ENCRYPTION CODE GENERATOR

Paul Magrath BA(Mod) MIEI
Trinity College Dublin, Ireland
magrathp@tcd.ie

1. Problem and Motivation

Introduction

This project investigates using a code generator to generate the various variants of an Advanced Encryption Standard (AES) encryption loop. AES (see 'Background', chapter 2) is a form of encryption that Intel will begin supporting in hardware in their microprocessors in 2010. An encryption loop is the iterative loop that iterates through all the data to be encrypted and performs steps necessary to encrypt the data. As such it is a computationally expensive loop, requiring a large amount of CPU time, and is, hence, a candidate for optimization in order to reduce the time taken.

A code generator, in this context, is a program that generates a number of different variants of a piece of code in order to find which combination of optimization techniques yields the best possible result (see 'AES Code Generator', chapter 3, for variants employed here). The use of code generators is an established technique for solving problems in optimizing for modern architectures, used primarily in the research community. It is ideal for optimizing a small piece of code that uses a vast amount of processing time and to which the best optimizations are not obvious. Code generators have been very successfully applied to several domains such as Spiral, which generates code for linear transformations [1], and FFTW, which generates code for computing the discrete Fourier transform [2]. The motivation for a code generator is to avoid the problems with code maintenance and code readability that almost inevitably result from hand-tuned assembly specific to the architecture it is written for. A code generator can tune itself to the architecture it is running on to find the best combination of optimizations for that architecture, while remaining readable and maintainable as it can be written in a high level language, such as C++.

In 2010 Intel will release processors that will have AES instructions built in to their instruction set. This will greatly reduce the cost of encryption, as it will be possible to perform the encryption much more quickly and efficiently than previously. This is part of an enhancement, and replacement, of the current Intel SIMD

instruction set, the Streaming SIMD Extension (SSE). The replacement instruction set will be known the Intel Advanced Vector Extensions (AVX).

This project takes a look at what would be the most likely ways that the use of such instructions as part of a standard AES encryption loop could be optimised, how this can be automated using a code generator, and presents the results of running this AES code generator to generate these optimisation combinations on different architectures and processors.

2. Background and Related Work

AES

AES (Advanced Encryption Standard) is one of the most popular algorithms used in symmetric encryption.

Originally published as Rijndael [3], it was adopted as a standard by the U.S. government in November 2001 [4], after a five-year standardization process involving fifteen competing designs. The standard comprises three block ciphers, AES-128, AES-192 and AES-256, adopted from a larger collection. A block cipher is a cipher that operates on fixed-length groups of bits, termed blocks, with an unvarying transformation. The block cipher takes in two inputs, the plaintext of the block and the secret key, and outputs the ciphertext (encrypted text) of the block. Each AES cipher has a 128-bit block size, which means that 128-bits of the plaintext are encrypted into ciphertext in each iteration of the encryption loop. AES-128, AES-192 and AES-256 have secret keys of sizes 128, 192 and 256 bits respectively where AES-128 is the least secure while AES-256 is the most secure.

When the length of data to be encrypted exceeds the block size, a mode of operation must be used [5]. The two that we will concern ourselves with are Electronic Code Book (ECB) and Counter (CTR). These will be discussed in detail in the AES Code Generator (chapter 3).

An outline of the algorithm for AES-256 encryption is:

- **Key Expansion**
  Using the Rijndael key schedule, the 14 round keys are extracted from the 256-bit secret key.
- **Initial Round** (round 0)

The initial round is simply the bitwise XOR of the round key to the plaintext (referred to as the 'state' during the encryption).

- **Rounds 1 through 12**

  Each of these rounds is comprised of a non-linear substitution step, a transposition step, a mixing step, and a bitwise XOR of the round key to the state.

- **Final Round** (round 13)

  The final round is identical to Rounds 1 through 12 except the mixing step is omitted.

It should be noted that the key expansion only has to be performed once for any given secret key. Hence, the encryption loop is composed only of the Initial Round, the Rounds, and the Final Round as the round keys extracted during key expansion can be reused for whatever many iterations of the encryption loop it takes to encrypt the entirety of the plaintext.

SIMD

SIMD (Single Instruction, Multiple Data) is a technique employed to achieve data level parallelism. In SIMD computer architecture, the computer exploits multiple data streams against a single instruction stream in order to perform operations that may be easily parallelized [6].

SIMD processors provide a way to utilize data parallelism in applications that apply a single operation to all elements in a data set, such as a vector or matrix [7].

SSE

Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture.

AVX (Advanced Vector Extensions) is an advanced version of SSE, which will appear in Intel products in 2010, and which features a 256 bit data path (widened from 128 bits in SSE4). AVX will provide six new instructions for symmetric encryption/decryption using the Advance Encryption Standard (AES) and one instruction performing carry-less multiplication (PCMULQDQ) which aids in performing finite field arithmetic (a type of arithmetic used in advanced block cipher encryption). These hardware-based primitives provide a security benefit apart from their speed advantage by avoiding table-lookups and hence protecting against software side channel attacks (attempts to discover the secret key by observing and analyzing the flow of information in the computer during the encryption process). [9]

However, we do not need to program in assembly in order to utilise any of these extensions to the instruction set. Instead we can use intrinsics. Intrinsics are special functions for which the compiler generally has a specific optimisation path and which the compiler encodes to one or more machine instructions. They represent a good middle ground between speed of execution and ease of use for the programmer. With modern optimising compilers, particularly with the Intel C++ Compiler, we can get nearly as good results as the best hand tuned assembly code, without compromising code readability or having to bother about register management.

OpenMP

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on many architectures, including Linux, Windows and Mac OS X. OpenMP gives programmers a simple model and interface for developing parallel applications [10].

Optimization

Optimization is the process of tuning the output of a compiler to minimize or maximize some attribute of an executable computer program. The most common goal is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied. Optimization is usually applied by the compiler, which will usually attempt to generate the fastest possible code. However, a programmer will sometimes attempt to help the compiler out by performing some optimizations manually as (s)he, in theory, knows the algorithms better.

The general themes of optimizing programs are:

- **Optimize the common case**
- **Avoid redundancy**
- **Less code**
- **Straight line code**
- **Locality**
- **Manage memory efficiently**
- **Parallelize**

Code Generators

A code generator is not a new idea to the world of research, nor is the work that code generators do which is in fact analogous to the work a programmer does when optimising a piece of code by hand for a fixed machine. Code generators take a piece of code and try and find the fastest version by trying combinations of different

techniques to optimize the code, just like a programmer would by hand. The difference is that a code generator does not have to make guesses like a programmer would to save time. A code generator can try all the conceivable combinations of the optimization techniques to find the best, without any manual intervention whatsoever. It is this lack of manual intervention that is so convenient and labour saving. The work of optimizing even a small piece of code by hand can take weeks by hand. A code generator can generate all the combinations possible almost instantly and requires no attention during the time consuming measuring of run times and selection of the fastest one. Hence, in exchange for a single, one time expense, a large time saving in the cost of running the code on a daily basis can be achieved. [2]

However, compilers do not make use of this technique for a couple of reasons. Firstly, because the number of versions of a program can become astronomically large, even when only a few transformations are considered. Secondly, it is often found that important performance improvements can only be achieved by transformations that require knowledge of the algorithm that cannot easily be extracted from the source code, even with modern compilers with algorithm recognition techniques [1]. Hence, domain specific code generators that generate variants of code for a specific problem are used instead of the integration of this optimization process into the compiler itself.

FFTW ("Fastest Fourier Transform in the West") is an excellent example. It is a library for computing the discrete Fourier transform and its various cases. It is composed of two parts. To compute a discrete Fourier transform, the user first invokes the FFTW planner (the code generator), specifying the problem to be solved, the "shape" of the input data, but not the data itself. The planner returns a plan, executable code that accepts the input data and computes the desired Fourier transform. This plan is the fastest for the user's machine and the user can then execute the plan as many times as desired. [2]

Another example is Spiral ("Signal Processing Implementation Research for Adaptable Libraries"), which is a code generator for the domain of numerical problems, namely digital signal processing algorithms. It was the first to demonstrate the power of machine learning techniques in automatic algorithm selection and optimization. It uses a formal framework to generate many alternative algorithms for a given transform and translate them into code. It then finds the one that is best tuned in that platform. [1]

Both of these projects have been very successful and showcase how powerful code generators can be. This project attempts to bring this solution to the domain of AES encryption in order to use it to find the fastest, best combination of variants to use on new architectures that support AES encryption with SIMD intrinsics.

3. Uniqueness of my approach

The uniqueness of my approach was in being the first to bring technique of using a domain specific code generator to the problem of AES encryption code optimization. The AES Code Generator is discussed here under a number of sections:

I. Correctness: An AES-256 implementation
II. The Generator
III. Simulating
IV. Testing

I. Correctness: An AES-256 implementation

The correctness of the input and output is confirmed by means of a customised AES-256 implementation.

The implementation described in this report emulates AES-256 and was based upon a byte-oriented portable C implementation in which all the lookup tables had been replaced with "on-the-fly" calculations [12]. The implementation is fully compliant with the specification and is highly portable. There was no assembler in the original code but I later wrapped the code with vector functions (SSE functions) and vectorised some, but not all, of the code. This was done so that the function signatures (i.e. the names and parameters of the functions) would be compatible with those of the AES instructions that Intel will introduce as part of AVX (the Advanced Vector Extensions, see Background, chapter 2).

Since the purpose of including the AES code is to check correctness, the slow speed of this implementation is not a problem. It is provided as a simple means of verifying that the implemented variants do not change the basic algorithm of the AES encryption loop that is the input to the generator.

II. The Generator

The generator system generates a large number of simple variations of a basic AES encryption loop. These various modifications of the code can then be run on a particular model of processor, and with various compiler switches, to find the best variant for that particular processor.

The generator was tested with two block cipher modes of operation of 256 bit AES: Electronic Code Book (ECB) and Counter (CTR). The generator can take an input file of an encryption loop of either mode of operation, electronic codebook (ECB) or counter (CTR), as its input file. These two types of block cipher mode

of operation were chosen because they were the easiest to parallelise. There are a number of different options available for generating the variants. Each of these options, is, in essence, a distinct optimization, or set of possible configurations of optimizations, that can be performed. The options are:

- Streaming store
- Unwind inner loop
- Use local variables
- Unwind outer loop
- Interleave
- OpenMP (parallel)
- Prefetch to cache
- Prefetch to register

In the following sections, these options are described in more detail.

A. Steaming store

In the Steaming Store option, we use an SSE instruction instead of storing the result to memory using a standard memory assignment. This variant uses the _mm_stream_si128 instruction to store the result directly to memory without polluting the caches. As with all the options to the generator, specifying it will generate the variants with the option enabled and disabled.

Before:

```
result = encrypt_final(result, *keys);
result = _mm_xor_si128(result,source[i]);
dest[i] = result;
```

After:

```
result = encrypt_final(result, *keys);
result = _mm_xor_si128(result,source[i]);
_mm_stream_si128(&(dest[i]),result);
```

Normally, when we write to memory, the cache is updated with the contents of the write so that if there is a read request for that information soon after the write, it can be recalled quickly. However, in this situation the information that is being written is the cipher text that has just been encrypted, and we want to keep the cache for memory that we are going to be accessing again such as round keys and the plain text to be encrypted.

However, this is an option that can be turned on or off as it can happen that the use of these instructions can interfere with the compiler's own optimizations. The generator can therefore experiment with both versions in combination with lots of other options.

B. Unwind inner loop

This variant unwinds the inner loop to the extent specified in the argument.

Loop unrolling is a technique that attempts to increase the execution speed of the program at the expense of its size. The loop is rewritten as a sequence of independent statements, hence reducing (in this case, eliminating) the overhead of evaluating the loop condition on each of the iterations and reducing the number of jumps and conditional branches that need to be executed.

There are two side effects of loop unrolling. These are an increased register usage in a single iteration to store temporary variables (but not in this case, as we are completely eliminating the loop rather than just unwinding it a little), and the code size expansion after the unrolling. Large code size can lead to an increase in instruction cache misses.

In this case, the loop is quite short (equal to the number of keys, 14) and hence a significant speed boost should be observed due to the removal of the control variable check, the 14 jumps and the conditional branches from the control flow.

Before:

```
for (int j = 1; j < nKeys; j++ ) {
result = encrypt_round(result, *(keys+j));
    }
```

After:

```
result = encrypt_round(result, *(keys+1));
result = encrypt_round(result, *(keys+2));
result = encrypt_round(result, *(keys+3));
...
result = encrypt_round(result, *(keys+12));
result = encrypt_round(result, *(keys+13));
```

Loop unrolling can also aid the compiler and the processor perform their own optimizations, such as instruction scheduling.

C. Use local variables

This variant uses local variables for the AES round keys instead of memory accesses (the round keys are sub-keys used for the individual rounds extracted from the cipher key using the Rijndael key schedule). This involves defining the variables, assigning them the round keys from their memory locations and updating all references in the input file to the memory location to refer to the variables instead. The idea is that assigning the round keys to variables should be a massive hint to the compiler to keep the round keys in registers rather than doing a memory access (it is faster to access registers than the L1 cache where the round keys will probably be).

It is also observable that the number of round keys that are stored in registers can have a negative impact on run time. This is due the impact that storing them in registers has on the number of registers available for other purposes, such as storing temporary variables such as results. This is particularly relevant when a high level of unwinding of the outer loop has also occurred, especially when it has been interleaved as well.

As such, up to 2^{14} different variants of the number of round keys stored in local variables as opposed to using memory accesses can be generated. There are 2^{14} different variants as there are 14 round keys that could be stored in local variables and hence 2^{14} different permutations of round keys in local variables and memory accesses. However, for testing purposes, only 14 permutations formed by loading successive round keys into local variables were looked at.

Before:

```
for ( i = 0; i < limit; i++ ) {
        ...
        result = encrypt_round(result, *(keys+1));
  result = encrypt_round(result, *(keys+2));
        result = encrypt_round(result, *(keys+3));
...
  result = encrypt_round(result, *(keys+12));
   result = encrypt_round(result, *(keys+13));
```

After:

```
        const vector_type key0 = keys[0];
        const vector_type key1 = keys[1];
        const vector_type key2 = keys[2];
                const vector_type key3 = keys[3];
        ...
                const vector_type key12 = keys[12];
        const vector_type key13 = keys[13];
        for ( i = 0; i < limit; i++ ) {
           ...
           result = encrypt_round(result, (key1));
    result = encrypt_round(result, (key2));
    result = encrypt_round(result, (key3));
    ...
    result = encrypt_round(result, (key12));
    result = encrypt_round(result, (key13));
```

Using local variables for the round keys also has the bonus that it allows the compiler to apply other optimizations much earlier in the process than if the round keys are left as elements of an array as it can be

difficult for the compiler to prove that the array is unaliased (i.e. does not reference the same location or variables as another pointer).

D. Unwind outer loop

This variant unwinds the outer loop to the extent specified in the argument. The technique, and the side effects, is the same as with unwinding the inner loop, except the effect is much greater as a result of the greater number of instructions involved.

There is a certain threshold where the returns from the unwinding begin to rapidly diminish and then the effect of the limited number of registers and the instruction cycle misses hits and run time increases again.

Before:

```
for ( i = 0; i < limit; i++ ) {
vector_type result;
result = _mm_add_epi64(
nonce,_mm_set_epi32(0,0,0,i)
);

// initial round
result = encrypt_initial(result, enckey);
// encryption
result = encrypt_round(result, (key1));
result = encrypt_round(result, (key2));
...
result = encrypt_round(result, (key12));
result = encrypt_round(result, (key13));
// final round
result = encrypt_final(result, key0);
result = _mm_xor_si128(result,source[i]);

dest[i] = result;
} // end outer loop
```

After:

```
for ( i = 0; i < (limit-2); i+=2) {
vector_type result;

// iteration 0
result = _mm_add_epi64(
nonce,_mm_set_epi32(0,0,0,i)
);
// initial round
result = encrypt_initial(result, enckey);
// encryption
result = encrypt_round(result, (key1));
result = encrypt_round(result, (key2));
        ...
result = encrypt_round(result, (key12));
result = encrypt_round(result, (key13));
// final round
result = encrypt_final(result, key0);
result = _mm_xor_si128(result,source[i]);

dest[i] = result;
// end of original outer loop


// iteration 1
result = _mm_add_epi64(
nonce,_mm_set_epi32(0,0,0,i+1)
);

// initial round
result = encrypt_initial(result, enckey);
// encryption
result = encrypt_round(result, (key1));
result = encrypt_round(result, (key2));
            ...
result = encrypt_round(result, (key12));
result = encrypt_round(result, (key13));
// final round
result = encrypt_final(result, key0);
result = _mm_xor_si128(result,source[i]);

dest[i+1] = result;
// end of original outer loop

} // end unrolled loop
```

E. Interleave

This variant interleaves an unwound outer loop to the extent specified in the argument. The idea is that by interleaving the loop, we reduce the number of instructions that are stalling due to a reliance on the result of a previous instruction.

Since each iteration is dealing with a different intermediate result, and since each iteration is dealing with its own intermediate result a number of times as it works through the encryption rounds, it makes sense the a number of operations of the same round, rather than the same iteration, after one another. Hence, the delay that would exist in an iteration between the first and the second round while waiting for the result from the first is filled by calculating the result of the first round of the next iteration.

Also, it is potentially very important for performance that the keys are used multiple times in sequence so that it is strictly necessary to load them just the once for each sequence of multiple instructions using that key.

```
for ( i = 0; i < (limit-2); i+=2) {
vector_type result;

result = _mm_add_epi64(
nonce,_mm_set_epi32(0,0,0,i)
);
// initial round
result = encrypt_initial(result, enckey);
// encryption
result = encrypt_round(result, (key1));
result = encrypt_round(result, (key2));
      ...
result = encrypt_round(result, (key12));
result = encrypt_round(result, (key13));
// final round
result = encrypt_final(result, key0);
result = _mm_xor_si128(result,source[i]);
dest[i] = result;
// end of original outer loop
result = _mm_add_epi64(nonce,_mm_set_epi32(0,0,0,i+1));

// initial round
result = encrypt_initial(result, enckey);
// encryption
result = encrypt_round(result, (key1));
result = encrypt_round(result, (key2));
      ...
result = encrypt_round(result, (key12));
result = encrypt_round(result, (key13));
// final round
result = encrypt_final(result, key0);

result = _mm_xor_si128(result,source[i]);
dest[i+1] = result;
// end of original outer loop
} // end unrolled loop
```

```
for ( i = 0; i < (limit-2); i+=2) {
vector_type result0;
vector_type result1;

result0 = _mm_add_epi64(
nonce,_mm_set_epi32(0,0,0,i)
);
result1 = _mm_add_epi64(
nonce,_mm_set_epi32(0,0,0,i+1)
);

// initial round
result0 = encrypt_initial(result0, enckey);
result1 = encrypt_initial(result1, enckey);
// encryption
result0 = encrypt_round(result0, (key1));
result1 = encrypt_round(result1, (key1));
result0 = encrypt_round(result0, (key2));
result1 = encrypt_round(result1, (key2));
result0 = encrypt_round(result0, (key3));
result1 = encrypt_round(result1, (key3));
...
result0 = encrypt_round(result0, (key11));
result1 = encrypt_round(result1, (key11));
result0 = encrypt_round(result0, (key12));
result1 = encrypt_round(result1, (key12));
result0 = encrypt_round(result0, (key13));
result1 = encrypt_round(result1, (key13));
// final round
result0 = encrypt_final(result0, key0);
result1 = encrypt_final(result1, key0);
result0 = _mm_xor_si128(result0,source[i]);
result1 = _mm_xor_si128(result1,source[i+1]);

dest[i] = result0;
dest[i+1] = result1;
// end of original outer loop
} // end unrolled loop
```

F. OpenMP

This variant includes the OpenMP pragma directive that are normally commented out for the other variants. This allows for the investigation of parallel versions of the code, for running on processors with multiple cores and/or

simultaneous multithreading.

Any of the other variants can be generated with the OpenMP variant to create both threaded and non-threaded versions of the same code to investigate which is the most efficient of the options.

Before:

```
for ( i = 0; i < (limit-2); i+=2) {
vector_type result0;
vector_type result1;
vector_type src0;
vector_type src1;
```

After:

```
#pragma omp parallel for
for ( i = 0; i < (limit-2); i+=2) {
vector_type result0;
vector_type result1;
vector_type src0;
vector_type src1;
```

G. Prefetch to cache

This variant uses the _mm_prefetch SSE intrinsic to prefetch source data to the cache. The instruction loads one cache line of data from the given address to a location closer to the processor. The idea is to prime the cache for the next iteration(s) of the encryption loop.

However, a balance must be found to ensure that priming the cache too far ahead does not poison the cache (i.e. if we prefetch too many lines into the cache, or prefetch the lines too soon, it may cause lines in the cache that we still need to be removed). Hence, the generator must generate a large number of variants in order to find a version that prefetches the source data just enough instructions ahead.

Also, the number of iterations before its use that source data is prefetched can have an impact on the speedup gained from the optimization so a number of variants must be produced in order to find the one with the greatest speed up.

Example:

```
_mm_prefetch((char const*)&source[i+2],_MM_HINT_T0);
```

H. Preload to register

This variant uses a variable to preload source data to a register. The idea is to touch the L1 cache so it is primed with the line from which the source data value is chosen for the register for the next iteration(s) of the encryption loop. This is important as the prefetch SSE instruction only brings things into the L2 cache.

However, there are only a limited number of registers that are available for use. Hence, the generator must generate a large number of variants in order to find the version that will lead to an improvement in runtimes.

Example:

```
const vector_type src2 = source[i+2];
```

Most Intel microprocessors are also capable of doing prefetching in hardware, so sometimes attempts at prefetching have little, no or even a negative effect, if the hardware prefetcher does a better job.

III Simulation

The new architecture that Intel will release in 2010 introduces six Intel SSE instructions that facilitate encryption. Four instructions, namely AESENC, AESENCLAST, AESDEC, and AESDELAST facilitate high performance AES encryption and decryption. The other two, namely AESIMC and AESKEYGENASSIST, support the AES key expansion procedure. Together, these instructions will provide a full hardware to support AES, offering security, high performance, and a great deal of flexibility. [13]

In order to simulate the performance effect of the actual AES instructions without the actual hardware that supports them, the generator supports replacing the AES encryption instructions in the input code with instructions that are of various different latencies (provided in #defines in the input code) that we can use as proxies for the purposes of testing the effect of optimizations. This allows the testing of the code with various different latencies to give an idea of the effect of the speedups implemented by the variants. This is done only with the four instructions that facilitate high performance AES encryption and decryption. The two instructions supporting AES key expansion are not emulated since, as key expansion is only done once no matter the amount of plaintext to be encrypted, there is little to no benefit in studying speedups of it especially as it is relatively quick.

For the purposes of this project, instructions of latencies two, three and five cycles respectively were chosen for comparison, using the available Intel documentation [12]. Intel documentation published to date [13] indicates that the initial chips supporting AES encryption in hardware will do so in six cycles. As such, the figures for the effect of the various optimizations on five cycle latency instructions are the most relevant for the first generation of hardware supporting the instructions (given that there are not currently any six or seven latency instructions in

the SSE instruction set). The figures for two and three latencies, however, represent what we can expect from the second or third generation of hardware.

IV Testing

In order to test my code and get my experimental results, I ran my generated code under a number of compiler flags and architectures. Specifically, both the GNU C Compiler and Intel C Compiler, 32 and 64 bit multiprocessors and machines with only a single core, dual cores and 8 cores.

In order to generate the detailed results I got for my experimental results (see the next section), I used PapiEx. PapiEx is a performance analysis tool designed to transparently and passively measure the hardware performance counters of an application using PAPI [15]. It uses Monitor to effortlessly intercept process/thread creation/destruction. The Performance API (PAPI) project specifies a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. [14] Monitor is a library that gives the user callbacks or traps upon events related to library/process/thread initialization/creation/destruction. [17]

Using PapiEx, I was able to get accurate information on the number of instructions issued, executed and completed, the amount of data cache misses and the number of stall cycles used, as well as the total number of cycles execution took.

At the time of writing, the Intel C Compiler did not yet support the compilation of AES instructions, however it should be trivial to test the correctness of the output using the Intel Software Development Emulator [18].

5. My Results and Contributions

AES Code Generator is a valuable means to find the most efficient and optimized AES code for any given architecture. In general for the architectures surveyed here, the interleaved, parallel variants seem to be the most efficient. This is most likely due to the use of all of the available cores by using multiple threads and to the reduction in loads to registers from caches caused by interleaving the outer loop of the encryption loop. The number of local variables used for storing round keys seemed to be consistently low on 32-bit architectures and medium on 64-bit architectures, reflecting the increased number of registers available on the Intel x86_64 64-bit architecture (16 instead of only 8 – as discussed in SSE section of chapter 2, ‘Background’).

To conclude, this report will outline what contributions this project has made to the current state of the art and briefly discuss what future work could be attempted that could build upon the progress achieved in this project.

Contributions

This project has made a number of contributions to the state of the art.

Firstly, it is one of the very first to deal with the new AES instruction set which will appear in the next generation of Intel processors.

Secondly, it provides a proof of the concept of using a code generator to provide all the various optimized variants of the standard AES encryption loop.

Thirdly, it extends the idea of self-tuning generators to the area of encryption code generation

References

[1] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo, "SPIRAL: Code Generation for DSP Transforms", Proceedings of the IEEE special issue on "Program Generation, Optimization, and Adaptation," Vol. 93, No. 2, 2005, pp. 232-275.

[2] Matteo Frigo and Steven G. Johnson, "The Design and Implementation of FFTW3," Proceedings of the IEEE 93 (2), 216–231 (2005). Invited paper, Special Issue on Program Generation, Optimization, and Platform Adaptation.

[3] Joan Daemen , Vincent Rijmen, "The Block Cipher Rijndael", Proceedings of the The International Conference on Smart Card Research and Applications, p.277-284, September 14-16, 1998.

[4] National Institute for Standards and Technology, "Announcing the Advanced Encryption Standard (AES)", Federal Information Processing Publication #197, 2001.

[5] Michael Flynn, "Some Computer Organizations and Their Effectiveness", IEEE Trans. Comput., Vol. C-21, pp. 948, 1972.

[6] Aart J.C. Bik, "Vectorization with the Intel Compilers", Intel, 2008.

[7] R.M. Ramanthan, "Extending the World's Most Popular Architecture", Intel, 2006.

[8] Nadeem Firasta, "Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency", Intel, 2008.

[9] Barbara Chapman, "Using OpenMP: Portable Shared Memory Parallel Programming", The MIT Press, 2007.

[10] Unknown, “Compiler optimization”, http://en.wikipedia.org/wiki/Compiler_optimization (last accessed on 3rd April 2009)

[11] Ilya O. Levin, “A byte-orientated AES-256 implementation”, http://www.literatecode.com/2007/11/11/aes256/, (last accessed on 4th April 2009).

[12] Shay Gueron, Intel Mobility Group Israel, “AES Instructions Set White Paper”, Intel, July 2008.

[13] Intel, “Intel 64 and IA-32 Architectures Optimization Reference Manual”, November 2007.

[14] P.Mucci, “PapiEx - Execute arbitrary application and measure hardware performance counters with PAPI”, http://icl.cs.utk.edu/~mucci/papiex/ (last accessed on 10th April 2009).

[15] P. Mucci et al, “A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters”, Proceedings of Supercomputing 2000, 2000.

[16] P. Mucci and N. Tallent, “Monitor - user callbacks for library, process and thread initialization/creation/destruction”, 2004.

[17] Mark Charney, “Intel Software Development Emulator”, Intel, http://www.intel.com/software/sde/ (last accessed on 14th April 2009).

[18] Robert Konighofer, “A Fast and Cache-Timing Resistant Implementation of the AES”, Proceedings of the Cryptographer’s Track at RSA Conference 2008, 2008.

[19] Intel, “Intel 64 and IA-32 Architectures Software Developers Manual”, November 2007.

[20] Guido Bertoni et al, “Efficient Software Implementation of AES on 32-Bit Platforms”, Proceedings of Cryptographic Hardware and Embedded Systems 2002: p159-171, 2002.