overload 178

DECEMBER 2023 £4.50

Use SIMD: Save the Planet

Andrew Drakeford demonstrates how SIMD (Single Instruction Multiple Data) can reduce your carbon footprint.

User Stories and BDD – Part 2, Discovery

Seb Rose continues his investigation of the term 'User Story'.

Dollar Origins

Paul Floyd shows how $ORIGIN can help when using tools from non-standard locations.

How to Write an Article

Frances Buontempo explains how easy it is to submit an article to Overload.

Afterwood

Chris Oldwood takes time to consider ghosts in the machine.

A magazine of ACCU

ISSN: 1354-3172

To connect with like-minded people visit accu.org

accu

December 2023

ISSN 1354-3172

Printing and distribution

Parchment (Oxford) Ltd

Cover design

Original design by Pete Goodliffe
pete@goodliffe.net

Cover photo by Daniel James. Fire escape staircases of the Lloyds building reflected in the building opposite.

Copy deadlines

All articles intended for publication in *Overload* 179 should be submitted by 1st January 2024 and those for *Overload* 180 by 1st March 2024.

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

Many of the articles in this magazine have been written by ACCU members – by programmers, for programmers – and all have been contributed free of charge.

Overload is a publication of the ACCU For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

General Knowledge and Selective Ignorance

It's very easy to assume everyone knows everything you know. Frances Buontempo points out that general knowledge is context dependent.

It is, perhaps, common knowledge to our regular readers that I always find an excuse for not writing an editorial. If you are new to this magazine, first, welcome, and second, bear with me. Allow me to explain why, as ever, I have no editorial for you. An editorial would involve a subjective viewpoint on a topic, giving a personal opinion which might open the floodgate to arguments. These, in turn, might stop me from finishing an editorial, so the most efficient way forward is not bothering. Besides, no one likes an argument. Furthermore, trying to spot if something is purely subjective can be difficult. Recently, the BBC news has insisted on reporting what 'BBC Verify' has discovered, slapping the label 'BBC Verify' on many of its new articles [BBC]. Many of the 'verifications' involve an assessment of images, trying to discern if they could be from a particular place by comparing known landmarks, streets and so on, as well as checking whether they have previously been visible on the internet, in order to pinpoint the veracity of the time the picture or film is claimed to have been taken. I am not convinced this approach would spot deep fakes or image alterations. I suspect most 'fake news' from bad actors involves sneaking emotive words, possibly along with real pictures, into a social media feed. We need to apply some discernment to anything we are told.

It can be very hard to spot untruths, and I am sure many of us can cite examples where we have changed our minds as more information has come to light. I recently watched *Union*, by David Olusoga [Olusoga23], which went through the history of the United Kingdom. I was aware of much of what he talked about, though there were some new bits of information. I was left thinking about the economic situation in the UK in 1973, and similarities with today's 'cost of living crisis'. I shall keep my opinions to myself, but I do see a coincidence with the Vietnam War causing a spike in the price of oil internationally and the cost of oil today. Further information can provide a new perspective on things.

Regardless of my opinions, verification is difficult. I have recently taken on some work with an open source Bounded Model Checker for C (CBMC) [CBMC]. I haven't worked on verification software before, so have lots to learn. This style of verification is very different to the BBC's attempt; however, both can spot things that are wrong but cannot prove things are right. Knowing something is wrong is one thing, but you will always be left with Rumsfeld-style known unknowns and even unknown unknowns. Proof is difficult, and software verification seems to nudge up against Gödel's incompleteness theorems very quickly. If I manage to understand the specifics fully, I might get round to writing an article, though I cannot promise it will be a complete guide to software verification. Hopefully, it will be consistent.

Mentioning 'Rumsfeld' and giving no reference rather assumes you know about the known unknowns phrase, [Wikipedia-1]. When we try to communicate, we need to assume some common knowledge, at very least a common understanding of a few words. It is all too easy to use colloquialisms specific to your own culture and experience when you write or speak. If I watch or listen to so-called 'general knowledge' quizzes, I often notice people answering sport-related questions fall into one of two categories. They either know or they don't know. I am firmly in the latter camp. Piecing together an intelligent guess is impossible with no background knowledge to go on. The same happens for other types of questions. If someone who has lived in the UK for the last ten years is asked about a British TV programme from the 1970s they are unlikely to know or even be able to guess the answer. General knowledge is usually geared towards specific contexts. This is why it is important to provide citations when writing. It allows readers to fill in gaps in their knowledge, as well as verifying your claims.

Many who take part in quizzes spend ages learning lists, say of US states or presidents. Points in a general knowledge quiz tend to demonstrate memorization of these lists or an ability to guess based on, say, Latin roots of words or similar. The culture and cohort specific questions don't seem to indicate any kind of cleverness. No-one said general knowledge is the same as being clever, to be fair. Sometimes playing quiz games with family or friends can be fun, but occasionally, someone says "Don't you even know that?" Very unkind. I've seen the same happen in teams of coders or even interviews, and am guilty myself. I interviewed someone for a C++ job years ago, and he hadn't come across `std::string`. I'd like to claim I didn't raise an eyebrow at this, but can't remember clearly. He got the job anyway and the first thing he said to me when he started was he'd been practising using the standard library, which was great. If someone doesn't know something you regard as essential knowledge, don't be arrogant and remember they might be able to learn about the subject, just as you did once.

I've been re-watching Babylon 5 [IMDB-1] recently. It's a 30 year old (!) TV series, based on a space station:

> located in neutral territory, … a major focal point for political intrigue, racial tensions and various wars over the course of five years.

This means various different aliens species interact. In order to communicate, they need some kind of commonality. At one point, they used the phrase "20 standard minutes". As I write this, the clocks have switched from BST to GMT in the UK, which will confuse me silly for a few weeks. I would imagine changing to 'std::minutes' might confuse me too. I wonder if a standard minute is an Earth minute, rather like UTC is GMT. David Olusoga might have something to say on the matter. Many 'common' ideas are drawn from a geographical and cultural

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD using AI and data mining. She's written a book about machine learning: *Genetic Algortithms and Machine Learning for Programmers*. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

context, making them seem more natural to some people than others. This provides diverse groups and teams with a challenge, which we should willingly embrace.

In order for people who speak different languages to communicate, we usually try to find a *lingua franca* [Wikipedia-2]. Literally meaning 'Frankish language', for many years, English has been used to communicate in business and the like. Lingua franca can also apply to mixed languages or Pidgins [Wikipedia-3]. These provide a general way to communicate, often using pre-existing languages, perhaps simplified or combined. People trying to communicate is one thing, but how do we make computers communicate? In some sense, the C language works as a lingua franca. Providing a C-API for a program, written in C++ or indeed one of many other languages, allows libraries or processes to be called directly. Using a narrow application programming interface hides away details, forming a very useful selective ignorance. There are various other ways to provide APIs, including REST, allowing services to talk. Hiding implementation details frequently enables better general discourse, and likewise making an API simple can facilitate usage. We don't always need to know historical details behind the definition of UTC in order to tell the time, but the extra details can be of interest. And get you points in a pub quiz.

In order to win at quizzes, you often need a diverse team of people with a mixture of knowledge, including someone who has memorized a list or two. The phrase "Not a lot of people know that" is often ascribed to Michael Caine [Wikipedia-4], since he had a tendency to rattle off obscure facts at the drop of a hat. Being a mind of 'useless' information might be very useful in a quiz. Is any information truly useless? I'm not sure. Certainly, someone pointing out everything they can think of about a topic might prove distracting. For example, someone telling a group why a decision was made about a code base can be useful insight in some circumstances; however, it might not help the team find the cause of a bug or figure out how to add a new feature. Watching Dominic Cummings speak to the Covid enquiry in the UK recently was 'interesting', or at least fascinating. He frequently rattled off lots of definitions and vague memories, interspersed with various expletives, managing to avoid answering the actual questions. A barrage of irrelevant information is always unhelpful. Discerning useless information is subjective, but spotting irrelevance is easier.

Now, if someone, or even something, can rattle off a stream of facts, does that makes them knowledgeable? We can watch AI chat software stream many words, sometimes including truths; however, this does not prove AI 'knows' anything. Similarly, we can type words into a search engine, and get results. The internet does not know anything, but can be a source of knowledge. As with books, or conversations, we need to find a way to pick through what is written or said, and draw our own conclusions. I asked ChatGPT if it knows or understands anything, and it replied,

> ChatGPT does not possess knowledge or understanding in the way humans do. It is a machine learning model that processes and generates text based on patterns it has learned from a large dataset of text. While it can provide responses that appear knowledgeable or understanding, it does not have true comprehension or consciousness.

To be fair, some people sometimes prattle off words and maybe do not have a proper understanding either. I notice of a lot of politicians and business people using the word 'nimble' recently. I suppose they had been saying 'agile' and someone, somewhere said "You keep using that word. I do not think it means what you think it means." [IMDB-2]

How do we gain understanding? A child may imitate adults while they learn language, and we actively encourage children to sing along with songs, allowing them to mispronounce words as they go. Over time, the child usually learns a few words correctly, and manages to use them outside the context of the song. As adults, we still have to keep learning new words. 'Have to' might be a bit strong, but new tech comes into being, so new words and phrases creep in, or old words get repurposed. Zoom, cloud, … I still have to suspend my disbelief/confusion if people say "sick" to indicate something is good. I notice some people struggling with the pronoun "they", probably for similar reasons. Unfamiliarity with a new context means you might misunderstand because of your expectations. I wonder if learning a new programming language has similar issues. Perhaps to begin with, you copy something from a book, blog or talk, like a child singing a song, or ChatGPT spewing forth code it found on the internet. Maybe you recognize a few parts of the code, but perhaps you stumble a bit. If you have experience of a similar language though, you may assume you understand a statement, but completely miss the point. Sometimes, a little bit of knowledge is a dangerous thing, to slightly misquote Alexander Pope [Wiktionary].

If you start working on a new codebase, in a language you know well, it can still take a while to find your way around. Even someone who has worked on a large codebase for years sometimes has to rely on a search to find definitions or usages. You can't always know all the things. Managing enough selective ignorance and using a few signposts or heuristics is usually enough to be able to find what you are after.

As the year draws to a close, take time to consider what, if anything, you have learnt this year. Have a look back through the articles in *Overload*, and if you are an ACCU member, look through this year's members' magazine, *CVu*, to pick your favourites. Then vote in our best article survey (see page 11).

Thank you for reading *Overload*, and thank you also to all our writers. Hopefully, we manage to keep you informed. Perhaps the citations provided clarify background details if you are interested, and give potential further reading.

We all have different strengths and weaknesses, so let's continue to help each other by sharing what we have learnt and encouraging each other by acknowledging our writers' hard work. Hopefully next year will bring more opportunities to continue to learn, while being careful to avoid irrelevant rabbit holes or simply learn lists of facts. Thank you for taking time to read this, and have a happy new year.

References

[BBC] BBC Verify: https://www.bbc.co.uk/news/reality_check

[CBMC] Bounded Model Checking for Software (for C and C++ programs): https://www.cprover.org/cbmc/

[IMDB-1] Babylon 5: https://www.imdb.com/title/tt0105946/

[IMDB-2] *The Princess Bride* – Quotes: https://www.imdb.com/title/tt0093779/quotes/

[Olusoga23] David Olugosa (2023) 'The Making of Britain': https://www.bbc.co.uk/programmes/p0gd25kn

[Wikipedia-1] Unknown unknowns: https://en.wikipedia.org/wiki/There_are_unknown_unknowns

[Wikipedia-2] Lingua franca: https://en.wikipedia.org/wiki/Lingua_franca

[Wikipedia-3] Pidgin: https://en.wikipedia.org/wiki/Pidgin

[Wikipedia-4] Michael Caine: https://en.wikipedia.org/wiki/Michael_Caine

[Wiktionary] 'A little knowledge is a dangerous thing': https://en.wiktionary.org/wiki/a_little_knowledge_is_a_dangerous_thing

User Stories and BDD – Part 2, Discovery

The term 'User story' is used in a variety of different ways. Seb Rose continues his investigation of the term, looking at detailed analysis.

This is the second in a series of articles digging into user stories, what they're used for, and how they interact with a BDD approach to software development. You could say that this is a story about user stories. And like every good story, there's a beginning, a middle, and an end. Welcome to the middle!

Previously ...

In the last article in this series [Rose22], we traced the origins of the user story. We saw that the term *user story* was used interchangeably with *story*, that stories were used as a *placeholder for a conversation*, and that this allowed us to *defer detailed analysis*. Now it's time to dive into the detailed analysis.

Last responsible moment

The reason we defer detailed analysis is to *minimise waste* – we don't want to work on features until we're reasonably sure that we're actually going to deliver them, and that work includes analysis. There's no value in having a detailed backlog that contains thousands of stories that we'll never have time to build.

The XP community approaches waste from another direction, with the concept of *you aren't gonna need it* (YAGNI) [C2-Wiki]. In a nutshell this tells us not to guess the future. Deliver only what you actually need today, not what you might need tomorrow – because you may never need it.

In *Lean Software Development: An Agile Toolkit* [Poppendieck03], the authors coined the phrase 'the last responsible moment' (LRM). This describes an approach to minimising waste based upon making decisions when "failing to make a decision eliminates an important alternative".

The key is to make decisions as late as you can responsibly wait because that is the point at which you have the most information on which to base the decision.

Accidental discovery

We want to defer making decisions until the *last responsible moment* because software development is a process of learning. We learn about the domain, what the customer needs, and the best way to use the available technology to deliver on that need. The more we learn, the better our decisions.

Learning something relevant after we've already made a decision is called *accidental discovery*. We made a decision believing we had sufficient knowledge, but we were surprised by an *unknown unknown* [Wikipedia]. The consequence of accidental discovery is usually rework (which is often costly), so our job is to minimise the risk of that happening.

Seb Rose Seb has been a consultant, coach, designer, analyst and developer for over 40 years. Co-author of the BDD Books series Discovery and Formulation (Leanpub), lead author of *The Cucumber for Java Book* (Pragmatic Programmers), and contributing author to *97 Things Every Programmer Should Know* (O'Reilly).

Risks and uncertainties are the raw materials we work with every day. We'll never be able to remove all the risks, nor should we want to. As Lister and DeMarco put it so eloquently in *Waltzing with Bears* [DeMarco03]:

> If There's No Risk On Your Next Project, Don't Do It.

Deliberate discovery

As professionals, we are paid to have answers. We feel deeply uncomfortable with uncertainty and will do almost anything to avoid having to admit to any level of ignorance. Rather than focus on what we know (and discreetly ignore what we're unsure of), we should actively seek out our areas of ignorance.

Daniel Terhorst-North [North10] proposed a thought experiment:

> What if, instead of hoping nothing bad will happen this time, you assumed the following as fact:
>
> - Several (pick a number) Unpredictable Bad Things will happen during your project.
> - You *cannot* know in advance what those Bad Things will be. That's what Unpredictable means.
> - The Bad Things will materially impact delivery. That's what Bad means.

To counter the constraint of "Unpredictable Bad Things happening", he suggests that we should invest effort to find out what "we are most critically ignorant of [and] reduce that ignorance – *deliberately discovering* enough to relieve the constraint."

By building deliberate discovery into our delivery process we are demonstrating our professional qualities – not admitting to incompetence. We are acting responsibly to minimise unnecessary risk and maximise the value that we deliver to our customer:

> Ignorance is the single greatest impediment to throughput. [North10]

Example mapping

Many agile teams will be familiar with *backlog grooming*/*refinement* meetings. The intention of these meetings is to refine our understanding of the stories on the backlog, but in my experience, they are often unstructured. Nor do they focus on uncovering what we, as a team, are ignorant of.

One of the most effective techniques for deliberate discovery is called Example Mapping – created and documented by my colleague, Matt Wynne [Wynne15]. It's an extremely simple, yet extraordinarily powerful way of structuring collaboration between team members, harnessing diverse perspectives to approximate the wisdom of crowds [Surowiecki05].

Example maps (see Figure 1, overleaf) give us a visual indicator of how big a story is and how well we understand it. Once the story is well

If there's one thing that hurts delivery teams more than anything else, it is inappropriately large stories

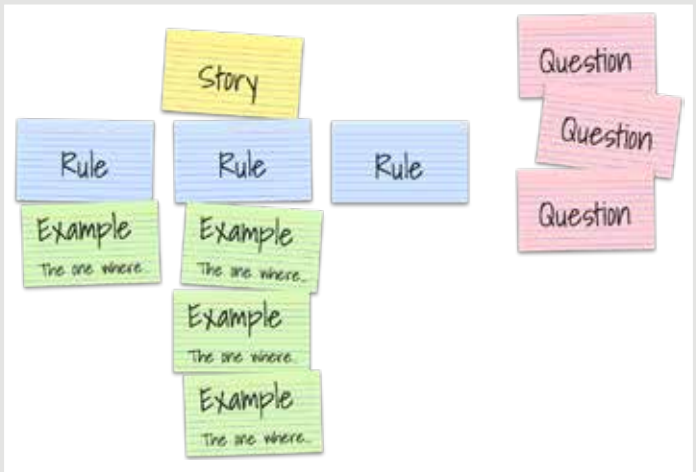

Figure 1

understood, we can use the example map to help the team split the story into manageable increments.

Stories all the way down

Stories start their life as *placeholders for a conversation*. As they get refined through the deliberate discovery process they become better understood, allowing us to decompose them into *detailed small increments*. ***Which we still call stories.***

The transformation from *placeholder for a conversation* to *detailed small increments* is not well understood by agile practitioners. There's informal usage in the agile community of the term *epic* as a label that identifies a large user story [Cohn22], but epics and user stories can both be *placeholders for a conversation*.

Nor are *detailed small increments* equivalent to *tasks*. Tasks are used to organise the work needed to deliver a *detailed small increment*. Each task is "restricted to a single type of work" [Cohn15] – such as programming, database design, or firewall configuration. Tasks, on their own, have no value whatsoever to your users. Each small increment, however, makes your product just a little bit more useful.

Same name, different purpose

By the time a piece of work is pulled onto the iteration backlog, the main purpose of the story is to *aid planning and tracking*. The title of each story is no longer all that important – the story is simply a container that carries the detailed requirements of the next small increment to be delivered.

Without the decomposition to *detailed small increments* that takes place during discovery, the stories will be too large. If there's one thing that hurts delivery teams more than anything else, it is inappropriately large stories. Nevertheless, most teams that I visit still work on stories that take weeks to deliver and this is still all too common in our industry. ■

References

[C2-Wiki] You Aren't Gonna Need It: https://wiki.c2.com/?YouArentGonnaNeedIt

[Cohn15] Mike Cohn 'The Difference Between a Story and a Task', posted on the Mountain Goat Software blog on 24 February 2015 and accessed on 21 November 2023 at https://www.mountaingoatsoftware.com/blog/the-difference-between-a-story-and-a-task

[Cohn22] Mike Cohn 'Epics, Features and User Stories', posted on the Mountain Goat Software blog on 8 November 2022, and accessed on 21 November 2023at https://www.mountaingoatsoftware.com/blog/stories-epics-and-themes

[DeMarco03] Tom DeMarco and Timothy Lister (2003) *Walking with Bears: Managing Risk on Software Projects*, published by Dorset House Publishing Co Inc, USA.

[North10] Dan North 'Introducing Deliberate Discovery', published on Dan North & Associates Limited blog on 30 August 2010. Accessed 21 November 2023 at https://dannorth.net/introducing-deliberate-discovery/

[Poppendieck03] Mary Poppendieck and Tom Poppendieck (2003) *Lean Software Development: An Agile Toolkit*, Addison-Wesley Professional

[Rose22] Seb Rose 'User Stories and BDD – Part 1' published in *Overload* 171, October 2022 and available at https://accu.org/journals/overload/30/171/rose/.

[Surowiecki05] James Surowiecki (2005) *The Wisdom of Crowds: Why the Many Are Smarter Than the Few*, Abacus.

[Wikipedia] 'There are unknown unknowns': https://en.wikipedia.org/wiki/There_are_unknown_unknowns

[Wynne15] Matt Wynne 'Introducing Example Mapping', published on 8 December 2015, and accessed on 21 November 2023 at https://cucumber.io/blog/bdd/example-mapping-introduction/

This article was published on Seb Rose's blog on 21 November 2019: https://cucumber.io/blog/bdd/user-stories-and-bdd-(part-2)-discovery/

If you have just read something that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

Use SIMD: Save The Planet

Writing efficient code is challenging but worthwhile. Andrew Drakeford demonstrates how SIMD (Single Instruction Multiple Data) can reduce your carbon footprint.

Some sources claim that data centres consumed 2.9% of the world's electricity in 2021 [Andrae15]. With the recent sharp increase in energy prices, many firms became aware of this cost. Additionally, trends in AI use and the near-exponential growth in both network and data services projected over the next few years [Jones18] suggest that this situation will only get worse.

A data centre's single purpose is to run our software (not heat the planet). But what if the processing cores that our software runs on have a unique, often unused, feature that would enable them to do the work several times faster? Would this also mean several times more efficiently? This feature is SIMD (Single Instruction Multiple Data).

SIMD is a parallel computing technology that allows processors to perform the same operation on multiple data elements simultaneously. By using SIMD instructions, a single CPU instruction can operate on multiple data elements in a single clock cycle. For example, with 256-bit SIMD registers, it is possible to process eight 32-bit floating-point numbers or sixteen 16-bit integers in parallel. This leverages the data-level parallelism inherent in many algorithms, such as mathematical computations, image processing, audio processing, and simulations. Figure 1 illustrates the element-wise addition of two sets of numbers using the SSE2, AVX2 and AVX512 instructions sets.

SIMD instructions use vector registers, which can hold multiple data elements. These registers allow the CPU to apply a single instruction to all the elements simultaneously, thus reducing the instruction count. Consequently, SIMD can provide a substantial speedup for tasks that exhibit regular and parallelizable data processing patterns.

The experiment

To determine if utilizing SIMD instructions can save energy, we performed a straightforward experiment on widely used x86 hardware. In this experiment, we developed implementations for a standard task using three different instruction sets: SSE2, AVX2, and AVX512. Our goal was to measure power consumption while running these implementations.

Experiment setup

We created a simple application called **dancingAVX512** for this purpose. This application cycles through each of the different SIMD implementations, with each one processing the workload in three bursts, separated by pauses. These pauses allow us to establish both baseline energy usage and the energy consumed during actual workload. The application runs continuously and is pinned to a single processor core. We monitor various core parameters, such as clock frequency, temperature, loading, and power consumption, using Open Hardware Monitor [OHM].

Andrew Drakeford A Physics PhD who started developing C++ applications in the early 90s at British Telecom labs. For the last two decades, he has worked in finance developing efficient calculation libraries and trading systems in C++. His current focus is on making quant libraries more ecologically sound. He is a member of the BSI C++ panel.

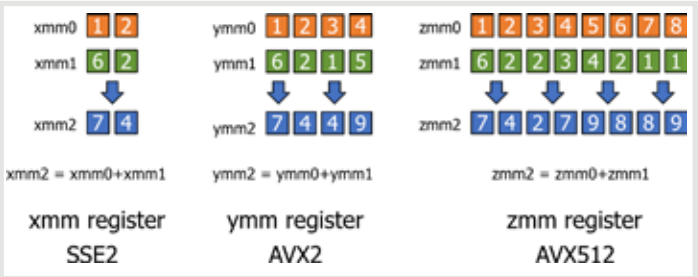

Figure 1

Additionally, we run a reference implementation of the same task using the C++ Standard Template Library.

Task selection

For our experiment, we chose the task of finding the maximum element in a vector of doubles. This task applies a reduction operation to a contiguous block of memory. Linearly traversing contiguous memory is highly predictable for hardware and allows efficient loading of data into vectorized registers. Moreover, reduction operations typically yield a scalar result, minimizing output size and memory writes. Additionally, we keep the vector size relatively small, it ranges from a few hundred to a few thousand doubles. This ensures that most of the processing occurs using data held in the fast L1 and L2 caches, reducing the chances of memory-related bottlenecks. These considerations are crucial to accentuating processing speed.

Implementation details

To implement the task, we leveraged the **DR3** framework [DR3]. The driver code for this implementation is shown in Listing 1 (overleaf). It defines a generic lambda function for the task, `maxDbl`. `maxDbl` simply calls the `iff` function which returns the results of an element wise selection from `lhs` or `rhs`, based on the results of the comparison operator.

The test loop corresponds to the experiment's main workload. Inside the main loop, `DR3::reduce` applies the `maxDbl` algorithm over the vector of doubles, `vec`. Internally, `DR3::reduce`'s inner loop is unrolled four times to achieve better performance.

Different, instruction set specific, versions of the workload for SSE2, AVX2, and AVX512 are created just by changing the enclosing namespace. This changes the SIMD wrappers (and instruction sets) used to instantiate the generic lambda and reduction algorithm. The code is compiled with both the Intel and Clang compilers to build two versions of the test executable **dancingAVX512**.

Disassembly

Clang compilation

When examining the compiled code, we observe that it produces highly efficient and compact assembly code, precisely what our experiment requires. In Listing 2 (opposite), we present the disassembly for the inner

The main difference between AVX2 and AVX512 disassembly is the register width, which affects the number of doubles processed per iteration

```
// The following lines determine which SIMD
// namespace is in use. Uncomment the desired
// namespace for the SIMD instruction set you
// wish to use.

//using namespace DRC::VecD8D; // avx512
     // - Enables 512-bit wide vector operations.

//using namespace DRC::VecD4D; // avx2 - Enables
     // 256-bit wide vector operations.

using namespace DRC::VecD2D;   // sse2 - Enables
     // 128-bit wide vector operations.

// A volatile double is used to store the result.
// Declaring it as 'volatile' prevents the
// compiler from optimizing it away.
volatile double res = 0.0;

// A lambda function 'mxDbl' is defined to return
// the maximum of two values.
// The 'iff' function selects between 'lhs' or
// 'rhs' based on the comparison.
auto maxDbl = [](auto lhs, auto rhs) { return
iff(lhs > rhs, lhs, rhs); };

// Generate a shuffled vector of size 'SZ'
// starting from value '0'.
auto v1 = getRandomShuffledVector(SZ, 0);

// Create a SIMD vector 'vec' with the values
// from 'v1'.
VecXX vec(v1);

// Loop 'TEST_LOOP_SZ' times, reducing the vector
// 'vec' using the 'mxDbl' function and storing
// the result in 'res'.
for (long l = 0; l < TEST_LOOP_SZ; l++)
{
  res = reduce(vec, maxDbl);
}
```

Listing 1

loops of both AVX2 and AVX512 workloads compiled using the Clang compiler.

Using `DR3::reduce` with a generic lambda function creates almost identical disassembly for the different instruction sets. In Listing 2, we can observe the assembly code generated for the inner loops of our experiment. Below is a breakdown of what is happening in the disassembly:

- The `vmovapd` instructions load values from memory into registers (`ymm4`, `ymm5`, `ymm16`, `ymm17` for AVX2, and `zmm4`, `zmm5`, `zmm16`, `zmm17` for AVX512).
- The `vcmppd` instructions perform element-wise comparisons, creating a mask (`k1`) that represents the results of the comparison, of the running maximums with the newly loaded values.
- The `vmovapd` instructions use the mask (`k1`) to update the running maximum values.
- The loop progresses through memory locations (`rbx`) and compares against the loop counter (`rcx`) to determine whether to continue iterating.

Intel compilation

The disassembly generated with the Intel compiler is shown in Listing 3 (overleaf). This has a much shorter loop and uses the `vmaxpd` (maximum of packed doubles) instruction. The unrolled inner loop calls four independent `vmaxpd` instructions. These compare and update the running maximum values (held in `zmm0`, `zmm1`, `zmm2` and `zmm3`) with values held in adjacent areas of memory pointed to by `zmmword ptr`s.

The main difference between AVX2 and AVX512 disassembly is the register width, which affects the number of doubles processed per iteration. As we move from SSE2 to AVX2 and then to AVX512, the number of elements processed doubles. This increase in processing capacity could be expected to double performance when executing the instructions at the same rate.

AVX2

```
1.  vmovapd   ymm4,ymmword ptr [rdx+rbx*8]
2.  vmovapd   ymm5,ymmword ptr [rdx+rbx*8+20h]
3.  vmovapd   ymm16,ymmword ptr [rdx+rbx*8+40h]
4.  vmovapd   ymm17,ymmword ptr [rdx+rbx*8+60h]
5.  vcmppd    k1, ymm0,ymm4,2
6.  vmovapd   ymm0{k1},ymm4
7.  vcmppd    k1,ymm3,ymm5,2
8.  vmovapd   ymm3{k1},ymm5
9.  vcmppd    k1,ymm1,ymm16,2
10. vmovapd   ymm1{k1},ymm16
11. vcmppd    k1,ymm2,ymm17,2
12. vmovapd   ymm2{k1},ymm17
13. add       rbx,10h
14. cmp       rbx,rcx
15. jle       doAVXMax512Dance+0B60h
```

AVX512

```
1.  vmovapd   zmm4,zmmword ptr [rdx+rbx*8]
2.  vmovapd   zmm5,zmmword ptr [rdx+rbx*8+40h]
3.  vmovapd   zmm16,zmmword ptr [rdx+rbx*8+80h]
4.  vmovapd   zmm17,zmmword ptr [rdx+rbx*8+0C0h]
5.  vcmppd    k1,zmm0,zmm4,2
6.  vmovapd   zmm0{k1},zmm4
7.  vcmppd    k1,zmm3,zmm5,2
8.  vmovapd   zmm3{k1},zmm5
9.  vcmppd    k1,zmm1,zmm16,2
10. vmovapd   zmm1{k1},zmm16
11. vcmppd    k1,zmm2,zmm17,2
12. vmovapd   zmm2{k1},zmm17
13. add       rbx,20h
14. cmp       rbx,rcx
15. jle       doAVXMax512Dance+390h
```

Listing 2

when the workload starts, the clock frequency and temperature rise, until an equilibrium is reached

AVX2

```
1. vmaxpd ymm0,ymm0, ymmword ptr [rdx+rax*8]
2. vmaxpd ymm1,ymm1, ymmword ptr [rdx+rax*8+20h]
3. vmaxpd ymm2,ymm2, ymmword ptr [rdx+rax*8+40h]
4. vmaxpd ymm3,ymm3, ymmword ptr [rdx+rax*8+60h]
5. add    rax,10h
6. cmp    rax, rsi
7. jle    doAVXMax512Dance+1370h
```

AVX512

```
1. vmaxpd zmm0,zmm0, zmmword ptr [rdx+rax*8]
2. vmaxpd zmm1,zmm1, zmmword ptr [rdx+rax*8+40h]
3. vmaxpd zmm2,zmm2, zmmword ptr [rdx+rax*8+80h]
4. vmaxpd zmm3,zmm3, zmmword ptr [rdx+rax*8+0C0h]
5. add    rax,20h
6. cmp    rax,rsi
7. jle    doAVXMax512Dance+1370h
```

Listing 3

std::max_element comparison

The disassembly of the inner loop of the `std::max_element` is shown in Listing 4. Even though it uses the vector `xmm` registers, the instruction names are suffixed by `sd`, for scalar double.

```
1. vmovsd   xmm0,qword  ptr [rcx]
2. vcomisd  xmm0,mmword ptr [rax]
3. cmova    rax,rcx
4. add      rcx,8
5. cmp      rcx,rbx
6. jne      doAVXMax512Dance+160h
```

Listing 4

The registers are used like scalars and the inner loop processes the elements sequentially, with only one double value considered per iteration. By way of contrast, the previous AVX512 listing, with a similar number of instructions, processes 32 doubles per iteration, potentially leading to a significant speed difference.

Results

Figure 2 and Figure 3, below and opposite, show the clock speed, temperature, load, and power recorded by the Open Hardware Monitor application [OHM] when the test application, **dancingAVX512**, is run on an Intel Silver Xeon 4114 (with a Skylake architecture).

The area under the power curve gives the total energy used. The area between the power curve when running a load and its background consumption level, is shaded. It corresponds to the energy used to compute the workload.

The figures show that when the workload starts, the clock frequency and temperature rise, until an equilibrium is reached. This is maintained until the task is completed. If a task runs for long enough, the run time gives a reasonable approximation of energy used when clock frequencies remain constant (and there is no reduction in power).

Figure 2 shows the power consumption for the Clang-cl build of **dancingAVX512**. The workload with the largest shaded area (and energy consumption) corresponds to `std::max_element`. For clarity, its

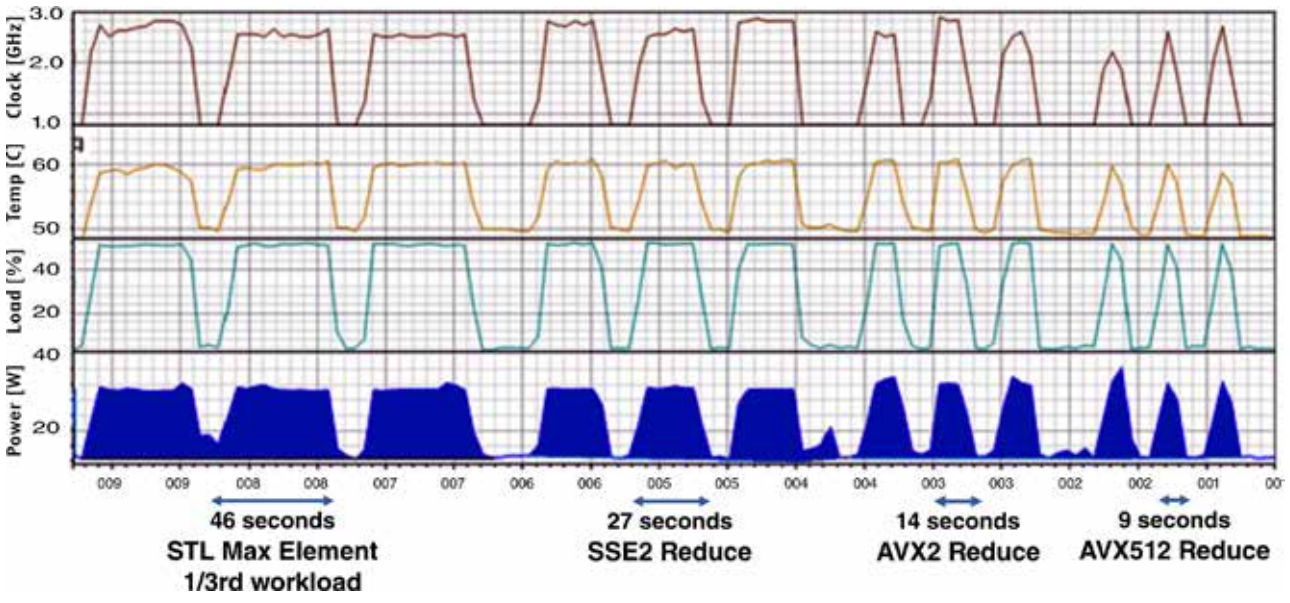

Figure 2

calling a member function on a group of objects of the same type, will always call the same method, so the contents of the instruction cache do not change

workload is reduced to a third of that done by the other implementations. However, its energy use is nearly twice that associated with the SSE2 implementation, suggesting that it uses between five and six times more energy. The regions of the graph corresponding to executing AVX2 and AVX512 workloads show a sequential halving of runtime and energy use (indicated by the shaded area). This is what we expected from the disassembly, under the assumption that the instructions are processed at the same rate.

Figure 3 shows the power consumption of the Intel (ICC2022) build of **dancingAVX512**. We note that switching from AVX2 to AVX512 workloads only reduces the runtime slightly; from 14 seconds down to 13 seconds. However, the energy used (shaded area), is noticeably less. Also, with the AVX512 implementation, the temperature impulse is much smaller than with the Clang-generated code. The top plot gives the CPU clock speed and shows that for AVX512 it only boosts up to about 1.6 GHz, as compared to the 2.6 GHz for the other runs. The Intel-compiled, AVX512 task is downclocked, making it run slower than the Clang build, however, it still saves energy when compared to the AVX2 run.

Our experiment clearly shows that when software uses SIMD instructions effectively, it can make a considerable difference to power consumption.

Next steps: saving the planet

Our codebase is available on GitHub [DR3], allowing enthusiasts to replicate our experiments. While results may vary due to compiler and hardware differences, substantial improvements in performance via SIMD are evident. We hope readers are inspired to integrate SIMD instructions into their existing C++ applications. Provided that the application is not IO bound and does not have workloads that are impossible to parallelize, SIMD could make significant improvements. However, initially, some might find limited performance gains. In this situation, the most common problems are caused by:

- **Memory layout**: In traditional OO C++ code, typical data structures may lead to inefficient memory access patterns.
- **Auto vectorization**: The compilers' auto-vectorizer does not always generate the vectorized code anticipated.

Memory layout

Memory bound, performance critical regions of legacy C++ applications often navigate diverse object collections, invoking varied virtual functions. This unpredictability in iteration leads to unpredictable memory access patterns which cause cache misses and hinder optimal CPU performance.

A SIMD speed-up will be of little use if the application's performance is memory-bound. Refactoring is needed to remove this limitation. After an initial performance measurement step which identifies the problem areas, a two-stage refactoring process can be used to ensure the codebase is SIMD-ready. It focuses on reorganizing data to support efficient, parallel computations by enabling:

- Predictable iteration.
- Efficient SIMD invocation.

Predictable memory access is achieved by segregating the inner loop's object collection. The original collection is divided into type-specific subsets. Iterating over and calling a member function on a group of objects of the same type, will always call the same method, so the contents of the instruction cache do not change. The subsets contain objects of the same

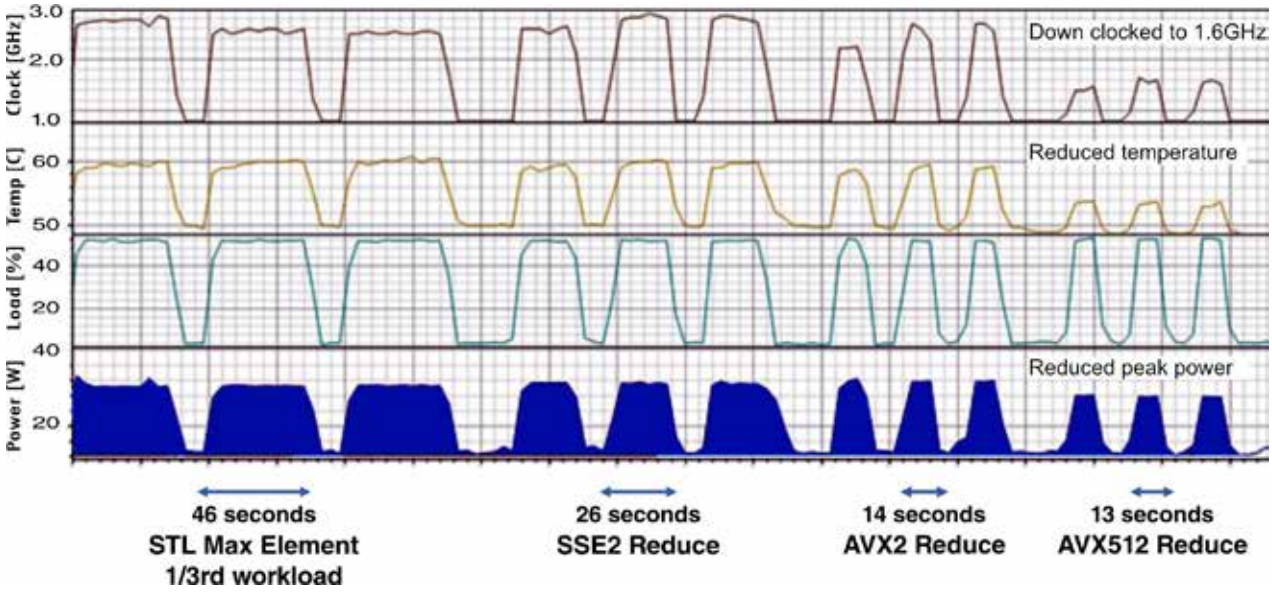

Figure 3

type, which are of course, the same size. This enables us to re-organize them as an **A**rray **o**f **S**tructures, (**AoS**). Furthermore, iterating over this layout has a very predictable memory access pattern, since the objects are all the same size. This lays down a foundation for predictable memory access.

Efficient SIMD invocation is achieved by transforming each segregated (**AoS**) collection into a **S**tructure **o**f **A**rrays (**SoA**). The array of structs of the same sub-type is converted to a new struct, where all its primitive member attributes are replaced by arrays. An individual object is represented by the collection of values found by slicing through all the member arrays at the same index. The **SoA** layout has the following benefits:

- *Contiguity:* Values of the same data attribute from multiple objects can be loaded into a SIMD register simultaneously, enhancing performance.
- *Optimized cache utilization:* Predictable sequential data access from the array data members in **SoA** means the CPU can prefetch data more effectively, reducing cache misses.
- *Simplified management:* In some cases, memory read by SIMD instructions needs to be aligned. Allocating memory to contain a multiple of a registers capacity and adding extra elements for padding, (if needed), simplifies processing the array. Processing padded arrays only loads a whole number of registers. Additionally, by ensuring that the first element in the padded array is aligned appropriately, all subsequent loads will satisfy the alignment requirements.

With the appropriate utilities to support transformations to **SoA**. [Amstutz18, Intel23] and alignment and padding, this is less arduous than one might think. A practical demonstration of vectorizing a function is given in [Drakeford22].

Refactoring with these strategies in mind not only sets the stage for effective SIMD integration, but one often sees substantial performance improvements even before SIMD comes into play. These strategies harness some of the core principles of data-oriented design [Fabian18, Straume20, Nikolov18], focusing on the properties of the typical data used by the application, and how it is accessed and processed rather than solely on its representation as a data structure.

Auto vectorization and approaches for SIMD development

Effective auto-vectorization requires loops with known sizes, simple exit conditions, and straight-line, branchless loop bodies [Bick10]. Loop-carried dependencies occur when the body of a loop reads a value computed in a previous iteration. Vectorized loops assume calculations are independent across iterations. Loop-carried dependencies break this assumption and deter the compiler from vectorizing the loop.

Sometimes, when a loop uses multiple pointers or complex pointer arithmetic, the compiler cannot be sure that the values referred to are not the same object in memory (i.e., aliasing) or an object previously written to (i.e., a loop carried dependency). It is unsafe for the compiler to generate vectorized code in such cases. However, using the compiler-specific keyword `restrict` with pointers, can guide the compiler. Essentially, `restrict` hints that the pointers are not aliases, meaning they don't point to the same memory location. Similarly, using `#pragma`s with loops offers directives to the compiler, suggesting how it should process the loop. Both these techniques can encourage the compiler to generate vectorized instructions.

More invasive approaches become necessary when the compiler does not automatically vectorize the code. Various strategies and tools are available. The choice depends on the project's specific requirements and constraints, and the available libraries and tools.

Approaches to SIMD development:

While auto-vectorization offers opportunities for optimization, manual intervention can often lead to more substantial gains. Here are some approaches to consider:

- **Open MP (4)**: A parallel programming model for C, C++, and Fortran, which uses `#pragma`as to guide compiler optimizations, though it isn't universally supported. [Dagum98]
- **ISPC compiler**: A performance-oriented compiler that generates SIMD code. [ISPC, Pharr12]
- **Domain-specific libraries**: These are libraries tailored and optimized for particular computational tasks, providing specialized functions and routines for enhanced performance.
  - **Intel's MKL**: Optimized math routines for science, engineering, and financial applications. [MKL]
  - **Eigen**: A high-level C++ library for linear algebra. [Guennebaud13]
  - **HPX**: A C++ Standard Library for parallelism and concurrency. [Kaiser20]
- **Compiler intrinsics**: Provide a more granular, assembly-level control by directly accessing specific machine instructions, allowing for finely tuned SIMD optimizations. [Kusswurum22, Fredrikson15, Ponce19]
- **SIMD wrappers**: Libraries that offer higher-level, portable interfaces for SIMD operations, making SIMD code more maintainable and readable. [Kretz12, VCL2, Creel20]
- **Generative functional approaches**: Utilize the C++ template mechanism (including auto) to generate code that calls the appropriate SIMD instructions.
  - **EVE library**: Is a notable example of this approach, see [Falcou21, Penuchot18, Drakeford22].
- **SYCL and Intel's One API**: Frameworks designed to develop cross-platform parallel applications, ensuring code runs efficiently across various hardware architectures [Khronos21].
- **Using C++ 17's Parallel STL:** Guide the compiler to use SIMD where possible by using the `std::execution::par_unseq` execution policy.

Conclusion

The scale of energy use by data centres means that it has become a significant contributor to global pollution. Our simple experiment shows that using SIMD instructions can drastically improve energy efficiency on modern hardware. If our software made more effective use of SIMD, this could help reduce pollution.

However, achieving significant efficiency gains hinges on suitable parallelization and optimal data layouts in memory. For legacy OO systems, this typically demands carefully re-engineering the context in which the vectorized instructions run. Refactoring performance-critical regions of such systems using data-oriented design principles could be a necessary first step to enable effective use of SIMD.

Before committing to any particular approach to SIMD development, always test the performance achievable with the chosen toolset, its suitability to your problem domain, and its compatibility with your target environment.

Please help your code to consume energy responsibly. ■

References

[Amstutz18] Jefferson Amstutz, 'Compute More in Less Time Using C++ Simd Wrapper Libraries', *CppCon* 2018, available at https://www.youtube.com/watch?v=8khWb-Bhhvs

[Andrae15] Anders Andrae and Tomas Edler (2015). 'On Global Electricity Usage of Communication Technology: Trends to 2030' *Challenges*. 6. 117-157. 10.3390/challe6010117.

[Bick10] Aarat Bick 'A Guide to Vectorisation with Intel C++ compilers', https://www.intel.com/content/dam/develop/external/us/en/documents/31848-compilerautovectorizationguide.pdf

[Creel20] Creel (Chris), 'Agner Fog's VCL 2: Performance Programming using Vector Class Library', https://www.youtube.com/watch?v=u6v_70opPsk

[Dagum98] Leonardo Dagum and Ramesh Menon, 'OpenMP: an industry standard API for shared-memory programming' *Computational Science & Engineering*, IEEE 5.1 (1998): 46-55.

[Drakeford22] Andrew Drakeford, 'Fast C++ by using SIMD Types with Generic Lambdas and Filters' at *CppCon* 2022 https://www.youtube.com/watch?v=sQvlPHuE9KY, 6 min 35 secs

[DR3] DR3 library at https://github.com/AndyD123/DR3

[Fabian18] Richard Fabian (2018) *Data-oriented design: software engineering for limited resources and short schedules*, Richard Fabian.

[Falcou21] Joel Falcour and Denis Yaroshevskiy, 'SIMD in C++20: EVE of a New Era', *CppCon* 2021, available at https://www.youtube.com/watch?v=WZGNCPBMInI

[Fredrikson15] Andreas Fedrikson 'SIMD At Insomniac Games: How We Do the Shuffle' *GDC* 2015, available at https://vimeo.com/848520074.

[Guennebaud13] Gaël Guennebaud (2013) 'Eigen: A C++ linear algebra library' Eurographics/CGLibs, available at: https://vcg.isti.cnr.it/cglibs/. See also https://en.wikipedia.org/wiki/Eigen_(C%2B%2B_library)

[Intel23] Intel SDLT SoA_AoS https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2023-1/simd-data-layout-templates.html

[ISPC] Intel ISPC (Implicit SPMD Program Compiler): https://ispc.github.io/ispc.html

[Jones18] N. Jones 'How to stop data centres from gobbling up the world's electricity' *Nature* 561, 163-166 (2018)

[Kaiser20] Kaiser *et al.*, (2020). 'HPX – The C++ Standard Library for Parallelism and Concurrency', *Journal of Open Source Software*, 5(53), 2352. https://doi.org/10.21105/joss.02352

[Khronos21] SYCL 2020 specification, launched 9 February 2021, available at https://www.khronos.org/sycl/

[Kretz12] M. Kretz and V. Lindenstruth (2012) 'Vc: A C++ library for explicit vectorization', *Software: Practice and Experience* vol 42, 11

[Kusswurum22] Daniel Kusswurum (2022) *Modern Parallel Programming with C++ and Assembly Language X86 SIMD Development Using AVX, AVX2, and AVX-512*, APress.

[MKL] Intel(R) math kernel library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

[Nikolov18] Stoyan Nikolov 'OOP Is Dead, Long Live Data-oriented Design', *CppCon* 2018

[OHM] Open Hardware Monitor: https://openhardwaremonitor.org/

[Penuchot18] Jules Penuchot, Joel Falcou and Amal Khabou (2018) 'Modern Generative Programming for Optimizing Small Matrix-Vector Multiplication', in *HPCS* 2018

[Pharr12] M. Pharr and W. R. Mark (2012) 'ispc: A SPMD compiler for high-performance CPU programming' 2012 *Innovative Parallel Computing (InPar)*, San Jose, CA, USA, pp. 1-13, doi: 10.1109/InPar.2012.6339601.

[Ponce19] Sebastian Ponce (2019) 'Practical Vectorisation', *Thematic CERN School of Computing*, available at https://cds.cern.ch/record/2773197?ln=en

[Straume20] Per-Morten Straume (2019) 'Investigating Data-Oriented Design', Master's thesis in Applied Computer Science, December 2019 NTNU Gjøvik https://github.com/Per-Morten/master_project

[VCL2] Vector Class Library (version 2): https://github.com/vectorclass/version2

Dollar Origins

Using tools from non-standard locations can be challenging. Paul Floyd shows how $ORIGIN can help.

At work, we usually have GCC and binutils installed to the same root. I've recently been working on a project that needs a more recent GCC so I've been building these fairly often. Since these days GDB gets bundled with **binutils**, I thought I'd benefit from a nice new shiny GDB.

Some things that are always problematic when building and installing to non-standard locations are the shared libraries. Many of the system libraries that get used will be the ones that ship with the OS. The C standard library, **libc**, is one such example. However, if you build with a C++ compiler other than the system one, the odds are that you will need to link with the C++ standard library that was built with that compiler. Linking is only half of the problem. Finding the library at runtime is the other. When you build GCC, it will tell you all about that at the end of the build. If the library is in a well-known location like `/usr/lib64` then all is well. If you have multiple different versions of, say, `libstdc++.so`, then things are a bit more complicated.

The bad way to find libraries is to use **`LD_LIBRARY_PATH`**. The problem with this is that you can't use it to choose different library versions. It's a colon-listed set of directories, and the first directory that contains the library being sought gets used. That soon deteriorates to the point where every application needs a wrapper script to set its own **`LD_LIBRARY_PATH`**. There is an alternative. **`RPATH`**. **`RPATH`** is, in effect, **`LD_LIBRARY_PATH`** compiled into an exe. For more reasons to avoid **`LD_LIBRARY_PATH`**, see George Southoff's blog [Southoff16].

Recently, I decided to rename one of my directories, since I'd been doing some work with GCCs 11 and 13. And that broke my GDB. The problem was that I'd used an absolute **`RPATH`** to build it. When I renamed the directory, the absolute path no longer matched and the link loader could no longer find a suitable `libstdc++`. That gave me lots of errors like

```
/path/to/gcc-13.1.0/bin/gdb: /lib64/libstdc++.so.6:
version 'GLIBCXX_3.4.20' not found (required by
/path/to/gcc-13.1.0/bin/gdb)
```

There is a better way to set your **`RPATH`**. You can use **`$ORIGIN`**. **`$ORIGIN`** is a way of specifying relative paths. It is not an environment variable – it gets baked into the executable. The link loader will replace **`$ORIGIN`** with the directory containing the exe. So, for my installation of GDB, I just need to give it an **`RPATH`** of `$ORIGIN/../lib64`. Sounds easy? Wrong! Whilst the link loader doesn't look for **`$ORIGIN`** in the environment, the shell thinks that it is a shell variable and **`make`** thinks that it is a **`make`** variable.

binutils/GDB uses **`autoconf`** and a configure script. I wrote a shell script to run **`configure`** for the project.

Starting with something naive:

```
../configure {various arguments} LDFLAGS="-Wl,
-rpath,$ORIGIN/../lib64"
```

The **`-Wl,-rpath,`** bit is the parameter to tell g++ acting as the linker driver to pass **`-rpath`** to the link editor.

In the generated Makefile I get

```
LDFLAGS = Wl,-rpath,/../lib64
```

That's no good. My shell script has interpreted **`$ORIGIN`** as an environment variable and replaced it. OK, so I'll escape the dollar in my script, making it **`\$ORIGIN`**. Now the Makefile contains

```
LDFLAGS = Wl,-rpath,$ORIGIN/../lib64
```

That looks better, so I build GDB and ... same error. I can check what **`rpath`** (if any) has been built into gdb as follows:

```
readelf -d gdb | grep rpath
```

that gives me

```
0x000000000000000f (RPATH)
Library rpath: [RIGIN/../lib64]
```

OK, I got something. The next problem is that binutils/GDB uses a hierarchical configuration. Running **`configure`** in my build directory just generates a top level Makefile. Running **`make`** reruns configure for each subdirectory. The `gdb` subdirectory `Makefile` contains

```
LDFLAGS =  -Wl,-rpath,RIGIN/../lib64
```

That means that the recursive **`make`** has played the same trick on me, though this time it has interpreted **`$O`** as a **`make`** variable.

I could try to 'escape the escape' with **`\\$ORIGIN`** in my script. That won't work as the first escape only protects the second escape, and the shell will still replace the environment variable. I could try a triple escape. That gives me **`-Wl,-rpath,\RIGIN/../lib64`**. The problem is that **`\`** isn't the escape character for Makefiles. In order to escape a **`$`**, you need a second **`$`**.

So, let's try **`-Wl,-rpath,\$\$ORIGIN/../lib64`** in my script. That gives me **`LDFLAGS = $$ORIGIN/../lib64`** in the outer Makefile but just **`LDFLAGS = -Wl,-rpath,/../lib64`** in the inner Makefile. That looks like a shell replacement when the outer **`make`** runs **`configure`** for the inner gdb directory. Those dollars need protecting in the outer Makefile. I didn't think that it was the right thing, but I tried **`\$\$\$\$ORIGIN`** in my script. That gave me **`-Wl,-rpath,60598ORIGIN/../lib64`** in the inner Makefile. Definitely shell replacement where **`$$`** gets replaced by the PID. I think that's enough trial and error. Let's try to reason about it.

- To protect a shell dollar it needs to be preceded by a **`\`**.
- To protect a shell backslash it needs to be preceded by a **`\`**.
- To protect a make dollar it needs to be preceded by **`$`**.

I then spent a while looking at the flow from my script that runs **`configure`** to the final gdb binary. I wanted to understand that flow in terms of

Paul Floyd has been writing software, mostly in C++ and C, for about 30 years. He lives near Grenoble, on the edge of the French Alps and works for Siemens EDA developing tools for analogue electronic circuit simulation. In his spare time, he maintains Valgrind. He can be contacted at pjfloyd@wanadoo.fr

successive executions of **`shell`** and **`make`**, so that I could understand what replacements get done and what escaping is needed.

So there is:

1. The starting shell
2. Outer **`make`**
3. Outer **`config.status`** shell
4. Outer recursive **`make`**
5. Inner **`config.status`** shell
6. Inner **`make`**
7. Linker shell
8. Final gdb **`rpath`**

Along the way, there's some **`awk`** self modification of the Makefiles, but thankfully that doesn't need any escaping.

Working backwards and applying the 3 protection rules described above that means that the strings need to be

1. **`$ORIGIN`**
2. **`\$ORIGIN`**
3. **`\$$ORIGIN`**
4. **`\\\$\$ORIGIN`**
5. **`\\\$$\$$ORIGIN`**
6. **`\\\\\\\$\$\\\$\$ORIGIN`**
7. **`\\\\\\\$$\$$\\\$$\$$ORIGIN`**
8. **`\\\\\\\\\\\\\\\$\$\\$\$\\\\\\\$\$\\$\$ORIGIN`**

I'm glad that I didn't persist with the trial-and-error approach to finding that. So the big question, does it work? Yes! As long as I keep the gdb binary at the same position relative to `../lib64`, I can move the lower directories around to my heart's content. One disadvantage is that this approach may not allow in-place execution of the binary.

Let's hope that binutils/GDB never adds a third level of recursion. The backslashes grow by $2x+1$ for every extra level of shell, so two more shells would mean that leading group would need 63 backslashes. That really would be a 'fistful of backslashes'. ■

Reference

[Southoff16] George Southoff 'LD_LIBRARY_PATH considered harmful', posted on 22 Jul 2016 and accessed on 21 November 2023 at https://gms.tf/ld_library_path-considered-harmful.html

How to Write an Article

Submitting an article for publication might seem daunting. Frances Buontempo explains just how easy it is.

Who writes articles in *Overload*? Taking names from the list of *Overload* authors on the ACCU website gives around 250 authors, without breaking down joint authorship. The top three authors have been Sean Corfield, with 80, Alan Griffiths with 74 and Sergey Ignatchenko with 69 articles. Figure 1 (overleaf) shows a histogram of authors with 10 or more articles to their name.

A consistent pattern emerges of a few people who contribute again and again – most regular *Overload* readers have never written an article. Some readers may have a blog, or join in a discussion on accu-general, or discuss something technical over a beer with colleagues one evening, but never get as far was writing it up for the ACCU.

The majority of the articles published here are from ACCU members, but from time to time other people submit articles. Indeed, as a peer reviewed journal, we are open to submissions from anyone. One of the benefits of such a journal is the feedback process. The article can be improved before being read by the public at large whereas a blog gets the feedback after it is published. Don't forget being published in a peer review journal counts as a few extra kudos points.

Idea

How do you decide what to write about? Bear in mind writing is usually a learning process. Even if you think you are the world's leading expert on a particular field, writing it up clearly will find gaps in your knowledge or spark off new ideas.

It can be worthwhile to simply write a summary style article, with the latest thinking on a subject you are interested in, perhaps delving way back to its beginnings years ago or simply introducing a new language feature. This will get you, and your readers, up to speed with a way of doing something.

Some articles have started life with a question on a discussion group, like accu general. In fact, my first *Overload* article, 'Floating point fun and frolics' [Buontempo09], started there. It can make life easier to just have a trail of comments and ideas to summarise if you don't feel you have enough ideas yourself. Alternatively, you can present a new technique, library or even language you have developed yourself. Either way, the audience may expect to see a few references, so it is possible to do further background reading on a subject.

If you're not sure whether to submit to *Overload* or *CVu*, try both and see what happens. The main points to bear in mind are:

- *Overload* is freely available, so anyone may read it. You may want to just try something in *CVu* first time, but this is not a requirement.
- *Overload* tries to take a more academic tone, so might tend towards more technical content, with more references.

Something like 'My first 'Hello World' program in JavaScript' might be more suited to *CVu*, while something like 'Advanced C++17' might be more suited to *Overload*.

If you do not have a full blown article, but just a sketch of an idea, it is OK to get in touch for some early feedback. We might be able to give a few pointers of further things to research or other ways of doing things.

If you do want to submit a full-blown article, try to give it some kind of structure. Simply having an introduction, main work and conclusion is far better than sending in a list of bullet points.

Spare a little thought for your target audience. How much background might need fully explaining? What can be covered by a reference and leave them to go read up if required? We are always open to other ideas, including but not limited to letters to the editor, for example if a previous article has set you thinking.

Submission

How do you submit an article to *Overload*? The best approach is to use email: Overload@ACCU.org. An easy-to-copy format, like Open Office, Word, or just plain text is best, though other formats are acceptable. Any diagrams should be attached as separate scalable graphics, so they can be positioned and sized easily for the final layout.

If you are demonstrating code you have written, it might not need listing in its entirety. Enough listings to get the main point across often work well. It is sensible to add a link to the whole codebase, for example a github repository, if relevant, though.

We also like to have a short biography, with a contact email. Your readers may get in touch and say 'Thanks'.

Some journals offer a template, expecting submissions in a specific font, with a specific size, number of columns and so on. We don't mind – the formatting and layout will happen later. We won't fuss too much about length either. Approximately one thousand words fill a page. Anything from one page upwards is ok. If your article is a 20 page epic, it might make more sense to split in into two or more mini-articles. This will depend on how many other pages have already been taken up, and where we can find a natural place in which to insert a break. More details on the format and structure can be found in 'Guidelines for contributors' [Overload07].

Feedback

If your article looks like a plausible candidate, it will get sent round the review team, and you will be emailed back comments. We try to make sure we get a mix of positive encouraging comments, nit-picks over typos and grammar, and suggestions of unclear parts that may need rewording. On top of this basic style feedback, be prepared for the reviewers to point out something you may have missed, for example newer language techniques, more succinct ways of doing things, pre-existing libraries you

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD using AI and data mining. She's written a book about machine learning: *Genetic Algortithms and Machine Learning for Programmers*. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com

If you do not have a full blown article, but just a sketch of an idea, it is ok to get in touch for some early feedback

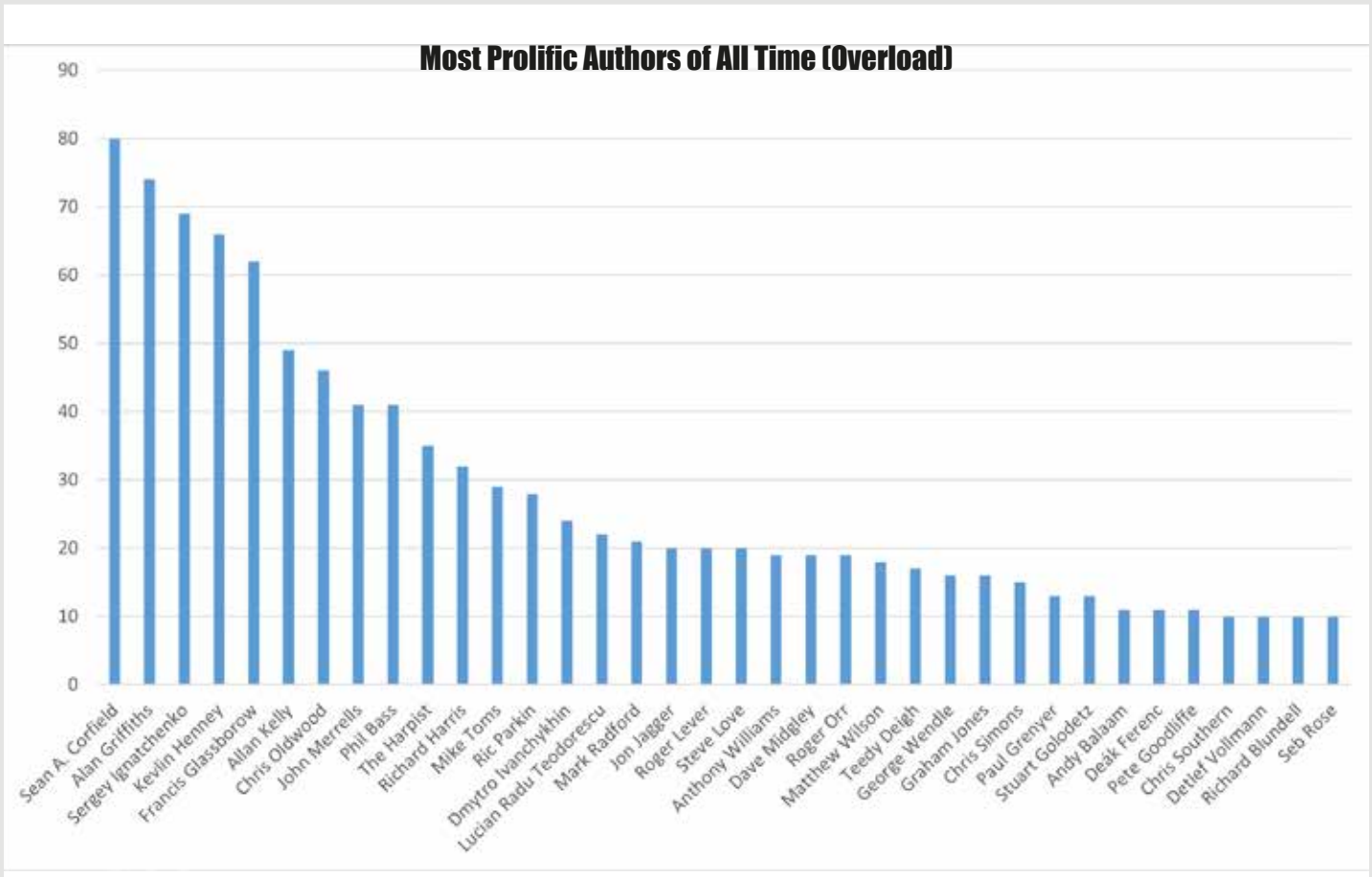

Figure 1

can use off the shelf. You are allowed to argue back, of course, but this process can make the articles more thorough and you may learn even more during the process. Once in a while, the only feedback simply says, "This is great." Not often, but it can happen. On very few occasions the potential author decides not to take on board the feedback, and the idea is taken no further.

Once all the articles have been reviewed they are sent to the production editor, and you will then receive a proof first-draft, showing the actual layout. At this stage everyone needs to keep their eye open for omissions, like second author's names, missing diagrams, copy and paste errors and other typos that have slipped under the net. Not matter how hard we try there always seems to be at least one in the final printed version.

Fame

Shortly after the drafts, the whole magazine will be pieced together. You are likely to see an announcement on accu-general, the accu.org webpage and possibly Twitter.

Obviously, if you are a member and have paid for it you will get a paper copy through your door at some point and can leave it lying around open on your desk to show off to all your friends and colleagues. If you aren't a member, you can ask for a printed copy – yours to show off and share with others. It might even persuade someone to join.

Almost nothing beats the sight of your name in print – try it. ■

References

[Buontempo09] Frances Buontempo 'Floating point fun and frolics' in *Overload* 91, available at http://accu.org/index.php/journals/1558

[Overload07] Overload 'Guidelines for Contributors' in *Overload* 80, available at http://www.accu.org/index.php/journals/1414

Afterwood

Halloween has been and gone. Chris Oldwood therefore takes time to consider the ghosts in the machine.

I always find writing an article at this time of year particularly challenging. Due to the lead time in publishing, we're currently celebrating Halloween, but by the time this hits the metaphorical shelves, we'll be enjoying the Christmas festivities. In the world of programming, this Halloween/Christmas duality is even immortalised in a joke that riffs on date formats and number bases to illustrate that 'OCT 31 == DEC 25'.

If you've ever worked with databases, you may have come across The Halloween Problem. This was an issue coined back in the 1970s, where a database update to adjust employee salaries used an index on the same column that was being updated. As each salary was increased, it caused the row to move further down the index and consequently was visited multiple times, meaning that every salary was bumped repeatedly until it exceeded the threshold specified for the raise. If you were one of the lowest paid employees in that company, then Christmas would definitely have come very early that year!

By the time I had started working with databases, all these issues had been worked out and the ACID guarantee had become a staple question at job interviews. It wasn't until the rise of the NoSQL movement in the late 2000s and the new breed of document-oriented databases like CouchDB and MongoDB that I started to become more aware of issues like The Halloween Problem, and its associated problems like Phantom Reads. The performance demands of 'Web 2.0 at scale' meant that the cool kids were happy to trade their ACID guarantees for throughput, and so the pendulum started to swing back the other way, and we also got to use the new excuse 'eventual consistency' whenever things didn't quite add up. In those early days, some database products traded off more than just the ACID guarantees. In the race to appear fastest in the benchmarks, they chose dangerous defaults which meant you couldn't even be sure if your request left the machine. The pitchforks came out, the 'NoSQL Considered Harmful' posts were written, and the pendulum swung back again towards the Pit of Success.

Even if you do manage to avoid the phantoms in your result-sets you'll struggle to escape one of the most curious perversions in relational database logic – the non-value `NULL`. After scratching your head wondering why your SQL query doesn't work as you expect when `NULL`s are present, you learn that one `NULL` does not equal another `NULL` and you need to litter your SQL code with `IS NULL`, `ISNULL`, `COALESCE`, etc. instead. Until, that is, you introduce certain aggregations, grouping, or sorting constructs, at which point `NULL` starts to feel like it does have equivalence semantics after all. But the SQL standards committee are a cunning bunch and with a little sleight of hand they sidestep the apparent similarities of equivalence by introducing the concept of 'distinctness' instead and any notion of equivalence remains merely a figment of your imagination.

If a database `NULL` is a value which doesn't exist then, in the world of floating-point numbers, the undead comes in the form of a nan – once a number, full of life, but now destined to walk the Earth turning every other number it meets into the undead, too. This one really is evil, though, as there is no standards committee to save you here if one creeps into your collection before you try and sort it. Depending on the language and sort implementation, you might be lucky and escape with a sequence that remains intact, whereas if you're unlucky, your sort won't complete until the heat death of the universe, making the outcome a moot point. I'm pretty sure IEEE754 wasn't what the late Fred Brooks had in mind when he warned us there was no silver bullet but maybe he also advised us to decorate our collection types with garlic in one of his lesser-known essays.

If you've dabbled in computing for even a small amount of time, you'll likely have experienced 'The Ghost in the Machine'. Like many real-world ghost sightings, they eventually get debunked. That elusive bug, which initially appears to be from another realm, turns out to be entirely real, and all too often self-inflicted. We might call it Undefined Behaviour to avoid frightening the children, but it's really a portal to another dimension where only those tooled up like Ash Williams will make it out alive. C++ in particular has the kind of power to contact the dead all too easy, although it tends to be hackers that celebrate its 'use after free' abilities.

Einstein famously coined the phrase 'spooky action at a distance' to describe the weirdness of quantum mechanics but you don't need to delve that far down the technology stack to experience the spookiness of hardware. Even though we're becoming less susceptible to the vagaries of some older technologies – like hard disk drives, aka 'spinning rust'– we are more reliant on network connectivity, meaning that Leslie Lamport's famous quip from the late 80s about distributed systems and being reliant on computers you didn't even know exist is becoming ever more prescient. Failing and loosely seated RAM chips also provide just enough of a distraction to make you question your sanity before declaring the host cursed.

The complexities of modern hardware and software can make it feel like you're constantly being haunted by a poltergeist as you struggle to reason why your code is not behaving the way you intended. Halloween might only be one day for normal people but for programmers it can feel like we're permanently living in The Upside Down. When you're battling with phantoms, daemons, and zombies, sometimes it can feel more like exorcism than programming. ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the comfort of his breakfast bar. He has resumed commentating on the Godmanchester duck race but continues to be easily distracted by messages to gort@cix.co.uk or @chrisoldwood

accu

Professionalism in Programming

Professional development
World-class conference

Printed journals
Email discussion groups

Individual membership
Corporate membership

Visit accu.org for details