P1026R0: A call for a Data Persistence (`iostream` v2) study group

Document #:	P1026R0
Date:	2018-05-06
Project:	Programming Language C++ Library Evolution Working Group

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

Responding to [P0939] *Direction for ISO C++*, a call for a new study group focused upon improving the standard library support for persisting data, specifically a modern replacement for `<iostream>` which is efficient on modern hardware. This *Data Persistence Study Group* would have the remit to:

1. Bring a low level file i/o library very thinly wrapping kernel syscalls into a portable standard library API, preserving all of the time and space complexities of the host platform, into an initial Technical Specification. See [P1031] *Low level file i/o*.

2. With recent advances by SG7 Reflection in mind, choose one of the existing major serialisation libraries as the base for a second Technical Specification standardising a more modern mechanism of object serialisation than `iostreams`.

3. Propose a comprehensive set of low level algorithms and containers whose in-memory representation is identical to their persisted representation on non-volatile storage as a third Technical Specification.

   Intel's persistent memory SDK and Boost.Interprocess would be good sources to draw from, perhaps with help from SG1 Concurrency in extending the C++ memory model to support persistent memory, and from SG5 Transactional Memory in implementing persistent memory transactions as Intel's SDK defines them.

Contents

1 Introduction

The C++ standard library has a reasonable collection of generic containers and algorithms for working with volatile memory, all with reasonable (amortised, or better) time and space guarantees. The original standard template library proposal, at its very beginning, did not assume that all memory was equal, rather it was to be through *Allocators* that the memory model for a container of objects was to be specified. To quote Stepanov [2]:

> During the design of STL and especially during the design of the allocator component, Bjarne observed that allocators, which encapsulate memory models, could be used to encapsulate a persistent memory model. The insight was Bjarne's, and it is an important and interesting insight. Several object database companies are looking at that. In October 1994 I attended a meeting of the Object Database Management Group. I gave a talk on STL, and there was strong interest there to make the containers within their emerging interface to conform to STL. They were not looking at the allocators as such. Some of the members of the Group are, however, investigating whether allocators can

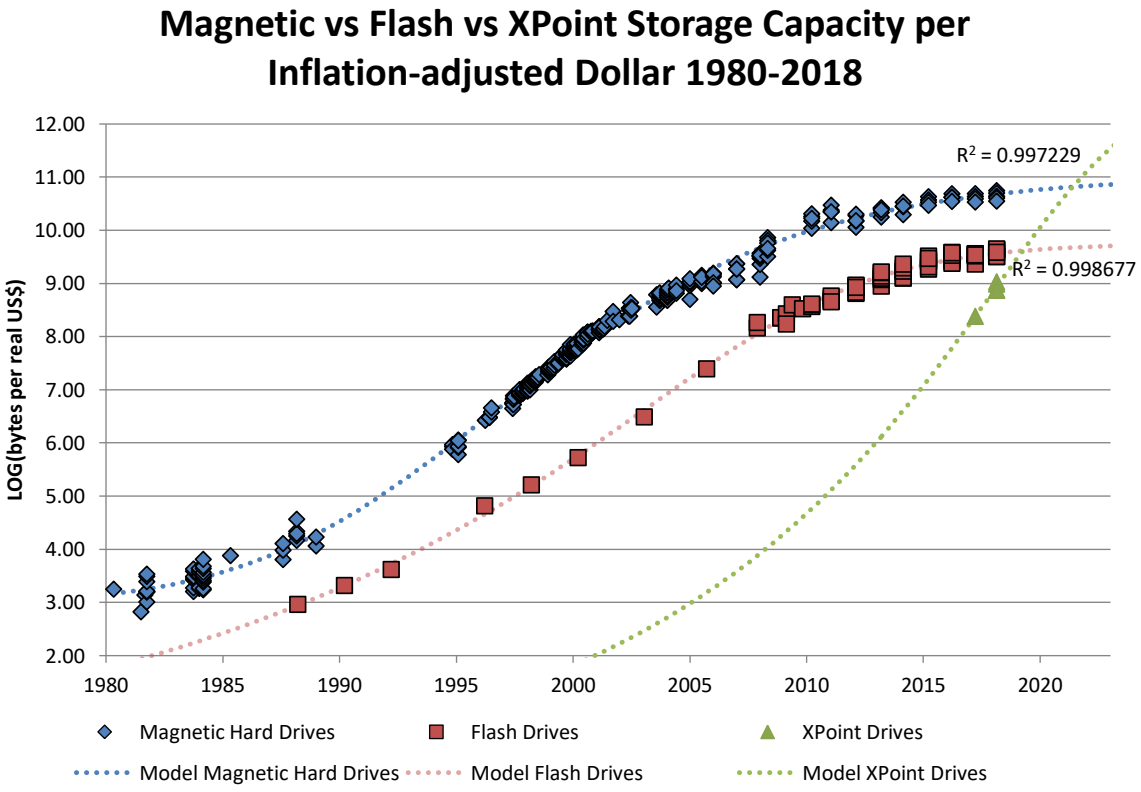


Figure 1: Magnetic vs Flash vs XPoint storage capacity per inflation-adjusted dollar 1980-2018.

be used to implement persistency. I expect that there will be persistent object stores with STL-conforming interfaces fitting into the STL framework within the next year.

As we now know, this did not occur.

Part of the cause was the enormous rise in the amount of volatile memory per dollar which occurred shortly after the interview (see Flash memory curve in Figure 1). Back in 1995, computers regularly came with less than one megabyte of RAM, and thus the ability to extend RAM with disc based storage with a finer granularity than that the system page size was seen at the time as highly important. However, as RAM dropped exponentially in price, and kernels implemented page file backed virtual memory more efficiently, the kernel implemented memory page swap file mechanism became good enough for most users. The pressure for the C++ standard to standardise more finely

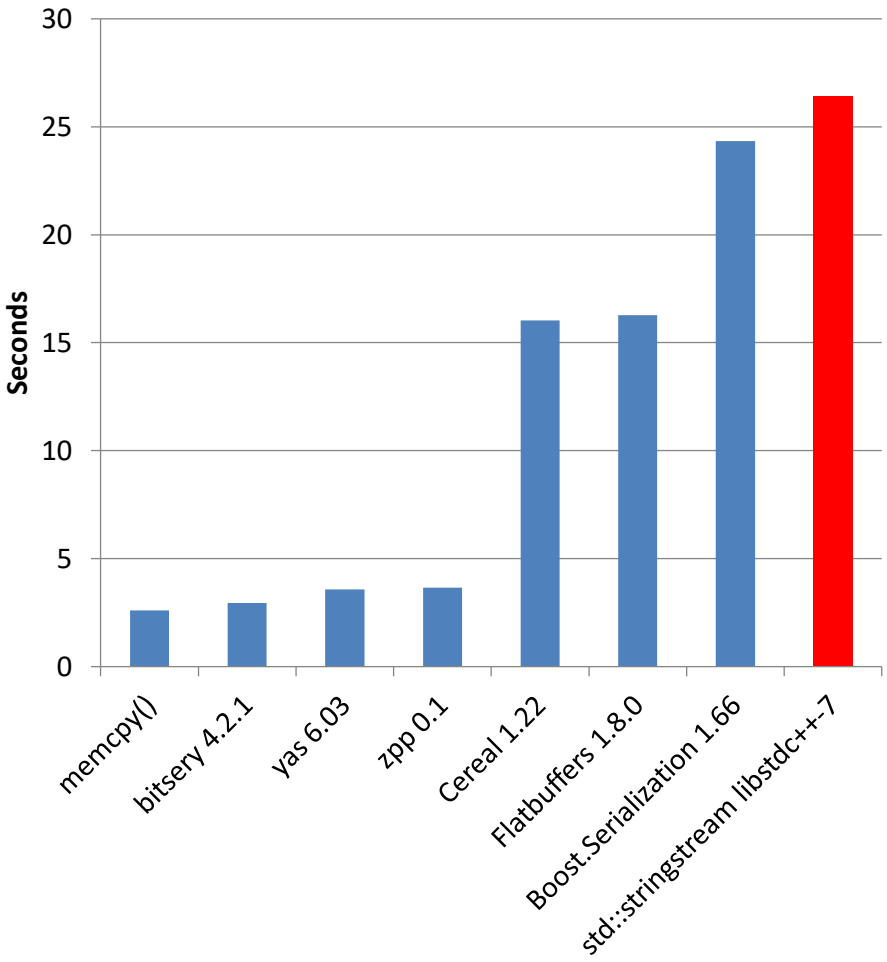
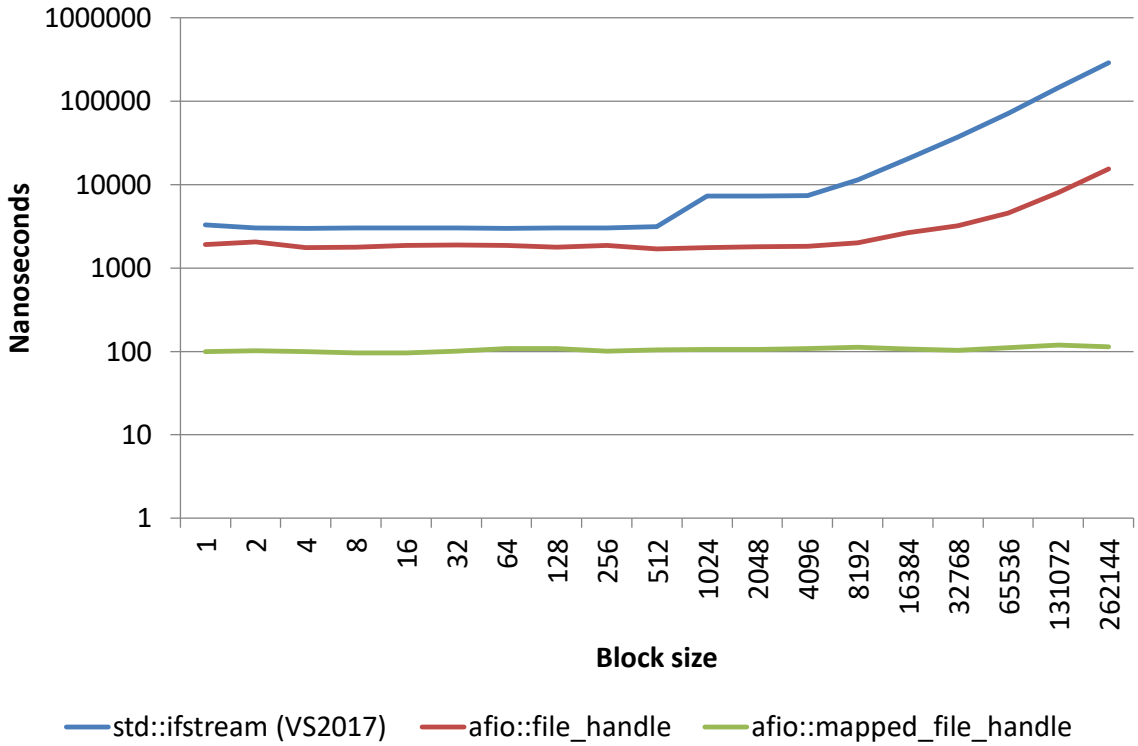


Figure 2: Performance of various serialization libraries as according to https://github.com/fraillt/cpp_serializers_benchmark. Lower is better. Note how the new generation of modern serialisation libraries are about eightfold quicker than `std::stringstream`, and approach hand written `memcpy()` in performance.

grained control of data persistence slackened.

The other part of the cause was that `iostreams` as standardised was 'good enough' for most people who needed to serialise and deserialise instances of objects to spinning rust storage. `iostreams` was always a bit of a legacy library in the C++ standard library. It is largely source compatible with at least one preceding i/o stream library design, and thus has oddities like opt-in exception throwing, lack of separation of formatting from serialisation, design inversion of customisation points plus some occasionally very surprising semantics, even to those who have been programming with it for twenty years or more. I can only speak of personal experience, but `iostreams` is the only library in the C++ standard library upon which I always still need to look up the reference documentation before I do anything more than trivial with it, because if I don't, I'll always be surprised. For me



Block size	1	4	16	64	256	1024	4096	16384	65536
`std::ifstream` (VS2017)	3301	2986	3017	2994	3020	7254	7389	20282	71538
`afio::file_handle`	1915	1766	1869	1873	1855	1750	1812	2633	4576
`afio::mapped_file_handle`	99	99	96	108	101	105	108	107	111

Figure 3: Latency differential between reads performed using `std::ifstream` and the proposed [P1031] *Low level file i/o library* as the size of the i/o increases. Test was conducted on a warm cache 100Mb file with random offset i/o, and represents the average of 100,000 iterations. Note the invariance to block size of the low level file i/o library's `file_handle` benchmark up to half the CPU's L1 cache size, demonstrating that no unnecessary memory copies have occurred. Note that the low level file i/o library's `mapped_file_handle` benchmark demonstrates no copying of memory at all.

at least, deep customisation of `iostreams` does not roll easily out of memory and practice, it feels too 'foreign'.

All warts aside though, `iostreams` is very powerful and flexible, capable of a very wide range of use cases and customisations, and performance is usually not a problem if gated by a spinning rust hard drive. Those who have found performance limits are usually those working exclusively avoiding storage e.g. socket i/o, shared memory IPC etc. See Figure 2 above.

Things have changed recently, however, and they will change a lot more again in the next few years, as one saw in the X-Point trend curve in Figure 1 above. The cause is the remarkable recent advance in the latency and bandwidth of non-volatile storage technology, with huge gains still to come in the near future. As shown in detail in the introduction to [P1031] *Low level file i/o library*, if you are using `<iostream>` on a recent MacBook Pro (which has a high end NVMe flash SSD), perhaps 10% of your i/o cost in a standard C++ program is due to your choosing `<iostream>` because of the unavoidable memory copies between the C++ program and the SSD hard drive. This is because of the no worse than 300 microsecond latency to perform a random 4Kb i/o on high end NVMe flash SSDs, as compared to the 26,000 microsecond worst case latency on a spinning rust hard drive[1]. This in turn means that latency in the software layers between the C++ program and the storage device have become exponentially more important as a percentage. See Figure 3 for a example of just how much latency the legacy `iostreams` design bring into file i/o over directly calling the kernel syscalls, with 72% more for a single byte and 1,463% more for 64Kb (the `file_handle` benchmark).

This is bad, but things get much worse with directly mapped storage (DAX) which is just around the corner, and ought to be common on consumer hardware in just a few years' time. Directly mapped storage appears to your program as random access memory, and when your program writes to such memory, **it is writing directly to non-volatile storage** (CPU caches notwithstanding). Such directly mapped storage is modelled by the `mapped_file_handle` benchmark, where `iostreams` is approaching *three orders of magnitude* worse latencies, at about six hundred fold worse.

These improvements to storage hardware fundamentally change how computer software must be designed to persist data efficiently. Hence the pressing need for the proposed *Data Persistence* study group, now.

[1] These are 99% percentile latency figures i.e. 99% of the time, an individual i/o latency will be below this figure.

2 Motivation and Scope

Firstly, we need to address if standardising this stuff is part of the Direction set for C++ in [P0939] which did not mention this as a priority for C++. Then we shall cover how the proposed pieces of standardisation could fulfil parts of the Direction set in P0939.

2.1 P0939 *Direction for ISO C++* did not state that getting closer to the kernel was part of the direction for C++

[P0939] said this:

> Technically, C++ rests on two pillars:
>
> - A direct map to hardware (initially from C)
> - Zero-overhead abstraction in production code (initially from Simula, where it wasn't zero overhead)

and:

> Over the long term, we must strengthen those two pillars
>
> - Better support for modern hardware (e.g., concurrency, GPUs, FPGAs, NUMA architectures, distributed systems, new memory systems)
> - More expressive, simpler, and safer abstraction mechanisms (without added overhead)

This makes no mention of closing the substantial gap of inefficiency between C++ and the operating system. Additionally, in personal conversations with Michael Wong, one of the Direction group, it was clear that the group had not considered getting C++ closer to the operating system kernel as a priority.

This paper asks the Direction group to refine their position on that. I think they have a choice of three options:

1. It could be decided that C++ has no business encoding, into the standard, use of features which were standardised into POSIX twenty or more years ago and which WG14 (C programming language) have not decided, to date, to standardise.
2. One could take an intermediate 'meta' position, that supporting GPUs, FPGAs, NUMA architectures and new memory systems would require changing the C++ memory model to incorporate Remote DMA (RDMA) whereby multiple tiers of ordering visibility would need to be specified, instead of the single tier currently in the standard. Such work would likely fall under the existing SG1 Concurrency. But apart from this abstract meta-support, nothing specific would be standardised.
3. Or one could take a fuller fat position, that facilities equivalent to those standardised in the decade old POSIX.1.2008 are widely implemented across all the major platforms, and C++

ought to expose more of those facilities in portable APIs which very thinly wrap those facilities with an absolute minimum of overhead.

Bare metal to the operating system kernel, as it were.

If Direction and WG21 do not consider getting closer to the operating system kernel as important i.e. the third option, then I need to know **now** so I can abandon my standardisation efforts on this topic, and redirect my energies elsewhere. To help with this decision, I have submitted six papers relating to my standardisation efforts to WG21 this meeting (which I shall also be physically attending), suggesting how one might go about the third option, specifically:

1. P1026 *A call for a Data Persistence (iostream v2) study group* https://wg21.link/P1026.
2. P1027 *SG14 Design guidelines for future low level libraries* https://wg21.link/P1027.

   This paper refines [P0939] *Direction for ISO C++* with concrete design guidelines for future low level library additions to the standard C++ library. Such low level libraries would historically have been an internal implementation detail of a standard library, P1027 calls for them to become public and standardised.

   P1028 and P1031 below meet these design guidelines.
3. P1028 *SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions: Throwing Values* https://wg21.link/P1028.

   This proposes a refactored, even lighter weight `<system_error>` v2 which fixes a number of problems which have emerged in the use `<system_error>` as hindsight has emerged. The replacement for `std::error_code`, `status_code`, is rarefied into a proposed `std::error` object for [P0709].

   P1031 *Low level file i/o* makes very extensive use of P1028, and would make use of P0709 if approved (for now it uses Boost.Outcome [1], a library-based substitute).
4. P1029 *SG14 `[[move_relocates]]`* https://wg21.link/P1029.

   This proposes a new C++ attribute `[[move_relocates]]` which lets the compiler optimise such attributed moves as aggressively as trivially copyable types. If approved, this would enable a large increase in the variety of types permissible in P1027's guidelines, plus P1028's standard error object would gain the ability to transport `std::exception_ptr` instances directly, a highly desirable feature for improving efficiency of legacy C++ exceptions support under P0709.

   All types consumed and returned in the APIs of [P1031] *Low level file i/o* have standard layout, are trivially copyable, or move relocating, as per the low level library design guidelines in P1027.
5. P1030 *Filesystem path views* https://wg21.link/P1030.

   This proposes a lightweight view of a filesystem path. Path views can help eliminate the often frequent copying of filesystem paths when calling a library such as P1031 below.
6. P1031 *Low level file i/o* https://wg21.link/P1031.

> This proposes a very thin portable API wrap of the kernel syscalls implementing file i/o, providing support for persistent memory, write reordering barriers, kernel cache control, sparse storage control, virtual memory control, byte range locking, lock files, temporary inodes, deadline i/o, asynchronous i/o, coroutined i/o, zero copy i/o, memory maps, and shared memory. A subset of (or near equivalent to) POSIX.1.2008 is required, with sufficient facilities on recent versions of all the major operating systems, including mobile and embedded.

Something perhaps not obvious until you read P1027 is that all of my standards work listed above is intended to be [P0829] *Freestanding C++* compatible i.e. without dependency on any STL or language facility not usable on embedded systems.

2.2 P0939 *Direction for ISO C++* asked for work on a Database interface

[P0939] said this:

> We feel that work on a database interface is badly needed, but since there seem to be no current work on that in the committee and no critical mass of expertise and interest, we can't put it on our list.

I have been working on bringing better data persistence to the C++ standard library since 2012, including submitting a previous edition of the proposed low level i/o library to a Boost peer review in 2015. The work in generating a complete rearchitecture of that reference library to meet Boost peer review feedback took me down some useful sidelines such as Outcome/Expected, but I believe now is the time to come out of (relative) stealth mode as it will take some years to ready this for standardisation.

I have not been working alone however. I have received direct assistance from Microsoft and Intel filesystem engineers. I am a (to date mostly silent) part of the persistent memory community, and they have been aware of my intention to propose this to WG21 for some time now. USENIX and the SDC are particularly abuzz with theoretical papers on new storage algorithms which implement interruption safe, concurrency safe, lock free data structures which work identically on non-volatile, as well as volatile memory; in RDMA, or local use cases.

For example, a recent paper at the 2018 USENIX Conference on File and Storage Technologies [3] demonstrated an implementation of a B+ tree which met all those criteria. A B+ tree is a core fundamental algorithm in almost any database implementation. A generic, customisable C++ template implementing that algorithm would be a huge value add to the C++ standard template library, and bring the C++ standard library much closer to being able to implement arbitrary databases simply by mashing together a few STL algorithms.

I appreciate that P0939 didn't exactly ask for this. Direction probably had SQL bindings in mind, or something similar. I find that a missed opportunity given the potential available here for C++ to extend and embrace the custom data persistence domain, one which is seeing profound disruption at the moment.

I have already built a proof of concept implementation of a transactional key-value store template primitive using the reference library for [P1031] *Low level file i/o*. It is not a database, rather it is a templated primitive from which you can construct *any arbitrary database* with very little code by the

user, and which works well even on embedded systems with Freestanding C++ (if they implement a filing system). This simple key-value store may have particular use to WG21 in the future, as it works perfectly well before `main()` is called. In other words, a future Modules based C++ program could load its shared Modules from the store during process bootstrap, and the C++ compiler could write compiled Modules into the key-value store, and atomically swap over a previous version of Modules for new version of Modules for some given program. There are endless possibilities.

Finally, and I know many reading this will find this silly, but what got me started down this track back in 2012 is the continuing impossibility, after forty years of software engineering, of portably updating two files at once as an atomic operation which is interruption safe. This is a critically important primitive to have in one's programming toolbox in order to write reliable software, and yet the current best available portable solution is to use SQLite which I find appalling[2]. This proposal would culminate in finally adding that simple primitive to C++: the ability to update more than one file in an all-or-nothing operation which is impervious to sudden process exit, sudden power loss, or any other surprise.

Fixing that is what has got me out of bed each morning to work on this since 2012, because the lack of that simple operation in portable code annoys me profoundly.

2.3 Unified page cache architecture kernels have become ubiquitous

The unified page cache kernel architecture treats RAM as a simple cache of file system data, lazily cached on first use only via the page fault. All memory is thus a map of some file data from somewhere, but i/o only occurs on first access. This allows the system to only use the exact physical RAM necessary for the processes running, rather than fill physical RAM with data never used. This in turn allows less RAM to run more processes and data than would be the case otherwise, but at the cost of spreading the i/o latency to load the data over every first access of a memory page.

Historically, kernels cached files separately to memory maps, and gave you physical memory when you asked for it. Thus, writing to a memory map of a file and using `write()` on the same file at the same offset was highly unwise as the two copies would become inconsistent. A lot of developers, even today, believe that this situation remains.

This has not been the case since the year 2000 for any of these platforms: Microsoft Windows, Linux, FreeBSD, NetBSD, Solaris, HPUX, all of which use a unified page cache architecture[3]. The only platforms of any user base significance with a split cache architecture are, to my knowledge, OpenBSD and QNX. And a few of the embedded operating systems implement no memory maps at all e.g vxWorks, eCos (LynxOS does implement them).

So I think it safe to say that the unified page cache architecture design has 'won', and POSIX.1.2008 standardises a reasonable collection of syscalls for detailed control over the virtual memory in page caching kernels. Yet C++ has zero awareness of any of this, and this makes C++ inefficient in quite a few important corner cases, mainly wherever one is working with medium to large sized ranges of data.

[2] Not because SQLite is bad, but it's massive overkill for what should be a very simple primitive operation universally available.

[3] As does Mac OS, as it was derived from BSD.

For example, when you resize to much larger a `std::vector<T>` where `T` is a trivial type, the STL will default initialise the newly allocated memory, often to all bits zero. This is what happens:

1. New memory returned by unified page cache kernels is merely the kernel zero page mapped repeatedly throughout the returned address range. Only upon first write is a page fault generated, and a real page of memory placed where the write was performed.
2. Each page fault to do this costs several hundred CPU cycles per page, sometimes many thousands of CPU cycles if no pre-zeroed pages are to hand in the kernel.
3. The STL then writes zeros, for a second time, all over pages already guaranteed to be zero, evicting valuable hot data from the CPU's caches.

The committee has historically taken the view that this is desirable behaviour, because it front loads the 'true' cost of the allocation at the time of allocation. However it is a highly inefficient approach. Modern kernels can be told, in a single syscall, to prefault a memory allocation, thus skipping the overhead of a page fault and kernel transition per memory page written. Additionally, one need not write zeros over memory already known to be all bits zero, especially if it is all prefaulted into memory. Finally, for large allocations exceeding the large page size (typically 2Mb), and where we know that the whole allocation will be used, the kernel can be asked for large pages which occupy a single TLB entry and reduce the latency of accessing a cold normal page considerably (a TLB fill can cost hundreds of CPU cycles). The ability to use single TLB entry large pages significantly improves the performance of working with datasets into the megabytes range.

As a second example, currently when we deallocate a medium to large sized vector, we probably end up invoking `free()` on that memory, which probably asks the kernel to deallocate the memory entirely, which in turn means that the kernel must clear those pages to all bits zero, again evicting from CPU caches valuable hot data. A better alternative is if we told the kernel that we no longer care about the contents of the deallocated memory. The kernel will make a note that those pages can be detached and used for new allocations if needed, but will otherwise leave them in place. Any write to any those pages will either fault in a new, cleared page, or unmark the existing page from being available for recycling. As one can see, this is a far more efficient way of deallocating memory because it avoids superfluous page clearing, which is especially valuable to the STL because it will default initialise the region anyway.

As a third and final example, it can be very useful for very large arrays of data to not prefault them all into existence at construction, and let the kernel allocate just the pages needed as they are written to. Instead of default constructing or clearing to zero parts of the array, one can instead ask the kernel to reset individual pages to all bits zero by swapping them for the kernel zero page, thus making the memory as if just freshly retrieved from the kernel, and helping the kernel to clear the dirty pages at its leisure in say an idle worker thread, rather than userspace forcing a clear immediately. This reduces overall memory pressure on the system and can help performance significantly by hinting to the kernel which memory pages contain valuable data, and which do not.

[P1031] *Low level file i/o* provides control functions for all of the above, exposing the POSIX.1.2008 functionality supporting virtual memory control. For medium to large ranges of memory allocations, it can make a big difference to STL container and algorithm performance. When accumulated in aggregate across all programs running in a system, it can make a **huge** difference.

As an analogy, right now C++ programs are running inside an emulated memory system with no awareness of the emulation. What virtual memory control built into the language and STL does is to *paravirtualise* that emulation. As with paravirtualised operating systems running inside many virtual machines, you receive large improvements in scalability and latency, thus allowing one to load more virtual machines onto each host server, improving cost benefit. The same would be the case for C++ programs, once they hint and tell the kernel what to do with memory: you would be able to run more C++ programs on the same hardware.

2.4 Persistent memory is coming

As Figure 1 showed, X-Point storage technology is currently undergoing exponential improvement in capacity per dollar with a much steeper improvement curve than any other storage technology. I do not expect it to exceed the capacity per dollar point of Flash storage however, as it is superior to Flash in every way, and thus I expect that Intel will choose to profit take on the new technology by keeping prices just above those of Flash. They will, however, wipe out the high margin segment of Flash storage entirely, leaving Flash storage for the budget minded only i.e. with markedly inferior performance to even today's high performance Flash storage devices.

X-Point works just like DRAM, and thus can go into DIMMs. NV-DIMMs based on X-Point will land later in 2018, complete with Intel chipset support. You will be able to fit either DDR4 DRAM or X-Point into your DIMM slots. There is every reason to believe that within a few years, that all high end computers, laptops and especially mobile devices will be using X-Point, or a similar non-volatile RAM storage, either as a complete replacement for volatile RAM, or as a supplement to it. I especially mention mobile devices because non-volatile RAM requires no constant refreshing to retain contents, and thus is extremely appealing to those wishing to reduce power consumption i.e. prolong battery life.

RAM which always retains its contents is a profound shift in computer technology. It has very wide implications for security, for the need for caching storage at all, and the whole structure and design of all database software, much of which will need to be completely rearchitected from first principles.

Major platform kernel support for persistent memory landed some years ago in the form of directly mapped storage (DAX) whereby your storage device appears as RAM to your CPU, and memory maps of files are direct maps of the storage device. Direct CPU support for persistent memory, in the form of dedicated CPU write buffer control opcodes, landed some years ago for ARM CPUs, who were unusually prescient for the future need for supporting this technology. Dedicated opcodes will land for Intel CPUs in the 2019 Icy Lake refresh.

Should C++ account for any of this now, or leave it until later?

This is a tricky question. On the one hand, CPU performance and RAM access latencies have seen mere linear improvement for more than a decade now. RAM capacity and GPU execution width continue to see exponential improvement, at least for now. This suggests that the ongoing reprioritisation for software architecture of exchanging, where possible, capacity utilisation for latency reduction will continue. This in turn is why I believe that C++ needs to focus, razor sharp, on eliminating all *amortised* complexities entirely in favour of *hard* complexity guarantees. This makes

building fixed latency budget software much easier, and that is where I see C++ continuing to be the dominant choice of programming language, **if WG21 acts now** to support the non-obvious causes of non-determinism in the language, and in its standard library.

Reworking the memory model of C++ to account for persistent memory is a huge undertaking, and best left to SG1 Concurrency and SG5 Transactional Memory. Multiple tiers of write ordering visibility will be needed, probably extending or modifying `std::atomic`, and I feel for those having to come up with standards wording to account for that. In the meantime, the 'everything is a file' paradigm, which has in such large part led to so much of present operating system design, is a useful bridge: write reordering barriers on file i/o also work when that file is on persistent memory, albeit at a non-trivial cost in efficiency. We can thus ship low level file i/o support aiming for C++ 23, and direct support for persistent memory in the language memory model itself for a later C++ edition.

2.5 Remote DMA is becoming more important

Disc i/o and network i/o are but specialised cases of remote DMA (RDMA) i.e. the direct reading or writing of remote memory. The higher end network cards contain CPU and RAM sufficiently powerful to class them as a separate computer, and disc controllers have become such powerful embedded computers that some allow you to run programs on them directly working with the raw storage, much as you would write a shader program for a GPU. I/O buffers for higher-end network and disc devices increasingly are mapped straight into user space memory such that when user code performs an i/o, there are zero copies of memory between the user code and the device.

The C++ standard does not need to – yet – standardise on some method of implementing RDMA. It would be better to see how the standardisation efforts at OpenFabric et al pan out first. That does not mean we need to stand still however. What we can do now – and with great utility for the DMA efficiency of disc and network i/o – is the following:

1. STL algorithms need to not silently cause hidden page copies by accessing a page locked for DMA, which can cause the kernel to silently copy a full memory page behind the scenes. To explain, on some architectures when an i/o is performed which is of a reasonable size, the kernel may pin the memory page(s) the buffer is in and have the device directly DMA to that page. By 'pin', I mean that the page is marked read-only or with no access at all until the DMA completes, and thus a page fault occurs should C++ access that page whilst it is locked. In order to ensure coherency i.e. what was in the page when the DMA was scheduled is not modified, the kernel may silently clone the page and modify the process' page tables, which also incurs a TLB shootdown across all CPUs (which involves sending them an interrupt, it's expensive).

   The STL needs to avoid causing this to occur by properly aligning i/o buffers, and using mitigation techniques such as double, triple or quadruple alternating buffers for i/o. It costs nothing on architectures where DMA can occur straight into CPU caches, but means the world for scalable performance on less fancy architectures.

2. STL algorithms need to learn when to use non-temporal stores to avoid evicting the CPU caches with data being DMAed elsewhere/persisted which will not be read again. For many

architectures, this also avoids needing to stall on CPU write buffer flushes for the written region which is a big gain, plus any hyperthreads can usually run at close to full speed whilst the other hyperthread is doing non-temporal stores.

3. As implicit memory allocation is a serious no-no for RDMA applications given the severe and non-deterministic execution time, a new suite of explicit-only capacity expanding containers is needed, ones whose in-memory representation and sequential ordering is identical to their persisted representation i.e. unexpected interruption would always leave these containers in a valid state.

   These have a wide range of use cases: interprocess shared memory which handles sudden process exit, memory mapped storage which handles sudden power loss, network endpoints which suddenly vanish, and so on. Unlike the existing STL containers, these proposed containers work with whole CPU TLB entry capacities, so typically 4Kb aligned and sized, 2Mb aligned and sized, 1Gb aligned and sized and so on. These, by definition, are ideal for maximum efficiency DMA without potential for silent page copies, and with minimum TLB entry imposed load on the DMA engine.

   There is a rich source of containers and algorithms to mine for standardisation in Intel's persistent memory SDK, and Boost.Interprocess.

3 References

[P0709] Herb Sutter,
Zero-overhead deterministic exceptions: Throwing values
https://wg21.link/P0709

[P0829] Ben Craig,
Freestanding proposal
https://wg21.link/P0829

[P0939] B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, M. Wong,
Direction for ISO C++
http://wg21.link/P0939

[P1027] Douglas, Niall
SG14 Design guidelines for future low level libraries
https://wg21.link/P1027

[P1028] Douglas, Niall
SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions
https://wg21.link/P1028

[P1029] Douglas, Niall
SG14 [[move_relocates]]
https://wg21.link/P1029

[P1030] Douglas, Niall
Filesystem path views
https://wg21.link/P1030

[P1031] Douglas, Niall
`Low level file i/o`
https://wg21.link/P1031

[1] *Boost.Outcome*
Douglas, Niall and others
https://ned14.github.io/outcome/

[2] *Al Stevens Interviews Alex Stepanov*
http://stepanovpapers.com/drdobbs-interview.html

[3] Deukyeon Hwang and Wook-Hee Kim, UNIST; Youjip Won, Hanyang University; Beomseok Nam, UNIST
Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree
Proceedings of the 16th USENIX Conference on File and Storage Technologies (2018)
https://www.usenix.org/system/files/conference/fast18/fast18-hwang.pdf