofish	16	528	162	3.26x	1.0

Table 3. Instruction counts, critical path lengths, and effective parallelism

For the critical-path analysis reported here we only count the instructions and cycles associated with the transformation of a plaintext block into a ciphertext block (ECB mode) with both the plaintext and ciphertext assumed to reside in registers. Unlike the real code whose performance is discussed later, here we ignore instructions associated with loading the plaintext, storing the ciphertext, and maintaining a software loop.

For all cases our instruction counts assume that sub-keys are re-loaded for every encrypted block. For some of the algorithms the expanded key schedule is sufficiently small that our generous register set (128 registers) would allow the sub-keys to be loaded just once in an outer loop and then let multiple blocks be encrypted without having to re-load them. Since sub-key loading is not a critical-path operation, the inclusion of these instructions in our count does not make the assessed speed any slower, rather, it has the effect of inflating the apparent parallelism of the algorithms.[‡]

Judged by their lengths of their critical paths, Rijndael and Crypton stand well ahead of the pack. E2, Mars, RC6, and Twofish form the second tier. The remaining algorithm, Serpent, trails the next nearest candidate by somewhat more than a factor of two.

The effective parallelism listed in the table is the ratio of the total number of instructions per block to the number of cycles in the critical path. This simple metric provides a useful indicator of how many concurrent execution units could be put to good use by each of the algorithms. In general we can expect an

‡. Note that the situation is different for real code running on finite machines - see later discussion.

algorithm to show little increase in performance once the number of concurrent execution units exceeds the effective parallelism. Of course, such an analysis is inexact since it does not account for the computational flowgraph being perhaps 'wide' in some places and 'narrow' in others. Nor does it account for specific resource constraints such as exhaustion of load / store resources. On the other hand, our later analysis using real code does substantially validate the value of this metric.

We comment that a large value for the effective parallelism is not per-se an indicator of 'goodness'. A case in point is the second implementation of E2 characterized in the table. This version achieves its shortened critical path by resorting to mostly byte operations, which has the effect of inflating the total number of operations needed to perform the round function. Since several byte operations can occur in parallel it gives this implementation a high effective parallelism, but in reality the algorithm would be making poor use of the resources of a highly parallel 32-bit processor if all it was doing was 8-bit operations.

Only Crypton, E2, and Rijndael show substantial promise of taking advantage of more than three or four concurrent execution units. Among the other candidates there is not enough variation in instruction-level parallelism for this characteristic to be a significant discriminating factor between them.

Of the two algorithms showing differences between their encrypt and decrypt critical paths (RC6 and Serpent), curiously both perform faster in the *decrypt* direction.

For RC6 this difference is accounted for by the sub-key addition in each round being in the critical path of the encryption flowgraph, but not in the critical path of the decryption flowgraph.

For Serpent this difference is accounted for in part by the inverse S-boxes having (on average) a shorter critical path than the S-boxes used for encryption (but see below). The remaining difference is due to the linear transformation used in each round being two cycles shorter in the reverse direction than in the forward direction.

The true critical path of Serpent is rather difficult to establish. This is partly because the S-boxes have many possible boolean implementations, and in general there may not be an implementation which has both the lowest instruction count and the shortest critical path. Another reason is that in general each of the four outputs of the S-box has a different delay from each of the four inputs such that when S-boxes are cascaded the critical path through the cascade is not necessarily the sum of the longest paths through the individual S-boxes. The possibility of multiple implementations for the linear mixing layer only compounds this complexity. For the paper analysis reported here we have, for the S-boxes, simply summed the critical paths of each of the S-boxes as implemented in the optimized C-code for Serpent provided in the first-round AES submissions. That this gives a pessimistic figure for Serpent's true critical path is evidenced by the compiled C-code results, reported in the next section, in which Serpent's performance *in practice* is found to exceed what our naïve analysis would predict is theoretically possible. Beyond this, there is reason to expect that alternative

implementations of the S-boxes could further shorten Serpent's critical path given that the current S-box implementations only show a parallelism factor of under 2.5x. Since in principle all four S-box output terms can be evaluated concurrently (which would give at least a 4x parallelism factor) we suspect that they have been tuned for the fewest instructions rather than shortest critical path.

E2 also offers several instruction-count / critical-path trade-offs for its round function. We have characterized two flowgraphs. The first is based on the optimized C-code implementation provided to NIST in the first-round AES submissions, and uses 63 32-bit instructions to perform the round function with a 16 cycle critical path. In the tables this is referred to as *NIST 32-bit*. The second case assumes that all operations are done byte-wise, except for loading the round sub-keys as 32-bit values (to keep the number of loads down). In this version we assume that sub-key XORing is moved off of the critical path but we allow 4 XOR gates delay for the *P*-function. This flowgraph takes 84 (mostly 8-bit) operations to achieve an 11 cycle critical path per round and is referred to in the tables as *8-bit w/32-bit sub-keys*. At a cost of at most an additional 24 XOR operations per round the critical path could be reduced by one additional cycle to its *theoretical minimum of 10 cycles per round*. We have not analyzed this case in detail.

For Crypton and Rijndael the picture is not quite as rosy as their critical path lengths would suggest. Their most efficient implementations have a somewhat higher ratio of table look-ups to other operations than do the other algorithms. This is represented by the column in table 3 which lists *loads per critical-path cycle*. This is simply the ratio of the number of memory references per encrypted block to the number of cycles in the critical path. For Crypton and Rijndael to run at the speed that their critical path lengths predict, they would need to perform on average almost three memory accesses per cycle.

Our hypothetical family of machines allows at most two memory references per cycle, resulting in both algorithms being memory constrained. When load / store constraints are taken into account their critical paths lengthen: Crypton uses 192 table-look-ups per block while Rijndael (for a 128-bit key) uses 160, and this is without re-loading sub-keys for each block. This extends their critical paths to at least 96 cycles per block and 80 cycles per block respectively. Thankfully, their sub-key arrays are not so large as to preclude the possibility of their being kept wholly in registers (at least when we have a total of 128 registers available). If we remove the sub-key loads from the instruction count, and take the memory-constrained cycle count as the critical path then Crypton and Rijndael show an effective parallelism of 6.04x and 6.05x respectively, which is still very high compared to the other candidates.

4 AES candidate performance on the family of finite CPUs

While the previous section concerned itself with assessing the theoretical limitations of the candidates given unlimited instruction-level parallelism, in this section we take real C-code implementations and compile them for our family of CPUs.

For each of the algorithms benchmarked we started with the optimized C-code implementations provided to NIST in the first-round AES submissions. Where it

Algorithm	Instruction issue slots									
	1	**1+1**	**2**	**2+1**	**3**	**4**	**5**	**6**	**7**	**8**
Crypton†	744	463	388	276	258	194	165	155	151	149
E2	1052	626	523	373	352	282	276	275	274	274
Mars	552	378	285	232	228	227	227	227	227	227
RC6	389	311	214	188	186	186	186	186	186	186
Rijndael	628	388	327	231	217	166	137	131	129	127
Serpent	1356	1216	714	624	541	472	468	468	468	468
Twofish	626	384	318	225	222	187	185	185	185	185

†. The results reported for Crypton are actually taken from 12-round Rijndael, as discussed in the text

Table 4. Encryption speed in cycles per block for each CPU configuration

had not already been done we unrolled the encryption routine until processing of an entire 128-bit block was expressed as straight-line code. Where appropriate we then expressed rotations and byte extractions using macros that efficiently invoke our CPU's native instructions for these operations. Our last transformation was to eliminate, as far as possible, variables accessed as array references, replacing them instead with simple variables that the compiler could assign to registers (an exception to this is that we have uniformly maintained sub-key accesses as array references in each implementation).

An exception to this strategy is Crypton, for which we have instead substituted figures from our optimized version of Rijndael in its 192-bit key mode. The validity of this substitution is that an optimized implementation of Crypton's round function differs from that of Rijndael only in the contents of the look-up-tables, and the contents of these have no impact on performance. Rijndael uses 12 rounds in its 192-bit key mode, which is the same number of rounds specified for Crypton.

We compiled the resulting C-code for each of our machine definitions and inspected the scheduled assembly code produced by the compiler. The results reported here are the number of cycles in the *static* instruction schedule. As explained in section 2.3, we can expect this to be closely representative of the *dynamic* performance that a machine of the specified architecture would achieve.

The number of cycles taken by each algorithm to encrypt one block on each variant of the CPU is shown in table 4, while the corresponding throughput (in bits per cycle) is graphed in figure 1.

Apart from Crypton and Rijndael, and to a lesser extent E2, the performance achieved on the largest machine (8 slots) is quite close to the theoretical performance limits predicted by our critical-path analysis. This is a good indicator that our code is being efficiently compiled, lending credence to the proposition that the performance trends shown in figure 1 are inherent characteristics of the algorithms rather than some quirk of the compiler or C-code implementation. That Serpent actually does *better* than our critical-path analysis predicts is due to our critical-path analysis for Serpent being only an approximation, as discussed in the previous section, while for the other algorithms it is exact. In the case of

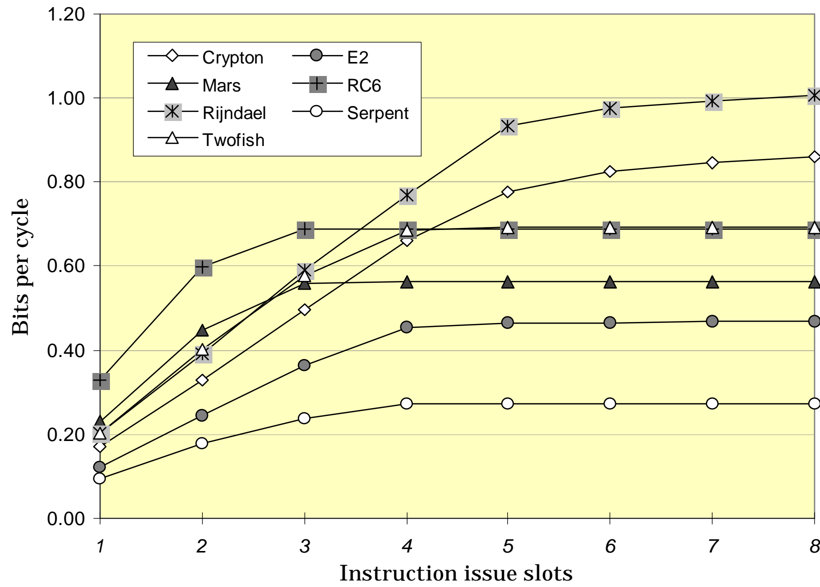

Fig. 1. Encryption speed versus instruction-level parallelism

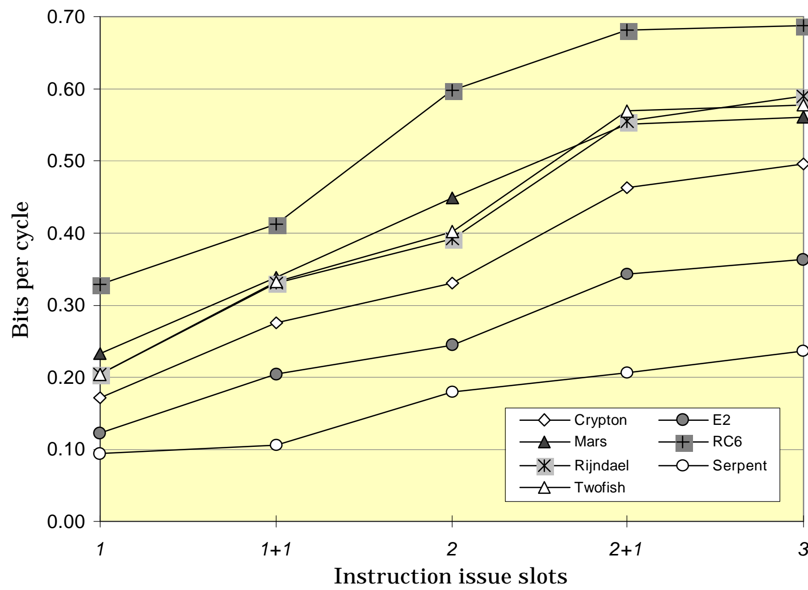

Fig. 2. Encryption speed versus instruction-level parallelism (detail)

Crypton and Rijndael the differences from the theoretical predictions are due to them becoming memory constrained (which our critical-path analysis does not consider). Their performance, more than the other algorithms, would likely benefit by not re-loading the sub-keys for every encrypted block. So far we have not applied this optimization to any of the candidate implementations.

E2 has the third highest ratio of memory accesses to critical-path cycles. Even though it is less than two loads per cycle it is still high enough to start becoming a limiting factor. The remaining algorithms, all with no more than about one memory access per critical-path cycle on average, do not appear to suffer a memory bottleneck.

Figure 2 shows that the *relative* performance of the algorithms remains substantially constant for up to three instruction issue slots. Thereafter, as seen in figure 1, Crypton and Rijndael whose performance had so far been in the middle of the pack, take the lead while the others flatten out due to their inability to take advantage of the increasing CPU resources.

RC6, having the least effective parallelism according to our critical-path analysis is, not surprisingly, the first to flatten out.

Twofish continues to make performance gains up to the addition of the fourth instruction-issue slot, at which point it essentially reaches performance parity with RC6. This result is not so surprising when one notes that Twofish has a slightly shorter critical path than RC6 (at least for *en*cryption). However, we draw a distinction from previous reports of Twofish matching or out-performing RC6 on 32-bit processors which have relied on RC6 being the victim of weaknesses in the CPU's implementation of rotations or integer multiplication. RC6 suffers no such penalty here since our machine architecture has a single-cycle rotate and three-cycle multiply.

For E2 we comment that only the *NIST 32-bit* version is represented in these results. The *8-bit w/32-bit sub-keys* version would likely show superior performance in the region of five to eight instruction issue slots, potentially even reaching parity with Twofish and RC6 by eight slots, but we have not coded this version.

While Serpent is the slowest among the surveyed candidates, we point out that there is expected to be some opportunity for improvement by modifying the implementation of its S-boxes. However, we don't expect that such changes will afford sufficient improvement to affect the relative ranking of the algorithms.

4.1 Normalized security comparisons

Biham [2] has proposed that a fairer basis for comparison among candidates than simply measuring their speed as specified, is to compare their speed when all are adjusted to give the same 'nominal' level of security. While such a normalization process is somewhat subjective, it is nonetheless clear that some algorithms have indeed been specified with a much greater degree of conservatism than others. While Biham's normalization factors should be acknowledged as those of an interested party, we accept them here as good-faith estimates and use them to adjust our results. Table 5 lists the adjusted number of rounds proposed by Biham for comparison purposes.

Algorithm	*'Official' number of rounds (128-bit key)*	*Number of rounds for 'normalized' security*†
Crypton	12	11
E2	12	10
Mars	16+16	12+8
RC6	20	20
Rijndael	10	8
Serpent	32	17
Twofish	16	12

†. According to Biham, see [2]

Table 5. Number of rounds used for *normalized security* comparisons

The relative performance of the algorithms with correction having been made for nominally equivalent security is shown in figures 3 and 4. For these results we have not re-run the algorithms with an altered number of rounds implemented, but have simply scaled the previous figures by the appropriate correction factor.

The primary effect of these adjustments is to promote Serpent's ranking to be essentially at parity with E2, and to demote RC6 to the middle of the pack.

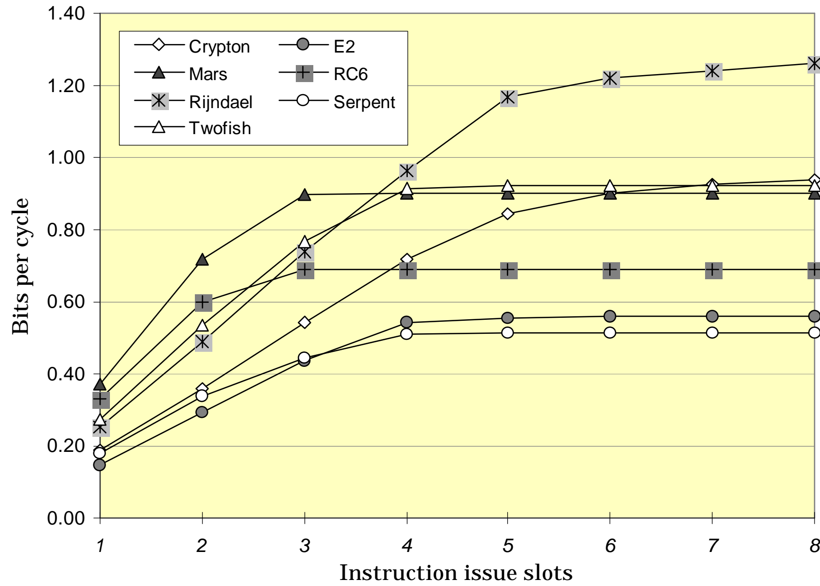

Fig. 3. Encryption speed for *normalized security* vs. instruction-level parallelism

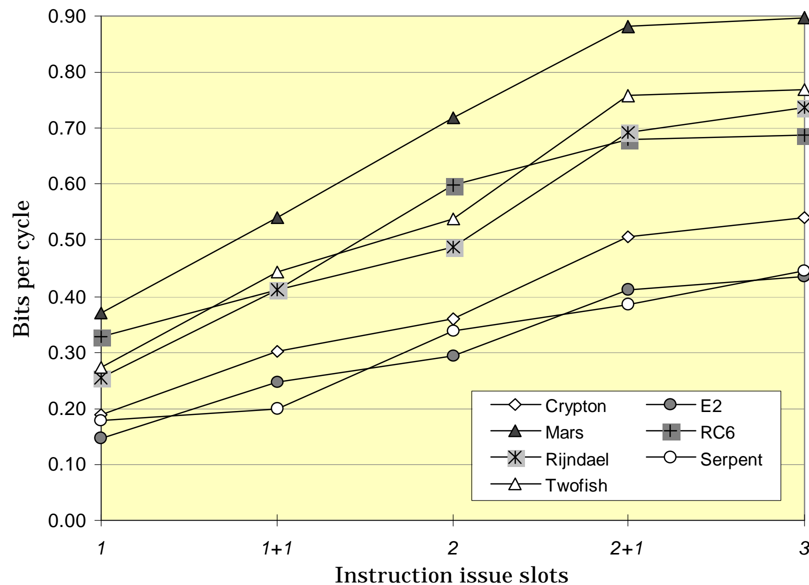

Fig. 4. Detail of Fig. 3.

5 Conclusion

We have examined some of the leading AES candidates for their ability to exploit the substantial instruction-level parallelism anticipated in future high performance processors. We performed a critical-path analysis to establish the theoretical performance limits of the algorithms and a practical experiment of compiling real C-code implementations of them for a family of hypothetical VLIW CPUs. The experimental results were substantially in agreement with the theoretical predictions.

With three instruction issue slots RC6 exhibits a distinct performance lead, with Rijndael, Twofish, and Mars tying for second place.

With four instruction issue slots, Twofish, RC6, Rijndael, and Crypton showed remarkably similar performance, all somewhat ahead of second placed Mars.

Of the candidates studied, Crypton and Rijndael show the greatest potential for benefit from more than four instruction issue slots. Since their speed on machines with quite modest resources is already very competitive, it can be expected that the future will only make them even more appealing from the performance perspective.

References

1. R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," NIST AES Proposal, Jun 1998.
2. E. Biham, "Design Tradeoffs of the AES Candidates," ASIACRYPT'98, October 20, 1998, http://www.cs.technion.ac.il/~biham/Reports/Slides/asiacrypt98-slides.ps.gz
3. 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.
4. C.S.K. Clapp, "Optimizing a Fast Stream Cipher for VLIW, SIMD, and Superscalar Processors," *Fast Software Encryption (Ed. E. Biham), LNCS 1267*, Springer-Verlag, 1997, pp. 273-287
5. J. Daemen and V. Rijmen, "AES Proposal: Rijndael," NIST AES Proposal, Jun 1998.
6. B. Gladman, "AES Algorithm Efficiency," http://www.seven77.demon.co.uk/aes.htm
7. L. Granboulan, "AES : analysis of the `RefCode`, `OptCCode` and `AddCode` submissions," http://www.dmi.ens.fr/~granboul/recherche/AES.html
8. F. Koeune, "Some figures about candidates performances," http://www.dice.ucl.ac.be/crypto/CAESAR/performances.html
9. C.H. Lim, "CRYPTON: A New 128-bit Block Cipher," NIST AES Proposal, Jun 1998.
10. H. Lipmaa, "AES Ciphers: speed," http://home.cyber.ee/helger/aes/table.html
11. NBS FIPS PUB 46-1, "Data Encryption Standard," National Bureau of Standards, U.S. Department of Commerce, Jan 1988
12. Nippon Telegraph and Telephone, "Specification of E2 - a 128-bit Block Cipher," NIST AES Proposal, Jun 1998.
13. Philips Semiconductors, http://www.trimedia.philips.com
14. R.L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin, "The RC6 Block Cipher," NIST AES Proposal, Jun 1998.
15. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Twofish: A 128-Bit Block Cipher," NIST AES Proposal, Jun 1998.
16. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson, "Performance Comparison of the AES Submissions," http://www.counterpane.com/aes-performance.pdf

Appendix A. Critical-path analysis details

Table 6 details the critical-path analysis the results of which are presented in summary form in section 3. It should be understood that this analysis is specific to the instruction set and corresponding latencies outlined in the text. In particular, the latency for Load and Multiply instructions may differ on other processors (for instance, the Pentium II has a four cycle latency for a 32 x 32-bit integer multiplication). Also, the lack of native support for rotations or byte extraction in the instruction set would cause the analysis to need adjustment.

Algorithm	*Number of rounds per block*	*Instructions per round*	*Number of instructions per block*	*Instructions in critical path (per round)*†	*Cycles in critical path (per round)*	*Cycles in critical path (per block)*	*Effective parallelism*
Crypton							
Main rounds	12	XOR(16), BXT(16), Load(20)	624	XOR(3), BXT(1), Load(1)	7	84	7.43x
Initial key addition	1	XOR(4), Load(4)	8	XOR(1)	1	1	8x
Total			**632**			**85**	**7.44x**
E2‡							
Main rounds (NIST 32-bit)	12	XOR(23), BXT(17), Load(23)	756	XOR(8), BXT(2), Load(2)	16	192	3.94x
Main rounds (8-bit w/32-bit sub-keys)	12	XOR(48), BXT(16), Load(20)	1008	XOR(5), Load(2)	11	132	7.64x
Outer rounds (both implementations)	2	XOR(4), AND(16), OR(12), MUL(4), Load(8)	88	XOR(1), AND(1), OR(2), MUL(1)	9	18	4.89x
Total (NIST 32-bit)			**844**			**210**	**4.02x**
Total (8-bit with 32-bit sub-keys)			**1096**			**150**	**7.31x**
Mars							
Forward keyed transformation	8	ADD(3), XOR(3), AND(1), ROT(6), MUL(1), Load(3)	136	ADD(2), XOR(2), AND(1), ROT(1), Load(1)	9	72	1.89x
Backward keyed transformation	8	ADD(3), XOR(3), AND(1), ROT(6), MUL(1), Load(3)	136	XOR(1), ROT(2), MUL(1)	6	48	2.83x
Forwards mixing	2	ADD(10), XOR(8), ROT(4), BXT(16), Load(16)	108	ADD(4), XOR(4), BXT(4), Load(4)	24	48	2.25x
Backwards mixing	2	SUB(10), XOR(8), ROT(4), BXT(16), Load(16)	108	SUB(2), XOR(4), BXT(4), Load(4)	22	44	2.45x
Key addition layers	2	ADD(4), Load(4)	16	ADD(1)	1	2	8x
Total			**504**			**214**	**2.36x**

Table 6. Instruction counts, critical path lengths, and effective parallelism

Algorithm	*Number of rounds per block*	*Instructions per round*	*Number of instructions per block*	*Instructions in critical path (per round)*†	*Cycles in critical path (per round)*	*Cycles in critical path (per block)*	*Effective parallelism*
RC6							
Main rounds (encrypt)	20	ADD(6), XOR(2), ROT(4), MUL(2), Load(2)	320	ADD(3), XOR(1), ROT(2), MUL(1)	9	180	1.78x
Main rounds (decrypt)	20	ADD(6), XOR(2), ROT(4), MUL(2), Load(2)	320	ADD(2), XOR(1), ROT(2), MUL(1)	8	160	2.00x
Key addition layers	2	ADD(2), Load(2)	8	ADD(0.5)	0.5	1	8x
Total (encrypt)			**328**			**181**	**1.81x**
Total (decrypt)			**328**			**161**	**2.04x**
Rijndael							
Main rounds	10	XOR(16), BXT(16), Load(20)	520	XOR(3), BXT(1), Load(1)	7	70	7.43x
Final key addition	1	XOR(4), Load(4)	8	XOR(1)	1	1	8x
Total			**528**			**71**	**7.44x**
Serpent							
S-box layers (encrypt)	32	AND/OR/XOR/NOT (18 average)	576	AND/OR/XOR/NOT (8.625 average)	8.625	≤276	≥2.09x
Inverse S-Box layers (decrypt)	32	AND/OR/XOR/NOT (17.875 average)	572	AND/OR/XOR/NOT (7.75 average)	7.75	≤248	≥2.31x
Linear transformations (encrypt)	31	XOR(8), Shift(2) ROT(6)	496	XOR(3), Shift(1) ROT(3)	7	217	2.29x
Inverse linear transformations (decrypt)	31	XOR(8), Shift(2) ROT(6)	496	XOR(3), Shift(1) ROT(1)	5	155	3.20x
Key addition layers	33	XOR(4), Load(4)	264	XOR(1)	1	33	8x
Total (encrypt)			**1336**			**≤526**	**≥2.54x**
Total (decrypt)			**1332**			**≤436**	**≥3.06x**
Twofish							
Main rounds	16	ADD(4), XOR(8), ROT(2), BXT(8), Load(10)	512	ADD(2), XOR(3), ROT(1), BXT(1), Load(1)	10	160	3.20x
Key addition layers	2	XOR(4), Load(4)	16	XOR(1)	1	2	8x
Total			**528**			**162**	3.26x

†. In cases where multiple paths have the same critical length we only list the instructions in one of the paths

‡. Two different implementations of E2 are characterized, one needing few instructions *(NIST 32-bit)*, the other having a shortened critical path *(8-bit w/32-bit sub-keys)*.

Table 6. Instruction counts, critical path lengths, and effective parallelism