Masaryk University
Faculty of Informatics

Argon2 function and hardware platform optimization for OpenSSL

Bachelor's Thesis

Čestmír Kalina

Brno, Fall 2019

Replace this page with a copy of the official signed thesis assignment and a copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Čestmír Kalina

Advisor: Ing. Milan Brož, Ph.D.

Abstract

Memory-hard password hashing function Argon2 has been adopted by applications and libraries alike, but it was as yet missing in OpenSSL library. Some of Argon2 users use OpenSSL. To remedy the need for an extra dependency or the need to maintain separate implementation maintenance, Argon2 is introduced into OpenSSL. As it is designed to be executed in parallel, threading support is also proposed for OpenSSL. Parts of Argon2 code are then optimized further for ARMv8.0-A architecture, in particular for the 64-bit Aarch64 execution state. This includes optimization of memory copying, preloading and optimization of the internal compression function. Performance oriented optimizations are then benchmarked on 5 different ARMv8.0-A machines, ranging from development boards to servers.

Keywords

Argon2, hash function, aarch64, ARM, optimization

Contents

1 Introduction

The problem of low key entropy is a common one in cryptographic applications. In an attempt to reduce the cost of an exhaustive search, a computationally expensive derivation of an actual key is usually performed. And while increasing computational complexity achieves the goal on a level playing field,[1] it is not sufficient to eliminate the advantage of an attacker using parallel hardware (e.g., using *Application Specific Integrated Circuit*). Schneier et al. observed [26] that, besides key-stretching, using moderately large amounts of RAM would serve to further increase the search cost. To remedy the said disparity, a class of *memory hard functions* was introduced. *Argon2* [8] is a particular family of memory hard functions that won the Password Hashing Competition of 2013 [1].

Argon2 family of functions has been adopted by applications (*cryptsetup* [10]), programming languages (*Haskell* [12], *PHP* [41]) and libraries (*libsodium* [42]), but it was as yet missing in *OpenSSL* [38] library – one which many of the current Argon2 users already do use (and, whenever Argon2 support is required, bundle or link-in an external Argon2 library [10]). To remedy the need for an extra dependency, Argon2 is introduced into OpenSSL [25].

The thesis opens with a brief description of both memory hard functions and optimization-relevant internals of the Argon2 family. The next chapter focuses on OpenSSL: it discusses necessary preliminary work, such as threading or signal masking introduction into OpenSSL, as well as an architecture-independent port of Argon2 into OpenSSL. This port serves as a basis for ARM-specific optimizations.

With the family of Argon2 functions present in OpenSSL, the text specializes to *ARMv8* architecture [32]. First, ARMv8 architecture fundamentals are recollected, followed by two chapters dealing with optimization. Optimization techniques are presented with examples of use.

The thesis then concludes with benchmarks of generic and optimized code. This is by nature hardware specific – to minimize the impact that any one hardware's quirks might contribute into the over-

1. Compare `https://en.bitcoin.it/wiki/Mining_hardware_comparison` and `https://en.bitcoin.it/wiki/Non-specialized_hardware_comparison`.

all collected data, measurements were made on a wide variety of hardware, ranging from low to middle end development boards to production servers.

2 Memory Hard Functions

This chapter begins with the introduction of the class of memory hard functions [40] and its two important sub-classes, followed by a brief description of the Argon2 [8] family. This description focuses primarily on aspects most relevant to optimization performed. For a complete description of Argon2, the reader is referred to [8].

2.1 Memory Hardness

As stated in the introduction, using moderately large amounts of RAM to increase the search cost was considered as a viable way to make the use of specialized hardware disadvantageous. This led Percival [40] to consider parametrizing key derivation algorithms by not just time, but space cost as well:

Memory-hard algorithm [40] An algorithm $\mathcal{A}$ on a Random Access Machine is said to be memory-hard if it uses $S(n)$ space and $T(n)$ operations, where $S(n) \in \Omega(T(n)^{1-\epsilon})$.

In other words, a memory-hard algorithm asymptotically uses almost as many memory locations as operations. In his treatment of memory-hard functions, Percival [40] adds the comment that "a widely used rule of thumb in high performance computing is that balanced systems should have one MB of RAM for every million floating-point operations per second of CPU performance" to illustrate feasibility of this approach.

Depending on whether or not a memory-hard algorithm $\mathcal{A}$ performs memory accesses dependently or independently of its input, we say that $\mathcal{A}$ is data dependent or independent, respectively.

A memory-hard function is specified via a memory-hard algorithm which evaluates it.

There are multiple memory-hard functions currently in use today; among others: Balloon password hashing function [9], scrypt [40] and Argon2, the winner of a Password Hashing Competition [1].

2.2 The Family of Argon2 Functions

Argon2 [8] is a family of memory hard functions. Differences between members of the Argon2 family range from the intended use to whether they use data-dependent or independent accesses. The reader is referred to Argon2 IETF RFC draft[1] [22]. It is important to recognize the Blake2b [7] function that Argon2 is based on.

Generally speaking, each Argon2 variant has two types of inputs:

1. primary: message P and nonce S
2. secondary: degree of parallelism p, memory size m, tag length τ, number of iterations t, version number v, secret value K, associated data X, Argon2 type y

Argon2 uses [8] internal compression function G (based on Blake2b's internal permutation) with two 1024-byte inputs and a single 1024-byte output and an internal hash function (Blake2b hash function is used).

2.2.1 Permutation $\mathcal{P}$

Blake2b's integral part is the so called round function – a transformation on 512 byte (Blake-256) or 1024 byte (Blake-512) words. Argon2 uses the same principle, with one notable difference: multiplication is performed as well as addition and bit-wise operations, with the motivation of increasing circuit depth (and thus the running time) of any ASIC implementation.

Permutation $\mathcal{P}$, as defined in Argon2 [8], operates on 8 16-byte inputs $S_0, \ldots, S_7$. It is instructive to split $S_0, \ldots, S_7$ into 16 64-bit words v_i:

$$S_i = v_{2i+1}v_{2i}, \text{ where } \|v_{2i}\| = \|v_{2i+1}\|$$

and view them as a 4×4 matrix W of 4 rows of 4 words of the form

$$W = (r_i), r_i = (v_{4i} \quad v_{4i+1} \quad v_{4i+2} \quad v_{4i+3}).$$

1. The RFC draft neared completion at the time of writing this thesis.

We can then express, using the notation that $c_i(W)$ corresponds to i-th column of W, diag(W) a diagonal of a matrix W, and $W_{\ggg i}$ is a matrix W circularly shifted to the left by i columns, a single round of $\mathcal{P}$ may be expressed as:

$$\begin{matrix} G'(c_1(W)) & G'(c_2(W) & G'(c_3(W)) & G'(c_4(W)) \\ G'(\text{diag}(W)) & G'(\text{diag}(W_{\lll 1})) & G'(\text{diag}(W_{\lll 2})) & G'(\text{diag}(W_{\lll 3})). \end{matrix}$$

where G' applied to (a, b, c, d) results in (imperatively stated):

$$\begin{aligned} G'(a, b, c, d) : \\ & a \leftarrow a + b + 2a_L b_L \\ & d \leftarrow (d \oplus a) \ggg 32 \\ & c \leftarrow c + d + 2c_L d_L \\ & b \leftarrow (b \oplus c) \ggg 24 \\ & a \leftarrow a + b + 2a_L b_L \\ & d \leftarrow (d \oplus a) \ggg 16 \\ & c \leftarrow c + d + 2c_L d_L \\ & b \leftarrow (b \oplus c) \ggg 63 \\ & \text{return}(a, b, c, d) \end{aligned}$$

The operation $+$ is an operation of addition mod 2^{64} and $\ggg$ is a 64-bit rotation to the right. For any 64-bit integer, the $\cdot_L$ operator truncates the upper 32 most significant bits. Long multiplication (64-bit operands) is assumed. The operation $\oplus$ indicates the XOR operation.

When no parallelism is used, G is iterated m times; see Figure 2.1.

2.2.2 Compression function $G(X, Y)$

Now to the integral part of Argon2 memory hardness: the compression function $G(X, Y)$. It operates on two 1024-byte blocks X and Y and can be expressed as:

1. compute $R = X \oplus Y$
2. view R as a 8×8-matrix of 16-byte registers $R_0, \ldots, R_{63}$
3. apply $\mathcal{P}$ row-wise and then column-wise to get Z (also an 8×8 matrix of 16-byte registers)

$$\begin{aligned}
(Q_0, \ldots, Q_7) &\leftarrow \mathcal{P}(R_0, \ldots, R_7) \\
&\ldots \\
(Q_{56}, \ldots, Q_{63}) &\leftarrow \mathcal{P}(R_{56}, \ldots, R_{63}) \\
(Z_0, Z_8, Z_{16}, \ldots, Z_{56}) &\leftarrow \mathcal{P}(Q_0, Q_8, Q_{16}, \ldots, R_7) \\
&\ldots \\
(Z_7, Z_{15}, Z_{23}, \ldots, Z_{63}) &\leftarrow \mathcal{P}(Q_7, Q_{15}, Q_{23}, \ldots, R_{63})
\end{aligned}$$

4. G outputs $Z \oplus R$

2.2.3 Variable-Length Hash Function H'

Let H_x be a Blake2b hash function with x-byte output, π be a projection of an 64-byte block to its least significant 32 bytes, $\tau < 2^{32}$ the 32-bit little-endian expressed tag length in bytes. If $\tau \leq 64$, define $H'(X) := H_\tau(\tau \| X)$, otherwise:

$$\begin{aligned}
V_1 &\leftarrow H_{64}(\tau \| X) \\
V_2 &\leftarrow H_{64}(V_1) \\
&\ldots \\
V_r &\leftarrow H_{64}(V_{r-1}) \\
V_{r+1} &\leftarrow H_{\tau - 32r}(V_r) \\
H'(X) &:= \pi(V_1) \| \ldots \| \pi(V_r) \| V_{r+1}
\end{aligned}$$

where $r = \lceil \frac{\tau}{32} \rceil - 2$.

2.2.4 Operation

Argon2 follows [8] the extract-then-expand concept: extract entropy from message and nonce by hashing it:

$$H_0 := H(p, \tau, m, t, v, y, \|P\|, P, \|S\|, S, \|K\|, K, \|X\|, X),$$

After that it fills the memory with m 1024-byte blocks. These blocks are organized into a matrix $B[i][j]$ of blocks with p rows (lanes) and $q = \lfloor \frac{m}{p} \rfloor$ columns. Blocks are computed as follows:

$$B^1[i][0] = H'(H_0, \underbrace{0}_{\text{4 bytes}} \| \underbrace{i}_{\text{4 bytes}})$$

$$B^1[i][1] = H'(H_0, \underbrace{1}_{\text{4 bytes}} \| \underbrace{0}_{\text{4 bytes}})$$

$$B^1[i][j] = G(B[i][j-1], B[i'][j'])$$

where $0 \leq i < p, 2 \leq j < q$ and i' and j' are determined differently across Argon2 versions. At subsequent iterations we repeat the procedure in a similar manner:

$$B^t[i][0] = G(B^{t-1}[i][q-1], B[i'][j'] \oplus B^{t-1}[i][0])$$
$$B^t[i][j] = G(B^t[i][j-1], B[i'][j'] \oplus B^{t-1}[i][j])$$

and at the final iteration, at time T, we compute the final block as XOR of the last column:

$$B_m = \bigoplus_{k=0}^{p-1} B^T[k][q-1].$$

The output tag is produced by applying H' on B_m.

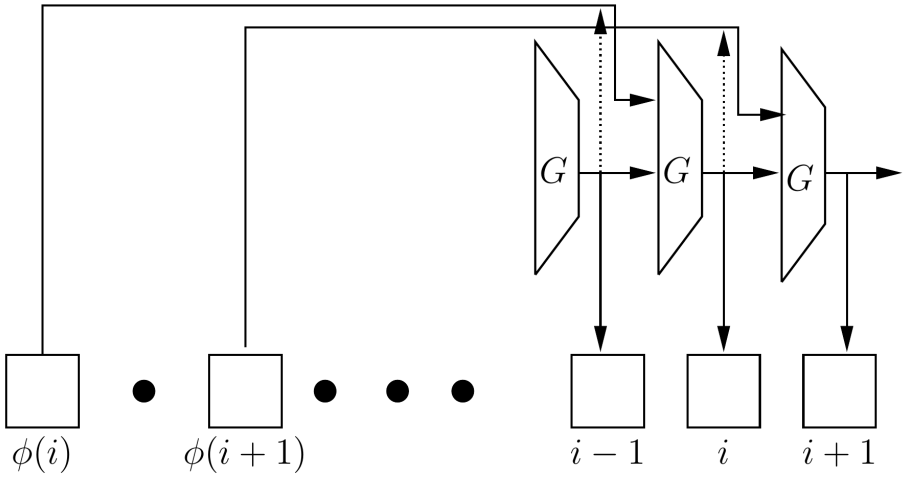

Figure 2.1: Argon2 mode of operation with no parallelism. [8]

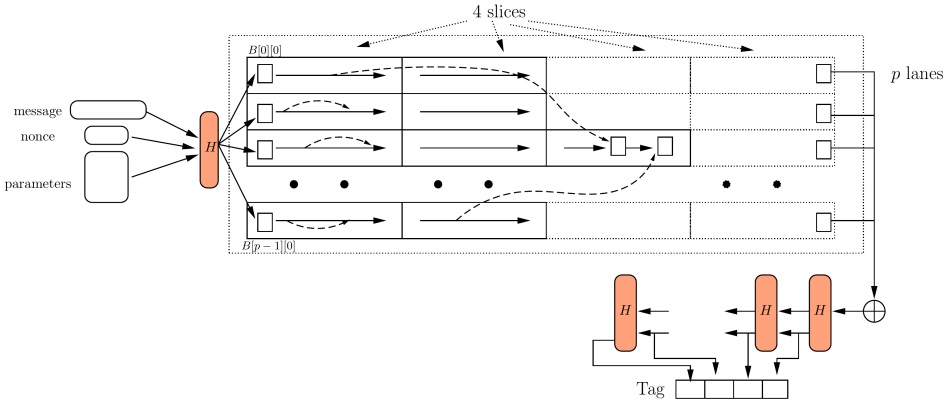

Figure 2.2: Single-pass Argon2 with p lanes and 4 slices. [8]

3 OpenSSL Integration

OpenSSL [38] is a general-purpose multi-platform cryptography library. This chapter discusses porting [25] of Argon2 into OpenSSL, as well as the necessary preliminary work.

3.1 Threading Dependencies

Introduction of threading to OpenSSL required architecture-agnostic wrappers for working with threads, mutexes, condition variables and signals (POSIX/WinAPI).

Remark: Threading support is proposed only for POSIX or Windows based systems. While POSIX systems are not restricted in any way, provided they do support signals and POSIX threads, the latter is required to satisfy the requirement:

`_WIN32_WINNT >= 0x0600`.

This guarantees that condition variable API [14] used in the Windows-specific code is available. Windows implementation makes use of both `CreateThread` [11] (processthreadsapi.h) and `_beginthread` [13] (C Runtime Library).

The Linux list [45] design pattern was adopted to OpenSSL.

3.2 Threading Support

Prior to the Argon2 port, OpenSSL supported cooperative multitasking (fibres) only [34]. As Argon2 was designed to be executed in parallel, support for preemptive multitasking (threading) was required, if OpenSSL's version of Argon2 were to have any real-world users.

Based on discussions with OpenSSL developers [24], two approaches to threading were proposed, discussed and implemented for pthread and Windows based systems.

Internal Threading OpenSSL creates threads independently, without restriction or supervision by the user application.

External Threading Summary of the implementation:

- OpenSSL library users provide worker thread to OpenSSL library to use, using OpenSSL-provided API.
- As a worker thread is created and goes to sleep; relying on a condition variable to wake it up.
- OpenSSL library users may provide callback to worker thread that determines worker lifespan. After a job is finished, user application is queried (via callback, if any) whether worker ought to terminate or not, given the current number of awaiting tasks.
- OpenSSL maintains a list of busy/idle worker threads provided to OpenSSL by the library user.
- An attempt to create a thread from inside OpenSSL merely files away task metadata (address, data) into a task queue. The task does not start until there is a worker available. No `pthread_create`, `_beginthreadex` or `CreateThread` or equivalent function is called!

After woken up[1] by job addition, worker thread attempts to pop the task queue (if not empty) and to execute the corresponding task. If the architecture supports fibres, the actual task is executed as a fibre on top of worker thread, allowing for future extensions to the design. After the executed task finishes, and a worker queries the user-provided callback whether it ought to service another task or terminate. Note that deadlocks may occur if this kind of threading is used and in-OpenSSL users rely on higher number of workers than currently present.[2]

Conceptually, threading implementation may be viewed as two distinct APIs, depending on the target user:

1. Barring potential spurious wake ups.
2. For example, a sole worker provided to OpenSSL, executing task T_1 that, at some point during execution, awaits for a lock released by T_2, which never gets executed due to resource constraints.

- OpenSSL library users, the goal here is to
  - give explicit permission to use threading,
  - disable threading,
  - provide callbacks executed upon signal receive,
  - provide workers for external threads.
- Internal OpenSSL users, the goal here is to:
  - provide implementation-agnostic wrappers for creating and joining threads,
  - facilitate signal masking/callback management.

3.3 Argon2 KDF Support

A **provider**, in OpenSSL terms, is a unit of code that provides one or more implementations for various operations for diverse algorithms that one might want to perform. [35]

Argon2 [8] relies on Blake2b [7], which is provided by OpenSSL already using providers – both as message digest and MAC. [37, 36] After a slight modification to Blake2b MAC in OpenSSL, in-Argon2 usage of Blake2b amounts to using the OpenSSL API. Most notably, this is required to implement the variable-length hashing function H' from the previous chapter.

Adding a new KDF into OpenSSL (in this case Argon2) amounts to:

- creating new Argon2 providers
- interfacing new providers, for example:
  `providers/default/defltprov.c`
  `providers/common/include/internal/provider_algs.h`
- implementing the Argon2 in `/crypto` (or alternatively directly in providers: `/providers/implementations/`),
- adding test vector(s) to `test/evp_kdf_test.c` (in the case of KDF)

- setting up the build system by filing appropriate:
  `/Configure`
  `/providers/default/kdfs/build.info`
  any relevant `build.info` in your directory tree
- new function definitions or metadata (SN, LN, NID):
  `apps/list.c`
  `apps/progs.c`
  `apps/progs.pl`
  `include/crypto/evp.h`
  `crypto/objects/objects.txt`
  `crypto/objects/obj_mac.num`
  `crypto/objects/obj_dat.h`
- adding extra KDF parameter types, definitions, or error types
  `include/openssl/kdf.h`
  `include/openssl/core_names.h`
  `include/openssl/evperr.h`

As this area is still in flux and is the most likely aspect to be outdated, it will not be discussed further.

4 The ARM Architecture(s)

ARM is not a single architecture, rather a family of architectures, one that is release and purpose (e.g., realtime, application) versioned. Only the latest application-purpose ARM, `ARMv8-A`, is used. This chapter outlines some of the fundamentals of `ARMv8-A` required for the optimization.

Remark: All material from this section about ARMv8 comes from the Arm®Architecture Reference Manual, Armv8, for Armv8-A architecture profile [32], unless stated otherwise.

Note: This text will not limit itself to any particular big.LITTLE setting, it will not assume availability of multiple NUMA nodes, support of `Aarch64` SVE or SVE2 extensions, or Statistical Profiling Extension. It will focus on ARMv8.0-A only.

4.1 Execution States

`ARMv8-A` supports two execution states: 32-bit and 64-bit state. An n-bit execution state is denoted `Aarch`n[1] and utilizes n-bit wide general purpose (GP) registers. Call *n word size*.

Execution occurs at one of four exception/privilege levels denoted from $EL0$ to $EL3$, and $PL0$ to $PL3$. In this thesis, no exception/privilege level restrictions are made and $EL0/PL0$ (unprivileged, used for user application execution) is assumed unless stated otherwise.

When in `Aarch64` execution state, the CPU executes `A64` instruction set; when in `Aarch32`, `A32` or `T32` (also called `ARM` or `Thumb`, respectively). Both `A64/A32` instruction sets have fixed instruction lengths of 32 bits, 16 bits in case of `T32`.

1. `Aarch32` aims to be backwards compatible with `ARMv7`.

4.2 Registers

In Aarch64 state, there are 31 GP registers in total (X0-X30). They are all readily available at all exception levels, IRQs, FIQs, etc. In Aarch32 there are only 15 GP registers available.[2]
The GP registers are used in function calls and can be dividided up into 4 groups:

X0-X7	Argument registers. Pass parameters to a function and to return a result. Can be used as: scratch registers, intermediate storage
X9-X15	Caller-saved temporary registers. If the caller requires these to be preserved across function calls, caller has to save in its own stack frame; can be modified by the called subroutine without the need to restore state
X19-X29	Callee-saved temporary registers. They may be modified as long as they get restored prior to returning
Others	Special purpose registers.

In addition to the GP registers, ARMv8-A also has 32 128-bit-wide registers: denoted V0-V31. These registers double as floating-point as well as SIMD registers: Advanced SIMD (NEON) share the same register file as the floating-point register file. Vn, Dn, and Sn are used to refer to the lower 128-bit, 64-bit, or 32-bit range, respectively.

Processor state is kept, depending on execution state, in:
Aarch32: $CPSR$ register (Current Program Status Register)
Aarch64: a collection of fields called $PSTATE$ (processor state)

4.3 Conditional Execution

ARMv8-A supports conditional execution of instructions based on ALU status flags in $CPSR/PSTATE$, notably of N, Z, C, V[3] flags – making

2. This is due to backwards-compatibility reasons. ARMv7 had only half the number of GP registers available during normal execution; some of which were banked. There is no register banking in Aarch64, so the remaining registers (not available in ARMv7) serve as banked copies in Aarch32 execution state.
3. N negative, Z zero, C carry, V overflow

the execution of an instruction dependent on the processor state, executing it or replacing it with a *NOP*. Add suffix S to an instruction to make it set ALU status flags in the *CPSR/PSTATE*.

```
1 ADD    c, x, y ; c := x + y, do not update flags
2 ADDS   c, x, y ; c := x + y, update flags
3 ADDEQS c, x, y ; if Z is set, then c := x + y
```

Listing 4.1: Conditional Execution Example

The aim is to reduce the number of branch instructions, since they are costly in terms of code density and processor cycles. The number of instructions that can be conditionally executed is significantly smaller in `A64`. It is important to note that while the instruction itself will not be executed when the condition is not met, it will still be fetched from memory. As a general rule of the thumb, a branch is faster than about 3 consecutive conditional instructions.

4.4 SIMD/Vector Extensions

ARMv8-A comes with multiple vector processing capabilities implemented as instruction set extensions. Implementation of these extensions is *not mandatory*.[4] Notably, this includes the NEON, SVE and SVE2 extensions.

SIMD instructions execute in their own separate pipeline(s) and SIMD-capable processors implement a new set of registers (they are not aliased to general purpose registers). That makes interleaving ARM and SIMD workloads highly suitable for optimization purposes. That said, SVE registers do extend NEON registers, so interleaving cannot be used in the same way across different SIMD extensions.

Note that while ARM to SIMD transfer is fast, the converse is not necessarily true – beware of pipeline stalls (e.g., if one modifies cacheline the other uses).

Promotion/demotion of types is supported by ARMv8 SIMD extensions: compound operations which combine type promotion with arithmetic operations are provided.

4. NEON is required in `Aarch64` execution state, but not generally.

4.5 The Barrel Shifter

ARM comes with a dedicated circuit capable of performing bit shifts and rotations prior to any arithmetic performed. One may use the barrel shifter with some instructions to apply bit shifts/rotations to a single operand of at little extra cost. The barrel shifter's use has been significantly reduced in `A64`.

4.6 Memory

Memory accesses are performed using via load and store instructions exclusively. `Aarch64` execution state supports 64-bit virtual addresses (VA), `Aarch32` 32-bit virtual address. Since instructions are fixed-size, it is impossible to directly load a 32-bit immediate constant into a register, and thus *PC* relative addressing is used. For these reasons it is recommended that stack storage be utilized to make use of stack-pointer-relative addressing. Data accesses may be little-endian or big-endian.[5] ARM recognizes two types of memory: Normal and Device:

Device Memory may be used for all memory regions where an access might have a side-effect (e.g., read to a FIFO location or a timer). Most importantly, speculative data accesses cannot be performed on Device memory.[6] Depending page table attributes, multiple accesses may (not) be merged (so called *gathering*, G/nG), re-ordered (R, nR), and writes early acknowledged (E, nE). For more information about these flags, see the reference manual.

Normal Memory is not sequentially consistent and employs a weakly-ordered model of memory[7]:

- the order of memory accesses to normal memory need not correspond to program order,
- the processor is able to re-order read and write operations,

5. See *SCTLR_ELn.EE* and *SCTLR_EL*1.*E*0*E* for more information.
6. With the exception of NEON operations during which the processor might read bytes not explicitly referenced, provided they are within 16-byte aligned block of explicitly referenced data.
7. Much weaker than a TSO model [23, 39, 43] found in SPARC or x86[15].

- a store by a thread may propagate to other threads in any order,
- multiple writes to different addresses may be interleaved arbitrarily.

To enforce memory order, memory barriers, fences or address dependencies must be used.[8]

4.7 Caches

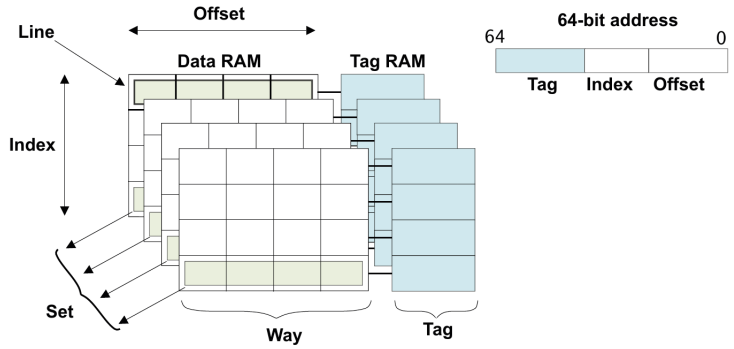

Figure 4.1: Cache Terminology [30]

ARMv8-A usually comes with separate L1 caches for instructions and data and a common L2 cache.

Cacheline (Figure 4.1) is an n-word chunk of contiguous data from memory. Each cacheline has an associated valid and dirty bit. A **cache miss** occurs as a result of a read/write operation on a noncached VMA.

To invalidate/clean cache (to PoU/PoC; or by set/way),

`DC <operation>{, <Xt>}` and `IC <operation>{, <Xt>}`

instructions may be used for data or instruction caches, respectively.

8. ARMv8-A supports instruction synchronization barriers (ISB), data memory barriers (DMB), data synchronization barriers (DSB) and one-way barriers.

4.8 Performance Counters

Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mis-predicted. They have a low impact on performance and form a basis for profiling applications to trace dynamic control flow and identify hot-spots.

4.9 Pipeline

For the purposes of optimization, it is important to mention how instructions themselves are processed by the processor: each instruction goes through the following (and more) stages: fetch, decode, execute, write-back. This process can be parallelized: as one instruction is being fetched, another decoded and so forth.

That said, concrete ARMv8 execution models vary across implementations; to avoid too general a discourse, in this section, we will talk about ARM Cortex®-A72. In the Cortex®-A72 instruction processing pipeline (see Figure 4.2) instructions are first fetched, then decoded into internal micro-operations (μops). From there, the μops proceed through register renaming and dispatch stages. Once dispatched, μops wait for their operands and issue out-of-order to one of eight execution pipelines. Each execution pipeline can accept and complete one μop per cycle. [31]
It is important to note that since ARMV8 is a pipelined system, it is subject to occurrence of structural and data hazards, pipeline interlocking and forwarding. In particular, optimization should take the following data dependencies into account (consider Listing 4.2):

```
1 ADD R1, R2, R3
2 SUB R6, R7, R1
3 MUL R1, R3, R5
```

Listing 4.2: Demonstration of data dependencies

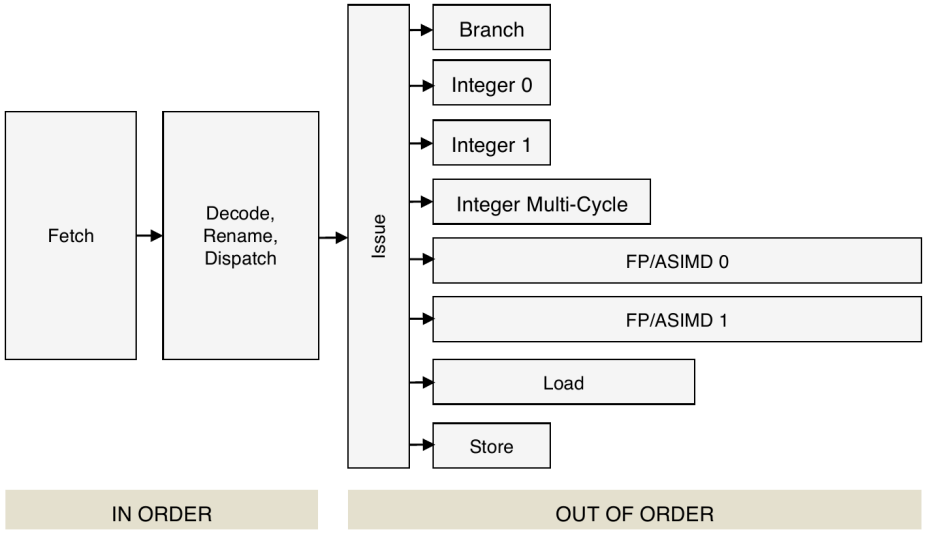

Figure 4.2: ARM Cortex®A72 Pipeline Overview[31]

RAW: Read After Write *lines 1-2, R1*: Result of the first instruction is used as source operand in the second – which is forced to wait for the completion of 1st instruction.

WAR: Write After Read (Anti-Dependency) *lines 2-3, R1*: Result of the third instruction depends on the second instruction's input.

WAW: Write After Write (Output Dependency) *lines 1 and 3, R1*

5 Optimization: Software Aspects

This chapter focuses on software aspects of optimization that assist in optimization from a software perspective: this may be recording and analyzing profiles, sampling performance data, or compiler specific topics that aid in code generation.

Disclaimer: This section contains architecture and operating system specific topics. Any restriction(s) will be denoted at the beginning of a section. Restrictions, if present, are not intended to be maximal or cover the breath of software that may support a given feature.

5.1 Indirect Functions

As previously mentioned, ARM comes with many ISA extensions – some of which are crucial to optimization. This is making the distribution of optimized code (especially with binary distributions) an important topic.

Restriction: An operating system using Executable and Linkable Format (ELF), using `binutils >= 2.20.1` and `glibc >= 2.11.1`.

There is a GNU extension to ELF format adding a new function type `STT_GNU_IFUNC` and new dynamic relocation `R_*_IRELATIVE`. [21] This is the (GNU) indirect functions – a mechanism postponing a selection of a particular function version to link time. A function call is resolved via PLT; PLT entry is changed to point to the desired version of the function, either at load time or at the first call.

```
static void fn_neon(void) { ... }
static void fn_arm(void)  { ... }
asm (".type fn, %gnu_indirect_function");
void * fn(unsigned long int hwcap) {
        if (hwcap & HWCAP_ARM_NEON)
                return &fn_neon;
        // ...
}
```

5.2 "Compiler-Friendly" C Language Constructs

Pointer Aliasing may impede compiler optimizations such as auto vectorization. Use the `restrict` keyword to hint no aliasing to compiler. In some cases (strict) aliasing may be circumvented using unions.

```
void foo(double *restrict r, double *restrict r2);

union {
        uint8_t  buf[1024];
        sometype data;
};
```

Pure functions [17] are functions that have no side-effects and whose return value depends only on the value of its arguments. Multiple calls to a pure function with the same arguments yield the same result. A compiler can eliminate common sub-expressions that contain pure function calls or move out loop invariant code outside of pure functions. An even stronger form of a pure function is a const function that may only use the arguments passed in and not any memory.

```
int foo (char *) __attribute__ ((pure));
```

Expect (with probability) [19] wrappers offer hints to compiler how to generate branches:

```
long __builtin_expect (long exp, long c)
long __builtin_expect_with_probability(
        long exp, long c, double probability
)

/* The following is adopted from the Linux kernel. */
#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)
```

You may use these functions to provide branch prediction information to the compiler.[1] If present, compiler may reorder branches to utilize

1. Note that compiler has usually its own set of heuristics that simulates assertions of this kind, such as: branch ending with returning a constant is probably not taken, branch from comparison using != is probably taken, == is probably not taken; branch to a basic block calling a cold function is probably not taken. For up to date data, see definitions for the branch prediction routines in the GNU compiler [20].

CPU pipeline better – the goal here is to minimize the amount of pipeline trashing that would otherwise have occurred.

They are especially good for error handling code. When used, a compiler can avoid size-increasing optimizations of an unlikely branch. It can even move the unlikely code to another section, causing it not to be loaded in an instruction cache, saving cache space and possibly cache trashing.

It is useful to use the gcc option `-fprofile-arcs` to collect the information rather than guessing code behaviour. An alternative to this is to use a multiple-staged compilation process with `-fprofile-generate` followed by `-fprofile-use`. Since this would be a major intrusion to OpenSSL build process, this approach is not adopted.
Finally, gcc option `-fdump-tree-profile_estimate` may also be used.

5.3 Software Profiling

5.3.1 perf

The Linux perf (perf_events) tool [2] provides rich generalized abstractions over hardware specific capabilities. Among others, per task, CPU and workload counters, sampling on top of these and source code event annotation. It can instrument CPU performance counters, tracepoints, kprobes, and uprobes. Tracepoints can be enabled by the perf command to collect information including timestamps and stack traces. It can record profiles and display statistics, down to an instruction level.

5.3.2 valgrind

Valgrind [6] is an instrumentation framework for building dynamic analysis tools. Valgrind tools can automatically detect many memory management and threading bugs, and profile your programs in detail.

5.3.3 pahole

Pahole [4] utilizes DWARF, CTF or BTF debug data to show data structure layout. In particular, it can display data structure field offsets,

holes created due to alignment and cacheline use (with respect to current CPU). It may produce the following output:

```
struct ossl_cmp_ctx_st {
        OSSL_cmp_log_cb_t    log_cb;          /*  0     8 */
        OSSL_CMP_severity    log_verbosity;   /*  8     4 */
        /* XXX 4 bytes hole, try to pack */
        ...
        char *               proxyName;       /* 56     8 */
        /* --- cacheline 1 boundary (64 bytes) --- */
```

It is important to note that while pahole output is useful in structure layout optimization, it is highly architecture specific. This remains true when optimizing to a specific architecture: cacheline boundary is the most obvious architecture-specific data that varies throughout different implementations, but it is not the only one: 32bit and 64bit environments influence the layout as well as LP64 vs LLP64.

5.3.4 pfunct

Pfunct uses debug data to display information regarding functions in the object code. It is capable of showing function sizes, functions declared inline but not inlined[2] or the converse.

5.4 Random Call-Stack Sampling

This diagnostic aims at providing a single piece of data: given multiple stack samples taken during program execution, are there any commonalities and if so, how frequent are they? Such common lines of code hint at possible optimization target.

2. Candidates for `__attribute__((always_inline))` or `__forceinline`.

6 Optimization Specific Aspects of ARMv8-A

This chapter opens up with a briefer on using assembly from within the C language, followed by ARMv8-specific optimizaiton aspects relevant to Argon2 [8].

6.1 Writing Assembly in C

6.1.1 The `asm` keyword

Architecture specific code may be written as inline assembly:

```
asm(
        code
        [: output_operand_list
        [: input_operand_list
        [: clobber_list]]]
);
```

For example:

```
int i, j, res;
asm(
        "ADD %w[result], %w[i1], %w[i2]"
        : [result] "=r" (res)
        : [i1] "r" (i), [i2] "r" (j)
);
```

Function calls between C and assembly code must follow the AAPCS64 [33] rules. For more information, see gcc docs [18]. Note that the compiler is free to reorder separate asm blocks.

6.1.2 NEON Intrinsics

NEON [28] intrinsic functions [29, 16] are similar to assembly programming in that they are substituted for specific instructions. They are safer to use as they leverage the type system: for example, `int16x4_t` is a vector containing four lanes each containing a signed 16-bit integer. Some intrinsics use an array of vector types of the form:
`<type><size>x<number of lanes>x<length of array>_t`.

When intrinsics are used, the compiler takes care of register allocation, function calling conventions, not to mention utilizing further compiler optimizations (e.g., instruction reordering, common expression elimination).

NEON intrinsics map closely to NEON instructions. [28] Each intrinsic has the form: `<opname>[q]_<type>` The optional q flag specifies that the intrinsic operates on 128-bit vectors. Note that mapping of intrinsics to instructions is not onto.

6.2 Copying Memory

Argon2 relies heavily on copying chunks of memory. In this section, we will review the steps we can take to optimize it. After a brief discussion of design aspects: alignment, prefetching, and blocking vs. non-blocking mode of operation, handful of select approaches are studied.

6.2.1 Alignment

Even though unaligned accesses are possible in ARMv8, the architecture generally performes better on word-aligned memory accessess. Coarser accesses allow the use of load multiple (LDM) to load multiple registers in a cycle (for example, Cortex A72 LDM has a throughput of $\lfloor \frac{\#\text{registers}+1}{2} \rfloor^{-1}$ [31]).

Memory copy in Argon2 is usually performed over 1K blocks (128 64-bit words) and thus need not be changed to conform to these requirements.

6.2.2 Prefetch

Prefetch from memory ($PRFM$) enables code to provide a hint to the MMU that data from a particular address with be used, soon. Effect of $PRFM$ is implementation defined. [32] A32 counterpart is PLD and PLI. The instruction syntax (A64) is:

```
PRFM <prfop>, <addr> | label
```

where `prfop` is a concatation of the following options:

```
Type     PLD or PST (prefetch for load or store)
Target   L1, L2, or L3 (target cache)
Policy   KEEP (keep in cache) or STRM (streaming data)
```

6.2.3 Load/Store Throughput

It is not uncommon for ARMv8 processors to have a separate load and store pipelines (e.g., ARM Cortex®-A72[31]). That means that it my execute one load μop and one store μop every cycle. Using discrete non-writeback forms of load and store instructions, interleaving them in so that a load is immediately followed by a store can improve speed.

6.2.4 Non-Blocking vs Blocking

Our goal is for memory copying to be non-blocking and to utilize the CPU pipeline during their execution in some other way. Using a *preload engine* (PLE) would allow us to fills ways of L2 cache while concurrently performing other tasks until receiving an interrupt informing us of operation completion. In the case of Argon2, however, this would not provide us with substantial speedup since the data region being copied is used in parts or in full by subsequent code and PLE use would not eliminate the bottleneck, while substantially increasing code complexity. Therefore, the rest of the section deals with blocking memory copy only.

6.2.5 Implementation

This subsection contains an overview of implemented memory copy routines. Both `Aarch32` and `Aarch64` routines were introduced, however, benchmarks were performed exclusively in `Aarch64` mode due to testing limitations. Since their implementation could not be covered by benchmarks, they were moved to Appendix A.1.[1]

All implementations were compared versus a base case, which is not optimized on purpose.

1. With the exception of Load-Multiple case that remained as a curiosity. It doesn't take up additional space.

Base Case Copy 64-bits at the time. Used as a reference.

```
1  .global memcpy_w32
2
3  memcpy_w32:
4      ldr  r3, [r1]
5      str  r3, [r0]
6      subs r2, r2, #4
7      add  r0, r0, #4
8      add  r1, r1, #4
9      bgt  memcpy_w32
10     ret
```

Load-Multiple [A32 only], Load-Pair [A64 only] In A32 one may load data into/store data from multiple registers in a single instruction (LDM/STM; Load/Store Multiple). A64 since reduced that only to a pair of registers (LPD/STP; Load/Store Pair).

The A32 variant loads contents from r1 (base address) into registers r3 through r10, incrementing address after each transfer and writing it back to r1 (similarly for store). The limited number of registers (compared to Aarch64) requires one to save up registers on stack prior to use, so there is a setup overhead. The A64 variant is straightforward.

```
1  .global memcpy_lm32
2
3  memcpy_lm32:
4      push  {r4-r10}
5
6  loop:
7      ldmia r1!, {r3 - r10}
8      stmia r0!, {r3 - r10}
9      subs  r2, r2, #32
10     bgt   loop
11     pop   {r4-r10}
12     ret
```

```
.global memcpy_lp64

memcpy_lp64:
    ldp  x3, x4, [x1]
    stp  x3, x4, [x0]
    subs x2, x2, #16
    add  x0, x0, #16
    add  x1, x1, #16
    bgt  memcpy_lp64
    ret
```

Interleaving Load/Store Since most ARM implementations (such as ARM Cortex®-A72 [31]) utilize separate load and store pipelines, interleaving Load (Pair) and Store (Pair) instructions may better utilize the CPU pipeline. The actual implementation is highly dependent the target hardware, e.g., the depth of its pipeline, instruction and data chacheline sizes.

```
1  .global memcpy_lp64_i
2
3  memcpy_lp64_i:
4      subs x2, x2, #128
5      ldp  x3, x4, [x1,#0]
6      stp  x3, x4, [x0,#0]
7      ldp  x3, x4, [x1,#16]
8      stp  x3, x4, [x0,#16]
9      ldp  x3, x4, [x1,#32]
10     stp  x3, x4, [x0,#32]
11     ldp  x3, x4, [x1,#48]
12     stp  x3, x4, [x0,#48]
13     ldp  x3, x4, [x1,#64]
14     stp  x3, x4, [x0,#64]
15     ldp  x3, x4, [x1,#80]
16     stp  x3, x4, [x0,#80]
17     ldp  x3, x4, [x1,#96]
18     stp  x3, x4, [x0,#96]
19     ldp  x3, x4, [x1,#112]
20     stp  x3, x4, [x0,#112]
21     add  x1, x1, #128
22     add  x0, x0, #128
23     bgt  memcpy_lp64_i
24     ret
```

NEON NEON approach's notable benefit is that NEON instructions do not use any of the generic purpose registers.

```
1  .global memcpy_neon64
2
3  memcpy_neon64:
4      ld1 { v0.2d, v1.2d, v2.2d, v3.2d }, [x1]
5      add x1, x1, #64
6      ld1 { v4.2d, v5.2d, v6.2d, v7.2d }, [x1]
7      add x1, x1, #64
8      st1 { v0.2d, v1.2d, v2.2d, v3.2d }, [x0]
9      add x0, x0, #64
10     st1 { v4.2d, v5.2d, v6.2d, v7.2d }, [x0]
11     add x0, x0, #64
12     subs x2, x2, #128
13     bgt memcpy_neon64
14     ret
```

NEON with Pre-Fetch NEON approach's may further be enhanced with a preload. Note that preload use, cache, load/store and the offset is highly dependent on the targeted class of devices.

```
1  .global memcpy_neon64_pld
2
3  memcpy_neon64_pld:
4      prfm pstl1strm, [x1, #0xC0]
5      ld1 { v0.2d, v1.2d, v2.2d, v3.2d }, [x1]
6      add x1, x1, #64
7      ld1 { v4.2d, v5.2d, v6.2d, v7.2d }, [x1]
8      add x1, x1, #64
9      st1 { v0.2d, v1.2d, v2.2d, v3.2d }, [x0]
10     add x0, x0, #64
11     st1 { v4.2d, v5.2d, v6.2d, v7.2d }, [x0]
12     add x0, x0, #64
13     subs x2, x2, #128
14     bgt memcpy_neon64_pld
15     ret
```

Mixed ARM and NEON with Preload This is an interleaved version of ARM and NEON instructions. Note that preload use, cache, load/store and the offset is highly dependent on the targeted class of devices.

```
1  .global memcpy_mixed64
2
3  memcpy_mixed64:
4      ldp  x3, x4, [x1]
5      stp  x3, x4, [x0]
6
7      prfm pstl1strm, [x1, #32]
8
9      ldp  x5, x6, [x1,#16]
10     stp  x5, x6, [x0,#16]
11
12     prfm pstl1strm, [x1, #96]
13
14     add x1, x1, #32
15     ld1  { v0.2d, v1.2d, v2.2d, v3.2d }, [x1]
16
17     ldp  x3, x4, [x1,#64]
18     stp  x3, x4, [x0,#96]
19
20     ldp  x5, x6, [x1,#80]
21     stp  x5, x6, [x0,#112]
22
23     add x0, x0, #32
24     st1  { v0.2d, v1.2d, v2.2d, v3.2d }, [x0]
25
26     add x0, x0, #96
27     add x1, x1, #96
28
29     subs x2, x2, 128
30
31     bgt memcpy_mixed64
32     ret
```

6.3 Optimization of G, G' and $\mathcal{P}$

Recall that G is the compression function on two 1024-byte blocks. G applies a permutation $\mathcal{P}$ on parts of its XOR-ed inputs. Permutation $\mathcal{P}$ operates on (128-byte) input, applying the round function G' on parts of its input. The round function G' operates on four 64-bit words. For more, see Section 2.2 or the original Argon2 paper [8]. This section deals with optimization aspects of $\mathcal{P}$ and therefore, barring preload optimizations, directly optimizes the compression function G.

6.3.1 Scalar Implementation

Using the Barrel Shifter Optimizes G'. As previously stated, ARM instructions often support the use of a barrel shifter. We can avoid a standalone rotation instruction if we replace, for example,

$$d \leftarrow (d \oplus a) \ggg 32 \qquad \text{with} \qquad d \leftarrow (d \oplus a)$$

and any subsequent use of d replaced by the value d rotated using the barrel shifter, e.g.,

```
1 ADD R0, R1, R2          ~> ADD R0, R1, R2, ROR #32
```

Load Multiple/Store Multiple Optimizes $\mathcal{P}$. The number of `Aarch64` registers as well as the available load pair/store pair operations make this yet another viable optimization way. The advantage of this approach is not only the relatively small number of load/store pair operations required, but also the speed in which the results may be propagated elsewhere (whereas with NEON, this is notoriously slow).

6.3.2 ASIMD/NEON Implementation

Optimizes $\mathcal{P}$. Recall that input data can be visualized as a 4 by 4 matrix W and calls:

$$\begin{array}{cccc} G'(c_1(W)) & G'(c_2(W) & G'(c_3(W)) & G'(c_4(W)) \\ G'(\mathrm{diag}(W)) & G'(\mathrm{diag}(W_{\lll 1})) & G'(\mathrm{diag}(W_{\lll 2})) & G'(\mathrm{diag}(W_{\lll 3})). \end{array}$$

This means that the first 4 calls to G' may be realized independently of each other, since each of them transforms a distinct column of the matrix (column step). Similar principle applies to the diagonal calls. NEON implementation utilizes this fact by processing adjacent columns (or diagonals) in pairs, storing them together in 4 vector 128-bit registers (C intrinsic type `uint64x2_t`) – individual vectors is stored in a lower/upper half of their respective NEON registers. The following notable intrinsics were used in the implementation:

Assembly	Intrinsic	Description
`vsli.64`	`vsliq_n_u64`	2-way 64-bit left-shift and insert
`vsri.64`	`vsriq_n_u64`	2-way 64-bit right-shift and insert
`vadd.i64`	`vaddq_u64`	2-way 64-bit register addition
`vshl.i64`	`vshlq_n_u64`	2-way 64-bit left-shift by constant
`vmull.u32`	`vmull_u32`	2-way 32-bit long multiply
`vadd.i64`	`vmovn_u64`	2-way 64-bit register addition
`veor`	`veorq_u64`	2-way 64-bit XOR
`vext.64`	`vextq_u64`	2-way 64-bit vector extraction

```
#include "arm_neon.h"

#define __ARGON2_ROT(x,y) do {                          \
        vsliq_n_u64(vshrq_n_u64(x, y), x, 64-y);        \
    } while(0)

#define __ARGON2_FBLAMKA(a,b) do {                      \
        a = vaddq_u64(vaddq_u64(a,b), vshlq_n_u64(      \
              vmull_u32(vmovn_u64(a), vmovn_u64(b)),    \
              1));                                      \
    } while(0)

#define __ARGON2_P(a,b,c,d) do {                        \
        FBLAMKA(a,b);                                   \
        d = veorq_u64(d,a);                             \
        d = __ARGON2_ROT(d, 32);                        \
        FBLAMKA(c,d);                                   \
        b = veorq_u64(b,c);                             \
        b = __ARGON2_ROT(b, 24);                        \
        FBLAMKA(a,b);                                   \
        d = veorq_u64(d,a);                             \
        d = __ARGON2_ROT(d, 16);                        \
```

```
        FBLAMKA(c,d);                                   \
        b = veorq_u64(b,c);                             \
        b = __ARGON2_ROT(b, 63);                        \
    } while(0)

#define __ARGON2_G(chunk, base, offset) do {            \
        uint64x2_t i0, i1, i2, i3, i4, i5, i6, i7;      \
        i0 = vld1q_u64(&chunk[base + offset*0]);        \
        i1 = vld1q_u64(&chunk[base + offset*1]);        \
        i2 = vld1q_u64(&chunk[base + offset*2]);        \
        i3 = vld1q_u64(&chunk[base + offset*3]);        \
        i4 = vld1q_u64(&chunk[base + offset*4]);        \
        i5 = vld1q_u64(&chunk[base + offset*5]);        \
        i6 = vld1q_u64(&chunk[base + offset*6]);        \
        i7 = vld1q_u64(&chunk[base + offset*7]);        \
        __ARGON2_P(i0, i2, i4, i6);                     \
        __ARGON2_P(i1, i3, i5, i7);                     \
        i8 = vextq_u64(i3, i2, 1);                      \
        i9 = vextq_u64(i2, i3, 1);                      \
        i2 = vextq_u64(i6, i7, 1);                      \
        i3 = vextq_u64(i7, i6, 1);                      \
        __ARGON2_P(i0, i9, i5, i3);                     \
        __ARGON2_P(i1, i8, i4, i2);                     \
        i6 = vextq_u64(i3, i2, 1);                      \
        i7 = vextq_u64(i2, i3, 1);                      \
        i2 = vextq_u64(i8, i9, 1);                      \
        i3 = vextq_u64(i9, i8, 1);                      \
        vst1q_u64(&chunk[base + offset*0], i0);         \
        vst1q_u64(&chunk[base + offset*1], i1);         \
        vst1q_u64(&chunk[base + offset*2], i2);         \
        vst1q_u64(&chunk[base + offset*3], i3);         \
        vst1q_u64(&chunk[base + offset*4], i4);         \
        vst1q_u64(&chunk[base + offset*5], i5);         \
        vst1q_u64(&chunk[base + offset*6], i6);         \
        vst1q_u64(&chunk[base + offset*7], i7);         \
    } while(0)
```

The above approach calculates two calls to G concurrently and we may use it as:

```
- for (i = 0; i < 8; ++i)
- BLAKE2_ROUND_NOMSG(blockR.v[16 * i], blockR.v[16 * i + 1],
-                    ..., blockR.v[16 * i + 15]);
+ __ARGON2_G(blockR.v, 16*i, 2);
+ __ARGON2_G(blockR.v, 16*(i+1), 2);
...
```

```
+ __ARGON2_G(blockR.v, 16*(i+6), 2);
+ __ARGON2_G(blockR.v, 16*(i+7), 2);

- for (i = 0; i < 8; ++i)
- BLAKE2_ROUND_NOMSG(blockR.v[2 * i], blockR.v[2 * i + 1],
-         blockR.v[2 * i + 16], blockR.v[2 * i + 17],
-         ..., blockR.v[2 * i + 112], blockR.v[2 * i + 113]);
+ __ARGON2_G(blockR.v, 2*i, 16)
+ __ARGON2_G(blockR.v, 2*(i+1), 16)
+ __ARGON2_G(blockR.v, 2*(i+2), 16)
...
+ __ARGON2_G(blockR.v, 2*(i+6), 16)
+ __ARGON2_G(blockR.v, 2*(i+7), 16)
```

6.4 Other ISA relevant aspects

6.4.1 Non-Temporal Load and Store Pair

LDNP, *STNP* instructions perform non-temporal read or write of a pair of register values. [32] They also hint to the MMU that caching is not useful for this data[2].

Non-temporal loads and stores relax the memory ordering requirements and thus might require the use of an explicit load barrier. This optimization didn't prove useful.

6.4.2 Conditional Execution vs Speculation

Conditional-register writes in an out-of-order processor such as Cortex-A57 have two side-effects [31], consider:

```
1 MULEQ R1, R2, R3
2 MULNE R1, R2, R4
```

The second multiply is dependent upon the result of the first multiply, not through one of its normal input operands ($R2$ and $R4$), but through the destination register $R1$.

The combined latency for these instructions is six cycles, rather than the four cycles that would be required if these instructions were not conditional (3 cycles latency for the first, one additional cycle

2. There is nothing prohibiting caching of the address, preloading or gathering. It just indicates to the MMU that caching is unlikely to increase performance.

for the second which is fully pipelined behind the first). So if the condition is easily predictable (by the branch predictor), conditional execution can lead to a performance loss. But if the condition is not easily predictable, conditional execution may lead to a performance gain since the latency of a branch mispredict is generally higher than the execution latency of conditional instructions. That, with the significant decrease of conditional execution instructions made any use in `Aarch64` disadvantegous.

6.4.3 Load/Store Exclusive Pair

This is useful for spinlocks for example. *LDXR* loads a value from memory address and attempts to claim an exclusive lock on an address *STXR* then writes a new value to that location only if the lock was successfully obtained and held.

7 Benchmarks

This chapter includes a summary of some of code benchmarks performed. The following hardware[1] was used:

- Broadcom BCM2711: Raspberry Pi 4 Model B
- Armada 8040: MACCHIATObin Double Shot
- ThunderX cn8890: Penguin Computing Valkre 2040 Gigabyte Blade
- ThunderX2 cn9975: Cavium ThunderX2 Sabre
- HP Proliant M400
- Qualcomm Centriq 2452

7.1 Memory Copy

7.1.1 Preloading

Preloading and NEON To explore just how advantageous preloading is, several measurements were performed with preloading before, in between and after consecutive NEON blocks, with different settings. Preloading scenario excerpt (`memcpy_neon64_pld` from Section 6.2.5):

```
1  <- Prefetch was explored here [1]
2  ld1 { v0.2d, v1.2d, v2.2d, v3.2d }, [x1]
3  add x1, x1, #64
4  ld1 { v4.2d, v5.2d, v6.2d, v7.2d }, [x1]
5  add x1, x1, #64
6  <- Prefetch was explored here [2]
7  st1 { v0.2d, v1.2d, v2.2d, v3.2d }, [x0]
8  add x0, x0, #64
9  st1 { v4.2d, v5.2d, v6.2d, v7.2d }, [x0]
10 <- Prefetch was explored here [3]
11 add x0, x0, #64
12 subs x2, x2, #128
```

1. Sometimes, to save space, hardware will be referred to using a substring of its name in this section. Substring used will identify the hardware uniquely.

The following table shows the best and worst prefetch scenarios on cn8890 and illustrates that is hard to use portably and using bad prefetch offset may have detremental effect on performance.

Prefetch Scenario	Time [s]
After NEON blocks, L2 Cache, Keep Cached Prefetch for Load, Offset 384 bytes	0.255987
After NEON blocks, L1 Cache, Streaming Prefetch for Store, Offset 0 bytes	1.317233

For more complete measurement information, the reader is referred to sections Appendix A.4.1 and A.4.2.

Preloading and Interleaved ARM/NEON Similarly to the NEON implementation of memory copy, preloading was used in interleaved ARM and NEON code as well. Again, machines with deep pipelines such as cn8890 provided the biggest difference in measured data. The following pseudo-code (modified excerpt of `memcpy_mixed64` from Section 6.2.5) illustrates preloading locations used.

```
1     stp  x3, x4, [x0]
2 #ifdef PRFOP1
3     prfm PRFOP1, [x1, PRFOFF1]
4 #endif
5     ldp  x3, x4, [x1,#16]
6     stp  x3, x4, [x0,#16]
7     add x1, x1, #32
8     ld1  { v0.2d, v1.2d, v2.2d, v3.2d }, [x1]
9 #ifdef PRFOP2
10    prfm PRFOP2, [x1, PRFOFF2]
11 #endif
12    ldp  x3, x4, [x1,#64]
13    ...
14    st1  { v0.2d, v1.2d, v2.2d, v3.2d }, [x0]
15 #ifdef PRFOP3
16    prfm PRFOP3, [x1, PRFOFF3]
17 #endif
```

The following table shows the best and worst preloading scenario on cn8890; what is more, illustrates the benefit that prefetch might have:

PRFOP1 PRFOFF1	PRFOP2 PRFOFF2	PRFOP3 PRFOFF3	Time [s]
`pstl1keep` Offset 512	`pstl1keep` Offset 256	`pldl1keep` Offset 512	0.200185
N/A	N/A	N/A	1.267637

For more complete measurement information, the reader is referred to sections Appendix A.5.

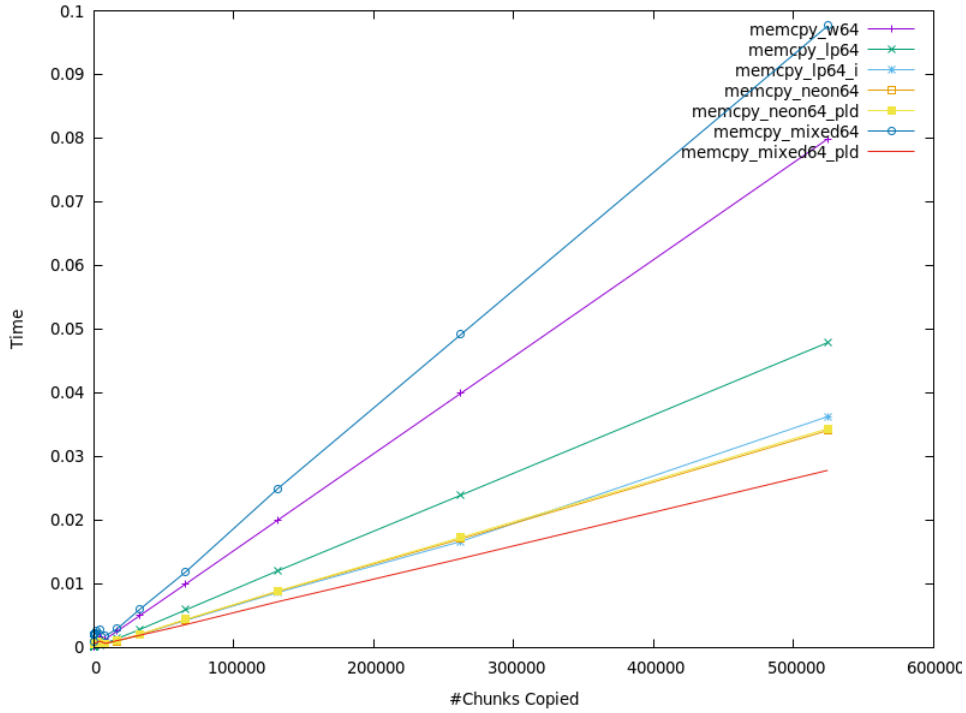

Figure 7.1: Comparison of Different Memory Copy Implementations Machine: Qualcomm Centriq 2452

7.1.2 Comparison of Proposed Methods

The following tables detail time-required for memory copy implementations of each of the proposed methods on two select machines. Measured data for these and other systems may be found graphed in Appendix A.3.

Measurement of memcpy implementations on cn8890:

Amount	LDP	LDP/STP Interleave	NEON	NEON with preload	ARM NEON Interleave	ARM NEON Interleave with preload
1	1.0	4.44	10.08	10.33	8.24	8.48
2	1.0	4.43	10.09	10.33	8.24	8.49
4	1.0	4.42	10.05	10.29	8.21	8.46
8	1.0	4.42	10.02	10.27	8.19	8.43
1024	1.0	2.15	4.57	4.66	3.75	3.65
2048	1.0	1.48	3.01	3.1	2.48	2.33
4096	1.0	1.3	1.99	2.07	1.94	1.43
8192	1.0	1.0	1.41	1.12	2.43	0.78
524288	1.0	0.38	0.46	0.25	0.42	0.16

Measurement of memcpy implementations on cn8890:

Amount	LDP	LDP/STP Interleave	NEON	NEON with preload	ARM NEON Interleave	ARM NEON Interleave with preload
1	1.0	3.27	3.28	3.28	10.77	3.27
2	1.0	3.28	3.28	3.28	10.77	1.8
4	1.0	3.28	3.28	3.28	10.77	3.28
8	1.0	3.26	3.26	3.26	10.71	3.33
1024	1.0	1.81	1.85	1.88	6.15	1.9
2048	1.0	1.34	1.11	1.43	4.59	1.46
4096	1.0	2.28	2.36	2.43	7.04	2.53
8192	1.0	0.81	0.83	0.84	2.56	0.88
524288	1.0	0.76	0.71	0.72	2.04	0.58

7.1.3 Comparison Permutation $\mathcal{P}$

This section compares time (in seconds) it took to complete 524288 iterations of $\mathcal{P}$ calculation. It compares vanilla C implementation with NEON and scalar implementations.

Machine	NEON Implementation	Scalar Implementation	Vanilla C
cn9975	0.634132	0.031966	0.303046
HP M400	0.760000	0.030000	0.280000
Armada 8040	0.829197	0.029125	0.332296
cn8890	0.809705	0.043538	0.518385
bcm2711	1.010660	0.037847	0.430943
Qualcomm 2452	0.457340	0.018164	0.399370

The NEON implementation is slower on all accounts. It is also the only implementation that didn't use assembly directly, rather used intrinsics and left the actual code generation up to a compiler. The actual code is very small and NEON loads and especially subsequent stores back are very costly. Also all the machines used in benchmarks are Cortex-A72, meaning 2 SIMD pipelines [31]; it is possible that pipeline stalls occured while computing the `fBlaMka` function.

The increased number of general purpose registers in `Aarch64` along with the load/store pair instructions allowed for the scalar implementation to perform better and without extensive stack utilization.

8 Conclusion

The thesis deals with the class of memory hard functions [40] and the family of Argon2 [8] functions, widely adopted winner of Password Hashing Competition [1], in particular.

The primary aim of the thesis was two-fold: to port the Argon2 family of functions to OpenSSL [38] and to research and implement architecture specific optimizations of the Argon2 family on the ARM architecture [32].

OpenSSL port of Argon2 family has been completed [25]; initially, a single-threaded Argon2 was integrated with MD, MAC and KDF interfaces of OpenSSL. After a brief discussion upstream [24], this has been reduced to KDF support only. Subsequent changes in OpenSSL required a rewrite and Argon2 integration has since adopted to the newly coined providers [35] interface.

While Argon2 may run in a single-threaded mode, any serious use mandates, due to performance reasons, the use of threads – after all, Argon2 was designed with the use of threading in mind. [8] Discounting fibre support [34], OpenSSL lacked threading support at the time; architecture-agnostic wrappers and support for threading and signal handling on pthread and Windows based platforms were introduced to OpenSSL. This addition was added to the Argon2 pull request [25].

With Argon2 and threading OpenSSL changes pending upstream, a way to further optimize Argon2 for ARMv8.0-A architecture [32] was sought. Through out the optimization process, gdb [3], radare2 [5], pahole [4], valgrind [6] and Linux's perf [2] were the tools used to determine bottlenecks in the code. Their use yielded candidates for architecture-based optimizations that deal with cache preloading, cache locality, vectorization, interleaving, and utilizing compiler hints, to name a few. A separate proof-of-concept was experimented on: using Device-nGnRnE, enforcing strong memory order and, among others, disallowing speculative accesses. Ways of incorporating optimized code into codebase were also explored.

Performance based optimizations were benchmarked. This included testing on 6 separate ARMv8.0 machines, ranging from low-cost embedded devices (Raspberry Pi 4 Model B), through development boards (MACCHIATObin Double Shot) to deep-pipelined servers

(Cavium ThunderX2 Sabre or Qualcomm Centriq 2452). A number of performance increasing changes were found, as well as approaches that failed and merit further research.

Future Work The first and immediate action is to resolve upstream OpenSSL discussion and merge the architecture-independent Argon2 implementation in-tree, as well as its dependencies: Blake2b changes, threading and signal handling among others. After that a separate merge request will be made to introduce the optimized version upstream.

Using SVE [32] rather than NEON could yield an extra performance increase and merits further research, especially if the optimization is targeted at machines with at least 256 bit SVE registers. A natural extension of this is to utilize Scalable Vector Extension version two (SVE2) ARM A-Profile technology that was early-disclosed this year [44].

Optimizing Blake2b for OpenSSL would increase the overall Argon2 performance as well. Furthermore, this would allow for a standalone Argon2 assembly implementation.

Using $LDXR/STXR$ instructions to implement architecture specific locking for OpenSSL or utilizing hardware locking elsewhere in the code is yet another possible optimization that would be interesting to explore.[1]

1. Given the number of times locking is used in Argon2 this is not crucial for Argon2.

Bibliography

[1] Password hashing competition. http://password-hashing.net/. [Online; accessed 10-October-2019].

[2] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2015. [Online; accessed 10-November-2019].

[3] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/, 2019. [Online; accessed 10-November-2019].

[4] pahole manpage. https://linux.die.net/man/1/pahole, 2019. [Online; accessed 10-November-2019].

[5] radare: Libre and Portable Reverse Engineering Framework. https://rada.re/n/, 2019. [Online; accessed 10-November-2019].

[6] Valgrind homepage. https://valgrind.org/, 2019. [Online; accessed 10-November-2019].

[7] J. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. *BLAKE2: Simpler, smaller, fast as MD5*, volume 7954 LNCS of *Lecture Notes in Computer Science* (*including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*). 2013.

[8] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S and P 2016*, pages 292–302, 2016.

[9] D. Boneh, H. Corrigan-Gibbs, and S. Schechter. *Balloon hashing: A memory-hard function providing provable protection against sequential attacks*, volume 10031 LNCS of *Lecture Notes in Computer Science* (*including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*). 2016.

[10] M. Brož. cryptsetup, Add Argon2 bundled library to crypto backend, commit 09d14a0b. https:

//gitlab.com/cryptsetup/cryptsetup/commit/09d14a0b6cb660af10eec9fcf55d61082334f7a3, 2017. [Online; accessed 10-November-2019].

[11] Windows Dev Center. CreateThread function. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread, 2019. [Online; accessed 10-November-2019].

[12] O. Charles and H.V. Riedel. argon2: Memory-hard password hash and proof-of-work function. https://hackage.haskell.org/package/argon2, 2018. [Online; accessed 10-November-2019].

[13] MicrosoftDocs Contributors. _beginthread, _beginthreadex. https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/beginthread-beginthreadex?view=vs-2019, 2018. [Online; accessed 10-November-2019].

[14] MicrosoftDocs Contributors. Condition Variables. https://docs.microsoft.com/en-us/windows/win32/sync/condition-variables, 2018. [Online; accessed 10-November-2019].

[15] Intel Corporation. Intel® 64 Architecture Memory Ordering White Paper. http://www.cs.cmu.edu/~410-f10/doc/Intel_Reordering_318147.pdf, August 2007. [Online; accessed 10-October-2019].

[16] gcc. ARM NEON Intrinsics. https://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/ARM-NEON-Intrinsics.html. [Online; accessed 10-November-2019].

[17] gcc. Common Function Attributes. https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes. [Online; accessed 10-November-2019].

[18] gcc. How to Use Inline Assembly Language in C Code. https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html. [Online; accessed 10-November-2019].

[19] gcc. Other Built-in Functions Provided by GCC. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. [Online; accessed 10-November-2019].

[20] gcc. Definitions for the branch prediction routines in the GNU compiler. https://gcc.gnu.org/viewcvs/gcc/trunk/gcc/predict.def?view=markup, 2019. [Online; accessed 10-November-2019].

[21] glibc. What is an indirect function (IFUNC)? https://sourceware.org/glibc/wiki/GNU_IFUNC, 2017. [Online; accessed 10-November-2019].

[22] IETF. The memory-hard Argon2 password hash and proof-of-work function. https://datatracker.ietf.org/doc/draft-irtf-cfrg-argon2/, 2019. [Online; accessed 10-November-2019].

[23] SPARC International Inc. The SPARC Architecture Manual, V. 8. https://www.gaisler.com/doc/sparcv8.pdf, 1992. [Online; accessed 10-October-2019].

[24] Č. Kalina. OpenSSL Argon2 KDF Support Introduction Merge Request Comment. https://github.com/openssl/openssl/pull/9444/#issuecomment-520057832, 2019. [Online; accessed 10-November-2019].

[25] Č. Kalina. OpenSSL Argon2 KDF Support Introduction Merge Request Commits. https://github.com/openssl/openssl/pull/9444/commits, 2019. [Online; accessed 10-November-2019].

[26] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. In *ISW '97: Proc. of the first international workshop on information security*, pages 121–134, 1998.

[27] ARM Ltd. What is the fastest way to copy memory on a cortex-a8? http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13544.html, 2011. [Online; accessed 10-October-2019].

[28] ARM Ltd. NEON™ Programmer's Guide, ARM DEN0018A. https://static.docs.arm.com/den0018/a/DEN0018A_neon_programmers_guide_en.pdf, June 2013. [Online; accessed 10-October-2019].

[29] ARM Ltd. ARM ® Compiler Version 5.05, User Guide, ARM DUI 0472K. http://infocenter.arm.com/help/topic/com.arm.doc.dui0472k/DUI0472K_armcc_user_guide.pdf, September 2014. [Online; accessed 10-October-2019].

[30] ARM Ltd. ARM® Cortex®-A Series, Version: 1.0, Programmer's Guide for ARMv8-A, ARM DEN 0024A. https://static.docs.arm.com/den0024/a/DEN0024A_v8_architecture_PG.pdf, March 2015. [Online; accessed 10-October-2019].

[31] ARM Ltd. Cortex®-A72 Software Optimization Guide, ARM UAN 0016A. https://static.docs.arm.com/uan0016/a/cortex_a72_software_optimization_guide_external.pdf, March 2015. [Online; accessed 10-October-2019].

[32] ARM Ltd. Arm®Architecture Reference Manual, Armv8, for Armv8-A architecture profile ARM DDI 0487D.a. https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf, July 2019. [Online; accessed 10-October-2019].

[33] ARM Ltd. Procedure Call Standard for the Arm®64-bit Architecture, ARM IHI0055D. https://github.com/ARM-software/software-standards/blob/master/abi/aapcs64/aapcs64.rst, October 2019. [Online; accessed 10-November-2019].

[34] OpenSSL. Cooperative Multitasking Implementation. https://github.com/openssl/openssl/tree/master/crypto/async, 2018. [Online; accessed 10-November-2019].

[35] OpenSSL. provider manpage. https://www.openssl.org/docs/manmaster/man7/provider.html, 2018. [Online; accessed 10-November-2019].

[36] OpenSSL. Blake2b MAC OpenSSL Implementation, commit 7c214f1092. https://github.com/openssl/openssl/tree/

7c214f1092f7622a1c2fdc5cfe70ddc94918daa3/providers/implementations/macs, 2019. [Online; accessed 10-November-2019].

[37] OpenSSL. Blake2b MD OpenSSL Implementation, commit 7c214f1092. https://github.com/openssl/openssl/tree/7c214f1092f7622a1c2fdc5cfe70ddc94918daa3/providers/implementations/digests, 2019. [Online; accessed 10-November-2019].

[38] OpenSSL. Homepage. https://www.openssl.org/, 2019. [Online; accessed 10-November-2019].

[39] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *International Conference on Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[40] C. Percival. *Stronger key derivation via sequential memory-hard functions*. BSDCan 2009. 2009.

[41] C. R. Portwood. PHP RFC: Argon2 Password Hash. https://wiki.php.net/rfc/argon2_password_hash, 2016. [Online; accessed 10-November-2019].

[42] C. R. Portwood. Argon2, Password hashing, libsodium documentation. https://libsodium.gitbook.io/doc/password_hashing#argon2, 2019. [Online; accessed 10-November-2019].

[43] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[44] N. Stephens. New technologies in the arm architecture. https://static.sched.com/hosted_files/bkk19/3c/BKK19-202_New-Technologies-in-Arm-Architecture.pdf, April 2019. [Online; accessed 10-November-2019].

[45] L. Torvalds et al. Linux Kernel, linux/include/linux/list.h . https://github.com/torvalds/linux/blob/master/include/linux/list.h, 2018. [Online; accessed 10-November-2019].

A Appendix

A.1 Aarch32 memory copy

The following code snippets [27] illustrate how memory copy would be performed/benchmarked on an Aarch32 system.

A.1.1 Base Case

```
memcpy_w32:
    ldr  r3, [r1]
    str  r3, [r0]
    subs r2, r2, #4
    add  r0, r0, #4
    add  r1, r1, #4
    bgt  memcpy_w32
```

A.1.2 Load-Multiple

In A32 one may load data into/store data from multiple registers in a single instruction (LDM/STM; Load/Store Multiple). A64 since reduced that only to a pair of registers (LPD/STP; Load/Store Pair). It loads contents from r1 (base address) into registers r3 through r10, incrementing address after each transfer and writing it back to r1 (similarly for store). The limited number of registers (compared to Aarch64) requires one to save up registers on stack prior to use, so there is a setup overhead.

```
memcpy_lm32:
    push  {r4-r10}
    stp  x3, x4, [x0]
    loop:
    ldmia r1!, {r3 - r10}
    stmia r0!, {r3 - r10}
    subs  r2, r2, #32
    bgt   loop
    pop   {r4-r10}
```

A.1.3 Interleaving Load/Store

```
memcpy_lp32_i:
    subs r2, r2, #64
    ldrd r3, r4, [r1,#0]
    strd r3, r4, [r0,#0]
    ldrd r3, r4, [r1,#8]
    strd r3, r4, [r0,#8]
    ldrd r3, r4, [r1,#16]
    strd r3, r4, [r0,#16]
    ldrd r3, r4, [r1,#24]
    strd r3, r4, [r0,#24]
    ldrd r3, r4, [r1,#32]
    strd r3, r4, [r0,#32]
    ldrd r3, r4, [r1,#40]
    strd r3, r4, [r0,#40]
    ldrd r3, r4, [r1,#48]
    strd r3, r4, [r0,#48]
    ldrd r3, r4, [r1,#56]
    strd r3, r4, [r0,#56]
    add  r1, r1, #64
    add  r0, r0, #64
    bgt  memcpy_lp32_i
```

A.1.4 NEON

```
memcpy_neon32:
    vldm r1!, {d0-d7}
    vstm r0!, {d0-d7}
    subs r2, r2, #0x40
    bgt  memcpy_neon32
```

A.1.5 NEON with Preload

```
memcpy_neon32_pld:
    pld  [r1, #0xC0]
    vldm r1!, {d0-d7}
    vstm r0!, {d0-d7}
    subs r2, r2, #0x40
    bgt  memcpy_neon32_pld
```

A.1.6 Mixed ARM and NEON with Preload

```
memcpy_mixed32:
    push    {r4-r11}
    mov     r3, r0

loop:
    pld     [r1, #192]
    pld     [r1, #256]

    vld1.64 {d0-d3},   [r1@128]!
    vld1.64 {d4-d7},   [r1@128]!
    vld1.64 {d16-d19}, [r1@128]!

    ldm     r1!, {r4-r11}
    subs    r2, r2, #128

    vst1.64 {d0-d3},   [r3@128]!
    vst1.64 {d4-d7},   [r3@128]!
    vst1.64 {d16-d19}, [r3@128]!

    stm     r3!, {r4-r11}

    bgt     loop
```

A.2 Overview of used perf commands

Listing A.1: **perf: Before Measuring**

```
# Various basic CPU statistics, system wide for 10 seconds:
perf stat -e cycles,instructions,cache-references,cache-misses,bus-cycles -a sleep 10
```

Listing A.2: **perf: Counting Events**

```
# Detailed CPU counter statistics (includes extras) for the specified command:
perf stat -d command

# Various CPU level 1 data cache statistics for the specified command:
perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores command

# Various CPU data TLB statistics for the specified command:
perf stat -e dTLB-loads,dTLB-load-misses,dTLB-prefetch-misses command

# Various CPU last level cache statistics for the specified command:
perf stat -e LLC-loads,LLC-load-misses,LLC-stores,LLC-prefetches command

# Count syscalls per-second: ???
perf stat -e raw_syscalls:sys_enter command

# Count system calls by type:
perf stat -e 'syscalls:sys_enter_*' command

# Count block device I/O events:
perf stat -e 'block:*' command
```

Listing A.3: **perf: Profiling**

```
# Sample on-CPU functions for the specified command, at 99 Hertz:
perf record -F 99 command

# Sample CPU stack traces (via frame pointers) at 99 Hertz:
perf record -F 99 -g -- command

# Sample CPU stack traces, using dwarf (dbg info) to unwind stacks, at 99 Hertz:
perf record -F 99 --call-graph dwarf -- command

# Sample CPU stack traces, once every 10,000 Level 1 data cache misses, for 5 seconds:
perf record -e L1-dcache-load-misses -c 10000 -ag -- sleep 5

# Sample CPU stack traces, once every 100 last level cache misses, for 5 seconds:
perf record -e LLC-load-misses -c 100 -ag -- sleep 5

# Sample on-CPU user instructions, for 5 seconds:
perf record -e cycles:u -a -- sleep 5

# Sample on-CPU user instructions precisely (using PEBS), for 5 seconds:
perf record -e cycles:up -a -- sleep 5

# Sample CPUs at 49 Hertz, and show top addresses and symbols, live (no perf.data file):
perf top -F 49
```

```
# Sample CPUs at 49 Hertz, and show top process names and segments, live:
perf top -F 49 -ns comm,dso

# Trace all minor faults with stack traces, until Ctrl-C:
perf record -e minor-faults -c 1 -ag

# Sample page faults with stack traces, until Ctrl-C:
perf record -e page-faults -ag

# Sample stacks at 99 Hertz, and, context switches:
perf record -F99 -e cpu-clock -e cs -a -g

# Sample stacks to 2 levels deep, and, context switch stacks to 5 levels (needs 4.8):
perf record -F99 -e cpu-clock/max-stack=2/ -e cs/max-stack=5/ -a -g

# Record cacheline events (Linux 4.10+):
perf c2c record -a -- sleep 10

# Report cacheline events from previous recording (Linux 4.10+):
perf c2c report
```

A.3 Performance Results of Memory Copy

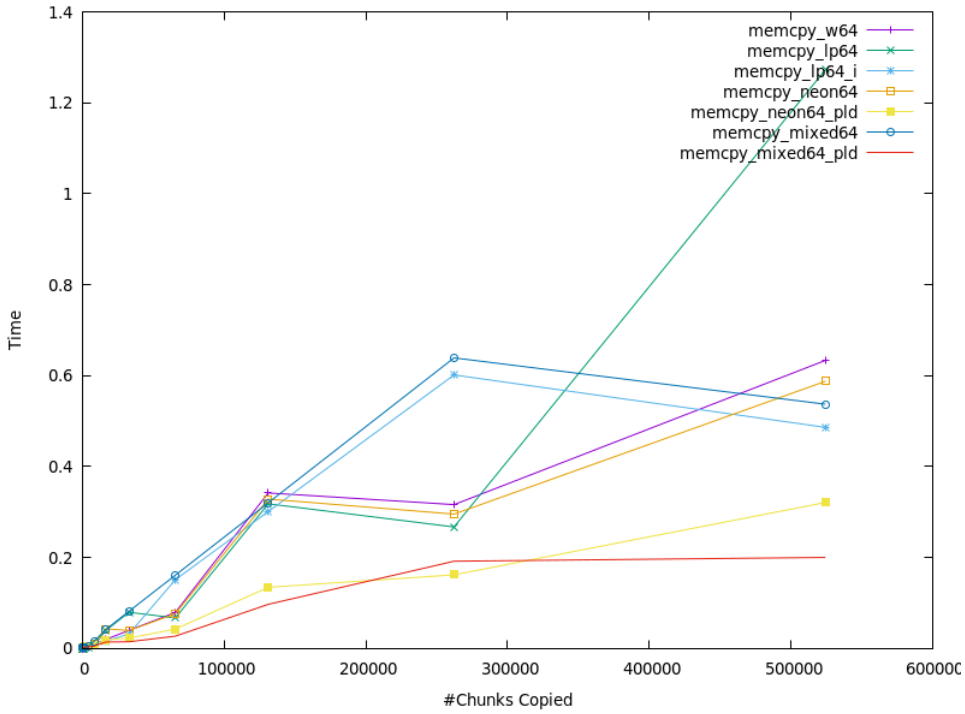

Figure A.1: ThunderX cn8890: Penguin Computing Valkre 2040 Gigabyte Blade

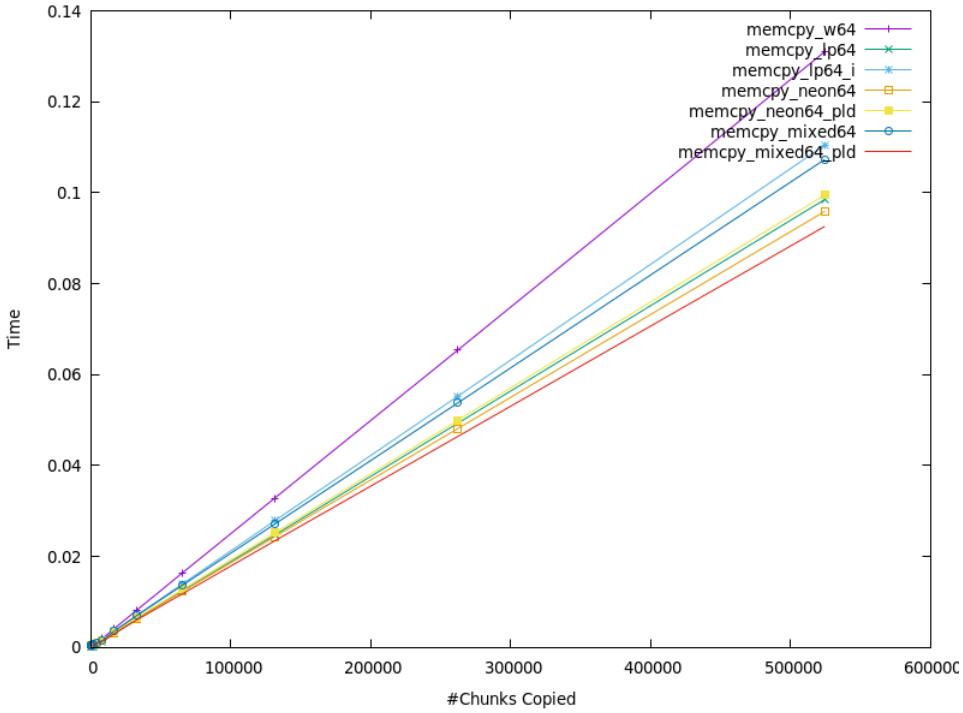

Figure A.2: ThunderX2 cn9975: Cavium ThunderX2 Sabre

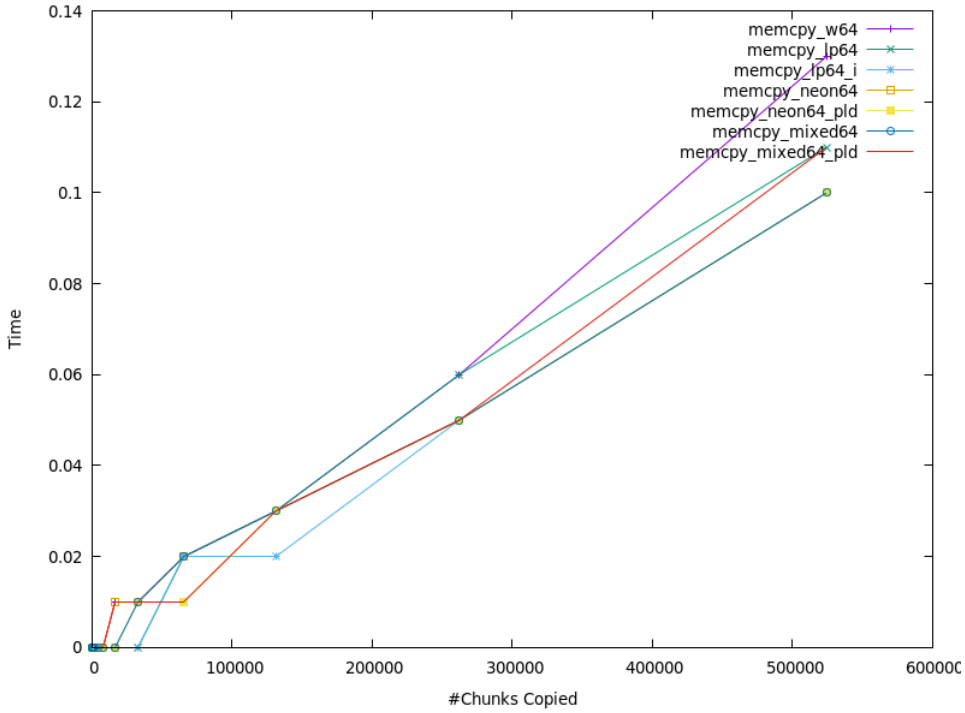

Figure A.3: HP M400

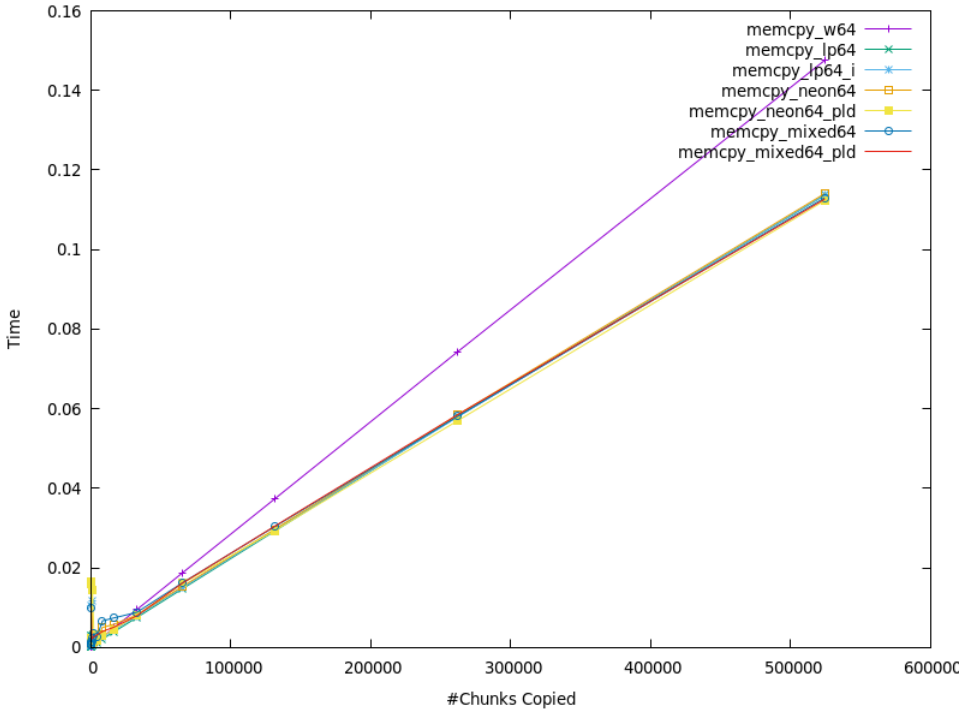

Figure A.4: Armada 8040: MACCHIATObin Double Shot

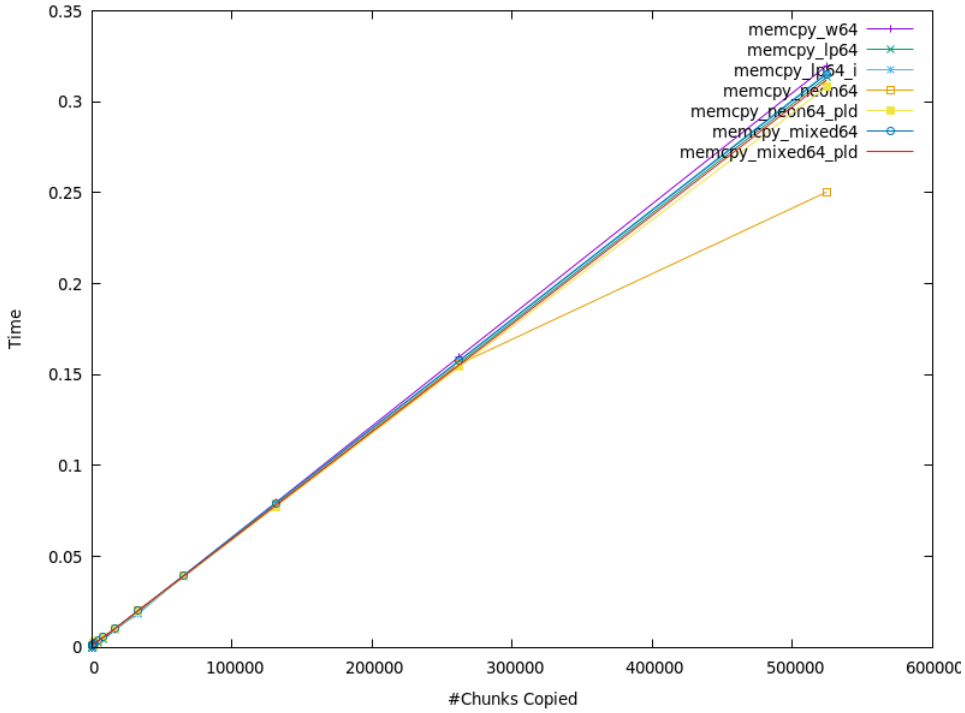

Figure A.5: Broadcom BCM2711: Raspberry Pi 4 Model B

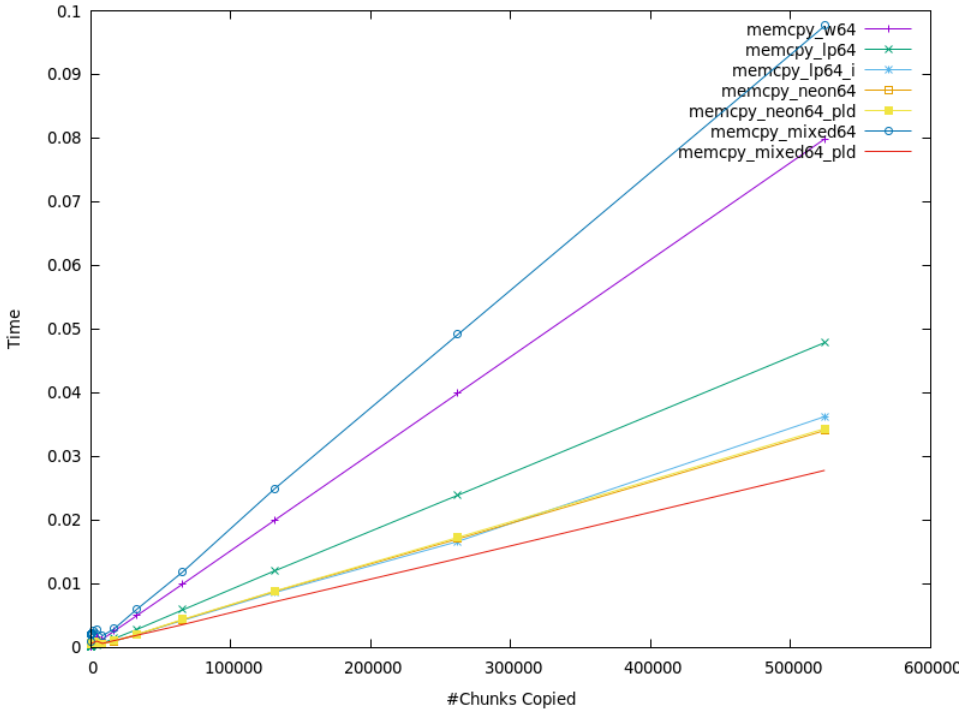

Figure A.6: Qualcomm Centriq 2452

A.4 Benchmark: NEON Preload

This section details time required (in seconds) to copy the same amount of memory (524288 chunks) across benchmark systems.

A.4.1 NEON Preload Tables

The following settings yielded the best/worst behaviour on tested systems:

1. Before 1st NEON block, L1 Cache, Keep Cached, Prefetch for Load, Offset 1024 bytes
2. Before 1st NEON block, L1 Cache, Keep Cached, Prefetch for Store, Offset 0 bytes
3. Before 1st NEON block, L1 Cache, Keep Cached, Prefetch for Store, Offset 128 bytes
4. Before 1st NEON block, L1 Cache, Streaming, Prefetch for Load, Offset 256 bytes
5. Before 1st NEON block, L1 Cache, Streaming, Prefetch for Load, Offset 768 bytes
6. Before 1st NEON block, L2 Cache, Streaming, Prefetch for Store, Offset 512 bytes
7. Before 1st NEON block, L3 Cache, Keep Cached, Prefetch for Store, Offset 640 bytes
8. Before 1st NEON block, L3 Cache, Streaming, Prefetch for Load, Offset 384 bytes
9. Before 2nd NEON block, L3 Cache, Keep Cached, Prefetch for Store, Offset 0 bytes
10. Before 2nd NEON block, L3 Cache, Streaming, Prefetch for Store, Offset 512 bytes
11. After NEON blocks, L1 Cache, Streaming, Prefetch for Store, Offset 0 bytes
12. After NEON blocks, L2 Cache, Keep Cached, Prefetch for Load, Offset 384 bytes

Machine	1	2	3	4	5
cn9975	0.105090	0.094883	0.102791	0.104605	0.106522
HP M400	0.100000	0.100000	0.120000	0.110000	0.100000
Armada 8040	0.115589	0.114654	0.113910	0.111123	0.115655
cn8890	0.297778	0.596271	1.264202	0.301849	0.443795
bcm2711	0.291434	0.311980	0.313710	0.310962	0.304966
Qualcomm 2452	0.035353	0.035858	0.033079	0.035311	0.035787

Machine	6	7	8	9	10
cn9975	0.104566	0.106249	0.104720	0.095378	0.104327
HP M400	0.100000	0.100000	0.110000	0.110000	0.100000
Armada 8040	0.119202	0.113813	0.113913	0.113890	0.119340
cn8890	0.259372	0.440914	0.256560	0.596181	0.259429
bcm2711	0.298141	0.290274	0.198008	0.317305	0.294494
Qualcomm 2452	0.040689	0.032463	0.035175	0.035205	0.034691

Machine	11	12
cn9975	0.095225	0.104487
HP M400	0.100000	0.100000
Armada 8040	0.113630	0.113832
cn8890	1.317233	0.255987
bcm2711	0.311624	0.306717
Qualcomm 2452	0.035554	0.034826

A.4.2 NEON Preload Summary

cn9975

Before 1st NEON block, L1 Cache, Keep Cached Prefetch for Store, Offset 0 bytes	0.094883
Before 1st NEON block, L1 Cache, Streaming Prefetch for Load, Offset 768 bytes	0.106522

HP M400

Before 1st NEON block, L1 Cache, Keep Cached Prefetch for Load, Offset 1024 bytes	0.100000
Before 1st NEON block, L1 Cache, Keep Cached Prefetch for Store, Offset 128 bytes	0.120000

Armada 8040

Before 1st NEON block, L1 Cache, Streaming Prefetch for Load, Offset 256 bytes	0.111123
Before 2nd NEON block, L3 Cache, Streaming Prefetch for Store, Offset 512 bytes	0.119340

cn8890

After NEON blocks, L2 Cache, Keep Cached Prefetch for Load, Offset 384 bytes	0.255987
After NEON blocks, L1 Cache, Streaming Prefetch for Store, Offset 0 bytes	1.317233

bcm2711

Before 1st NEON block, L3 Cache, Streaming Prefetch for Load, Offset 384 bytes	0.198008
Before 2nd NEON block, L3 Cache, Keep Cached Prefetch for Store, Offset 0 bytes	0.317305

Qualcomm 2452

Before 1st NEON block, L3 Cache, Keep Cached Prefetch for Store, Offset 640 bytes	0.032463
Before 1st NEON block, L2 Cache, Streaming Prefetch for Store, Offset 512 bytes	0.040689

A.5 Benchmark: Interleaved ARM/NEON Preload

This section details time required (in seconds) to copy the same amount of memory (524288 chunks) across benchmark systems. The tables below illustrate the best and worst results on different systems; n-th column maps to PRFOPn and PRFOFFn, if either of the two is well-defined.

cn9975

`pldl1keep` Offset 512	`pldl2keep` Offset 256	`pldl3keep` Offset 512	0.087315
`pstl3keep` Offset 512	N/A	`pstl1strm` Offset 256	0.183988

HP M400

N/A	N/A	`pldl1keep` Offset 256	0.100000
`pstl2strm` Offset 512	`pldl2strm` Offset 512	`pldl1strm` Offset 256	0.120000

Armada 8040

`pstl1keep` Offset 256	N/A	`pldl2keep` Offset 256	0.111054
`pldl3strm` Offset 512	`pldl3keep` Offset 256	`pldl3keep` Offset 512	0.142107

cn8890

`pstl1keep` Offset 512	`pstl1keep` Offset 256	`pldl1keep` Offset 512	0.200185
N/A	N/A	N/A	1.267637

bcm2711

N/A	N/A	`pldl1strm` Offset 512	0.198548
N/A	`pstl2keep` Offset 256	`pldl1strm` Offset 256	0.511537

Qualcomm 2452

N/A	N/A	`pldl3strm` Offset 256	0.026883
`pstl1keep` Offset 512	`pstl2strm` Offset 256	`pstl1keep` Offset 256	0.051841

A.6 Benchmark: Permutation Optimization

This section details time required (in seconds) to calculate 524288 iterations of $\mathcal{P}$ across benchmark systems.

cn9975

Iterations	NEON	Load Pair StorePair	C
1024	0.001287	0.000076	0.000594
65536	0.079305	0.004011	0.037895
131072	0.158524	0.007864	0.075762
262144	0.317145	0.015714	0.151519
524288	0.634132	0.031966	0.303046
1048576	1.283361	0.062812	0.606211

HP M400

Iterations	NEON	Load Pair StorePair	C
1024	0.000000	0.000000	0.000000
65536	0.090000	0.010000	0.030000
131072	0.190000	0.000000	0.070000
262144	0.380000	0.010000	0.140000
524288	0.760000	0.030000	0.280000
1048576	1.520000	0.070000	0.560000

Armada 8040

Iterations	NEON	Load Pair StorePair	C
1024	0.019175	0.000121	0.001286
65536	0.124347	0.003593	0.041591
131072	0.217263	0.007165	0.083123
262144	0.427577	0.014572	0.166105
524288	0.829197	0.029125	0.332296
1048576	1.607901	0.058231	0.664409

cn8890

Iterations	NEON	Load Pair StorePair	C
1024	0.001593	0.000094	0.001022
65536	0.101228	0.005454	0.064813
131072	0.202440	0.010896	0.129608
262144	0.404861	0.021776	0.259201
524288	0.809705	0.043538	0.518385
1048576	1.619883	0.087062	1.036777

bcm2711

Iterations	NEON	Load Pair StorePair	C
1024	0.001980	0.000082	0.000845
65536	0.126415	0.004765	0.053523
131072	0.252777	0.009475	0.107395
262144	0.505392	0.018967	0.215132
524288	1.010660	0.037847	0.430943
1048576	2.081701	0.077758	0.859831

Qualcomm 2452

Iterations	NEON	Load Pair StorePair	C
1024	0.002120	0.000086	0.001784
65536	0.072288	0.002220	0.042253
131072	0.131555	0.004501	0.087396
262144	0.247022	0.008988	0.194019
524288	0.457340	0.018164	0.399370
1048576	0.913953	0.035913	0.776470

A.7 Running in `Device-nGnRnE` mode

Remark: This section will demonstrate `Device-nGnRnE` on Linux based systems build using GNU libc[1], only. For information about what `Device-nGnRnE` memory is, the reader is referred to Section 4.6 or ARM Architecture Reference Manual [32]. It is also meant as a proof-of-concept only – one that makes sense for Argon2i only. It proposes using different memory access policy in security-sensitive sections, to aid in mitigating side-channel attacks by effectively forbidding speculative data accesses. Beware that `Device-nGnRnE` assumes that data is always 4-byte aligned, causing alignment fault[2] otherwise. Userspace doesn't usually count with these stringent restrictions. Note that trying to execute code from a region marked as Device is unpredictable.

It is not possible to redily allocate `Device-nGnRnE` memory directly from userspace, so first `Device-nGnRnE` userspace memory allocator must be created. OpenSSL uses malloc under the hood (that in turn is a wrapper of mmap/brk), so overriding malloc suffices to get the desired functionality on the userland-side.

A.7.1 Hooking `malloc`

In glibc, malloc is defined as a weak symbol so it can be overwritten by the application or a shared library (no preloading necessary). To obtain the address of the original libc malloc, either use a GNU extension in glibc:

```
dlsym(RTLD_NEXT, "malloc")
```

or the glibc-specific:

```
extern void *__libc_malloc(size_t size);
```

The wrapper uses the MSB of the size argument to determine whether `Normal` or `Device-nGnRnE` memory ought to be used. In the case of the former, `__libc_malloc` is called, otherwise `mmap` is called directly

1. This proof of concept can be easily adopted for non-GNU libc systems using, say, custom shared library and LD_PRELOAD.
2. Or, prior to `52d7523d84d534c241ebac5ac89f5c0a6cb51e41`, a kernel oops.

(with a specific file-descriptor facilitated by the kernel module passed as argument).

A.7.2 Kernel Module

To mark memory as `Device-nGnRnE`, `pgprot_noncached` can be used:

```
vma->vm_page_prot = pgprot_noncached(
        vma->vm_page_prot
);
if (io_remap_pfn_range(vma, vma->vm_start,
                       vma->vm_pgoff,
                       vma->vm_end - vma->vm_start,
                       vma->vm_page_prot))
return -EAGAIN;
```

This is not really a performance optimization as it is significatly slower than Normal memory variant. Since it is highly dependent on the operating system and the presence of a tailored kernel module, it is not fit to be included in OpenSSL architecture-specific port.

Acronyms

ASIC Application Specific Integrated Circuit

FPGA Field Programmable Gate Array

VMA Virtual Memory Area

ELF Executable and Linkable Format

ISA Instruction Set Architecture

PLT Procedure Linkage Table

SVE Scalable Vector Extension

API Application Programming Interface

IRQ Interrupt ReQuest

FIQ Fast Interrupt reQuest

TSO Total Store Ordering

KDF Key Derivation Function

MAC Message Authentication Code
MSB Most Significant Byte

LSB Least Significant Byte

MMU Memory Management Unit

TLB Translation Lookaside Buffer

GP General Purpose (Register)

XOR Exclusive OR

CTF Exclusive OR

Glossary

LLP64 A 64-bit data model. LLP64 model maintains compatibility with 32-bit code by leaving both int and long 32-bit.

LP64 A 64-bit data model. In an LP64 data model, int variables are still 32-bits wide, but long integers and pointers are 64-bits wide.

PoU Point of Unification (PoU) for a core is the point at which the instruction and data caches and translation table walks of the core are guaranteed to see the same copy of a memory location.

μop Micro-operation (or μop) is an internal representation of an architectural instruction handled by the microprocessor.

Tracepoint Instrumentation points placed at logical locations in code, such as for system calls, TCP/IP events, file system operations, etc. Tracepoints have negligible overhead when not in use.