# Optimizing a Fast Stream Cipher for VLIW, SIMD, and Superscalar Processors

Craig S.K. Clapp

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

**Abstract.** The mismatch between traditional cipher designs and efficient operation on modern Very Long Instruction Word, Single Instruction Multiple Data, superscalar, and deeply pipelined processors is explored. Guidelines are developed for efficiently exploiting the instruction-level parallelism of these processor architectures.
Two stream ciphers, WAKE-ROFB and WiderWake, incorporating these ideas are proposed. WAKE-ROFB inherits the security characteristics of WAKE, from which it is derived, but runs almost three times as fast as WAKE on a commercially available VLIW CPU. Throughput in excess of 40 MByte/s on a 100 MHz processor is demonstrated. Another derivative, WiderWake, whose security characteristics are not directly transferrable from WAKE runs in excess of 50 MByte/s on the same processor.

## 1 Introduction

Much of existing cipher design stems from an era when processors exhibited little or no instruction parallelism or concurrency. Given such processors, the route to achieving the ultimate in performance for software based encryption algorithms was to reduce to a minimum the total number of operations required to encrypt each symbol. Examples of algorithms that epitomize this strategy are RC4[1], SEAL[10], and WAKE[1].

The latest generation of processors gain much of their performance improvements by having deeper pipelines and a greater degree of available parallelism than their predecessors. This dictates additional design criteria for algorithms that are to fully benefit from these changes. Until now little attention has been paid to optimizing ciphers to run on such architectures.

We examine the performance limitations of some existing ciphers on the new CPU architectures and make suggestions for design practices to maximize speed. Some of these recommendations are contrary to practices currently favored in the design of ciphers for software execution.

Two related cipher families incorporating these ideas are presented as working examples of the potential for substantially increased throughput available on recently introduced processors.

[1] RC4 is a trademark of RSA Data Security Inc. Discussion of RC4 herein refers to the cipher described under that name in [11].

## 2 Exploiting Parallelism in Modern Processors

Instruction-level parallelism in modern processors is expressed in several forms:

- **Pipelining** - The execution pipe is split into several stages with registers in between. The clock rate can be increased because only a small amount of work is done in each stage of the pipe. It takes several machine cycles from the time an instruction enters the pipe until the time the result comes out, however, during that time additional instructions can be fed into the beginning of the pipe so long as they don't depend on results that are not yet available. Thus there is concurrency between instructions that are in different stages of the pipeline.

As well as being a technique used in single threaded processors, pipelining is typically used in the execution units of each the following architectures.

- **Superscalar** - Multiple execution units are implemented in one CPU. Several of them can accept instructions on each machine cycle. Assignment of instructions to the available execution units, and resolution of data dependencies between instruction streams, are tasks performed by the CPU at run-time.
- **Very Long Instruction Word (VLIW)** - Like superscalar, multiple execution units are implemented in one CPU. Several execution units can accept instructions on each machine cycle. Unlike superscalar however, VLIW instructions are assigned to specific execution units and data dependencies between instruction streams are resolved, at *compile-time*, i.e. the compiler generates assembly code that accounts for the CPU's pipeline delays in each path. So, two members of a VLIW CPU family having different pipeline delays may need different assembly code even if their instruction sets are identical.
- **Single Instruction, Multiple Data (SIMD)** - A single instruction stream is applied simultaneously to several data elements. This technique is especially favored for accelerating video and graphics processing. The Intel MMX instruction set recently added to the Intel Pentium line is one example.

Some authors have optimized their algorithms for 64-bit CPUs in their quest to extract more performance from recent processors, an example being the secure hash Tiger[2]. However, direct support for 64-bit arithmetic continues to be substantially limited to the niche market of high-end RISC CPUs.

Meanwhile, Intel-Architecture CPUs and other 32-bit processors for desktop and embedded applications are eagerly embracing a combination of superscalar, VLIW, and SIMD techniques. SIMD instructions have also been added to some of the high-end 64-bit RISC CPUs to accelerate video and graphics operations.

In SIMD architectures it is common for carry generation to take extra instructions, causing multiple precision arithmetic to be somewhat inefficient. For this reason we suggest that for widest applicability algorithms should avoid 64-bit arithmetic. $n \times 32$-bit SIMD compatibility is the preferred way of taking advantage of data paths wider than 32-bits.

A common assumption in the design of software-oriented algorithms is that table-look-ups are inexpensive operations provided that the table fits inside the processor's cache[1, 10, 12]. On modern deeply pipelined processors, memory

accesses, even to the CPU's local cache, typically have a longer latency than simple arithmetic or logical operations. Also, in highly parallel CPUs it is rare to have as many concurrent ports into the data cache as the number of parallel execution paths. Consequently, table-look-ups on such processors can be much more expensive relative to arithmetic or logical operations than they would be on a non-pipelined single-threaded CPU. To mitigate this, algorithms should be designed so that other useful work can proceed in parallel with table-look-ups.

Bit-rotations within a word have found favor in a number of recent algorithms[6, 10], with some including *data-dependent* rotations[9]. Common SIMD instruction sets, MMX included, do not directly support rotations within each of the multiple data operands, so a rotation must be synthesized by merging the results of two simple shifts, at a cost of three instructions. On a SIMD machine, *data-dependent* shifts and rotates can be substantially more costly than their fixed-length counterparts since applying different shifts to each operand is commonly not supported.

So, for efficient SIMD operation, shifts are preferred over rotations, and fixed-length shifts and rotations are preferred over data-dependent ones.

VLIW and superscalar architectures do not necessarily show a penalty for rotations versus shifts or for data-dependent versus fixed, except that rotations may only be efficient for the native word length.

## 3 Parallelism in Existing Ciphers

A characteristic of many existing ciphers is that a symbol of plaintext undergoes numerous inherently sequential operations on its way to becoming ciphertext, and the next plaintext symbol cannot be processed until the preceding ciphertext symbol is known. Encryption using a block algorithm such as DES[7] in cipher-feedback (CFB) mode[4] is a classic example of this characteristic[2]. The DES rounds are inherently performed sequentially, and the level of parallelism within a DES round is small except for eight parallel S-box look-ups. Unfortunately, table-look-ups are one of the *least* parallelizable operations on a modern CPU.

*Coarse-grained* parallelism can be forced on a system by for instance interleaving several independent cipher streams, but for a *single* CPU to get more throughput this way the associated replicated instruction streams need to map efficiently to the *fine-grained* instruction-level parallelism of modern processors. As a minimum, interleaving requires replication of the state variables that differ with each instance of the cipher, such as chaining variables or initialization vector dependent look-up-tables. The enlarged amount of state necessary to execute multiple instances of a cipher concurrently may *reduce* performance if it causes the cache capacity to be exceeded, or if it causes variables that otherwise could be held in CPU registers to instead be accessed from memory.

[2] While *encryption* in CFB mode is inherently a serial process, *decryption* using a block cipher in CFB mode offers unlimited block-level (coarse-grained) parallelism. This parallelism arises from the same absence of feedback loops in the decryption computational flowgraph that accounts for this mode's finite error extension characteristic.

**Table 1. Performance characteristics of various encryption algorithms**

| | RC4 | SEAL | WAKE-CFB |
|---|---|---|---|
| Number of bits ciphered per iteration | 8 | 128 | 32 |
| Number of 32-bit operations per iteration | 15 | 68 | 24 |
| Number of 32-bit operations per byte | 15 | 4.25 | 6.0 |
| Number of 32-bit operations in critical path (Add/XOR/mask, table-look-up, other) | 6 (2, 2, 2 stores) | 35 (27, 8, 0) | 17 (13, 4, 0) |
| Number of cycles in critical path | 10 | 51 | 25 |
| Normalized critical path | 10cycles/byte | 3.2cycles/byte | 6.25cycles/byte |
| Apparent parallelism (ops-per-iteration / ops in critical path) | 2.5x | 1.94x | 1.41x |
| Critical-path efficiency (ops in critical path / cycles in critical path) | 0.60 | 0.69 | 0.68 |
| P-factor (Apparent parallelism x critical-path efficiency) | 1.5 | 1.33 | 0.96 |
| | | | |
| Benchmarked performance on 32-bit VLIW CPU | 10.6cycles/byte | 3.5cycles/byte | 6.38cycles/byte |

To determine the upper limit on performance of a cipher on a suitably parallel processor architecture we attempt to identify the software *critical path* through the algorithm. This is the path through the algorithm from one output symbol to the next, that has the largest *weighted* instruction count, the weighting being the number of cycles of latency associated with each type of instruction.

For instance, on most processors the result of a simple operation like an addition or XOR can be used in the subsequent cycle - these instructions are said to have a one cycle latency. However, reading from memory, even when the data is in the CPU's local cache, will typically take several cycles. Data read from cache commonly suffers a two or three cycle latency on modern deeply pipelined processors, while one recent introduction has it is as high as five cycles. Table-look-ups are inherently memory references, since even if the table is so small that it could reside in the generous register space of some modern processors, it cannot be placed there because the instruction sets do not support indirect referencing of registers.

Table 1 compares the theoretical and benchmarked encryption performances of RC4, SEAL, and WAKE on a 32-bit CPU with a RISC-like instruction set. The critical paths assume a memory-read latency of three cycles. All other operations are assigned a latency of one cycle. The total number of operations per iteration includes reading a plaintext input buffer, applying the cipher, and writing the ciphertext to an output buffer. For purposes of comparison loop overhead has been ignored when counting operations since it is essentially common to all the algorithms and can generally be reduced to insignificant levels by sufficient loop unrolling.

The 'apparent parallelism' listed in the table is the ratio of the total number of operations per iteration of the cipher to the number of operations in the critical

path. It gives an indication of the most execution units that the algorithm could take *some* advantage of.

Another metric - the critical-path efficiency, tells what fraction of the cycles in the critical path are actually used to issue an instruction. In a pipelined processor the unused cycles here represent an opportunity for performing additional operations without resorting to more execution units or slowing the computation. Ideally an algorithm will have enough parallelism to exploit these 'bubbles' in the critical-path pipeline and *still* make use of multiple execution units.

A gauge of the most execution units that can be *fully* exploited by an algorithm can be obtained by taking the ratio of the total number of operations per iteration of the cipher to the number of cycles in the critical path. This is the same as the apparent parallelism times the critical-path efficiency, and is referred to in the table as the P-factor (for parallelism-factor). Note that since the critical path is a function of both the algorithm *and* the associated instruction latencies, the values for these metrics will change if the processor under consideration has different latencies from those assumed here. The instruction latencies used for the table are those of the processor actually used for benchmarking.

In order for an algorithm to fully exploit a superscalar, VLIW, or SIMD processor the P-factor needs to be substantially greater than unity. Ideally it should be no less than the number of parallel execution paths available in the target CPUs.

For some recently introduced processors the number of parallel execution paths is in the range of four to eight. Clearly, with P-factors of less than two, the encryption algorithms examined here cannot efficiently exploit the resources of such processors.

## 4 Design Strategy for a New Cipher

The strategy for developing a new cipher was to attempt to apply these principles to an already existing fast cipher, hopefully in a way that could leverage the security claims of the original. Two candidates were considered as the starting point for this exercise, RC4 and WAKE.

RC4 offers opportunity for speed-up both by use of wider data paths, and by an increase in concurrency. Schneier[11] suggests a modification to RC4 to exploit wider data paths, but only at the unacceptable expense of an exponential growth in the associated look-up-table that precludes extension to 32-bits. In some ways ISAAC[5] can be viewed as an extrapolation from RC4 to 32-bit and 64-bit data paths while not particularly addressing RC4's modest P-factor.

WAKE is already very fast, makes good use of a 32-bit datapath, avoids rotations, and its regularity lends itself to the possibility of efficient mapping to SIMD architectures. Its only weakness is a lack of concurrency principally brought about by cascaded table-look-ups. In addressing this limitation, another attraction over RC4 is WAKE's lack of self modification of its look-up-table.

WAKE was chosen as the candidate cipher for modification.

## 5 WAKE

In [1] Wheeler introduced the cipher WAKE. It uses a mixing function, $M(x, y, T)$, that combines two 32-bit inputs, $x$ and $y$, into one 32-bit output with the aid of a key-dependent $256 \times 32$-bit look-up-table, $T$. By constraining the values in the upper byte lane of the otherwise 'random' entries of $T$ to form a permutation of the numbers 0 to 255, the mixing function is made reversible in the sense that knowledge of the output word and one of the two input words is sufficient to uniquely specify the other input word. The mixing function, and its inverse, are shown in Fig. 1.

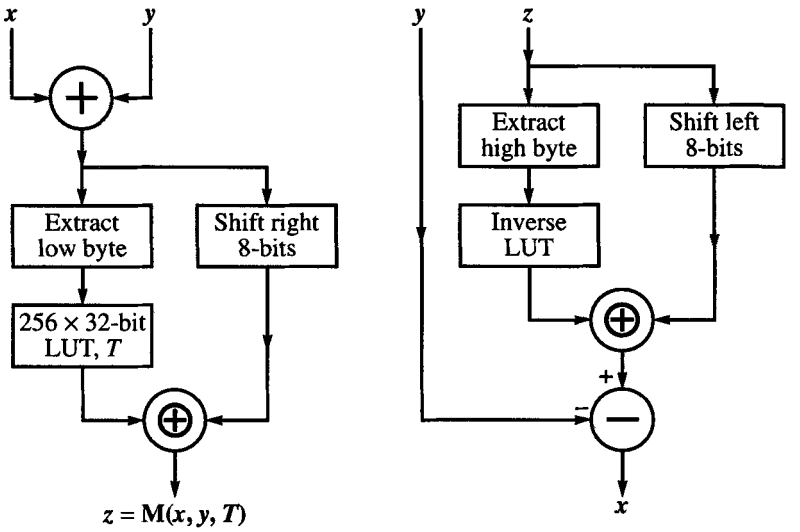


**Fig. 1** a) WAKE's Mixing function b) Inverse mixing function

WAKE consists of cascading four of these mixing functions with registered feedback around each one and overall feedback around the group. Four stages are chosen as the minimum number needed for complete diffusion. Fig. 2a shows WAKE in cipher-feedback mode as originally described.

In addition to its use in cipher-feedback mode, Wheeler suggests that WAKE is suitable for the production of a pseudo-random sequence for use as a stream cipher by XORing with the plaintext. This mode, shown in Fig. 2b and referred to here as WAKE-OFB, is used as the basis for the new ciphers since it conveniently circumvents the complaint that WAKE in cipher-feedback mode is susceptible to a chosen-plaintext attack.

Using the assumptions given for table 1, the mixing function has a critical path of 6 cycles, three of them for the table-look-up and one each for the other three operations in the path. WAKE-OFB cascades four mixing functions, for a total critical path of 24 cycles.

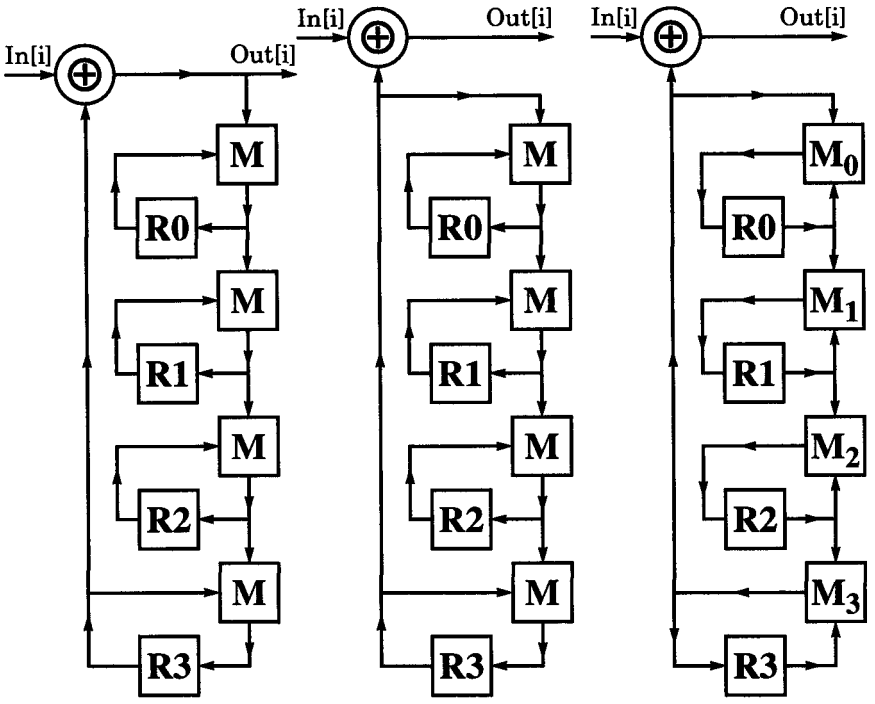


**Fig. 2** a) WAKE-CFB b) WAKE-OFB c) WAKE-ROFB

## 6 Taking a Step Backwards

Cryptographic security of a pseudo-random sequence, as used for a stream cipher, demands that no one part of the sequence can be predicted from any other part of the sequence. This security property makes no distinction between the forward and time reversed versions of the pseudo-random sequence. Thus, if a *reversible* pseudo-random number generator (PRNG) produces a cryptographically secure sequence then that same generator running in reverse must also produce a cryptographically secure sequence.

WAKE's next-state function is designed to be reversible in order to maximize the cipher's expected cycle lengths, thus when used in OFB mode it forms a reversible PRNG.

Fig. 2c shows the flowgraph for WAKE-OFB run in reverse which we will refer to as WAKE-ROFB. It differs from the forward version in that the direction of data through the registers is reversed, resulting in its stepping backwards through its states. Strictly speaking, to achieve the *actual* time-reversed sequence the mixing functions should also be replaced by their inverses. However, for convenience we can just as easily leave them unchanged, reasoning that the forward or inverse mixing functions will provide equally good but different encipherment, just as Wheeler in [1] reasons that performing an arithmetic right shift on signed 32-bit operands inside the mixing function is as good, but different from, performing a logical shift as is done in his reference implementation. In any case,

the forward and inverse mixing functions have identical critical-path lengths so choosing one over the other does not affect the performance analysis.

An interesting result of this reversal of the state machine is that the longest path from the output of one register to the input of another is now through only two mixing functions ($M_3$ and $M_0$), instead of through all four as in the original flowgraph. At first this might suggest that the critical path has been halved. However, closer inspection reveals that the critical path has in fact been reduced by a factor of *three* since at any given time three out of the four mixing functions can be evaluated concurrently, and three out of four registers updated, so over a period of four mixing function evaluations each of the registers can be updated three times, as illustrated by the following pseudo-code:

- Starting with known values for R0 through R3:
  - Evaluate $M_1$, $M_2$, and $M_3$ in parallel, update R1, R2, and R3
  - Evaluate $M_0$, $M_2$, and $M_3$ in parallel, update R0, R2, and R3
  - Evaluate $M_0$, $M_1$, and $M_3$ in parallel, update R0, R1, and R3
  - Evaluate $M_0$, $M_1$, and $M_2$ in parallel, update R0, R1, and R2
- All four registers have now been updated exactly three times. Each update of R3 allows another word to be ciphered. Repeat the sequence until done.

*Thus, simply by running WAKE-OFB backwards we can achieve a threefold increase in parallelism while claiming identical security.*

Wheeler offers that WAKE's security can be enhanced by increasing the number of stages from the four given, albeit at a reduction in speed. Indeed, for the original WAKE, both the total computation and, more importantly, the critical path, are essentially proportional to the number of stages, with the result that performance inherently declines as stages are added.

Now let's consider adding stages to WAKE-ROFB. Just as 4-stage WAKE-ROFB had three times as much parallelism as WAKE-OFB, a time reversed 5-stage version has *four* times as much parallelism as its non-reversed counterpart. Where 4-stage WAKE-ROFB had a critical path of $\frac{4}{3}$ mixing function evaluations, 5-stage WAKE-ROFB has the slightly shorter critical path of $\frac{5}{4}$ mixing function evaluations.

Counter to our intuition, we note that *adding stages can actually speed-up the cipher*, even though the total amount of work per word ciphered increases in proportion to the number of stages. The caveat here is of course the need for the processor to exhibit adequate parallelism. However, additional stages always harm the cipher's performance on a *single-threaded* CPU, and also on a CPU with parallelism once all of its parallelism has been exploited. For this reason, and also because further stages give progressively diminishing return in critical-path reduction, it is unattractive to extend beyond five stages.

The principal attraction of the 5-stage version is that its fourfold parallelism is a better fit for typical SIMD datapaths than the 4-stage version's threefold parallelism. Similarly, optimal loop-unrolling for the 5-stage version involves a factor of four, while for the 4-stage version loop-unrolling by a multiple of three is most efficient, which may be inconvenient for some applications.

## 7 Adding to the Flowgraph - WiderWake

To further increase WAKE's parallelism and shorten its critical path we investigate inserting additional pipeline stages in the algorithm's flowgraph. The trick is to do so without compromising its security characteristics. Our goal is to reduce the critical path to just *one* mixing function evaluation, while keeping the total computation down to that of 4-stage WAKE so that single-threaded CPU performance is not impaired.

For algorithms that include overall feedback - WAKE included, adding pipeline stages cannot be done without changing the nature of the algorithm. This means that the modified version does not necessarily inherit the security characteristics of the original scheme.

As an example, Fig. 3a shows a rearrangement of WAKE with its registers now acting as pipeline stages. This simple expedient increases the available parallelism by a factor of four over WAKE since now all four mixing functions can be evaluated concurrently.

However, it may be observed that unlike the original WAKE, the version of Fig. 3a has no direct way of determining its *previous* state. Indeed, some states may not have a *unique* previous state, while others may have no previous state at all. The lack of bijectivity in this version's state transition function would, due to the birthday paradox, result in its expected cycle lengths being dramatically shorter than those of the original (i.e. inferior security). This illustrates the caution necessary in making modifications to the flowgraph.

In order to achieve our critical-path goal, every mixing function output must be registered before becoming a mixing function input. Given this, we note that for the flowgraph to be reversible we additionally require that at least one of the mixing functions has an input node in common with the input of a register (this guarantees that when the flowgraph is reversed there becomes at least one mixing function with registers defining two of its three nodes). Since this input cannot be that of any of the mixing function output registers without violating our critical-path constraint, we conclude that for an $n$-stage flowgraph we need a minimum of $n + 1$ registers in order to meet our objectives.

One such arrangement, with the minimum five registers needed for four stages, is shown in Fig. 3b.

This general topology, in which an additional register is added in the feedback loop of one or more of the stages, we will refer to as *WiderWake*.

We identify specific instances of the topology by the total number of stages and the number of modified feedback loops, following the naming convention illustrated by Fig. 3b and Fig. 3c.

The available parallelism is determined by the overall number of stages, it does not change with the number of feedback loops that are modified.

Each modified feedback loop adds another 32-bits of state to the state machine, increasing the complexity of the generator, and potentially its security.

With four-stage WiderWake we can modify at most three of the four stages (WiderWake$_{4+3}$). This is because if all four stages are modified we get a common factor of two between the number of registers in the outer-loop (4 registers) and

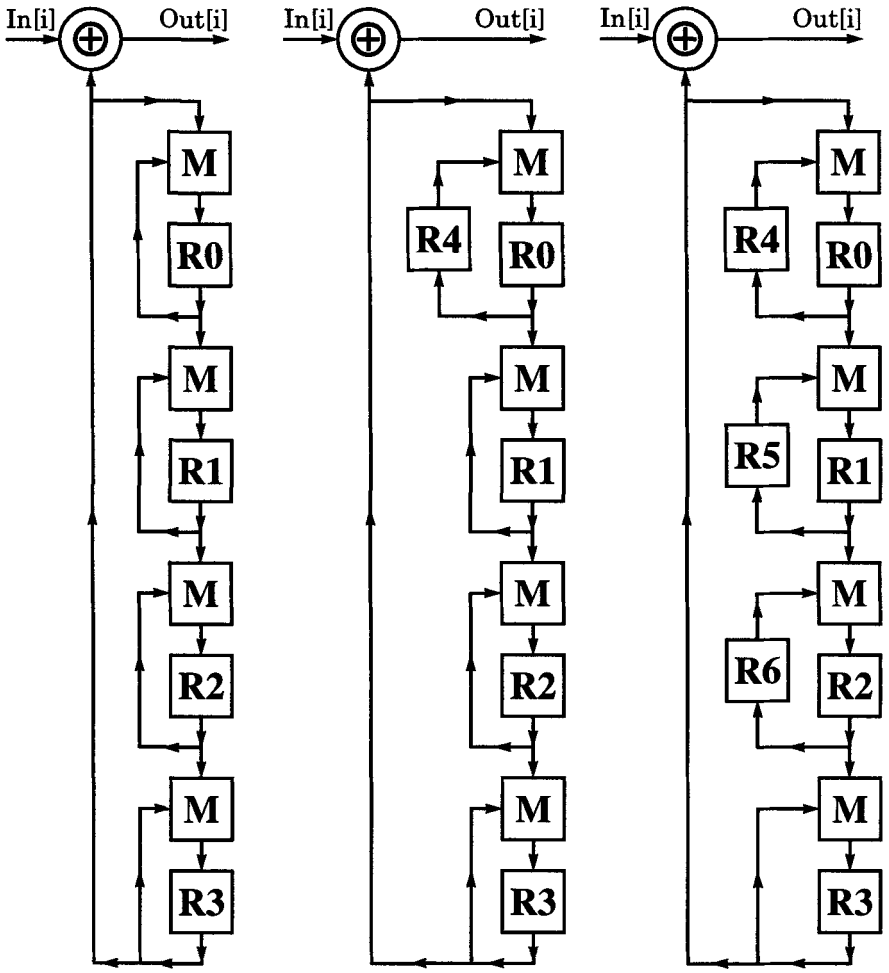


**Fig. 3** a) Pipelined WAKE (non-reversible) b) WiderWake$_{4+1}$ c) WiderWake$_{4+3}$

each minor loop (2 registers each). This causes the state machine to degenerate into two interleaved 128-bit generators instead of being a single 256-bit generator. The same problem does not afflict generators having an *odd* number of stages, however it is questionable whether modifying the stage which is tapped for the generator's output does anything to enhance security since it does not increase the amount of hidden state.

The most useful variants are suggested to be WiderWake$_{4+1}$, and WiderWake$_{4+3}$. The former has the benefit of using the fewest registers, and so may be best suited for backward compatibility with older register-impoverished processors. The latter has the highest generator complexity given the minimum number of four stages.

**Table 2. Performance characteristics of WAKE and WiderWake**

| | WAKE-OFB | | WAKE-ROFB | | WiderWake | |
|---|---|---|---|---|---|---|
| | 4-stage | 5-stage | 4-stage | 5-stage | 4+1 | 4+3 |
| Total generator state | 128-bits | 160-bits | 128-bits | 160-bits | 160-bits | 224-bits |
| Hidden state | 96-bits | 128-bits | 96-bits | 128-bits | 128-bits | 192-bits |
| Bits ciphered per iteration | 32 | 32 | 32 | 32 | 32 | 32 |
| 32-bit ops. per iteration | 24 | 29 | 24 | 29 | 24 | 24 |
| 32-bit ops. per byte | 6.0 | 7.25 | 6.0 | 7.25 | 6.0 | 6.0 |
| Number of 32-bit operations in critical path | 16 (12, 4, 0) | 20 (15, 5, 0) | 5.33 ($^{12}/_{3}$, $^{4}/_{3}$, 0) | 5.0 ($^{15}/_{4}$, $^{5}/_{4}$, 0) | 4 (3, 1, 0) | 4 (3, 1, 0) |
| Cycles in critical path | 24 | 30 | 8 | 7.5 | 6 | 6 |
| Normalized critical path | 6.0 cycles/byte | 7.5 cycles/byte | 2.0 cycles/byte | 1.9 cycles/byte | 1.5 cycles/byte | 1.5 cycles/byte |
| Critical-path speed-up | 1x (ref.) | 0.8x | 3.0x | 3.2x | 4.0x | 4.0x |
| Apparent parallelism | 1.5x | 1.45x | 4.5x | 5.8x | 6.0x | 6.0x |
| Critical-path efficiency | 0.67 | 0.67 | 0.67 | 0.67 | 0.67 | 0.67 |
| P-factor | 1.0 | 0.97 | 3.0 | 3.87 | 4.0 | 4.0 |
| | | | | | | |
| Benchmarked performance on 32-bit VLIW CPU | 6.23 cycles/byte | 7.73 cycles/byte | 2.32 cycles/byte | 2.28 cycles/byte | 1.90 cycles/byte | 1.85 cycles/byte |
| Throughput at 100MHz | 16.1MB/s | 12.9MB/s | 43.1MB/s | 43.9MB/s | 52.6MB/s | 54.1MB/s |
| Relative speed | 1x (ref.) | 0.81x | 2.69x | 2.73x | 3.28x | 3.37x |

## 8 Performance

A Philips TriMedia processor was chosen for performance comparisons between the algorithm variants as a vehicle that amply demonstrates the potential for improvement given adequate parallelism.

The TriMedia processor is a VLIW CPU containing five 32-bit pipelined execution units sharing a common set of registers. All execution units can perform arithmetic and logical operations, but loads, stores, and shifts are only supported by a subset of them.

Table 2 compares the characteristics of WAKE in output-feedback mode with the algorithm variants proposed herein. Encryption and decryption have identical characteristics. Assumptions used are the same as those given for table 1.

The benchmark conditions were a 100 MHz TriMedia processor running performance-optimized C-code to encrypt a buffer several times larger than the on-chip data cache. Comparable source-code optimizations were applied to all cases. The advertised benchmark performance includes all loop-overhead and cycles lost to cache misses, memory accesses, etc. No off-chip cache was present.

The benchmarked performance of 5-stage WAKE-ROFB is seen to marginally exceed that of the 4-stage version, demonstrating that extra stages can improve

performance for this topology in practice as well as theory.

WiderWake$_{4+3}$ shows very similar performance to WiderWake$_{4+1}$, as predicted. The slight improvement of WiderWake$_{4+3}$ over WiderWake$_{4+1}$ can be attributed to the former having fewer critical-length paths than the latter, thus easing the compiler's task of scheduling the code to simultaneously minimize all critical-length paths.

WAKE-ROFB and WiderWake achieve speed-ups over WAKE-OFB of better than 2.5x and 3x respectively. With performance of better than 2 cycles per byte, the WiderWake variants allow encryption or decryption at speeds in excess of 50 MByte/s on a 100 MHz processor. WAKE-ROFB exceeds 40 MByte/s in both 4-stage and 5-stage versions.

## 9 Cipher-Feedback Mode

Any of these generators could in principle be used in CFB mode while maintaining their speed, by taking the overall feedback from the ciphertext instead of direct from the final stage, just as in WAKE-CFB. However, the user is cautioned against doing so without further study. In CFB mode none of these generators can claim direct equivalence to WAKE's security. In particular, unlike WAKE-CFB where each ciphertext symbol affects the next ciphertext symbol, the fed-back ciphertext in all these new versions would take several cycles before it again influences the stream. So, as a minimum there might need to be some special processing to avoid weaknesses at the start and end of the stream.

In fairness to WAKE, it should be pointed out that WAKE-CFB itself offers fourfold parallelism in *decrypt* mode, an advantage not shared by WAKE-OFB. This comes about because the overall feedback of WAKE-CFB's encryption flowgraph becomes feed-forward during decryption, thereby breaking the outer feedback loop. The critical path then becomes just that of the minor loops around each stage, i.e. just one mixing function evaluation. After the first four ciphertext symbols have 'filled the pipe' all four mixing functions can be evaluated in parallel as each new ciphertext symbol arrives. That WAKE-CFB's decryption parallelism is not *unlimited* is a reflection of the fact that WAKE-CFB suffers *infinite* error extension under decryption, unlike a block cipher operating in CFB mode.

## 10 Table Initialization and Stream Re-synchronization

For a complete stream cipher definition two further components - look-up-table initialization (key scheduling), and stream re-synchronization (Initialization Vector processing), need to be specified.

Wheeler supplies an ad-hoc table-generation routine based on a 128-bit key (referred to as the *table-key*). We retain this routine with minor modifications to remove ambiguity from the original definition. On a single-threaded CPU this routine runs in about the time it takes to encrypt 1000 bytes with any of

the 4-stage variants. However, this routine is much less capable of exploiting instruction-level parallelism than our optimized cipher variants, so that on a CPU having substantial instruction-level parallelism table initialization can take more like 2000 to 3000 byte-encryption times. Improving on this is an area for future study.

Re-synchronization is achieved by setting the registers to a new state that is a hashed combination of the table-key and an initialization vector (IV). We propose a 64-bit IV for compatibility with common block ciphers operated in OFB mode. Our minimalist resync procedure consists of seeding the WiderWake state machine with a simple combination of the table-key and the IV, and stepping the generator until they are satisfactorily mixed, discarding the generator's output along the way. For an $n$-stage generator we choose to step the generator $2n$ times. This is enough so that all registers achieve avalanche at least once, and the output register achieves avalanche at least twice. For WiderWake$_{4+1}$ this resync process takes about as long as ciphering 32 bytes.

The resistance of this simple resync procedure to related-key cryptanalysis is unproven. This may represent an exposure if resync intervals are especially frequent, or if an attacker has the ability to force resyncs at will[3]. The first defence against such a weakness would be to step the generator more times during resync. Alternatively a secure hash such as SHA-1[8] could be used to robustly combine the table-key with the IV, however on a CPU that can exploit WiderWake's parallelism this is as costly as ciphering several hundred bytes.

## 11 Conclusion

We have illustrated the opportunities for performance gains through exploiting the instruction-level parallelism of current generation CPUs. We suggest that efficiency on these processors should be among the design criteria for new software-oriented ciphers.

The example ciphers are presented without a supporting security analysis. Cycle lengths have been determined experimentally for several models having shorter word-lengths and found to be as expected. It remains to be established what length of cipher stream can be safely exposed between IV changes and between table-key changes. Cryptanalysis is invited.

## References

1. D. J. Wheeler, "A Bulk Data Encryption Algorithm", *Fast Software Encryption (Ed. R. Anderson), Lecture Notes in Computer Science No. 809,* Springer-Verlag, 1994, pp. 127-134
2. R. J. Anderson and E. Biham, "Tiger: A Fast New Hash Function", *Fast Software Encryption (Ed. D. Gollmann), Lecture Notes in Computer Science No. 1039,* Springer-Verlag, 1996, pp. 89-97
3. J. Daemen, R. Govaerts, and J. Vandewalle, "Resynchronization Weaknesses in Synchronous Stream Ciphers", *Advances in Cryptology - EUROCRYPT '93 Proceedings,* Springer-Verlag, 1994, pp. 159-167

4. ISO/IEC 10166, "Information Technology - Modes of operation for an n-bit block cipher algorithm", International Organization for Standardization / International Electrotechnical Commission, 1991
5. R. J. Jenkins, "ISAAC", *Fast Software Encryption (Ed. D. Gollmann), LNCS 1039,* Springer-Verlag, 1996, pp. 41-49
6. B. S. Kaliski and M. J. B. Robshaw, "Fast Block Cipher Proposal", *Fast Software Encryption (Ed. R. Anderson), LNCS 809,* Springer-Verlag, 1994, pp. 33-40
7. NBS FIPS PUB 46-1, "Data Encryption Standard", National Bureau of Standards, U.S. Department of Commerce, Jan 1988
8. NIST FIPS PUB 180-1, "Secure Hash Standard", National Institute of Standards and Technology, U.S. Department of Commerce, April 1995
9. R. L. Rivest, "The RC5 Encryption Algorithm", *Dr. Dobb's Journal,* v. 20, n. 1, January 1995, pp. 146-148
10. P. Rogaway and D. Coppersmith, "A Software-Optimized Encryption Algorithm", *Fast Software Encryption (Ed. R. Anderson), LNCS 809,* Springer-Verlag, 1994, pp. 56-63
11. B. Schneier, *Applied Cryptography, Second Edition,* John Wiley & Sons, 1996, pp. 397-398
12. B. Schneier, "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)", *Fast Software Encryption (Ed. R. Anderson), LNCS 809,* Springer-Verlag, 1994, pp. 191-204

## Appendix A. WiderWake$_{4+1}$ Reference Implementation

The following C-code is a functional reference only. It does not represent all the performance optimizations embodied in the benchmarked source-code.

```
/* WiderWake4+1, Version 1.0, functional reference */
typedef unsigned long UINT32;     /* 32-bit unsigned integer */
/* Array sizes - UINT32 T[256], t_key[4], IVec[2], r_key[5]; */
/* addition is modulo 2^32,    >> is right shift with zero fill */
#define M(x,y,T)  (((x)+(y)) >> 8) ^ T[((x)+(y)) & 0xff]

void ofb_crypt(UINT32 *In, UINT32 *Out, int length, UINT32 *T, UINT32 *r_key)
{
   UINT32 R0, R1, R2, R3, R4, R0a, R1a, R2a, R3a;
   int i;
   R0=r_key[0];  R1=r_key[1];  R2=r_key[2];  R3=r_key[3];  R4=r_key[4];
   for (i = 0; i < length; i++)
   {
      R3a = M(R3, R2, T);     /*                                    */
      R2a = M(R2, R1, T);     /*    All four mixing functions       */
      R1a = M(R1, R0, T);     /*   can be evaluated in parallel     */
      R0a = M(R4, R3, T);     /*                                    */
      Out[i] = In[i] ^ R3;    /* Execution can overlap with mixing functions */
      R4 = R0;  R3 = R3a;  R2 = R2a;  R1 = R1a;  R0 = R0a;
   }
   r_key[0]=R0;  r_key[1]=R1;  r_key[2]=R2;  r_key[3]=R3;  r_key[4]=R4;
```

```
void resync(UINT32 *T, UINT32 *t_key, UINT32 *IVec, UINT32 *r_key)
  /* mix t_key with initialization vector and discard first eight words */
{
   UINT32 temp[8];  /* bit-bucket */

   r_key[0] = t_key[0] ^ IVec[0];  r_key[1] = t_key[1];
   r_key[2] = t_key[2] ^ IVec[1];  r_key[3] = t_key[3];
   r_key[4] = IVec[0];

   ofb_crypt(temp, temp, 8, T, r_key);
}


void key_sched(UINT32 *t_key, UINT32 *T)  /* expand t_key to look-up-table T */
{
   UINT32 x, z, p, t0;
   static UINT32 tt[8] = { 0x726a8f3b, 0xe69a3b5c, 0xd3c71fe5, 0xab3c73d2,
                           0x4d3a8eb3, 0x0396d6e8, 0x3d4c2f7a, 0x9ee27cf3 };

   for (p = 0; p < 4; p++) { T[p] = t_key[p]; }

   for (p = 4; p < 256; p++)
      { x = T[p-4] + T[p-1];  T[p] = (x >> 3) ^ tt[x & 7]; } /* (UINT32)x */

   for (p = 0; p < 23; p++) { T[p] += T[p+89]; }

   x = T[33];  z = (T[59] | 0x01000001) & 0xff7fffff;

   for (p = 0; p < 256; p++)
      { x = ( x & 0xff7fffff) + z;  T[p] = (T[p] & 0x00ffffff) ^ x; }

   x = (T[x & 0xff] ^ x) & 0xff;  t0 = T[0];  T[0] = T[x];

   for (p = 1; p < 256; p++)
      { T[x] = T[p];  x = (T[p ^ x] ^ x) & 0xff;  T[p] = T[x]; }

   T[x] = t0;
}
```

## Appendix B. Test Case

```
void test(void)
{
   UINT32 t_key[4] = { 0x12345678, 0x98765432, 0xabcdef01, 0x10fedcba };
   UINT32  IVec[2] = { 0xbabeface, 0xf0e1d2c3 };  /* Initialization Vector */

   UINT32  text[4] = { 0x1234abcd, 0xa0b1c2d3, 0x1a2b3c4d, 0x55667788 };

   UINT32 T[256], r_key[5];
   int i;

   key_sched(t_key, T);              /* Schedule key */
   resync(T, t_key, IVec, r_key);    /* Initialize generator state using IV */

   for (i = 0; i < 256; i++)         /* Encrypt text buffer 256 times */
      { ofb_crypt(text, text, 4, T, r_key); }

   for (i = 0; i < 4; i++) { printf("0x%08lx  ", text[i]); }  printf("\n");
}

/* final text[] == { 0x94739922, 0xb251752f, 0x1de1f2fe, 0x405f83dd } */
```