Ensuring Fast Implementations of Symmetric Ciphers on the Intel Pentium 4 and Beyond [⋆]

Matt Henricksen and Ed Dawson

Information Security Institute, Queensland University of Technology,
GPO Box 2434, Brisbane, Queensland, 4001, Australia.
{m.henricksen, e.dawson}@qut.edu.au

Abstract. Cipher design is a multi-faceted process. Many designers focus on security, or present novel designs, but neglect to consider the impact on their ciphers' efficiency. This paper presents simple guidelines for ensuring efficient symmetric cipher implementations on the Intel Pentium 4 and associated architectures. The paper examines the suitability of a handful of ECRYPT eSTREAM ciphers for the platform, including Dragon, HC-256, MAG, Mir-1, Phelix, and Py.

Key words: stream cipher, implementation, Intel Pentium 4, Dragon, HC-256, MAG, Mir-1, Py, RC4

1 Introduction

The days of slow symmetric ciphers are over. It is widely considered that efficiency of ciphers is nearly as important as their security. The benchmark set in software by the Advanced Encryption Standard (AES) [4] is around fifteen cycles per byte on the Intel Pentium 4 [13]. Symmetric ciphers that do not equal or surpass this benchmark are widely regarded as non-competitive [1]. Word-based stream ciphers are, in many cases ([1], [3], [18]), notably faster than block ciphers, while not demonstrably less secure.

In 2005, the ECRYPT NoE eSTREAM stream cipher project released a call [5] for stream cipher primitives. Of the thirty-five candidates, twenty-two were profiled as suitable for software implementation, with efficiencies ranging from three cycles [1] through to a sluggish 1,400 cycles per byte [11]. The diversity of results reveals that many cipher designers do not understand either the need for efficiency, or how to achieve it in their ciphers.

This paper discusses the design of ciphers with respect to the Intel Pentium 4. As this architecture is backwards compatible with the entire Intel x86 line, most of the advice holds for the other members of the Pentium family, such as

[⋆] This work was partially funded by Australian Research Council Discovery Project Grant DP0450920.

[1] For example, in the second round of the ECRYPT eSTREAM project, stream ciphers that did not compete with the block cipher AES were "archived" [6].

the Pentium III. It also holds for the Pentium-D, which contains two Pentium 4 Prescott chips in a single package. Much of the advice is, at a high level, similar to that found in the paper of Schneier and Whiting [15], published nearly a decade earlier. This is a natural consequence of Intel's strategy of backwards compatibility in the Pentium line of processors.

The guidelines within this paper are prioritized as *High*, *Medium* or *Low* according to their impact. They are presented at an algorithmic rather than implementation level. It is easy to obtain an efficient implementation of a cipher that has been well designed. Conversely, it may prove to be impossible to derive a fast implementation if attention is not paid to efficiency during the cipher's design phase. The designer of a cipher and its optimizer are frequently different people. This paper is presented to help the designer to understand how to facilitate the job of the optimizer. For a good discussion about efficient implementation of ciphers on recent architectures, see [13].

Section 2 describes how to commence the task of cipher design with implementation considerations in mind. Sections 3 and 4 respectively describe the impact that the number of cipher variables and state size have on the cipher's speed. Section 5 shows how to choose cipher operations for maximum speed. Section 6 describes how special features of the Intel Pentium 4 may be useful in implementing ciphers. Section 7 contains a discussion and summary of results.

2 Grounding Cipher Design in Reality

Many cipher designers hold the misconception that they can derive an accurate estimate of their cipher's efficiency by counting the number of its operations. This belief is reinforced by the prediction of Schneier and Whiting [15] in 1997 that all personal computing processors were converging rapidly to a RISC (Reduced Instruction Set Computing) architecture. History has shown this trend to be mythical. The Pentium 4 architecture is very complex, and this way of obtaining a benchmark on the cipher is inaccurate.

Consider the MAG cipher specification [16], which computes the relative efficiency of that cipher to RC4 by counting the number of operations in their update functions. It is concluded there, without the support of empirical evidence, that 32-bit MAG is four times faster than 8-bit RC4, due to the number and nature of their respective operations. The results given in Section 7 show that a basic implementation of MAG is more than four times slower than a basic implementation of RC4. Similar counting strategies are employed for HERMES [12], the estimate of which is at least qualified by specifying an 8-bit RISC architecture, and for MIR-1 [14], which also mistakenly assumes a lower bound of one operation per clock. Counting operations is not an effective method for calculating cipher efficiency.

Guideline 1 **(High)** Run practical tests during the design process to gauge the efficiency of the cipher.

A corollary to this is “don’t make up concrete timing information based upon your paper design”. The reality of a cipher’s efficiency is connected not just with the contained operations, but also with how well its design is matched to the target architecture. This means that the performance of a cipher will vary even between variants of a processor (for example, the Willamette, Northwood and Prescott variants of the Intel Pentium 4).

For example, MUGI [17] looks simple on paper: it uses an Linear Feedback Shift Register (LFSR), s-boxes, byte swapping, and simple arithmetic operations. However, it runs slowly on the Pentium 4, at around 25 cycles/byte [8], being a 64-bit cipher implemented on a 32-bit platform. The simple operation counting outlined above does not determine the impact of this design-architecture mismatch. The story is similar for 64-bit MIR-1 [14]. The second guideline not only leads to a better approximation of the performance of the cipher, but also to a more efficient cipher.

Guideline 2 **(High)** Understand (in general terms) the architectures on which the cipher is likely to be implemented.

By paying attention to the Intel register set, how data is transferred between the registers and memory, and how it is processed by the CPU, the cipher designer can learn how to make a more efficient cipher. The designer can also manipulate architecture-specific features to this end.

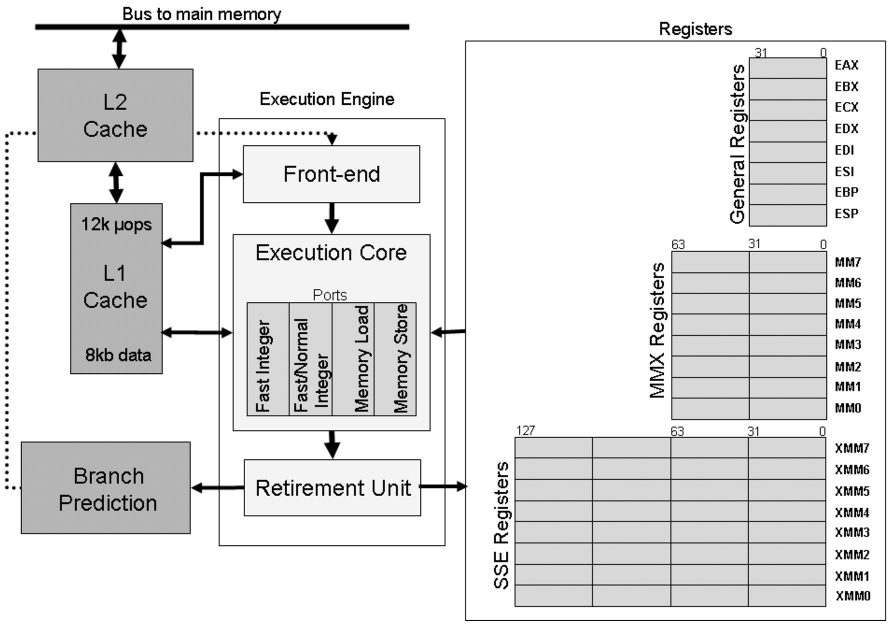

Fig. 1. Intel Pentium 4 Architecture

3 The Pentium 4's Register Set

Registers are very fast memory locations on the CPU die that operate at clock speed, rather than the slower speeds of general-purpose memory. Ideally, each variable in the cipher algorithm can be mapped directly to a register, rather than to general-purpose memory. The Intel Pentium 4 is register-poor in that the programmer has direct access to only eight 32-bit general purpose registers, shown at the right of Figure 1.

Not all registers are treated equally: for example, the multiplication instruction MUL works exclusively upon the EAX and EDX registers; common string operations operate on ESI and EDI; and stack-based operations relating to procedure calls and local variables use EBP and ESP. This reduces the availability of these registers without precluding their use. The instructions that operate on these eight registers, and the rules for dealing with them are found in [9].

Most members of the the Pentium family sport additional registers that can also be used by the programmer. The Multimedia Extensions (MMX) duplex a series of eight 64-bit registers on the back of the Floating Processor Unit (FPU). Members of the family from Intel Pentium III onwards contain eight independent 128-bit (XMM) registers as part of the Streaming Extensions (SSE). The MMX and XMM registers are controlled by their own sets of specialized (and in most regards, limited) instructions. So there are three sets of registers, and three sets of mostly incompatible instructions.

Register pressure occurs when, at a given time, there are more cipher variables to deal with, than there are registers in which to store them. In this case, some of the registers need to be stored in slower memory. This is likely to have a negative impact on the performance of the cipher.

Guideline 3 **(Medium)** For optimal efficiency, the number of variables in any locality of the cipher algorithm should not exceed the number of usable registers.

A good example of adherence to this guideline can be seen in the stream cipher Phelix [18]. Its designers chose to use a 160-bit internal state in their cipher, because this directly maps to five 32-bit registers, meaning that the state does not need to be stored in slower memory. In Mir-1, register pressure is generated through the use of the four sixty-four bit variables in its *loop_update_state* function. These variables translate into eight thirty-two bit variables x_i^t, $0 \leq i < 8$. Because each updated variable x_i^{t+1} depends upon all x_i^t, eight additional registers are required. Ignoring the intermediates, this results in a demand for ten registers at any time. This problem is insoluble using just the general purpose registers.

4 Transferring Data

The Pentium 4's registers are accessible at clock speed (between 1.3 GHz and 3.8 GHz). The memory from which the registers acquire their data is up to three times slower, and there is a latency penalty of as much 75 cycles, depending upon the chip variant, the front side bus, and how the data transfer is managed. The Pentium 4 contains mechanisms to reduce memory bottlenecks, one of which is a series of high-speed *caches* positioned between the CPU and the main memory. Data is summoned, on demand, to the caches from the main memory. When the data is required by the CPU, but not available within the registers or the caches, then a efficiency penalty (hidden to the cipher designer) is incurred.

The Pentium 4 generally has two caches: the first, the L1 cache, operates at clock speed. In the Willamette and Northwood, the L1 cache has a data capacity of eight kilobytes. In the Prescott, this capacity is increased to sixteen kilobytes. The L2 cache has a shared code and data capacity of between 256 and 2,048 kilobytes. There is L1 cache access latency of between two and four cycles, which increases to as much as sixteen cycles for the Willamette/Northwood's L2 cache (twenty-three cycles for the Prescott). These latencies are incurred if the data is not present in the registers, and has to be summoned from the cache.

The L1 cache stores up to 12,000 micro-operations (μops), which are decomposed assembly code instructions. Most instructions are converted into between one and four μops. Programs which contain more than 12,000 μops are partially stored in the L2 cache or memory. While manipulating the caches is the task of the implementer rather than that of the cipher designer, there is a simple guideline that the designer can use to benefit the implementer.

Guideline 4 **(Medium)** Design the cipher so that both its code and the state fit within the L1 cache.

This guideline is ranked as Medium, because a cipher that does not fit within the L1 cache is still shielded from the effects of slow memory by the much larger L2 cache. Phelix [18] adheres to this guideline, with its 160-bit state and small code size easily fitting into the Northwood's L1 cache. The block cipher LOKI97 [2] uses two s-boxes, one 13×8 and the other 11×8. Together this amounts to ten kilobytes of memory, and the s-boxes will not fit into the Willamette/Northwood's L1 cache. The tabular form of the AES also causes pressure in the L1 cache, as noted by its degraded performance [20].

One of the more interesting examples in this regard is Py [1]. It is one of the fastest ECRYPT ciphers on the Pentium III, with a throughput of one byte for each 2.8 cycles. Py uses two large tables. Elements in each table are modified using elements from both tables. The "novelty" of Py is the *rolling array*, which uses an old optimization trick commonly applied to LFSRs.

The traditional way to represent a LFSR with l stages is to use a circular buffer B of size $m = l$. A pointer $p, B \leq p < B + m$ indicates the location of stage W_0. The element W_1 is located at $p+1$ if $p+1 < l$ otherwise at location B. When the LFSR clocks, the feedback is written into the LFSR (usually at W_0), and the pointer increments by one. No elements are moved, or changed, except for where the feedback is placed. At each step, the pointer has to be checked to see if p equals $B + m$, in which case it is set to B. Any access into the LFSR also need to be bound checked.

Py's designers reason that the check on p is an expensive overhead. Py mitigates this expense by trading off memory. The trick it uses to allocate a buffer that is much larger than the LFSR, that is, $m \gg l$. As time passes, the pointer p moves along this buffer, and the feedback element normally written at W_0 is instead written at W_l. When the pointer reaches the end of the buffer, all of the elements representing the LFSR, from $B+m-(l+1)$ to $B+m-1$ are physically copied back to B through to $B+l$. The bound check on p needs to be made once in $(m/|W|)$ words of outputs, where $|W|$ is the size of the output word. This strategy can be adapted to any cipher in which the feedback is written only to the contiguous positions at the start of the LFSR. The copy operation employed by Py is expensive, but the cost is amortized over all of the intervening operations. Thus Py is at its most effective when the size of the buffer approaches the size of the L1 cache.

The default size for the buffers in the Py code submitted to ECRYPT is $b = 4000$ stages, where each stage is 32 bits. There are two buffers, so the state size is 32 kilobytes, which is commensurate with the size of the Pentium III's L1 cache. Clearly the cipher is less efficient on the Pentium 4, since the copy operation needs to be called four times more frequently to accommodate the much smaller size of that machine's L1 data cache. Benchmarks for Py with varying buffer sizes are given in Table 1 in Section 7.

5 Processing Data

From the cache, the data moves to the CPU. The cipher designer needs to understand how the data is processed, and from there choose instructions that optimize that processing.

5.1 The Pentium 4's Execution Engine

The CPU operates on data summoned to the registers, using its three-phase execution engine. The first phase, the Instruction Front-End converts assembly instructions into μops and supplies them in the order provided through the instruction portion of the L1 cache to the Instruction Execution core.

In theory, the *super-scalar* Instruction Execution core is able to execute six μops (for example, six exclusive-ors) within a single clock-cycle. It re-orders the instructions provided by the Instruction Front-End to provide the fastest possible execution, withstanding dependencies between the operands. The core has four ports that provide access to its execution units, including

1. port #0 has a fast integer unit and a MMX/SSE move unit[2]. The port can dispatch two fast integer operations per cycle; or one fast integer operation and one MMX/SSE move or store operation. Fast integer operations include logical operations, addition, subtraction and exclusive-or.

[2] Floating point operations are ignored in this discussion, since floating point arithmetic rarely features in symmetric cipher design

2. port #1 has a fast integer unit, and a normal integer unit. The port can dispatch two fast integer operations per cycle. An integer multiply, shift or rotate, or MMX/SSE operation can be substituted for the second fast integer operation.
3. port #2 deals with fetching memory for the general purpose registers. It can deal with one operation per cycle.
4. port #3 deals with storing data to memory. It can deal with one operation per cycle.

When the execution units have processed the μops, the third phase of the execution engine, the Instruction Retirement Unit, takes them and retires them in the correct order to maintain program correctness. Although the instruction core can execute six μops in parallel, the instruction retirement unit retires three μops in one clock cycle. This is the upper bound on the CPU throughput.

Guideline 5 **(Medium)** To take advantage of the Pentium 4's super-scalar ability, ensure that all instruction ports are occupied at all times for optimal execution.

There are some mechanisms within the Pentium 4 architecture, such as register renaming, which work to maximize the efficiency of the core (which is why this guideline has a medium priority). But the cipher designer needs to consider the types and order of operations used within a cipher. For example, if the cipher design requires two parallel executions of a multiplication, they will be executed serially as only port #1 handles general-purpose multiplication. Alternatively, if the cipher design requires parallel execution of a multiplication and an addition, the operations can be shunted to ports #1 and #0 respectively for simultaneous execution.

5.2 Throughtputs and Latencies

Each instruction is associated with a *latency* and a *throughput*. The latency of the instruction is the delay incurred before the CPU can operate on the next dependent instruction. For example, the latency affects the timing of the instruction sequence $a = a \oplus b; a = a \ll c$. The throughput is the delay before an independent operation can be scheduled on the same execution unit. For example, throughput affects the timing of $a = a \oplus b; c = c \oplus d$.

On the Intel Pentium 4, fast integer operations such as 32-bit addition, subtraction, exclusive-or and logical operations such as AND and OR all have throughputs and latencies of $\frac{1}{2}$ cycle (except on the Prescott, where these operations have latencies of one cycle). Two of these $\frac{1}{2}$ cycle operations can be scheduled and completed on a fast integer port within each clock cycle (assuming that their operands are dependency-free and located within the registers). As there are two fast integer ports, this means that up to four fast operations can be processed within the execution unit per cycle! This is a peak rate, and is subject to the limiting factor of three operations per cycle on the Instruction Retirement unit.

Shifts and rotates are executed on the normal integer unit, and on the Willamette/Northwood, have a worst-case latency of four cycles. On the same platforms, multiplication has a worst case throughput on five cycles, and latency of up to eighteen cycles. Timings for all operations can be found in [10].

Many 32-bit compilers offer decent support for 64-bit operands, and translate simple 64-bit arithmetic and logical operations into the underlying 32-bit instruction codes quite efficiently. A sixty-four bit exclusive-or can be simulated using two thirty-bit exclusive-ors. A sixty-four bit addition can be simulated using two thirty-bit additions, a comparison, and possibly an additional carry. Sixty-four bit rotation can be simulated using four thirty-two bit shifts and two exclusive-ors. On a per-bit basis, none of these operations is much worse than on 32-bit operands.

Mir-1's *loop_state_update* sub-function updates four sixty-four bit variables using the simple operations of addition, shifting, AND, OR, and multiplication. A sixty-four bit multiplication can be simulated on thirty-two registers using four multiplications, and eight additions. Considering the dependencies between operations, the four multiplications in *loop_state_update*, responsible in the production of one eight byte keystream word, expend as much as one hundred and twelve clock cycles!

Guideline 6 **(High)** Choose operations with low latency and low throughput. Avoid expensive operations such as multiplication or division, unless necessary.

The fastest ECRYPT ciphers on this platform (HC-256 [19], Phelix [18], and Py [1]) adhere to this guideline and also do not make extensive use of s-boxes or table-lookups. Unlike other cipher primitives, s-boxes do not take single or fractional clock cycles. For example, a single 8×8 s-box lookup $y = S(x)$ on a 32-bit machine may take up to three operations, as shown in Figure 2.

```
movzx eax, DWORD PTR _x$[ebx]      ; copy source to index register
mov   ecx, DWORD PTR _sbox[eax*4] ; copy address of s-box
mov   DWORD PTR _y$[ebx], ecx     ; perform s-box lookup
```

Fig. 2. Assembly Code for 8×8 S-box Lookup

Because of the dependencies between them, these operations all occur in serial. Register pressure prevents complete parallelization of more than two neighbouring s-box lookups. Additionally, the cache issues demonstrated in Section 4 come into play.

Guideline 7 **(Medium)** S-boxes are large and slow. Use with caution.

This guideline has a Medium priority because the s-box provides a potentially rich source of non-linearity. Many ciphers, including HERMES [12], use

8×8 s-boxes, whereas Dragon [3] uses 32×32 s-boxes. The use of larger s-boxes improves their non-linearity, but also poses a practical problem: whereas 8×8 s-boxes require 256 bytes of storage, 32×32 s-boxes require sixteen gigabytes.

This is clearly not practical, so in Dragon a virtual s-box is constructed as $y = S_0(x_0) \oplus S_1(x_1) \oplus S_2(x_2) \oplus S_3(x_3)$ where $x = x_0 \| x_1 \| x_2 \| x_3$. The memory requirement of the virtual s-box is further reduced to two kilobytes by reusing one of the 8×32 s-boxes, avoiding a likely L1 cache overflow. Performing a lookup on Dragon's virtual 32×32 s-box consists of four 8×32 s-box lookups, isolating individual bytes in the source word, using three ANDs and three shifts, and combining the results using three exclusive-ors. Four index registers are required for a single 8×32 s-box. As there are six s-box implementations, and seven registers, register pressure means that the s-box invocations are compelled to be serialized. This was considered by the designers of Dragon as a necessary security efficiency trade-off.

Guideline 8 **(High)** Maximize productivity for processor work.

Having chosen a series of efficient operations that provide the cipher with necessary security, make sure that the work performed by processor to execute those operations is not wasted. The most obvious way to ensure this is to output keystream in blocks that are multiples of the word size used by the cipher's internal state. HC-256, Py, RC4, and MIR-1 all output keystream blocks that are same size as an internal state word. Dragon has a 32-bit internal state word, and outputs 64-bit words. These all represent good usage of the work performed by the processor. Conversely, MAG outputs a block one-quarter the size of the internal state word, which wastes much of the work performed by the processor.

5.3 Branch Prediction

Some ciphers use branching as an intrinsic part of the algorithm. For example, MAG's concise update function contains just four assignments, including one based upon the result of an inequality comparison (that is, a branch).

The Intel Pentium 4 has a branch prediction unit that analyzes the anticipated path of the branch, executes it, and prepares the result by the time the branch expression is evaluated. When the branch is mis-predicted, the results are discarded and many cycles of CPU time are wasted, depending upon the size of the execution pipeline. The Pentium 4 has an exceptionally large pipeline, with as many as 31 stages, compared to the ten stages of the Pentium III. Thus a branch mis-prediction will have much worse consequences on the former. Unfortunately, the branch prediction algorithms of the Pentium 4 perform badly with random data, causing a delay of up between 24 and one hundred cycles with each mis-prediction. This situation is worse than if the Pentium 4 contained no branch prediction at all [7].

Guideline 9 **(Medium)** Avoid branching within the cipher algorithm.

Branch mis-prediction plays a major role in the poor performance of MAG. Yet RC4, to which MAG is most closely compared by the latter's designer, contains no branching of this type. The small size of MAG's update function means that the CPU has no chance to recover from the mis-prediction penalties. This is one reason as to why, despite the similar number of operations, MAG is four times slower than RC4. If branches can not be omitted, in some cases the pipeline penalties can be avoided by careful coding (for example, using the CMOV or SETcc instructions).

6 Instruction Extensions

The MMX and SSE extensions have potential as a valuable avenue for cipher implementors. The MMX extensions offer 47 new instructions on eight 64-bit registers. The SSE2/SSE3 instructions bring a wide range of integer and floating point operations to a set of eight 128-bit XMM registers. Many of the instructions are variations based around the size of the operands, which can have widths of 8, 16, 32, 64, or (for SSE) 128 bits. The primary benefit of MMX/SSE, aside from the additional registers, is that one instruction processes multiple operands in parallel.

The new instruction sets are somewhat deficient, and not particularly efficient. The SSE instructions have typical throughputs and latencies of two cycles. This puts them on an even footing with the fast integer operations on general purpose registers, which operate on 32-bits and execute in half a cycle.

Most valuable to cryptographers are the instructions for parallel execution of simple arithmetic and logical operations, and for a faster multiplication. The MMX and SSE instructions cannot address memory, which poses problems with s-boxes. S-boxes can be implemented on the general purpose registers, and indexed using operands computed using MMX/SSE. There is a strong penalty of six cycles for transferring data between the XMM registers and general purpose memory, making it problematic to implement any cipher containing s-boxes using MMX/SSE. Also the MMX and SSE instruction sets lack branching instructions, which can be simulated at a price on the general purpose registers.

Guideline 10 **(Medium)** Do not assume that there is a significant advantage in using MMX or SSE instruction extensions.

In particular, stream ciphers based on small functions that incorporate s-boxes, or other indirect addressing mechanisms, are probably not amenable for implementation using MMX or SSE. Some of the ECRYPT ciphers that are not suitable for efficient implementation using these extensions include Dragon, Py and HC-256.

7 Discussion and Conclusion

In symmetric cipher design, security is paramount and efficiency is a secondary consideration. However, it is an area in which there are now many successful candidates, fulfilling both security and efficiency requirements. The ciphers are judged now, on the platforms that we have available. A cipher designer who does not understand at a high-level, the architecture of the target machine, should expect sub-optimal performance from the cipher. This results in a non-competitive cipher.

The performance of some sub-optimized stream ciphers is shown in Table 1, for varying payload lengths. The metrics are gathered from an Intel Pentium 4 (Northwood) processor, running at 2.6 GHz, with an 8 kilobyte L1 data cache. For each test, one gigabyte of data is encrypted. When the specified payload has been encrypted, IV rekeying occurs. This overhead is incorporated into the metric, which is expressed as megabits per second. Py-x represents Py with a rolling array $x\times$ the state size of Py implemented using traditional techniques.

Table 1. Performance of Modern Stream Ciphers on the Intel Pentium 4 (Northwood)

Cipher	Word Size (bits)	Internal State (bits)	Features	Payload (bytes)			
				100	1,000	10,000	10^9
				Throughput (mbits/s)			
Dragon	32	1,088	large s-boxes	407	708	1,028	1,380
HC-256	32	65,536	very large tables	7	66	414	1,372
MAG	32	4,096	frequent branching	< 1	7	43	163
MIR-1	64	768	64-bit multiplication	198	362	419	464
Py-1	32	1,560	no rolling array	59	258	491	602
Py-4	32	6,240	large state	68	270	443	797
Py-16	32	24,960	very large state	65	256	446	801
RC4	8	2,048	small and fast	244	488	622	729

Those ciphers that most closely adhere to the guidelines issued in this paper — Py, Dragon, and HC256 – are the fastest on the Intel Pentium 4. The metrics of all the ciphers shown degrade with frequent rekeying, but those worst affected are those that rekey by updating each element in a large state (such as HC-256), or by excessively updating each element (for example, MAG, which during rekeying discards 2^{14} bytes of keystream, for each 100-byte payload). These metrics imply that designers need to pay attention to the key agility of their ciphers. They also suggest that the rolling arrays of Py lose their advantage on architectures with small cache sizes, such as the Intel Pentium 4.

This paper does not suggest that ciphers should be designed exclusively for the Intel Pentium 4. Most of the guidelines should be considered irrespective of the targeted architecture. All that is required to implement the guidelines is some knowledge about computer architectures, including the register set; the memory layout including the size and speed of the caches; the types, throughputs and latencies of available instructions; and special features of the architecture, such as SSE2.

References

1. E Biham and J Seberry. Py (Roo): A Fast and Secure Stream Cipher Using Rolling Arrays, 2005. Available at http://www.ecrypt.eu.org/stream/py.html.
2. L Brown and J Pieprzyk. Introducing the new LOKI97 block cipher. In *First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1998. Available at http://csrc.nist.gov/encryption/aes/.
3. K Chen, M Henricksen, L Simpson, W Millian, and E Dawson. Dragon: A fast word based cipher. In *ICISC '04*, volume 3506 of *LNCS*, pages 33–50, 2004.
4. J Daemen and V Rijmen. Rijndael. In *First Advanced Encryption Standard Candidate Conference, National Institute of Standards and Technology (NIST)*, August 1998. Available at http://csrc.nist.gov/encryption/aes/.
5. ECRYPT. Call for stream cipher primitives, April 2005. Available at http://www.ecrypt.eu.org/stream/call/.
6. ECRYPT. ESTREAM End of Phase 1, March 2006. Available at http://www.ecrypt.eu.org/stream/endofphase1.html.
7. A Fog. How to optmize for the Pentium family of microprocessors, 2004. Available at http://www.agner.org/assem/pentopt.pdf.
8. M Henricksen and E Dawson. Rekeying issues in the MUGI stream cipher. In *SAC'2005*, volume 3897 of *LNCS*. Springer-Verlag, 2006.
9. Intel Corporation. *IA-32 Intel Architecture Software Developers Manual Volume 2: Instruction Set Reference*. Intel Press, 2001.
10. Intel Corporation. *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*. Intel Press, 2001.
11. C Jansen, T Helleseth, and A Kholosha. Cascade Jump Controlled Sequence Generator (CJCSG), 2005. Available at http://www.ecrypt.eu.org/stream/ciphers/pomaranch/pomaranch.pdf.
12. U Kaiser. Hermes Stream Cipher, 2005. Available at http://www.ecrypt.eu.org/stream/hermes.html.
13. M Matsui and S Fukuda. How to maximize software performance of symmetric primitives on Pentium III and 4 Processors. In *FSE 2005*, LNCS, pages 398–412. Springer-Verlag, 2005. To appear.
14. A Maximov. A new stream cipher "Mir-1", 2005. Available at http://www.ecrypt.eu.org/stream/mir1.html.
15. B Schneier and D Whiting. Fast software encryption: Designing encryption algorithms for optimal software speed on the Intel Pentium Processor. In *FSE '97*, volume 1267 of *LNCS*. Springer-Verlag, 1997.
16. R Vuckovac. MAG My Array Generator (a new strategy for random number generation), 2005. Available at http://www.ecrypt.eu.org/stream/mag.html.
17. D Watanabe, S Furuya, H Yoshida, and K Takaragi. A new keystream generator MUGI. In *FSE'03*, volume 2365 of *LNCS*, pages 179–194. Springer-Verlag, 2003.
18. D Whiting, B Schneier, S Lucks, and F Muller. Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive, 2005. Available at http://www.ecrypt.eu.org/stream/phelix.html.
19. H Wu. A New Stream Cipher HC-256, 2004. Available at http://eprint.iacr.org/2004/092.pdf.
20. E Young. Re: More LTC timings... Posting to sci.crypt usenet group on 18 June, 2003.