P1026R1: A call for an 'Elsewhere Memory' study group

Document #:	P1026R1
Date:	2018-10-07
Project:	Programming Language C++ Evolution Working Group Library Evolution Working Group SG12 Undefined behaviour study group SG14 Low latency study group
Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

A call for a new study group focused upon improving the language and standard library support for working with memory located elsewhere to the CPU running the C++ program. Such a study group would have the remit to:

1. Bring a low level virtual and mapped memory library very thinly wrapping kernel syscalls into a portable standard library API, into an initial Technical Specification. See [P1031] *Low level file i/o*.
2. Propose a set of changes to the C++ memory and object model to support working with memory mapped from elsewhere without invoking undefined behavior e.g. storage devices, graphics cards, NUMA.

Changes since R0:
- Based on feedback from Rapperswil, renamed paper from 'A call for a Data Persistence (iostream v2) study group' to 'A call for an Elsewhere Memory study group'.
- Based on feedback from Rapperswil, very significantly shortened paper to bare essentials, dropped mention of longer term stuff like serialisation, elsewhere containers etc. These can be dealt with after we get over the hump of the memory model changes.

The following WG21 Rapperswil meeting attendees indicated that they would have an interest in participating in the proposed Study Group:

- Niall Douglas
- Bryce Adelstein Lelbach
- JeanHeyd Meneide
- Peter Bindels
- Nevin Liber
- Nathan Myers
- Tony van Eerd
- Mathias Stearn
- Axel Naumann

[*Note:* If you are a WG21 committee member and would like your name to be added to the above list, please email me. – end note]

The following domain experts from industry have formally indicated that they would have an interest in participating in the proposed Study Group:

- Vladimir Petter, Windows kernel and server division, Microsoft.
- Piotr Balcer, Optane persistent memory team, Intel.

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 [1]:

> 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 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 memory elsewhere on disc 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 grained control of data held elsewhere slackened.

Things have changed recently, however, and they will change a lot more again in the next few years. The primary driver is the rise of heterogeneous compute where an increasing number of specialised compute resources are offloading work from the general purpose CPU, thus facilitating an enormous improvement in parallelism and bandwidth.

For example, a large capacity cheap consumer hard drive is likely to use shingled magnetic recording (SMR) technology to improve storage density per dollar. SMR platters are extremely slow to write, about 10Mb/sec. Such drives therefore have a conventional PMR cache of about 20-25Gb such that writes of up to that amount at a time[1] appear to be as fast as a pure-PMR hard drive (150Mb/sec) costing significantly more money. During idle time, the drive asynchronously transfers data from its PMR cache onto the slow SMR tracks, taking about 45 minutes to completely drain a full PMR cache. The consumer usually never notices how slow their hard drive truly is, as reads from SMR tracks are as fast as reads from PMR tracks.

The computing power necessary to implement such sophistication is highly non-trivial[2]. The LSI

[1]Note that the install of a fresh Microsoft Windows 10 is about 20Gb.

[2]https://www.usenix.org/system/files/conference/fast15/fast15-paper-aghayev.pdf

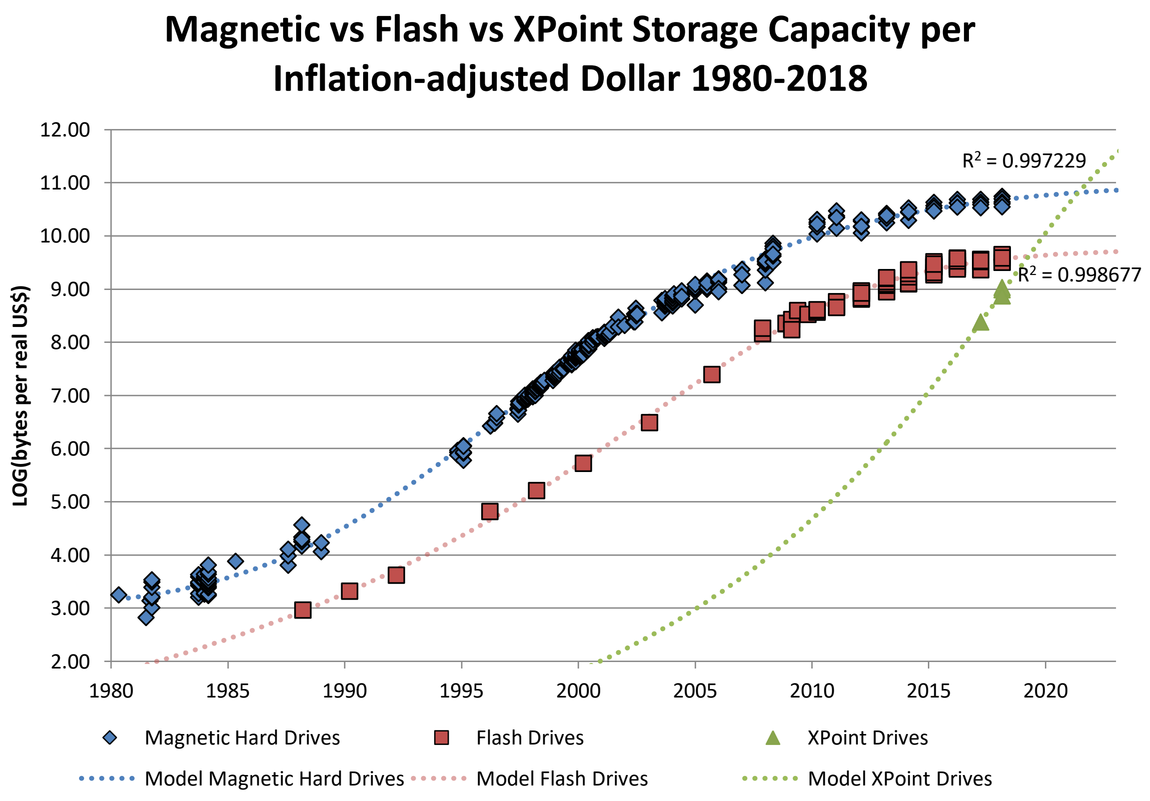

Figure 1: Magnetic vs Flash vs XPoint storage capacity per inflation-adjusted dollar 1980-2018.

controller used in these SMR hard drives has at least two quite powerful CPU cores of a similar design and arrangement to a SSD controller, and it runs at least several million lines of code, as much software complexity as an entire operating system two decades or so ago. Such a drive, along with *any* hard drive of recent years, is as powerful as a whole PC of only a decade or so ago.

Similar specialisation can be seen if you want to do a lot of math where the memory access patterns suit embarrassing parallelism. The TensorFlow symbolic maths library, for example, can run on CPUs or GPUs, and more usually many of the latter where possible. The potential gains in the amount of math performable per time unit are enormous, assuming that one can keep the GPU fed with data so that it doesn't stall.

As one can infer from the rise of heterogeneous compute, more and more problems implemented in C++ are going to be solved by tying together locally placed, hetergeneous compute resources over ultra low latency, high bandwidth links, where C++ running on say one or more hard drive controllers will need to work with C++ running on one or more GPUs, and with C++ running on one or more CPUs. Some of these C++ programs may be freestanding ([P0829]), some may be hosted. It should not matter which form they take – all should be able interoperate with one another without relying on undefined behaviour, and using the standard C++ memory and object model.

We thus need to improve how well C++ programs can interoperate with other C++ programs running elsewhere, and the first step towards that is support for working with memory elsewhere.

2 Motivation and Scope

We need the following in future C++ standards:

1. Awareness in the C++ standard that more than one C++ program can be executing at a time. Note that this is very different from multiple threads of execution within the single C++ program currently recognised by the standard.

2. Reasonable support in the standard for working with memory located elsewhere i.e. memory shared between heterogenous compute resources implemented by Direct Memory Access (DMA). This specifically means support at the language and library levels for memory mapped into the C++ program from elsewhere e.g. CPU cache write reordering control[3].

3 Papers in current work queue

3.1 [P1031] *Low level file i/o*

Bring a low level virtual memory and memory mapped file library very thinly wrapping kernel syscalls into a portable standard library API, into an initial Technical Specification. See [P1031]

[3]Atomics only specify to the CPU what *apparent* constraints on reordering there must be between concurrent threads of execution. Unless told otherwise, the CPU caches flush writes to main memory in an order disregarding atomic sequencing i.e. acquire, release and sequentially consistent atomics have no effect outside the CPU caches.

Low level file i/o.

3.2 Future paper on C++ memory and object model

Propose a set of changes to the C++ memory and object model to support memory mapped from elsewhere e.g. storage devices, graphics cards, NUMA. This probably will be sent to SG12 *Undefined Behaviour* before anything else.

It is currently expected that this paper will propose that elements of the CompCERT Memory Model v2 [2] ought to be merged into the standard C++ memory and object model. It is believed that these should be sufficient for the compiler to formally reason about memory mapped into the current C++ program from other C++ programs running elsewhere.

4 Papers relevant to the proposed study group

[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

[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

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

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

[2] Xavier Leroy, Andrew Appel, Sandrine Blazy, Gordon Stewart,
The CompCert Memory Model, Version 2
https://hal.inria.fr/hal-00703441/document