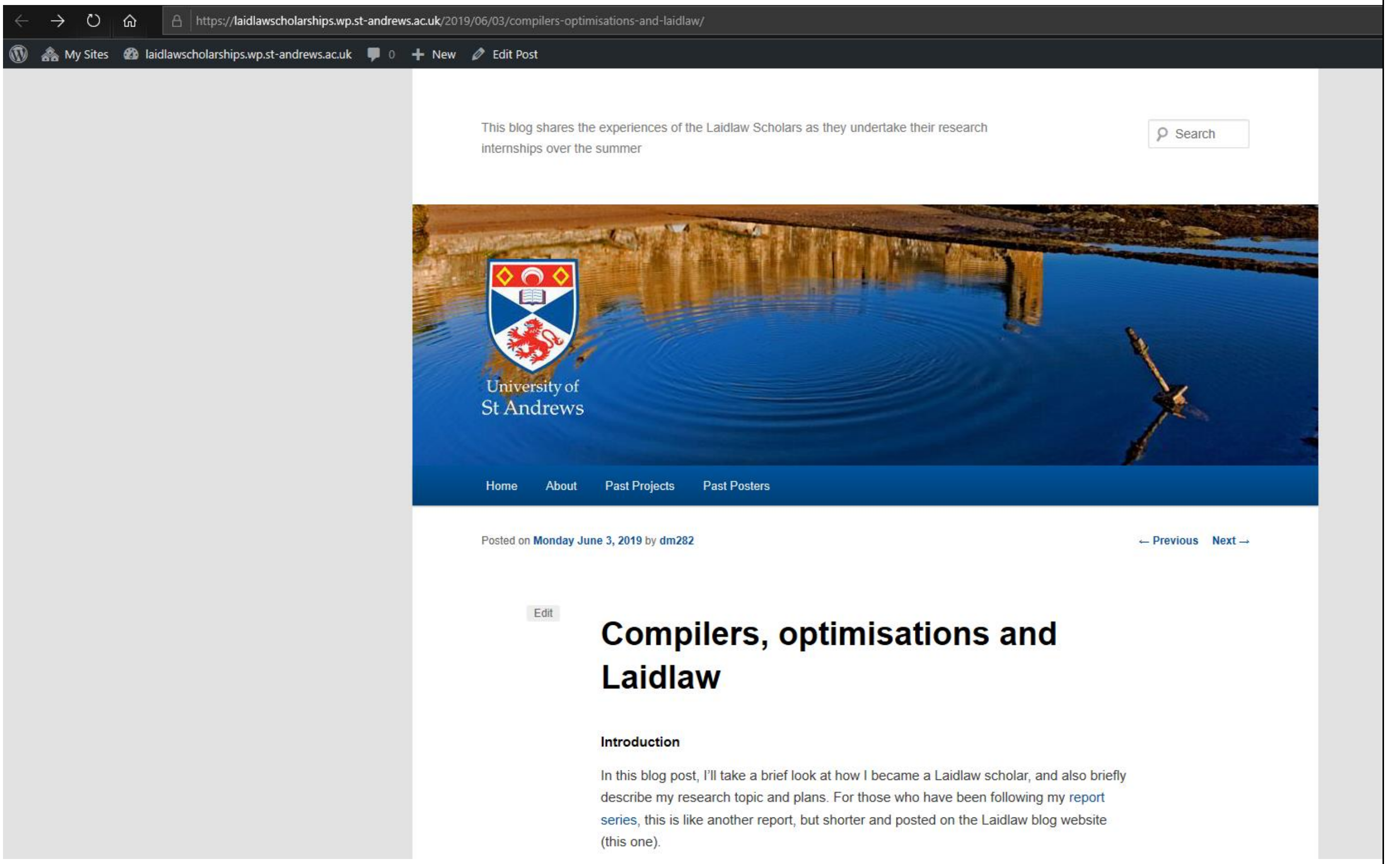
Report **5**

Compiling and optimising Laidlaw

Note:



https://laidlawscholarships.wp.st-andrews.ac.uk/2019/06/03/compilers-optimisations-and-laidlaw/
My Sites
laidlawscholarships.wp.st-andrews.ac.uk
0
New
Edit Post
Search
University of St Andrews
Home
About
Past Projects
Past Posters
← Previous
Next →
Edit

This blog shares the experiences of the Laidlaw Scholars as they undertake their research internships over the summer

Posted on Monday June 3, 2019 by dm282

Compilers, optimisations and Laidlaw

Introduction

In this blog post, I'll take a brief look at how I became a Laidlaw scholar, and also briefly describe my research topic and plans. For those who have been following my report series, this is like another report, but shorter and posted on the Laidlaw blog website (this one).

As part of my Laidlaw requirements, I published a blog post on the Laidlaw Scholars' blog just after my research started, and can be thought as an abridged version of the first section of this report. My blog post has parts about what I *thought* I'd do during the summer, and they were slightly different from the research I actually did this summer. I recommend that you read that blog post in addition to this report.

This report can be thought of as an extended version of that blog post that goes deeper into the technical side behind my research (and also before that).

Accessibility: The first part is intended for a general audience, and while such members are welcome to read Part 2, having a computing or technical background would be useful.

Version 1.0
Compiled on 04/07/19

Part 1: Getting there

Background

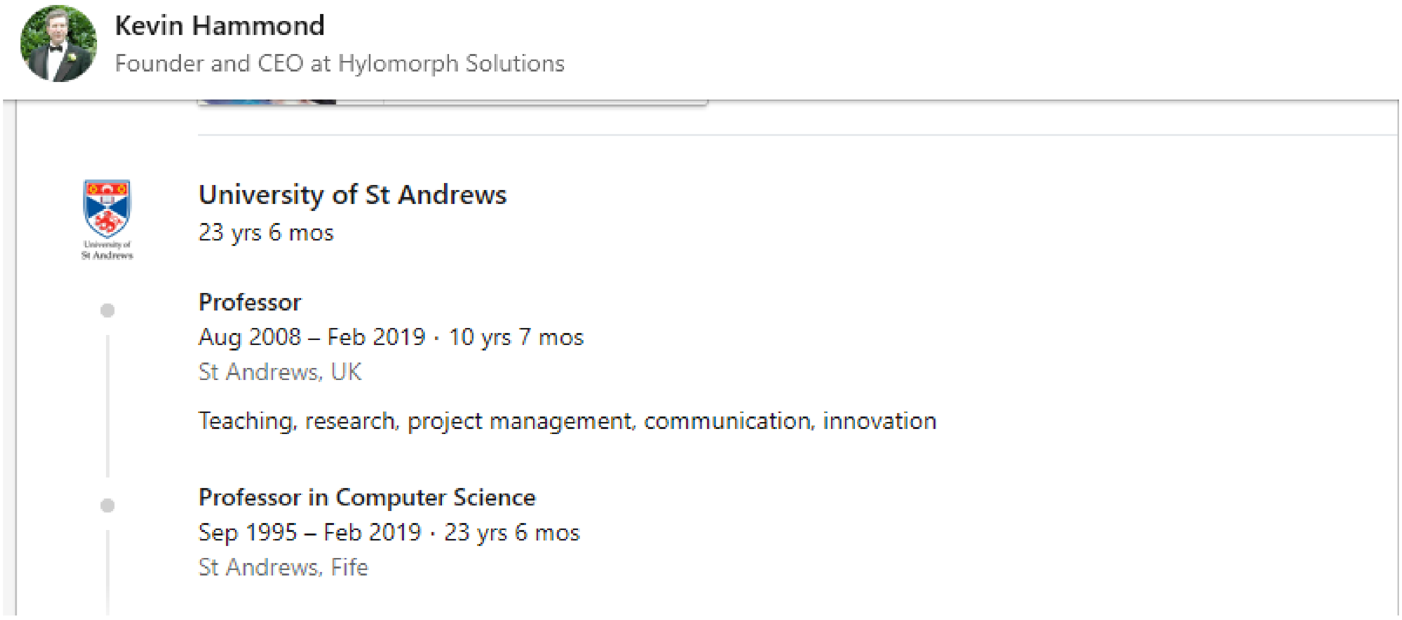
As a second-year student (at that time) in Computer Science and Mathematics, one of my major aims was to *try* to secure an internship. This did not go as well as I had hoped – generic rejections were common – and a common issue was that most of them explicitly preferred penultimate-year students (3rd year in my case).

The first time I got to know about this scholarship was through one of my fellow direct entrants in Computer Science (which was possible as there were only five of us). He had obtained the scholarship last year but repeated second year as he had decided to change programmes.
Soon after, there was an information briefing session, and then the list of pre-defined topics. But,

- There were no pre-defined topics available for Computer Science, and,
- Out of the Maths topics, one was about history, and the other was about fractals, and while that seemed vaguely interesting, a talk with that supervisor revealed that it was more suited for a third year student, and I was looking for something which more combined both my subjects.

So, I had to go for a student-defined topic. That was not very difficult, as I already had something in mind. The inspiration between the topic is explained in a further section.

But then the next challenge was to find a supervisor. I looked at the list of lecturers on the Computer Science website, and while my first instance was to consider the CS2002 lecturers (as that module had a significant component in C and Systems Programming, two sections that I thought my topic would be relevant at), I eventually short-listed two *other* lecturers, and then contacted the first one, planning to contact the second if the first one did not respond in time (I had about a week left then).



Kevin Hammond
Founder and CEO at Hylomorph Solutions

University of St Andrews
23 yrs 6 mos

Professor
Aug 2008 – Feb 2019 · 10 yrs 7 mos
St Andrews, UK

Teaching, research, project management, communication, innovation

Professor in Computer Science
Sep 1995 – Feb 2019 · 23 yrs 6 mos
St Andrews, Fife

Little did I know that he would be in the university for not much longer.

Fortunately, the first one responded quite quickly and was happy to supervise my work. With that done, the next step was to fill in the application (which asked me to write three essays). As usual, it took me a lot of time to write them, and I worked till 4 am to finish them (the deadline was 12 noon on the same day). *This remains the record for the latest time I've been awake as of writing.*

Then it was just waiting. It must be noted that at that time, it was like any other internship application: it had an entry in my internship application database, and the probability[1] was set as default. The risk of not being accepted anywhere was seriously considered, and I had no reason to believe that I would get this one than any other. When the Semester 1 results came out in late January (which they explicitly said would be considered), I was very unsure that I would progress: 16.47/20 wasn't the worst, but I knew other applicants who had done better.

But on 31st January 2019, I got an email which read:

> Further to your application, we would like to invite you to interview for the Laidlaw Undergraduate Research and Leadership Scholarship for 2019/2020. Interviews will take place on **Monday 11th February 2019 in the Pearce Room, Gateway building (on the lower level).** The interview will consist of a group task and one-to-one interviews...

And that meant that I had qualified for the second level, which was (and still is) considered unusual to me and had some importance – it was the *second ever* application which I had passed the first round.

I decided to take February 11th at 10 am. At the Gateway building, most of the candidates wore suits, but I didn't bother about that and simply continued to wear my trusty jacket, while wondering whether that would go against me. Soon after, it was my turn to get interviewed, and the very first question (from Alex Stanley) was:

"Define self-leadership"

I had heard of *leadership* before, but not "self-leadership", and hence had to quickly conjecture one, and answered that and the other questions to the best that I could. Note that there were none of the questions which I had anticipated for (like asking for my motivations in applying to the scholarship, maybe even asking about my proposed topic).

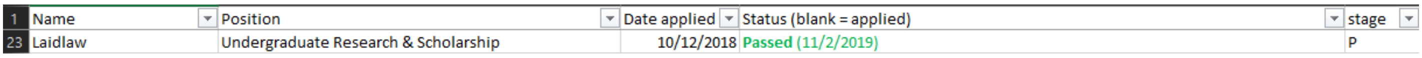
After that, I had to write a *self-evaluation* about myself, listing what I thought about my strengths and weaknesses in five minutes. Then I, along with some other candidates, were subjected to a group task, which was an interesting component that had links with boats, survival and teamwork. Overall, I liked my very first assessment centre experience, and didn't mind showing up 15 minutes late for the MT2505 lecture.

That very evening, when I was working at John Honey, I received an email which had my application decision (earlier than expected!). I opened the email, knowing well that not getting this would give me my second *2F* (second-level failure), and means that I have to continue applying for internships (which was a draining and time-consuming experience). Naturally I looked for keywords which would confirm this (think "unsuccessfully"/"unfortunately"). But this is what I got instead:

> Thank you for attending the interviews today. We are delighted to inform you that we would like to offer you a Laidlaw Undergraduate Research and Leadership Scholarship for **2019/2020**. Many congratulations on being selected. There was a considerable level of competition for these scholarships and we were very impressed with the standard of applications and the range of research projects that were submitted.
>
> The panel thought you made a very strong application...

-- ?? I actually got it! This is, and remains, the only *P* (there is no nP) in my application database:

[1] For stage S, $P(W) = \begin{cases} 1\%, & S = 1 \\ 10(S-1)\%, & S > 1 \\ 0, & S = nF \end{cases}$, where nF is a failure at stage n.



1	Name	Position	Date applied	Status (blank = applied)	stage
23	Laidlaw	Undergraduate Research & Scholarship	10/12/2018	Passed (11/2/2019)	P

That day, my internship application machine shut down, and I were to apply for very few (mainly special cases) after that.

Pre-internship activities

The welcome email noted that there would be a launch event on 6th March, along with the "first leadership weekend" for two days (?). I had little to do after that, other than having an initial meeting with my supervisor.

The launch event was at a hotel and featured a talk by the Principal (which was quite interesting), and a presentation about research ethics normally meant for PhDs, with lunch also being provided. Oh, and the supervisors were also present (which we weren't told beforehand). Overall, it was interesting.

And about the leadership weekend? Well, even though some people leaked the location to me beforehand, we were effectively kept in the dark about where we would be for that weekend (except that it was outside St Andrews). For the benefit of future scholars who may be reading this page, no further information about that weekend is provided here (other than the below photograph).

The other things that I had to do were to apply for summer accommodation, and many scholars had to complete a risk assessment if they were travelling (didn't apply to me). Additionally, I had to complete a few Epigeum (online research courses), which had quite a few snags. To start with, I decided to do it just after my second semester exams finished, but the system was demanding payment to do the courses when this clearly wasn't supposed to be the case. It took quite some back-to-forth communication between the Epigeum team, Laidlaw team, and me, was not an isolated issue and seemed to do with some migration of their website resources. Once the snags were resolved, I finished them in a few hours.

Accommodation

This broadly fell into two categories. Some used private accommodation (either by staying in the same flat they were staying term-time, by renting one, or on one case staying in the house of another scholar), while others took the university accommodation route, which is what I went for. This involved applying though a form (and annoyingly while they said that they'll have a list of the scholars, they *still* wanted my supervisor to write a note to that effect – why?). Disappointedly, the only options were DRA (ensuite but self-catered) and Fife Park (*not* ensuite and self-catered) – I was hoping that there would be something not DRAFP and closer to Jack Cole *or* town. But amongst the two, the latter was significantly cheaper, so I picked that.

Their offer came some time later, and it was weird. I requested accommodation from June 1st to July 8th, but their response was:

- Stay in DRA (*at a different building!)* from 1st to 7th June
- Stay in Fife Park from 7th June onwards

I suspect this was due to the existing occupants having to vacate, but this meant that I had to follow a convoluted arrangement instead. Why not just allow me to stay at my existing Lindsay room for one more week? I don't quite know why, but I suspect those rooms were to be used as a hotel.

This meant that I had a difficult time moving all my stuff from Lindsay to Nansen (where my room for a *week* would be at) on May 31st, and I was pretty tired by the end of the day. I did not arrange any of my stuff back in anticipation for the second phase of moving.

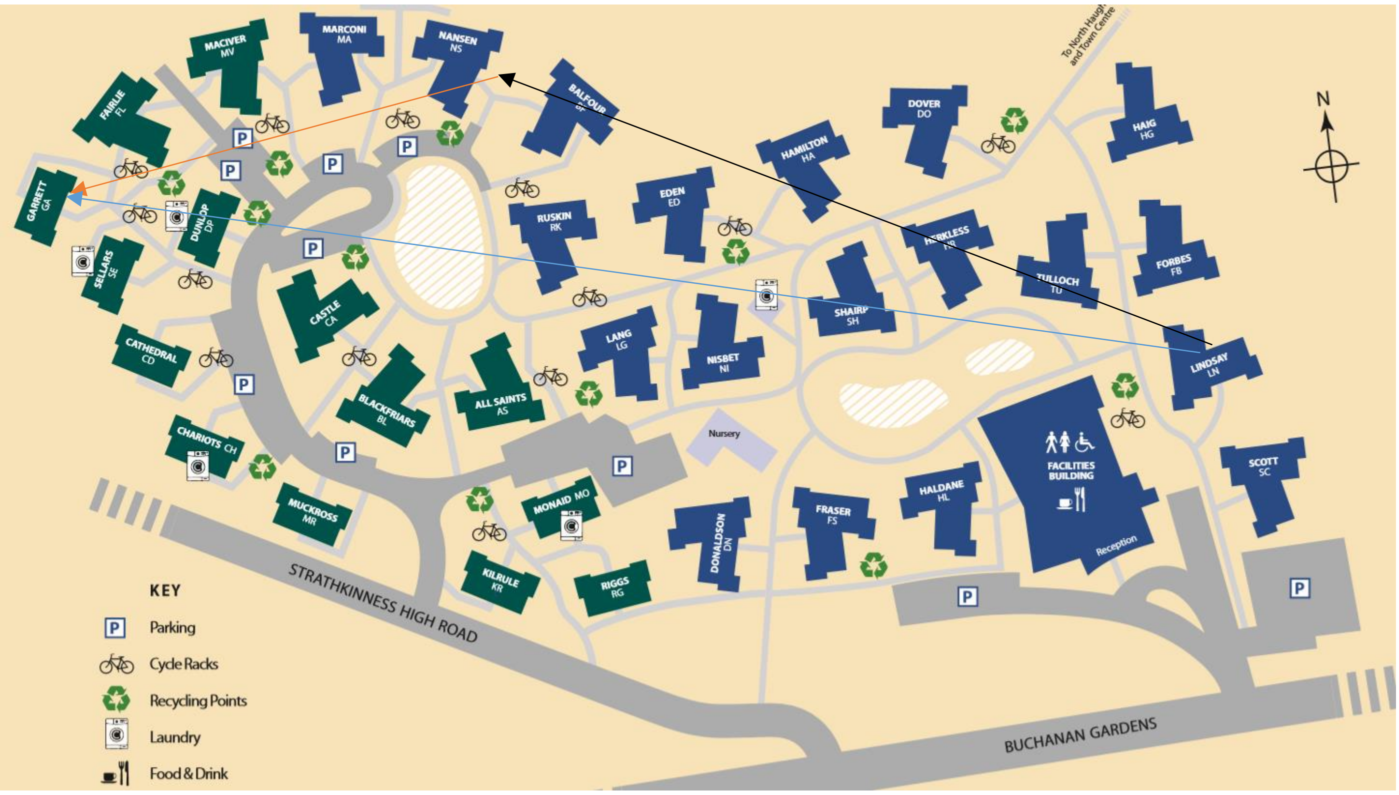
On 7th June, I moved my stuff all over again from Nansen to Garrett (in Fife Park). This was easier than the first time, partially because all my stuff was in a 'packed' form ready to carry, and partially because of the shorter distance. I finished this phase in around 4 hours total, less than the first phase.

And how's Fife Park? Pretty standard. It is not ensuite, so there are two bathrooms for every five occupants. And having previously lived in ensuite accommodation, I don't quite get the clamour that some students have for having an attached bathroom. It's hardly bad – there was a free bathroom most of the time – and the most I had to otherwise wait was 10 minutes[2]. The kitchens are mostly the same.
Garrett, being one of the newer buildings, uses the matriculation card for authentication, which eliminates the need for carrying separate cards like I had to at DRA (however, I could not figure out how to keep the door unlocked on demand like one can at DRA with a double-dip, which was very annoying).

My flatmates were all Laidlaw scholars except one who was undertaking a Statistics internship with the school – three (including myself) of which followed the convoluted arrangement (the other one came later) – and they were fine as expected.

But all this meant a longer walk to Jack Cole and anywhere else in town. DRA/FP is already not near *anywhere*, and Garrett is only further (click map for source):

[2] And they are usually cleaned externally from what I have heard – even though ours wasn't, the bathrooms were usually clean. I much prefer this to the inconsistently strict room/bathroom inspections that otherwise happen at DRA.



Black: Lindsay to Nansen; Orange: Nansen to Garrett; Light blue: Lindsay to Garrett. This is on top of the 12-minute walk to Jack Cole from Lindsay. With this, it took around 24 minutes to go from Garrett to the Main library, and I'm generally told that I walk faster than average.

During the summer

Research is not the only thing us scholars have to do. For instance,

- Every odd week, we have *leadership sessions*, which are held in a large room and feature talks related to leadership development, with related activities. It is a good place to chat with other scholars as well.
- On the other weeks, we have to complete *action learning sets*. They are approximately one-hour discussions with a few other students (this is pre-allocated), and one of the students have to write a report and upload it on Moodle (another keeps tabs on the meeting), with others describing the issues they have, and discussion about that follows.
- Us Laidlaw scholars have a 2019 Facebook group, wherein we organise *Cake Friday* (but I **hate** cake!) and other related informal activities. This included celebrating a scholar's birthday.
- We need to write a leadership essay (not published here, nor is this that one) and upload a video (this should be coming soon as of writing!)

It should be noted that there are no exceptions with regards to the former two; if one can't attend the leadership session, an essay on that session (which is recorded) would have to be submitted instead.

Action Learning Set in the library on 24th June 2019. The member on the left is barely in the picture. I am not in the picture being the photographer.

"Cake Friday" on 28th June 2019 – which was held outside at the garden in St Salvador's Quad as opposed to the library the former two times – and the scholars look fairly contented with vegan cake and strawberries, amongst others.

The third leadership weekend at Parliament Hall, where the topic was about video editing (and us scholars must make one!).

Getting the topic

Now I go back to explain what led me to pick my research topic. The main inspiration was from my benchmark program that I first made in August 2016 when still at school. In that, I had noted that on Visual Studio 2017 (Release; v141), the performance between different machines weren't normal at all: a CPU which is usually thrice as fast was only about 15% on my program (in fact, it seemed to vary with clock speed *alone*!). What confused me further was that on Visual Studio 2008, also with the Release config, the performance increased as expected. The below table shows some sample scores on VS2017:

Device	Processor	Max (turbo) clockspeed (GHz)	Core/Thread count	Benchmarker score
RM Education	Intel Atom	1.8	2/4	122.3
Acer Iconia W3-810	Intel Atom Z2760	1.8	2/4	136.7
Surface Pro 4	Core i5-6300U	2.9 - 3	2/4	1407.4
MacBook Air 2013	Core i5-4250U	2.3 - 2.6	2/4	1169.7

Dell Inspiron 6400 **(reference)**	Core2 Duo T7200	2	2/2	1050
School computer 1	Core i7-6700	3.7-4	4/8	2013
University computer 1 [PC5-*x*]	Core i5-6500	3.3-3.6	4/4	1744.8

The *score* was computed by the program using the following formula:

$$S = 40\sum_{i=1}^{5}\left[\sum_{j=1}^{5}\frac{R_j}{T_j}\right]$$

T_j is the time taken to compute each stage: the program would compute the time taken to find the first n Pythagorean triplets [note: $j: \{1,2,3,4,5\} \to n: \{100,250,500,1000,1500\}$] and then normalise with R_j, the time taken by the reference computer. This was repeated fives time to reduce variability. While some of the stages could take as little as one millisecond, this was carefully monitored and compared with the other stages to ensure that did not affect scores.

But why is this happening? And what happened if it was, say Java code (I had done something similar in CS2101)? Or a different program? What factors *could* affect the performance? That's what I wanted to investigate.

Part 2: The research summer

Non-research activities are covered in Part 1 under the section *During the summer*, and this section is arranged in a (roughly) chronological order.

Note: Please see the Summary section if you are looking for an overview of what I did in a simpler format.

The beginning

I started by undertaking some preparatory work by running my program on Linux compilers and see what they get – after all, I had tested it only on Windows up to now. The results showed that most of the performance gains occurred between `/O0` (optimisation disabled) and `/O1` (core optimisations enabled), and then thinned out from there. (On Windows, the Debug configuration is `/O0` and Release configuration is `/O2` (`/O1` has a different meaning on Windows and is equivalent to `/Os` on Linux – there are still performance gains, but not as much as the code is optimised for size rather than speed)).

Compiler setting	g++ [O0]	g++ [O1]	g++ [O2]	g++ [O3]	VS2017 [O0]	VS2017 [O2]
Score[3]	218.2	1588.3	1591.6	1595.4	74.4	1598.3

We can see from the table that while the Debug configuration in VS2017 scores substantially lower than

[3] Tested on a CS PC3-x lab machine, with a i5-4440 clocked at 3.1 GHz and with a 0/1/1/2 turbo boost configuration for 4/3/2/1 active cores.

their Linux counterparts, this no longer becomes the case in the Release configuration. While I don't have a concise explanation for this, I tried running on the Debug configuration again, but with the `/Ob2` flag set. This improved the scores back to `/O0` in Linux but note that it is *not* a sufficient reason by itself.

A potential flaw was also discovered for the shortest test case. The time precision in Windows is only 1 ms ($10^{-3}\ s$). And while Linux can distinguish between finer periods of time, this meant that if the time taken was less than 2 ms (or 1.5 ms?), Windows would report it as taking 1 ms (half the time!). The maximum scoring error would be

$$\frac{10^{-3}}{10^{-3}} * \frac{1500}{5} = 300 \text{ points total (or 60 points for each case)}$$

But in reality, it alternated between 1 ms and 2 ms, which reduced the actual error. But this was monitored carefully by also comparing the scores with the time taken to complete the longer cases, and the difference was low.

This will however mean that the benchmark will have to be rescaled in the near future, as the risk of it taking too little time (< 1 ms) is real – in which case it will report as ∞ and hence not give a total score.

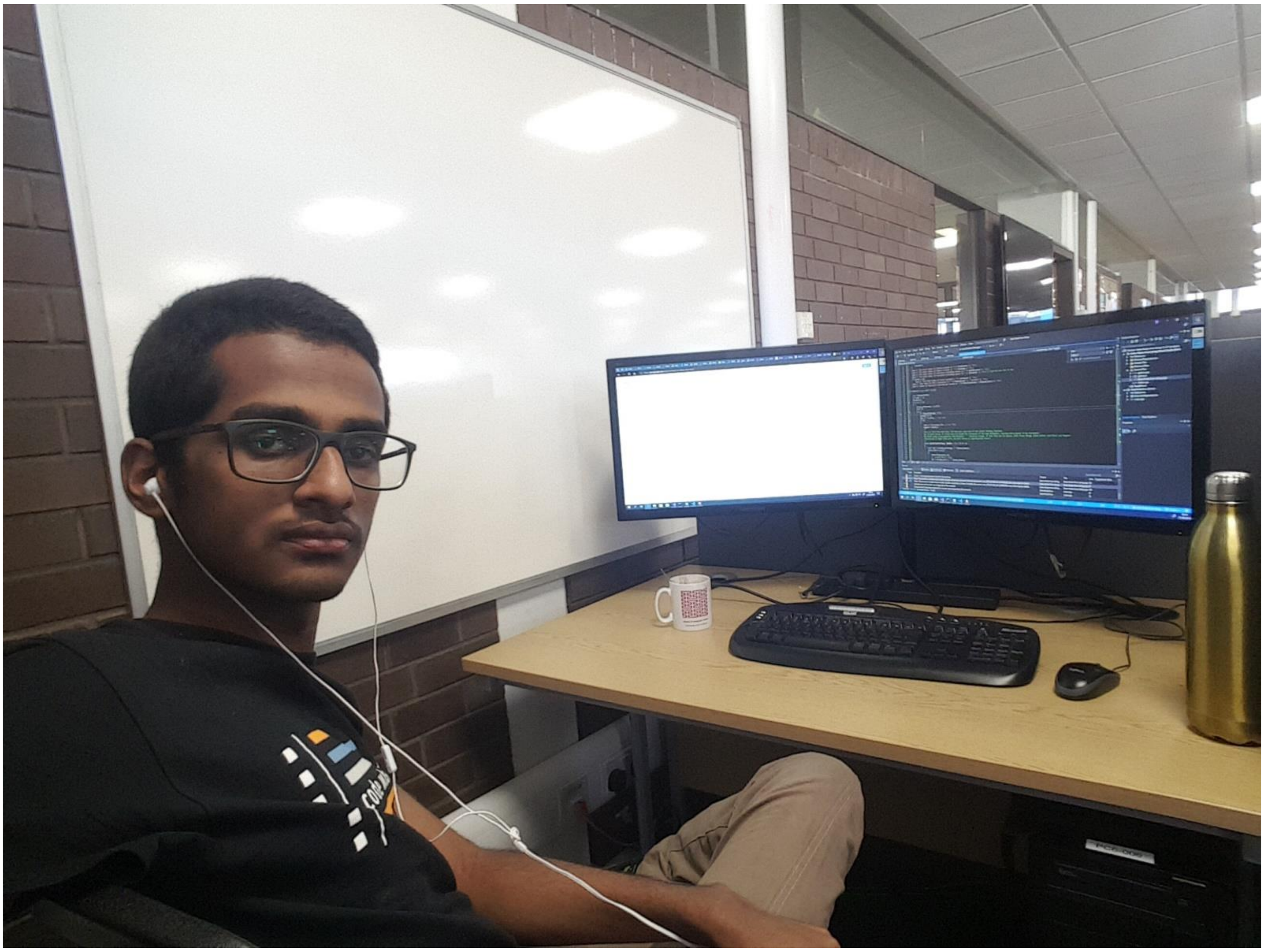
Role of fixit

When I was mass-running benchmark tests on Linux, I wanted to do the same on Windows. As it is impractical to have to run it individually each time, I remotely ran the tests on Linux using SSH (and with the `tmux` multiplexer for longer tasks). While all lab machines have Windows, RDP (Remote Desktop Connection) is disabled, and hence I can't remotely run tests there! This proved to be annoying as I could not do any work from my room, for instance (and neither could I run long tasks for fear of someone restarting the machine). So I contacted fixit (the technical support team of the CS department) to request them to allow me to remotely connect to the Windows lab machines.

They denied the request with the reasoning that it was useless as the Windows lab machines were configured to sleep after 30 minutes (this is true) but mentioned that they could set up a separate machine for me instead. I responded in the affirmative, but then the next response was literally to ask me when my internship ended (why?). This made me think that they weren't going to approve my request, and I had to do something else in the meantime (which I'll be explaining in the next section).

But then a week later, to my surprise, they set up for me a clean PC6-*x* machine which was configured to be RDP enabled. The CPU was a i5-4460, which was literally a PC3-*x* CPU with a 100 MHz higher base clock (the turbo configuration remained the same at 0/1/2/2). So I could *almost* emulate the i5-4440 by using ThrottleStop (or an equivalent configuration in the BIOS) to change the turbo configuration to 0/0/1/1 – almost the i5-4440, but I could not reduce the base clock nor can one set a negative turbo config (which would not make sense). But all my experiments were single core, so this did not matter much in the end.

The presence of this PC ended up being very helpful, more than just cross-platform comparison. For example, when I had to give my laptop (i.e, my workstation) for repair, I did all my research work on this PC – with the bonus that I could work from anywhere – an university computer or even my phone (which worked better than expected even at the phone's native 2560x1440 resolution!). Additionally, I did not have to carry my laptop anymore.



A regular day working on the dedicated PC.

--

Later on, I was facing another issue which held up testing on the Macs for some time. To start with, SSH was disabled on the Mac, and worse they slept after 10 minutes, stopping my run which could take hours! While they denied the SSH request on the basis that my use-case was unsupported and that they were explicitly meant for Mac/iOS development, they gave me an interesting alternative – `caffeinate` [man page] – which I had never heard of before. It blocks the machine from sleeping when this is prefixed before my testing run and was just enough for what I needed.

To summarise, fixit has been reasonably helpful with my demands which could be regarded as somewhat hard for a mere second year undergraduate, and especially considering that nearly all of my requests made during term-time have been turned down (which was what made me apprehensive this time around).

Looking at Linpack

Now that I had completed measurements with regards to my program, my supervisor suggested that I compare the results with that of a more well-known benchmark program. So I settled for Linpack, an open-source benchmark program which uses Gaussian elimination of matrices as its metric.

The results were a bit unexpected. On Clang, it largely followed the trajectory followed by my benchmark program (i.e, a major jump in performance between `/O0` and `/O1` with the improvements thinning out from there).

However, this was *not* the case for g++:

Machine	g++ [O0]	g++[O1]	g++[O2]	g++[O3]
PC5-x	1142	4339	4370	9370
PC3-x	952	3972	4081	7635
Lyrane	570	3035	3319	5400

Note: The metric is MFLOPS (megaflops per second).

We see a similar trend for all but the `/O3` flag, where the performance suddenly jumps up – and in fact, varies properly with actual performance improvement like `/O0`! At least one of the specific optimisation flags between `/O2` and `/O3` must somehow be responsible for this increase, which I haven't observed before. But what could be the reason? And what exactly causes the massive performance jump between `/O0` and `/O1` anyway (which held true in all cases)?

The -*f* optimisation flags

What actually happens between `/O0` and `/O1`? Actually, the compiler sets a series of *flags* which are enabled when the optimisation is set when compiling. But what exactly are the flags?

The documentation for this was poor. The gnu website mentions the flags which are switched on between each optimisation stage. So, then I have to iterate through each flag to find out which flag(s) is/are responsible for this increase.

This meant that I had to write a C++ program invoking a lot of system calls to automate this process. The full code is not shown for posterity and is available on GitHub under the *Benchmarker* repository.

```
for (int i = 0; i < opt1 * 1 - 1; i++)
       {
             string str = "g++";
             str = str + " -o " + "bench " + "benchmark.cpp " + optimisations[i % opt1];
             const char* c = str.c_str();
             system(c);
             string str2 = "./bench ";
             str2 = str2 + argv[2] + " " + optimisations[i % opt1];
             const char* c2 = str2.c_str();
             system(c2);
       }
```

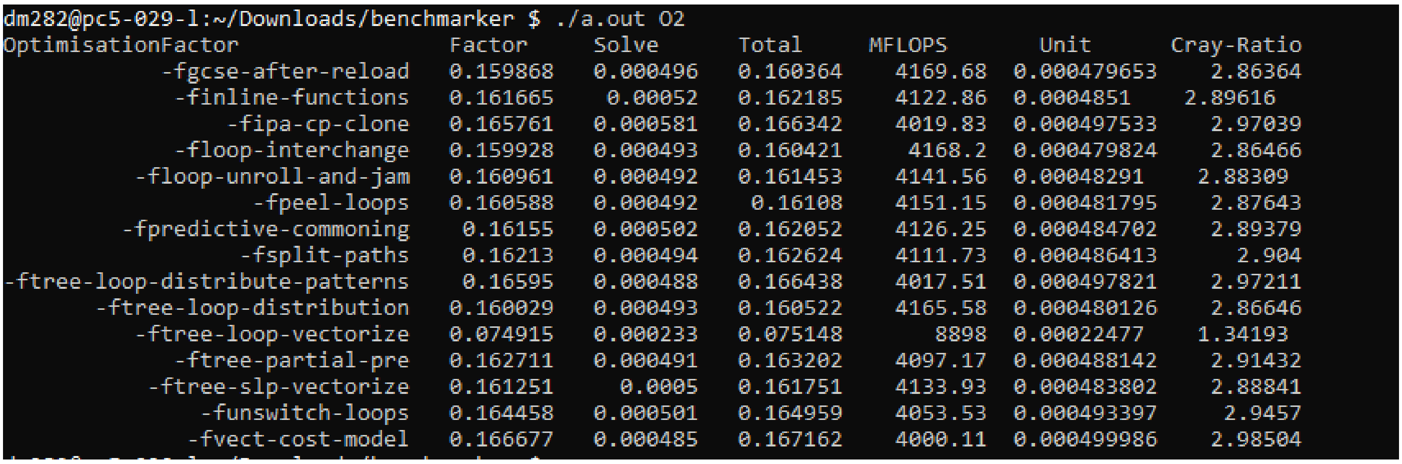
Note: opt1 is the length of the character array; the number of optimisations enabled between `/O0` and `/O1`.

This also meant that the original Linpack benchmark had to be modified so that flag comparisons could be easily done. But they are two separate programs, with their own stdout, so I printed them to stderr instead and configured the outputs carefully for that reason.

But the results were strange.
No change. None.
The benchmark scores that came out where so consistent that I was wondering whether there was some flaw in my code (perhaps the unoptimized version was being run instead?).

It turned out that g++ forcefully disables them at `/O0` or `/Og` – so the -*f* optimisation flags themselves have no effect in this case. Yes, it was in the documentation. Yes, I missed it for some time.

With this disappointment, I reran the tests with the `/O1` and `/O2` configurations. As expected, there was no change for each individual flag between `/O1` and `/O2` (as the overall score hardly changed to begin with), but that was not the case with one flag between `/O2` and `/O3`:



```
dm282@pc5-029-l:~/Downloads/benchmarker $ ./a.out O2
OptimisationFactor                Factor     Solve      Total     MFLOPS       Unit      Cray-Ratio
            -fgcse-after-reload   0.159868   0.000496   0.160364    4169.68  0.000479653    2.86364
             -finline-functions   0.161665    0.00052   0.162185    4122.86  0.0004851    2.89616
                 -fipa-cp-clone   0.165761   0.000581   0.166342    4019.83  0.000497533    2.97039
             -floop-interchange   0.159928   0.000493   0.160421     4168.2  0.000479824    2.86466
          -floop-unroll-and-jam   0.160961   0.000492   0.161453    4141.56  0.00048291    2.88309
                   -fpeel-loops   0.160588   0.000492    0.16108    4151.15  0.000481795    2.87643
         -fpredictive-commoning    0.16155   0.000502   0.162052    4126.25  0.000484702    2.89379
                  -fsplit-paths    0.16213   0.000494   0.162624    4111.73  0.000486413      2.904
-ftree-loop-distribute-patterns    0.16595   0.000488   0.166438    4017.51  0.000497821    2.97211
       -ftree-loop-distribution   0.160029   0.000493   0.160522    4165.58  0.000480126    2.86646
          -ftree-loop-vectorize   0.074915   0.000233   0.075148       8898  0.00022477    1.34193
             -ftree-partial-pre   0.162711   0.000491   0.163202    4097.17  0.000488142    2.91432
           -ftree-slp-vectorize   0.161251     0.0005   0.161751    4133.93  0.000483802    2.88841
               -funswitch-loops   0.164458   0.000501   0.164959    4053.53  0.000493397     2.9457
              -fvect-cost-model   0.166677   0.000485   0.167162    4000.11  0.000499986    2.98504
```

The cat's out of the bag! It's loop vectorisation that makes all the difference between `/O2` and `/O3` for Linpack!

I also took some time to try to find out *why* exactly loop vectorisation made such a large performance improvement. This blog is a good source that explains the impact of vectorisation and inlining of functions.

Now I tried to do the same with my benchmark program to determine whether a similar phenomenon could be deduced (at least for `/O0` to `/O1`). Sadly, the scores were consistent across the board. A bizarre problem occurred when I tried adding *ALL* the flags between `/O0` to see whether I would get `/O1`-level scores. And I did not (!!). [Post-script: they make no effect, see above]

Then I tried to start with `/O1` and then *disable* each flag selectively, and the scores were the same. This made me wonder: there is something external (which is not documented by the GNU website) that actually makes the big impact between `/O0` and `/O1`, but what is it? Also, there wasn't any reference on the GNU website which told me how to *disable* an optimisation flag – it took a StackOverflow post for me to find out. For reference, one replaces the prefix `-f` with `-fno` to achieve it on `g++`.

But what about Clang? That's a whole different story.

Clang and LLVM

To start with, the g++ optimisation flags did not apply to clang, so I had to look up the exact flags which make the difference between `/O0` and `/O1`.

But it took this excellent StackOverflow post to answer the question – the official websites of both Clang and LLVM (which this compiler is based upon) did not help at all – they didn't tell anything about /*Ox* other than they were optimisations themselves! Come on… and it only became worse as I found *NO* reference anywhere on the internet as to how a flag should be disabled – not even on StackOverflow! But `/O1` *disables* some flags, so how do I do it myself? As of writing I don't know.

I decided to test based on only the flags which would be *enabled* themselves as a result of this. But even that was difficult. To start with, most of the optimisations occur not by Clang, but LLVM – this means that passing the LLVM optimisation flags to Clang would not work. Hence actually doing that involves splitting the compilation into three steps: one to convert to LLVM, the second to actually apply the LLVM optimisation, and finally make Clang compile with that LLVM-optimised code:

```
string str = "clang";
str = str + " -emit-llvm -S -c linpackm.cpp -O0";
```

```
const char* c = str.c_str();
string str2 = "opt -S -o lin.ll linpackm.s ";
str2 = str2 + optimisations[i % opt1];
const char* c2 = str2.c_str();
string str3 = "clang lin.ll -o lin -lstdc++";
const char* c3 = str3.c_str();
cerr << setw(31) << optimisations[i % opt1] << "  ";
system(c);
system(c2);
system(c3);
system("./lin");
```

And then I ran the tests again. No difference! Worse, when I set /O1 in opt, I got /O0-level scores! That confused me as I again had to consider the possibility that my code was incorrect.

Turns out that my code indeed was not running at all and the scores were coming from an old copy (which was the reason I added code to clean after every run). But that was not all. There was no indication that the code did not run – it was silently failing but the errors associated from that never showed up in the terminal, which explains why it took quite some time before I realised what was going on.

But even after the bugs were fixed, I was still getting /O0-level performance. So I tried adding the flags in clang instead. Here we see that order of the arguments matter. This did *not* work:

```
clang -O1 -emit-llvm -S -c linpackm.cpp
opt -S -o lin.ll linpackm.s
clang lin.ll -o lin -lstdc++
```

But this *did* make an impact:

```
clang -emit-llvm -S -c linpackm.cpp -O1
opt -S -o lin.ll linpackm.s
clang lin.ll -o lin -lstdc++
```

The scores were also unusual: at 2200 MFLOPS (/O1: 3950) for a PC5-*x* PC, it was somewhere between /O0 and /O1. This suggested that some of the optimisations were occurring at the first stage, but not all. But how do we then extract the other optimisations in effect? Apply the flag on the *last* clang step as well!

```
clang -emit-llvm -S -c linpackm.cpp -O1
opt -S -o lin.ll linpackm.s
clang lin.ll -o lin -lstdc++ -O1
```

With this, we get /O1-level performance. But then is it possible for us to find out how much of an impact the optimisations on the last Clang step make? No, as the performance drops to /O0-level scores if /O1 is attached to *only* the last Clang step:

```
clang -emit-llvm -S -c linpackm.cpp
opt -S -o lin.ll linpackm.s
clang lin.ll -o lin -lstdc++ -O1
```

I'm not sure whether this means that the presence of setting optimisation on the first clang step is a pre-requisite for optimisations on the second clang step to take effect. Also setting the flag on opt and the second clang flag isn't enough either – this raises the question on whether setting /O1 on LLVM does anything. We could check this.

```
dm282@pc3-039-l:~/Downloads/benchmarker/clangflagtests $ clang -emit-llvm -S -c
linpackm.cpp
dm282@pc3-039-l:~/Downloads/benchmarker/clangflagtests $ opt -O1 -S -o lin.ll
linpackm.s
dm282@pc3-039-l:~/Downloads/benchmarker/clangflagtests $ diff lin.ll linpackm.s
1c1
```

```
// some changes with the declare lines
2639c2639
<   tail call fastcc void @__cxx_global_var_init()
---
>   call void @__cxx_global_var_init()
```

Most of the changes are with the declare lines (which are not shown above). While a few changes are interesting (for example tail-call optimisation), the core attributes aren't different, which made me wonder whether anything is changing.

Now we go back to the core question: what about the individual flags themselves? With Linpack, I could not find any particular flag that could be responsible by themselves in any of the optimisation flags. But with my benchmark program, while the results were the same for `/O0` (Clang's `/O1` performance is same as `/O0` for Linux), the cat can't hide any longer when we test the flags between `/O1` and `/O2`:

```
dm282@pc3-039-l:~/Downloads/benchmarker/clangbenchmarktests $ cat test.txt
-inline 1633.157104
-mldst-motion 238.439911
-gvn 240.046371
-elim-avail-extern 239.578308
-slp-vectorizer 235.590912
-constmerge 236.118927
```

It's function inlining this time – and while that is somewhat relevant to loop vectorisation as was observed in g++ on Linpack – they aren't exactly the same (partially because the process of inlining does not vectorise a function, and also if that was the case, we should have seen this on g++).

Now we have some results on Linux, but what about Mac? That's for the next section.

On the Mac

To start with, I tried to run the same g++ optimisations on the Mac, but they instead returned errors saying that the optimisations weren't supported. A closer look explains why that is the case. On the Mac, the g++ compilers use the LLVM backend, meaning that they operate much closer to clang on Linux with regards to the functionality and optimisation used. So, g++ and clang can be used interchangeably for the purposes of this section.

So, we first take baseline measurements for the default optimisation settings:

Machine	g++ [O0]	g++[O1]	g++[O2]	g++[O3]
PC5-x	1142	4339	4370	9370
PC3-x	952	3972	4081	7635
Lyrane	570	3035	3319	5400
Mac [i5-5675R]	1126	4067	4032	4096

The `/O0` score is very close to PC5-*x*, but nothing for concern otherwise.

Before going any further, it is important to take a look at the configuration of the Mac (mac2-*x*). The CPU is a i5-5675R, which is clocked at 3.1 GHz, but with a better 3/3/5/5 turbo configuration. This means that it is closer to the i5-6500 on the PC5-*x* configurations (3.2 GHz, 1/2/3/4) than the PC3-*x* CPUs in theory. However, the other major difference is the presence of a L4 cache. The R and C series on Intel desktop CPUs typically come with an Intel Iris GPU, and such models are configured with a shared 128 MB L4 cache which can also be used by the CPU. It can boost performance in some cases, so this could make some of the analysis and measurements a bit challenging.

Testing with each individual flag mirror the results obtained for Linux – no difference with `/O0->/O1`, but loop vectorisation making all the difference for `/O1->/O2`.

How do the scores compare when my Benchmarker program is run instead? Actually, the scores do not follow any of the usual trajectories:

Optimisation	g++ [O0]	g++ [O1]	g++ [O2]	g++ [O3]
Score	107.7	1006.6	2243.6	2246.7

So the initial score is pretty low (like VS 2017's Debug), jumps up to scores which are not quite `/O1` when the `/O1` optimisation is applied, but then we see a large performance improvement between `/O1` and `/O2` – so much that a score of 2243 is the highest ever score recorded (and beat an i7-7700K clocked over 4 GHz which was the previous record-holder). Again, it is difficult to ascertain whether the presence of a L4 cache could play a role here; what I do remember is that on a MacBook Air 2013 (i5-4250U which does *not* have a L4 cache), the scores obtained on the Release config using XCode and CLion on macOS were comparable to the scores obtained under Windows. To test this would involve configuring the Macs with Bootcamp on Windows or finding an equivalent PC with an L4 cache, both options which seem very unlikely as of writing (and I don't want to bother fixit too much; I think they are annoyed enough with my current requests).

I also tested the flags to see if the reasons for the large performance improvement between `/O1` and `/O2` are the same as on Linux, and they were (function inlining). But despite the run taking well over half a day (as the baseline `/O0` performance), there wasn't any difference for `/O0->/O1`, which was disappointing.

Note: It might so be the case like g++ - like they are explicitly disabled with no override. I could not find any documentation to prove – or disprove it - however.

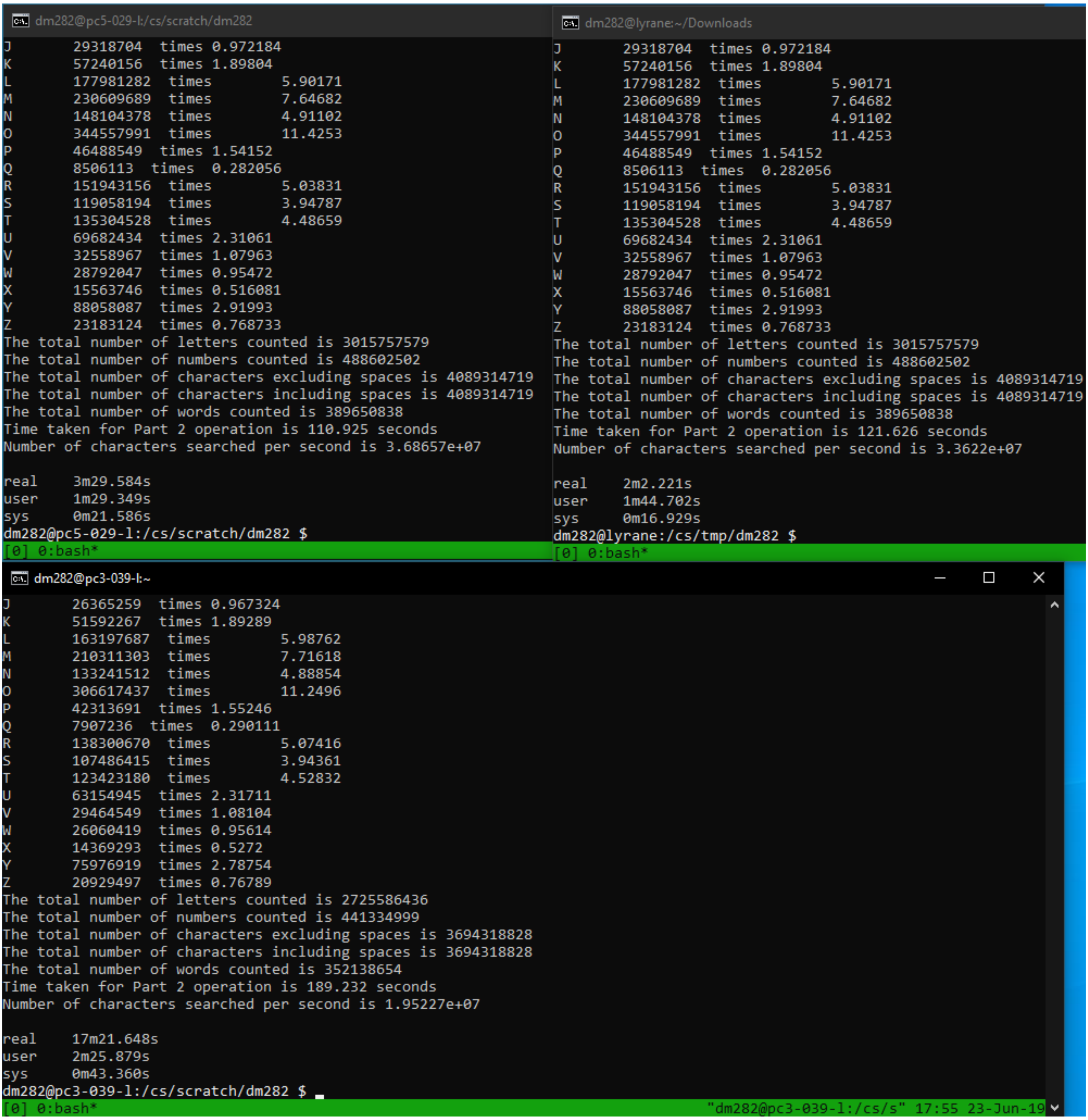
Another microbenchmark

After I had completed tests with Linpack and my benchmark program, the next step was to consider other microbenchmarks and see how they scale. For a short time I considered one of my simple programs which takes in contents of a file and prints their frequency of words in it. It was simple, and it worked, but it had a few shortcomings:

- There was a static character array of 8191 bytes
- The program failed if there were no spaces (as that was the delimiter)
- It is quite I/O intensive, which might have to be considered

So, I spent some time fixing the bugs in it (adding code to detect lines, and switching from static to dynamic memory allocation), and it works now, but has a couple of bugs which do not detract from benchmarking.

But then the scores (time taken) show why this benchmark is unsuitable for our purposes:



```
J       29318704  times 0.972184
K       57240156  times 1.89804
L       177981282  times        5.90171
M       230609689  times        7.64682
N       148104378  times        4.91102
O       344557991  times        11.4253
P       46488549  times 1.54152
Q       8506113  times  0.282056
R       151943156  times        5.03831
S       119058194  times        3.94787
T       135304528  times        4.48659
U       69682434  times 2.31061
V       32558967  times 1.07963
W       28792047  times 0.95472
X       15563746  times 0.516081
Y       88058087  times 2.91993
Z       23183124  times 0.768733
The total number of letters counted is 3015757579
The total number of numbers counted is 488602502
The total number of characters excluding spaces is 4089314719
The total number of characters including spaces is 4089314719
The total number of words counted is 389650838
Time taken for Part 2 operation is 110.925 seconds
Number of characters searched per second is 3.68657e+07

real    3m29.584s
user    1m29.349s
sys     0m21.586s
dm282@pc5-029-l:/cs/scratch/dm282 $
[0] 0:bash*
```

```
J       29318704  times 0.972184
K       57240156  times 1.89804
L       177981282  times        5.90171
M       230609689  times        7.64682
N       148104378  times        4.91102
O       344557991  times        11.4253
P       46488549  times 1.54152
Q       8506113  times  0.282056
R       151943156  times        5.03831
S       119058194  times        3.94787
T       135304528  times        4.48659
U       69682434  times 2.31061
V       32558967  times 1.07963
W       28792047  times 0.95472
X       15563746  times 0.516081
Y       88058087  times 2.91993
Z       23183124  times 0.768733
The total number of letters counted is 3015757579
The total number of numbers counted is 488602502
The total number of characters excluding spaces is 4089314719
The total number of characters including spaces is 4089314719
The total number of words counted is 389650838
Time taken for Part 2 operation is 121.626 seconds
Number of characters searched per second is 3.3622e+07

real    2m2.221s
user    1m44.702s
sys     0m16.929s
dm282@lyrane:/cs/tmp/dm282 $
[0] 0:bash*
```

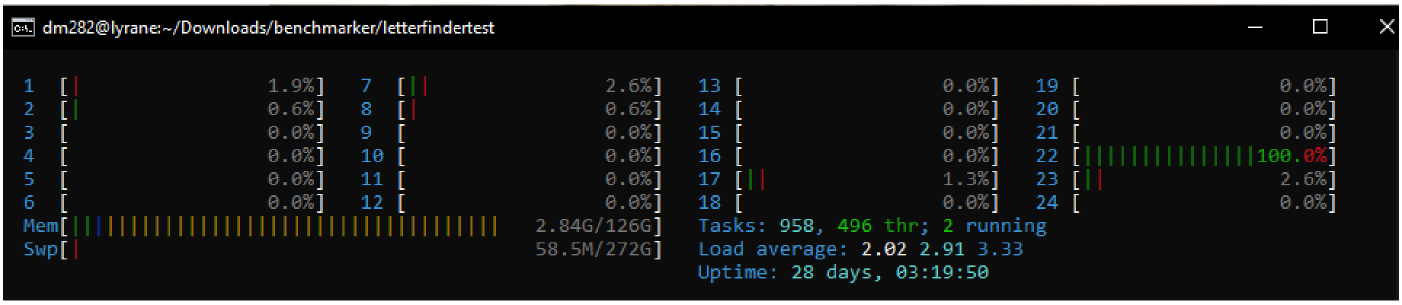
```
J       26365259  times 0.967324
K       51592267  times 1.89289
L       163197687  times        5.98762
M       210311303  times        7.71618
N       133241512  times        4.88854
O       306617437  times        11.2496
P       42313691  times 1.55246
Q       7907236  times  0.290111
R       138300670  times        5.07416
S       107486415  times        3.94361
T       123423180  times        4.52832
U       63154945  times 2.31711
V       29464549  times 1.08104
W       26060419  times 0.95614
X       14369293  times 0.5272
Y       75976919  times 2.78754
Z       20929497  times 0.76789
The total number of letters counted is 2725586436
The total number of numbers counted is 441334999
The total number of characters excluding spaces is 3694318828
The total number of characters including spaces is 3694318828
The total number of words counted is 352138654
Time taken for Part 2 operation is 189.232 seconds
Number of characters searched per second is 1.95227e+07

real    17m21.648s
user    2m25.879s
sys     0m43.360s
dm282@pc3-039-l:/cs/scratch/dm282 $ _
[0] 0:bash*                                                        "dm282@pc3-039-l:/cs/s" 17:55 23-Jun-19
```

Looking solely at the time taken as reported by the program, one would think that the Lyrane server has done extremely well, beating out the faster PC3-*x* computer. However, the program isn't correct here – the PC3-*x* computer took much more time to finish than the other two.

Along with the time taken as reported by the program, I also used the time command to report the actual time taken along with the time taken by the process and that taken by the kernel. The results show that we are badly I/O bottlenecked – the Lyrane uses a RAID array – and while the PC5-*x* configuration can keep up with its SSD, the poor PC3-*x* machines with their HDD is simply no match here. This explains why the real time is far more for the PC3-*x* machine, but the process itself did not take much longer (and could have done faster if it had a better I/O). The time taken as reported by the program is essentially user+sys (not wall clock as such).

When running the `htop` command, one can see the impact of the I/O – the HDD is under full load (but so was the CPU when its utilisation was checked through top):

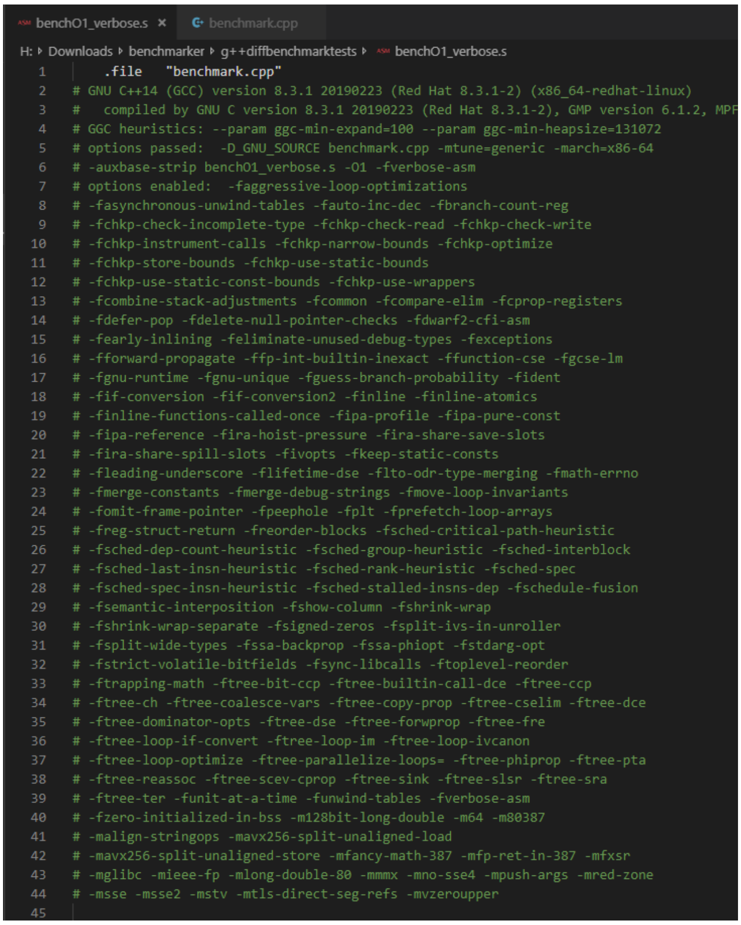


```
1  [|                    1.9%]   7  [||                   2.6%]   13 [                     0.0%]   19 [                     0.0%]
2  [|                    0.6%]   8  [|                    0.6%]   14 [                     0.0%]   20 [                     0.0%]
3  [                     0.0%]   9  [                     0.0%]   15 [                     0.0%]   21 [                     0.0%]
4  [                     0.0%]   10 [                     0.0%]   16 [                     0.0%]   22 [|||||||||||||||100.0%]
5  [                     0.0%]   11 [                     0.0%]   17 [||                   1.3%]   23 [||                   2.6%]
6  [                     0.0%]   12 [                     0.0%]   18 [                     0.0%]   24 [                     0.0%]
Mem[||||||||||||||||||||||||||||||||||||     2.84G/126G]   Tasks: 958, 496 thr; 2 running
Swp[|                                        58.5M/272G]   Load average: 2.02 2.91 3.33
                                                           Uptime: 28 days, 03:19:50
```

As a result of these findings, this program was not used for any further benchmarking results, and any results obtained till then were discarded.

The first breakthrough

I wanted to take a deeper look on what really was going on – so started to take a look at the assembly with the `-fverbose-asm` as that one puts very helpful comments which make it much easier to understand what is going on. And there comes my facepalm moment – the flags which are enabled by g++ - *right in front of my eyes*:

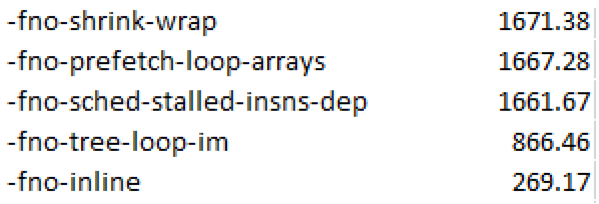

benchO1_verbose.s ×
benchmark.cpp
H: ▸ Downloads ▸ benchmarker ▸ g++diffbenchmarktests ▸ benchO1_verbose.s

```
	.file	"benchmark.cpp"
# GNU C++14 (GCC) version 8.3.1 20190223 (Red Hat 8.3.1-2) (x86_64-redhat-linux)
#	compiled by GNU C version 8.3.1 20190223 (Red Hat 8.3.1-2), GMP version 6.1.2, MPF
# GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
# options passed:  -D_GNU_SOURCE benchmark.cpp -mtune=generic -march=x86-64
# -auxbase-strip bench01_verbose.s -O1 -fverbose-asm
# options enabled:  -faggressive-loop-optimizations
# -fasynchronous-unwind-tables -fauto-inc-dec -fbranch-count-reg
# -fchkp-check-incomplete-type -fchkp-check-read -fchkp-check-write
# -fchkp-instrument-calls -fchkp-narrow-bounds -fchkp-optimize
# -fchkp-store-bounds -fchkp-use-static-bounds
# -fchkp-use-static-const-bounds -fchkp-use-wrappers
# -fcombine-stack-adjustments -fcommon -fcompare-elim -fcprop-registers
# -fdefer-pop -fdelete-null-pointer-checks -fdwarf2-cfi-asm
# -fearly-inlining -feliminate-unused-debug-types -fexceptions
# -fforward-propagate -ffp-int-builtin-inexact -ffunction-cse -fgcse-lm
# -fgnu-runtime -fgnu-unique -fguess-branch-probability -fident
# -fif-conversion -fif-conversion2 -finline -finline-atomics
# -finline-functions-called-once -fipa-profile -fipa-pure-const
# -fipa-reference -fira-hoist-pressure -fira-share-save-slots
# -fira-share-spill-slots -fivopts -fkeep-static-consts
# -fleading-underscore -flifetime-dse -flto-odr-type-merging -fmath-errno
# -fmerge-constants -fmerge-debug-strings -fmove-loop-invariants
# -fomit-frame-pointer -fpeephole -fplt -fprefetch-loop-arrays
# -freg-struct-return -freorder-blocks -fsched-critical-path-heuristic
# -fsched-dep-count-heuristic -fsched-group-heuristic -fsched-interblock
# -fsched-last-insn-heuristic -fsched-rank-heuristic -fsched-spec
# -fsched-spec-insn-heuristic -fsched-stalled-insns-dep -fschedule-fusion
# -fsemantic-interposition -fshow-column -fshrink-wrap
# -fshrink-wrap-separate -fsigned-zeros -fsplit-ivs-in-unroller
# -fsplit-wide-types -fssa-backprop -fssa-phiopt -fstdarg-opt
# -fstrict-volatile-bitfields -fsync-libcalls -ftoplevel-reorder
# -ftrapping-math -ftree-bit-ccp -ftree-builtin-call-dce -ftree-ccp
# -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-cselim -ftree-dce
# -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre
# -ftree-loop-if-convert -ftree-loop-im -ftree-loop-ivcanon
# -ftree-loop-optimize -ftree-parallelize-loops= -ftree-phiprop -ftree-pta
# -ftree-reassoc -ftree-scev-cprop -ftree-sink -ftree-slsr -ftree-sra
# -ftree-ter -funit-at-a-time -funwind-tables -fverbose-asm
# -fzero-initialized-in-bss -m128bit-long-double -m64 -m80387
# -malign-stringops -mavx256-split-unaligned-load
# -mavx256-split-unaligned-store -mfancy-math-387 -mfp-ret-in-387 -mfxsr
# -mglibc -mieee-fp -mlong-double-80 -mmmx -mno-sse4 -mpush-args -mred-zone
# -msse -msse2 -mstv -mtls-direct-seg-refs -mvzeroupper
```

And to surprise me even further, even `/O0` had some flags enabled (like aggressive loop optimisations). I thought `/O0` = optimisations disabled? And how can I trust your own documentation when that does not contain all the flags that is given by the compiler (yes, versions were virtually identical)? Oh come on…

With that, I dumped all the new flags (even the ones which `/O0` enables by default) and re-ran all the benchmarks.

And this cat nearly escaped – but it was caught. While there were still no changes with `/O0`, when I took `/O1` and removed flags,



-fno-shrink-wrap	1671.38
-fno-prefetch-loop-arrays	1667.28
-fno-sched-stalled-insns-dep	1661.67
-fno-tree-loop-im	866.46
-fno-inline	269.17

This was over five runs averaged, and the effect is clear: the mean was 1682.9 on a PC5-*x* configuration and while σ (standard deviation) was over 150 amongst the flags, it dropped to under 12 when the two lowest scoring ones were dropped. No change with the Linpack benchmark sadly – I was really hoping that they would be.

But that meant another question: what relation does `tree-loop-im` have with inlining, if any? After all, we still get a sizable 870 points if the former alone is dropped but drop to `/O0`-level performance if the latter alone is dropped. From their names they don't seem to be the same at all though.

It's time to look at the assembly.

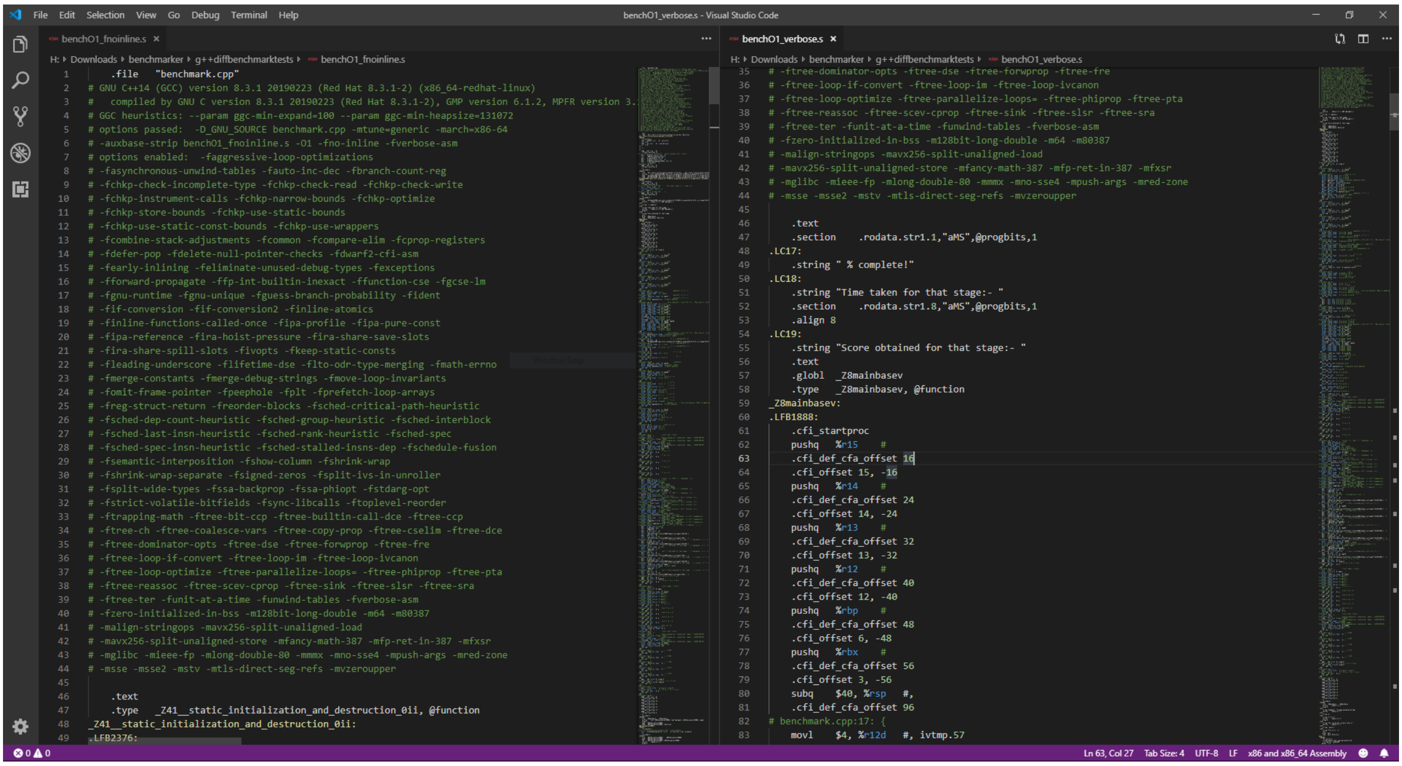
Poking at assembly

With most of my data collected, it was time to go deeper on what is going on. To start with, I decided to modify my program so that it would compile the flags to assembly and then use the `diff` command, piped to `wc`, to find out which flags actually did anything to being with. And in fact, looking at g++, many flags did nothing (assembly output was the same to that of `/O0`), and some more had changes but were superficial (mostly to the labels rather than actual assembly modification). And despite the g++ website stating that many of the optimisation flags would be ignored, I did get some flags which seemed to have some changes in the assembly itself. But then why is there no change in the score?

It got laughable with clang. Out of the 78 flags, only **2** had *any* difference, and the differences were negligible. So much for effort? And I can't test `/O1` without a flag as you haven't still told me how to do it, dear Clang documentation…

After collecting data for the differences in g++ and clang, and with my benchmark program and Linpack, I decided to take a deeper look on the parts where I know has a difference in scores – the *fno-inline* and -*fno-tree-loop-im* flags. This involves some assembly, made tastier by the fact that we set the verbose option to make it easier to understand.

Reminder: Remember that we're using AT&T (GAS) assembly, so the destination comes *after* the source!


File
Edit
Selection
View
Go
Debug
Terminal
Help
benchO1_verbose.s - Visual Studio Code
benchO1_fnoinline.s
H: › Downloads › benchmarker › g++diffbenchmarktests › benchO1_fnoinline.s
benchO1_verbose.s
H: › Downloads › benchmarker › g++diffbenchmarktests › benchO1_verbose.s
0 0
Ln 63, Col 27
Tab Size: 4
UTF-8
LF
x86 and x86_64 Assembly

```
	.file	"benchmark.cpp"
# GNU C++14 (GCC) version 8.3.1 20190223 (Red Hat 8.3.1-2) (x86_64-redhat-linux)
#	compiled by GNU C version 8.3.1 20190223 (Red Hat 8.3.1-2), GMP version 6.1.2, MPFR version 3.
# GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
# options passed:  -D_GNU_SOURCE benchmark.cpp -mtune=generic -march=x86-64
# -auxbase-strip bench01_fnoinline.s -O1 -fno-inline -fverbose-asm
# options enabled:  -faggressive-loop-optimizations
# -fasynchronous-unwind-tables -fauto-inc-dec -fbranch-count-reg
# -fchkp-check-incomplete-type -fchkp-check-read -fchkp-check-write
# -fchkp-instrument-calls -fchkp-narrow-bounds -fchkp-optimize
# -fchkp-store-bounds -fchkp-use-static-bounds
# -fchkp-use-static-const-bounds -fchkp-use-wrappers
# -fcombine-stack-adjustments -fcommon -fcompare-elim -fcprop-registers
# -fdefer-pop -fdelete-null-pointer-checks -fdwarf2-cfi-asm
# -fearly-inlining -feliminate-unused-debug-types -fexceptions
# -fforward-propagate -ffp-int-builtin-inexact -ffunction-cse -fgcse-lm
# -fgnu-runtime -fgnu-unique -fguess-branch-probability -fident
# -fif-conversion -fif-conversion2 -finline-atomics
# -finline-functions-called-once -fipa-profile -fipa-pure-const
# -fipa-reference -fira-hoist-pressure -fira-share-save-slots
# -fira-share-spill-slots -fivopts -fkeep-static-consts
# -fleading-underscore -flifetime-dse -flto-odr-type-merging -fmath-errno
# -fmerge-constants -fmerge-debug-strings -fmove-loop-invariants
# -fomit-frame-pointer -fpeephole -fplt -fprefetch-loop-arrays
# -freg-struct-return -freorder-blocks -fsched-critical-path-heuristic
# -fsched-dep-count-heuristic -fsched-group-heuristic -fsched-interblock
# -fsched-last-insn-heuristic -fsched-rank-heuristic -fsched-spec
# -fsched-spec-insn-heuristic -fsched-stalled-insns-dep -fschedule-fusion
# -fsemantic-interposition -fshow-column -fshrink-wrap
# -fshrink-wrap-separate -fsigned-zeros -fsplit-ivs-in-unroller
# -fsplit-wide-types -fssa-backprop -fssa-phiopt -fstdarg-opt
# -fstrict-volatile-bitfields -fsync-libcalls -ftoplevel-reorder
# -ftrapping-math -ftree-bit-ccp -ftree-builtin-call-dce -ftree-ccp
# -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-cselim -ftree-dce
# -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre
# -ftree-loop-if-convert -ftree-loop-im -ftree-loop-ivcanon
# -ftree-loop-optimize -ftree-parallelize-loops= -ftree-phiprop -ftree-pta
# -ftree-reassoc -ftree-scev-cprop -ftree-sink -ftree-slsr -ftree-sra
# -ftree-ter -funit-at-a-time -funwind-tables -fverbose-asm
# -fzero-initialized-in-bss -m128bit-long-double -m64 -m80387
# -malign-stringops -mavx256-split-unaligned-load
# -mavx256-split-unaligned-store -mfancy-math-387 -mfp-ret-in-387 -mfxsr
# -mglibc -mieee-fp -mlong-double-80 -mmmx -mno-sse4 -mpush-args -mred-zone
# -msse -msse2 -mstv -mtls-direct-seg-refs -mvzeroupper

	.text
	.type	_Z41__static_initialization_and_destruction_0ii, @function
_Z41__static_initialization_and_destruction_0ii:
.LFB2376:
```

```
# -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre
# -ftree-loop-if-convert -ftree-loop-im -ftree-loop-ivcanon
# -ftree-loop-optimize -ftree-parallelize-loops= -ftree-phiprop -ftree-pta
# -ftree-reassoc -ftree-scev-cprop -ftree-sink -ftree-slsr -ftree-sra
# -ftree-ter -funit-at-a-time -funwind-tables -fverbose-asm
# -fzero-initialized-in-bss -m128bit-long-double -m64 -m80387
# -malign-stringops -mavx256-split-unaligned-load
# -mavx256-split-unaligned-store -mfancy-math-387 -mfp-ret-in-387 -mfxsr
# -mglibc -mieee-fp -mlong-double-80 -mmmx -mno-sse4 -mpush-args -mred-zone
# -msse -msse2 -mstv -mtls-direct-seg-refs -mvzeroupper

	.text
	.section	.rodata.str1.1,"aMS",@progbits,1
.LC17:
	.string " % complete!"
.LC18:
	.string "Time taken for that stage:- "
	.section	.rodata.str1.8,"aMS",@progbits,1
	.align 8
.LC19:
	.string "Score obtained for that stage:- "
	.text
	.globl	_Z8mainbasev
	.type	_Z8mainbasev, @function
_Z8mainbasev:
.LFB1888:
	.cfi_startproc
	pushq	%r15	#
	.cfi_def_cfa_offset 16
	.cfi_offset 15, -16
	pushq	%r14	#
	.cfi_def_cfa_offset 24
	.cfi_offset 14, -24
	pushq	%r13	#
	.cfi_def_cfa_offset 32
	.cfi_offset 13, -32
	pushq	%r12	#
	.cfi_def_cfa_offset 40
	.cfi_offset 12, -40
	pushq	%rbp	#
	.cfi_def_cfa_offset 48
	.cfi_offset 6, -48
	pushq	%rbx	#
	.cfi_def_cfa_offset 56
	.cfi_offset 3, -56
	subq	$40, %rsp	#,
	.cfi_def_cfa_offset 96
# benchmark.cpp:17: {
	movl	$4, %r12d	#, ivtmp.57
```

It's not that hard to understand what the program is doing with the baseline /O0 flag, but there are a variety of changes that occur with /O1, some of which are harder to decipher than others.

For instance, consider this block of assembly:

/O0	/O1

`# benchmark.cpp:33:          k = 100;`
`    movsd   .LC1(%rip), %xmm0   #, tmp164`
`    movsd   %xmm0, k(%rip)  # tmp164, k`
`    jmp .L23    #`
	`# benchmark.cpp:33:          k = 100;`
	`    movq    .LC6(%rip), %rax    #, tmp228`
	`    movq    %rax, k(%rip)   # tmp228, k`
	`    jmp .L29    #`

The /O0 uses a signed double (i.e, octoword) (as the variable *k* is indeed a double), but /O1 switches that for a quadword (float)! This could be because 100 can easily be represented as a quad (probably it would plan to use cqto if the value could no longer fit in a quadword).

Another interesting trick by the compiler is to use bitwise operators to 'reset' registers:

```
pxor    %xmm0, %xmm0
```

This operation makes use of [pxor](#) (vector bit-wise XOR operator) to reset the value of register xmm0 to zero (the idea is that when represented in bits, they will have the same value, but 1 XOR 1 = 0 and 0 XOR 0 = 0). But this is hardly the only way that the compiler could have done. [Any of these](#) would have done the same thing:

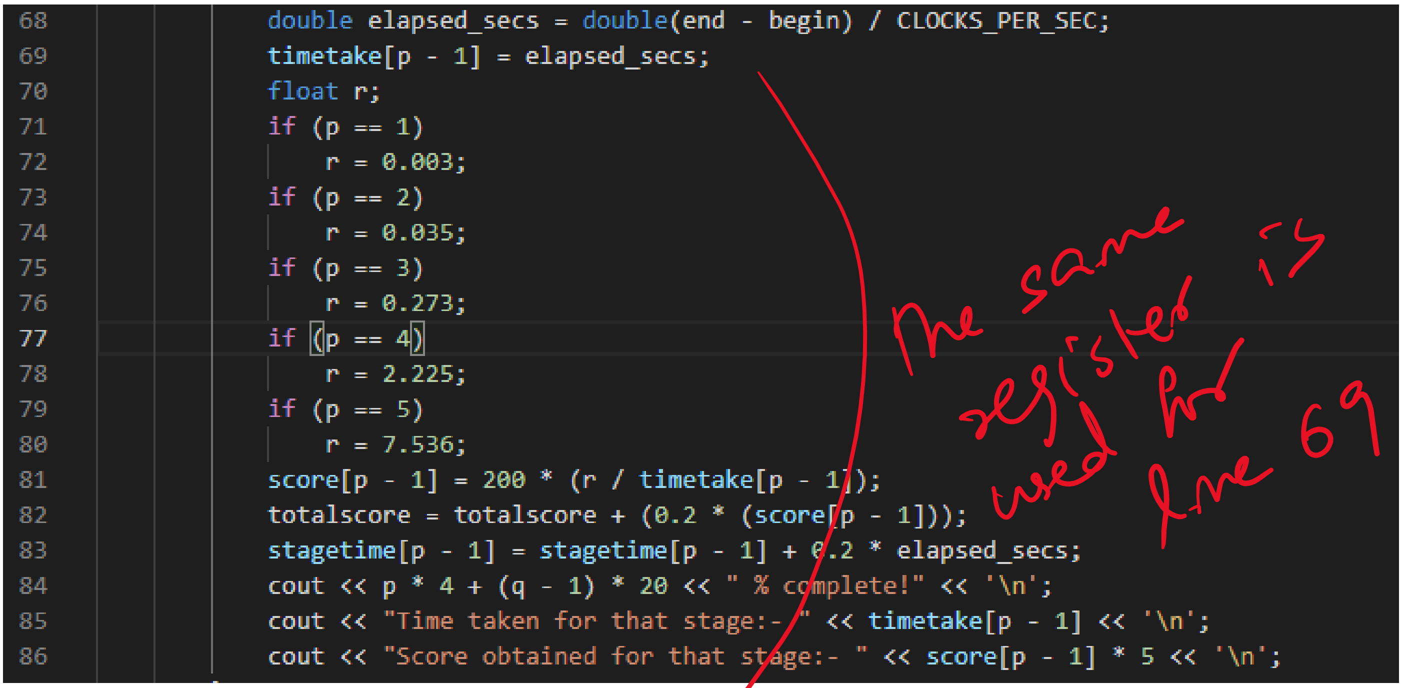
```
xorl    %eax, %eax

mov     $0, %eax

andl    $0, %eax
```

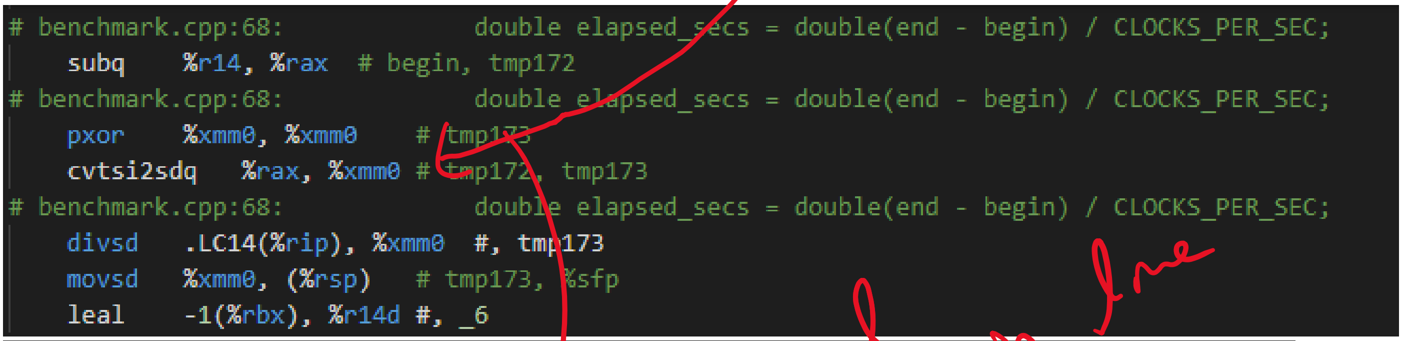
But the compiler would take the first option as it thinks it is the best choice for performance (it's technically more complex than that, and the StackOverflow answer linked above does a good job in explaining that).

The last general optimisation approach that will be described here is its tendency to retain using the same registers when it knows that it is going to be used immediately again. While /O0 does the expected steps of copying and moving between different registers, /O1 does nothing. The compiler cleverly senses that the same value is going to be used in the next few steps, and just carries the register containing that value all though! Note that there is no line 69 in the assembly for /O1 for this reason.

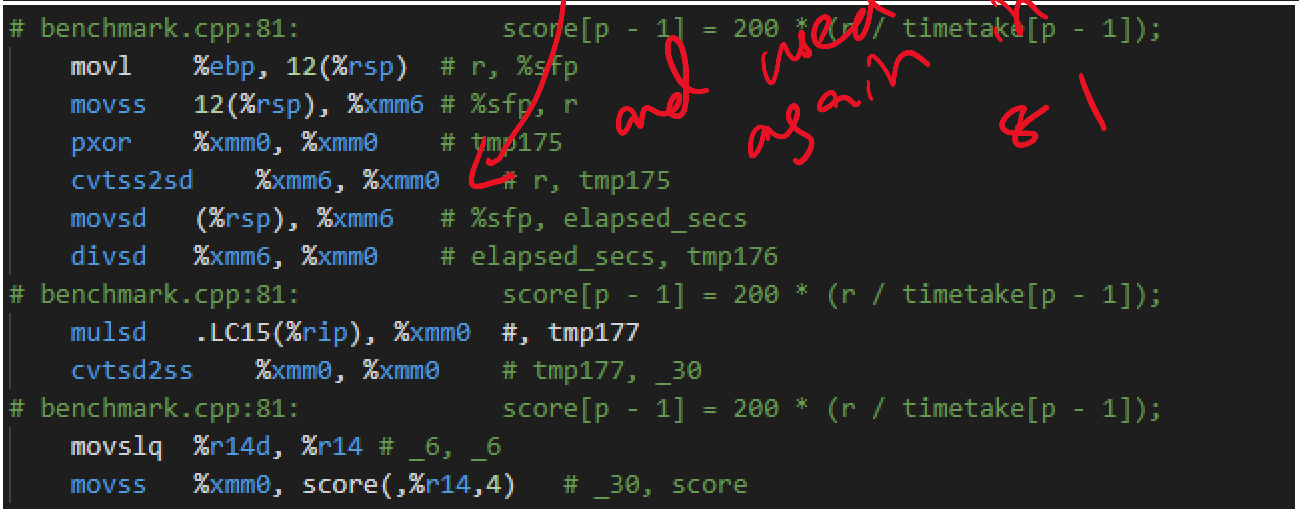


```
68              double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
69              timetake[p - 1] = elapsed_secs;
70              float r;
71              if (p == 1)
72                  r = 0.003;
73              if (p == 2)
74                  r = 0.035;
75              if (p == 3)
76                  r = 0.273;
77              if (p == 4)
78                  r = 2.225;
79              if (p == 5)
80                  r = 7.536;
81              score[p - 1] = 200 * (r / timetake[p - 1]);
82              totalscore = totalscore + (0.2 * (score[p - 1]));
83              stagetime[p - 1] = stagetime[p - 1] + 0.2 * elapsed_secs;
84              cout << p * 4 + (q - 1) * 20 << " % complete!" << '\n';
85              cout << "Time taken for that stage:- " << timetake[p - 1] << '\n';
86              cout << "Score obtained for that stage:- " << score[p - 1] * 5 << '\n';
```

The same register is used for line 69



```
# benchmark.cpp:68:             double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
    subq    %r14, %rax  # begin, tmp172
# benchmark.cpp:68:             double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
    pxor    %xmm0, %xmm0    # tmp173
    cvtsi2sdq   %rax, %xmm0 # tmp172, tmp173
# benchmark.cpp:68:             double elapsed_secs = double(end - begin) / CLOCKS_PER_SEC;
    divsd   .LC14(%rip), %xmm0  #, tmp173
    movsd   %xmm0, (%rsp)   # tmp173, %sfp
    leal    -1(%rbx), %r14d #, _6
```



```
# benchmark.cpp:81:             score[p - 1] = 200 * (r / timetake[p - 1]);
    movl    %ebp, 12(%rsp)  # r, %sfp
    movss   12(%rsp), %xmm6 # %sfp, r
    pxor    %xmm0, %xmm0    # tmp175
    cvtss2sd    %xmm6, %xmm0    # r, tmp175
    movsd   (%rsp), %xmm6   # %sfp, elapsed_secs
    divsd   %xmm6, %xmm0    # elapsed_secs, tmp176
# benchmark.cpp:81:             score[p - 1] = 200 * (r / timetake[p - 1]);
    mulsd   .LC15(%rip), %xmm0  #, tmp177
    cvtsd2ss    %xmm0, %xmm0    # tmp177, _30
# benchmark.cpp:81:             score[p - 1] = 200 * (r / timetake[p - 1]);
    movslq  %r14d, %r14 # _6, _6
    movss   %xmm0, score(,%r14,4)   # _30, score
```

and used again in line 81

← this is equal to score[p-1] and not timetake, and this register will be (hence) reused in line 86.

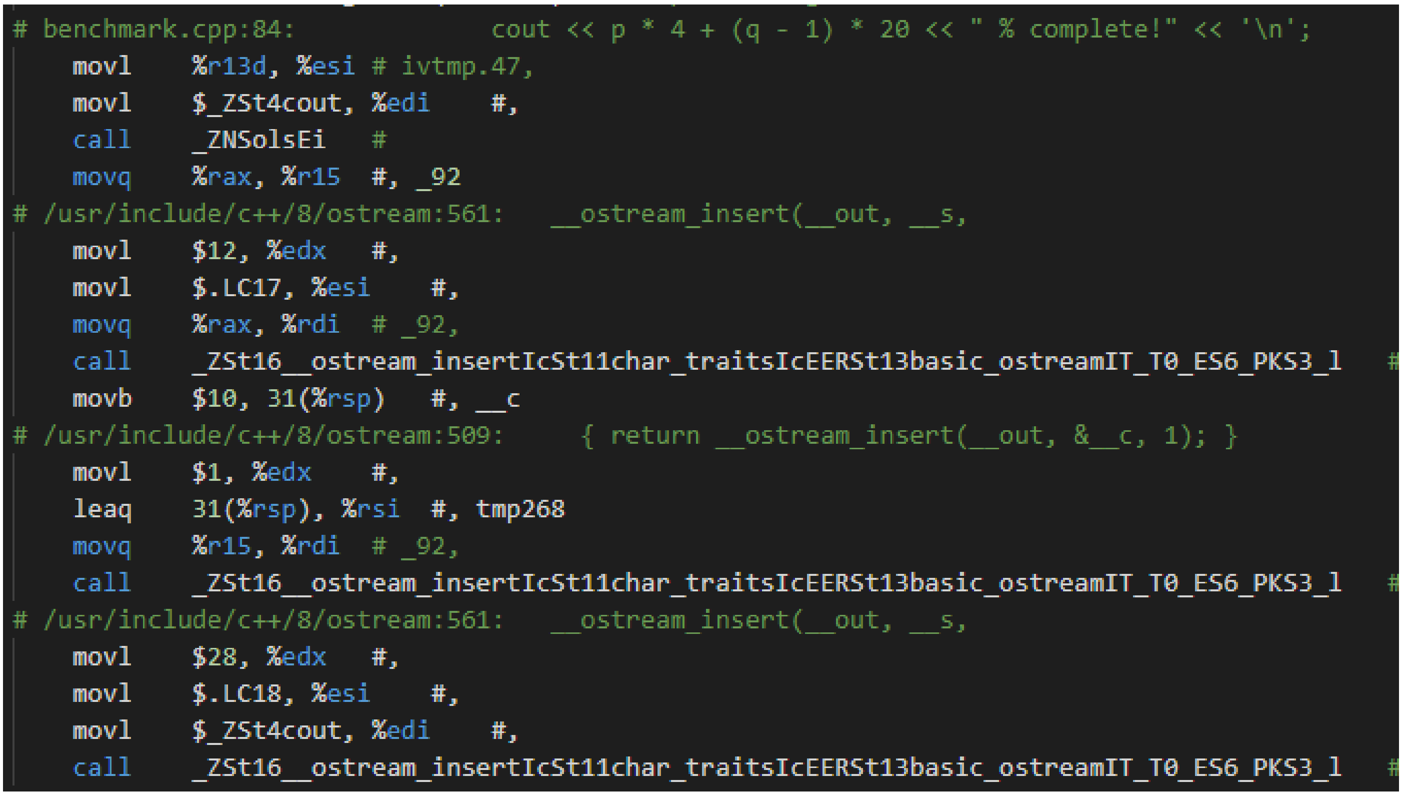
But I haven't yet covered the elephant in the room: *function inlining*. That's for the next section, and where we get our second breakthrough.

Function inlining

It wasn't clear to me what relevance function inlining had in the program. After all, the very notion of function inlining involves [inserting the definition](#) directly into the caller stream – it's like using a #define macro, but much safer and without the safety problems that the former entitles.

But that is most useful in programs which employ multiple functions – which my benchmark program does not with the gist of the program contained in a single function (the only other function is `main`). That means that we need to get to the bottom of this by making use of the assembly – is there *some* way this could still be happening?

Immediately, I found some unusual comments in the `/O1` assembly:

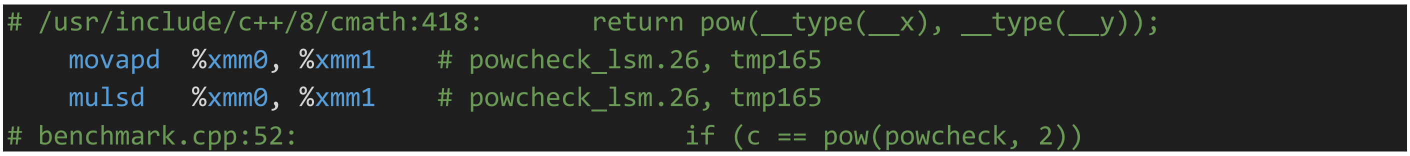


```
# benchmark.cpp:84:             cout << p * 4 + (q - 1) * 20 << " % complete!" << '\n';
    movl    %r13d, %esi # ivtmp.47,
    movl    $_ZSt4cout, %edi    #,
    call    _ZNSolsEi   #
    movq    %rax, %r15  #, _92
# /usr/include/c++/8/ostream:561:   __ostream_insert(__out, __s,
    movl    $12, %edx   #,
    movl    $.LC17, %esi    #,
    movq    %rax, %rdi  # _92,
    call    _ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l   #
    movb    $10, 31(%rsp)   #, __c
# /usr/include/c++/8/ostream:509:     { return __ostream_insert(__out, &__c, 1); }
    movl    $1, %edx    #,
    leaq    31(%rsp), %rsi  #, tmp268
    movq    %r15, %rdi  # _92,
    call    _ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l   #
# /usr/include/c++/8/ostream:561:   __ostream_insert(__out, __s,
    movl    $28, %edx   #,
    movl    $.LC18, %esi    #,
    movl    $_ZSt4cout, %edi    #,
    call    _ZSt16__ostream_insertIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_PKS3_l   #
```

These lines suggested that these optimisations were perhaps occurring *outside* the program, and I was concerned that this would mean having to look at ostream to see if, and how, inlining could occur there. My supervisor suggested removing all output statements (which I was sceptical, because I was not convinced that could explain such a large performance difference), but I still did it. And there was no difference. Something else must explain what is going on.

I took another look at the assembly. And while most of the ostream comments were now gone, I still tried to understand what was going on in the assembly with `/O1` (and `/O1` without function inlining). And while I was not able to fully explain a few oddities (like that at line 44), something else caught my eye – with the power function:

/O1:



```
# /usr/include/c++/8/cmath:418:       return pow(__type(__x), __type(__y));
    movapd  %xmm0, %xmm1    # powcheck_lsm.26, tmp165
    mulsd   %xmm0, %xmm1    # powcheck_lsm.26, tmp165
# benchmark.cpp:52:                         if (c == pow(powcheck, 2))
```

```
    ucomisd %xmm1, %xmm2    # tmp165, c_lsm.28
    jp  .L14    #,
    jne .L14    #,
```

/O1 without function inlining:

```
# benchmark.cpp:52:                           if (c == pow(powcheck, 2))
    movl    $2, %edi    #,
    call    _ZSt3powIdiEN9__gnu_cxx11__promote_2IT_T0_NS0_9__promoteIS2_XsrSt12__is_intege
rIS2_E7__valueEE6__typeENS4_IS3_XsrS5_IS3_E7__valueEE6__typeEE6__typeES2_S3_  #
# benchmark.cpp:52:                           if (c == pow(powcheck, 2))
    movsd   c(%rip), %xmm1  # c, c.15_14
# benchmark.cpp:52:                           if (c == pow(powcheck, 2))
    ucomisd %xmm1, %xmm0    # c.15_14, _13
    jp  .L23    #,
    jne .L23    #,
```

Wait, so they aren't the same? The `/O1` case is actually straightforward – the power function `pow (x,2)` is inlined as x * x. But that isn't the case with `/O0` – it passes 2 to a register and calls power (which is somehow weirdly represented here). Perhaps that could be the problem?

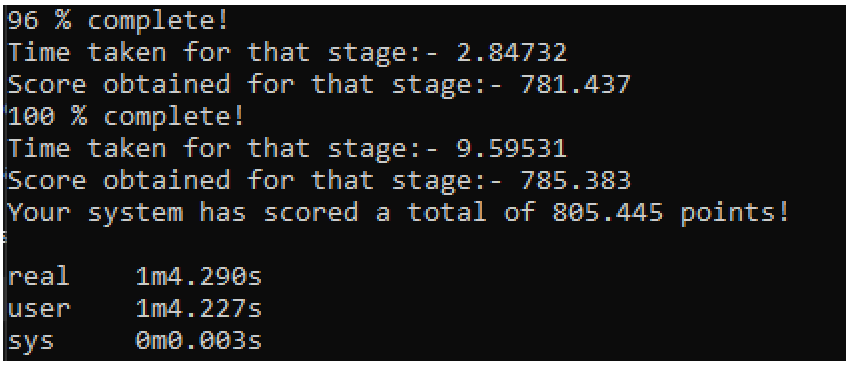
To test this, we simply change

```
if (c == pow(powcheck, 2))
```

To

```
if (c == (powcheck * powcheck))
```

I then gave the benchmark another run (with the baseline `/O0` case). And a giant cat comes out this time:



```
96 % complete!
Time taken for that stage:- 2.84732
Score obtained for that stage:- 781.437
100 % complete!
Time taken for that stage:- 9.59531
Score obtained for that stage:- 785.383
Your system has scored a total of 805.445 points!

real    1m4.290s
user    1m4.227s
sys     0m0.003s
```

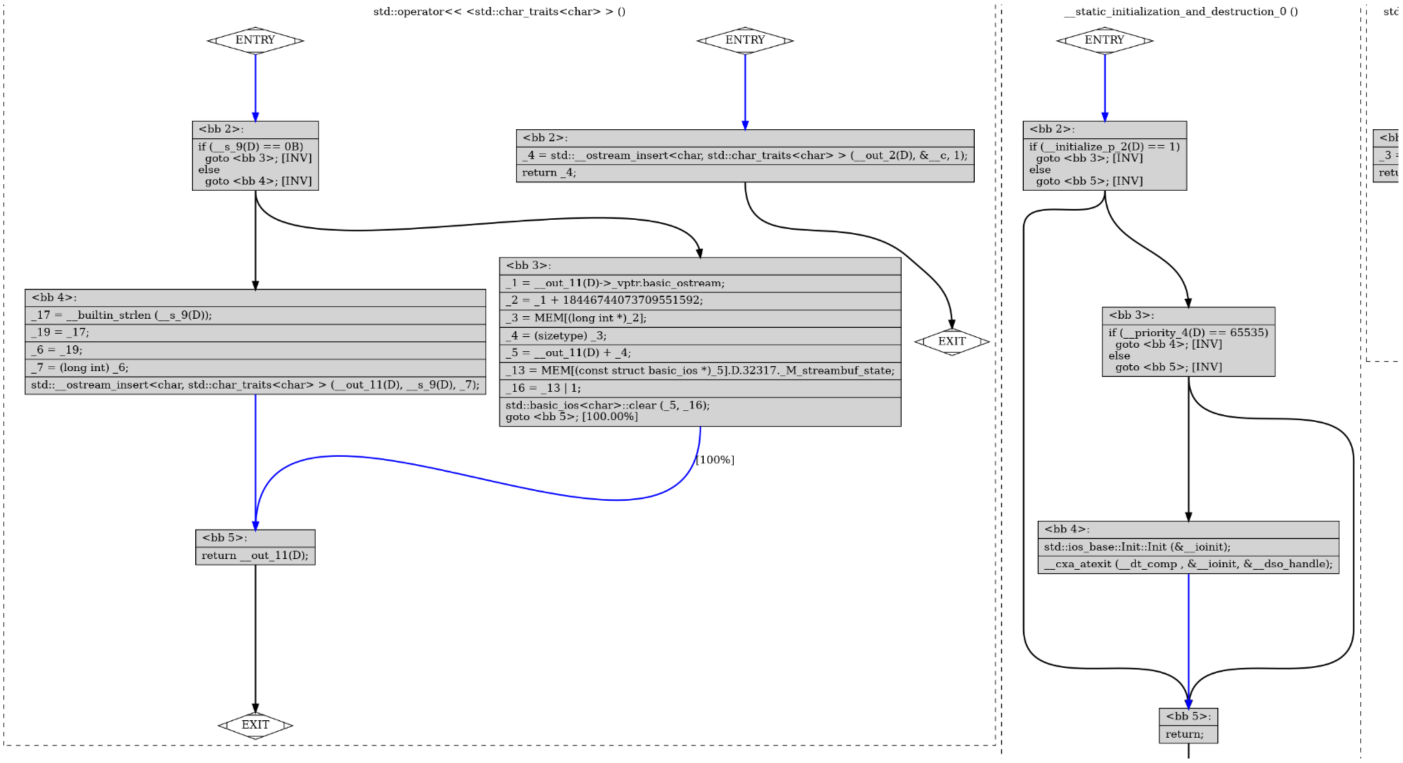
We get a performance increase of nearly **3** times (previous score = 270 points) just by changing one line! This effectively halves the delta between `/O0` and `/O1`. The power function must be seriously inefficient on g++. And it really is. Clang seems to have much less of a performance loss (but then it is harder to explain what is going on there). Maybe g++ is using a slower but more accurate method?

With this, we now need to take a look at `tree-loop-im` – to confirm, I gave the benchmark another run with each individual flag switched off from `/O1` – and this time, *fno-inline* gives an `/O1` score.

Trees and GIMPLE

Now we take a look at the last optimisation flag. But what does "`ftree-loop-im`" even mean? Looking at its man page says that it "Perform loop invariant motion on trees". While loop invariant (which is simply any condition that holds true for the duration of the loop) motion is the process of optimising loops by moving calculations which do the same thing *inside* the loop *outside* (or other similar conditions), what does *tree* mean here?? After all, the only thing I know about a 'tree' is that of an ADT (abstract data type). It turns out that *trees* are how g++ represents its data internally. But then do we find out what actually happens in these 'trees'? Decompose them.

Using the flag `-fdump-tree-all-graph`, we get a bunch of g++ intermediate code (like LLVM in clang), and .dot files, which can be converted into graphs using GraphViz.


std::operator<< <std::char_traits<char> > ()
ENTRY
ENTRY
EXIT
[100%]
EXIT
__static_initialization_and_destruction_0 ()
ENTRY
std

```
<bb 2>:
if (__s_9(D) == 0B)
  goto <bb 3>; [INV]
else
  goto <bb 4>; [INV]

<bb 4>:
_17 = __builtin_strlen (__s_9(D));
_19 = _17;
_6 = _19;
_7 = (long int) _6;
std::__ostream_insert<char, std::char_traits<char> > (__out_11(D), __s_9(D), _7);

<bb 5>:
return __out_11(D);

<bb 2>:
_4 = std::__ostream_insert<char, std::char_traits<char> > (__out_2(D), &__c, 1);
return _4;

<bb 3>:
_1 = __out_11(D)->_vptr.basic_ostream;
_2 = _1 + 18446744073709551592;
_3 = MEM[(long int *)_2];
_4 = (sizetype) _3;
_5 = __out_11(D) + _4;
_13 = MEM[(const struct basic_ios *)_5].D.32317._M_streambuf_state;
_16 = _13 | 1;
std::basic_ios<char>::clear (_5, _16);
goto <bb 5>; [100.00%]
```

```
<bb 2>:
if (__initialize_p_2(D) == 1)
  goto <bb 3>; [INV]
else
  goto <bb 5>; [INV]

<bb 3>:
if (__priority_4(D) == 65535)
  goto <bb 4>; [INV]
else
  goto <bb 5>; [INV]

<bb 4>:
std::ios_base::Init::Init (&__ioinit);
__cxa_atexit (__dt_comp , &__ioinit, &__dso_handle);

<bb 5>:
return;
```

```
<bb
_3 =
retu
```

An output from one of the graphs (they are really large as one of the nodes can be disproportionally larger than the others).

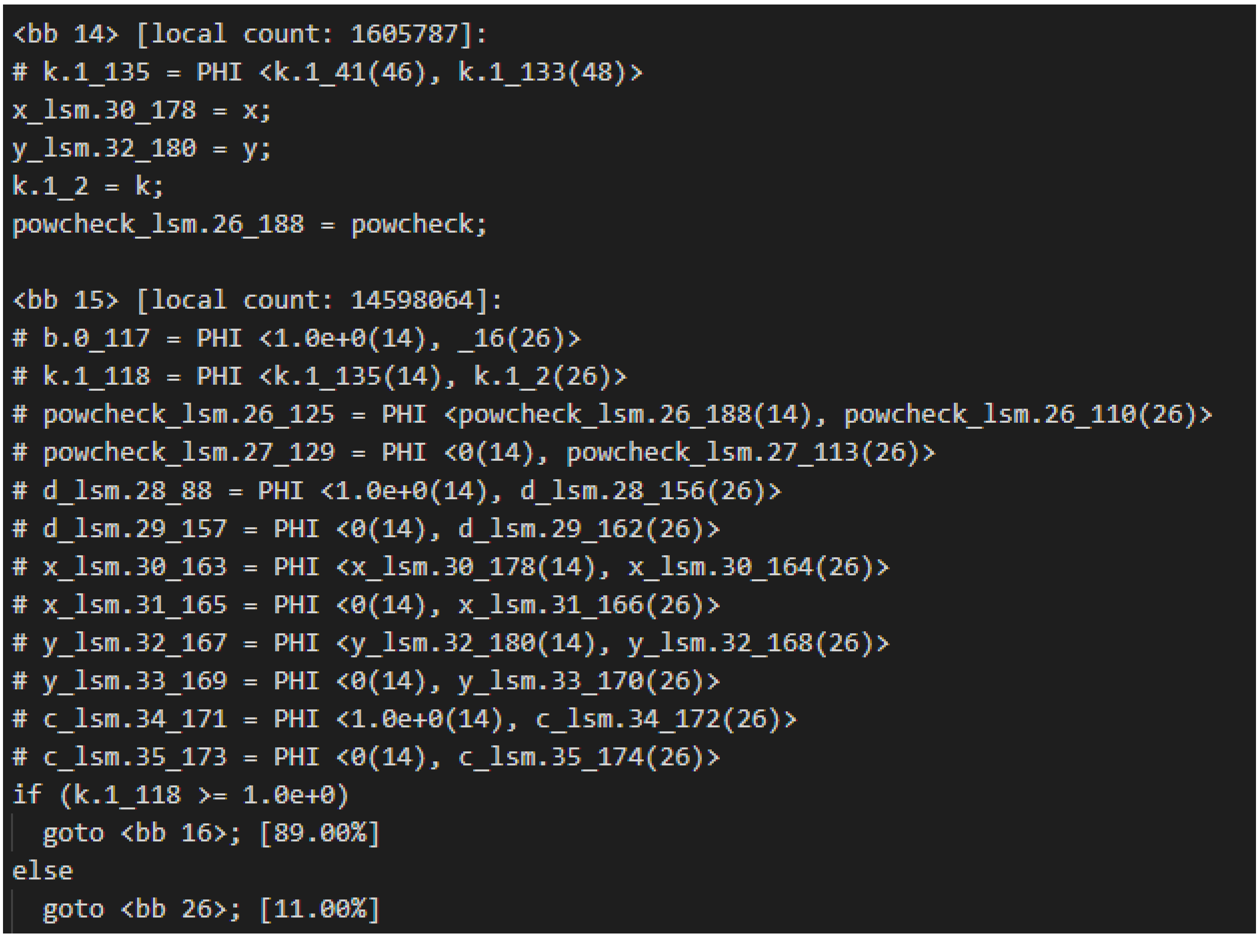
Unfortunately, there were two major issues:

- Many of the output files were actually identical to the extent that their checksums matched,
- It was a hit or miss – the other times I would get segfaults (why?).

But that wasn't going to discourage me. I decided to attack the LLVM-like code directly without looking at the graphs. While the above flag will also produce this, I used a separate flag `-fdump-tree-all`.

This left me with a bunch of funny files that I could not explain – why *so many* files – and again many of them were the same (huh?). So I diff'd the two folders to check the files that I should *actually* concentrate at. Turned out that there were many, but I doubled down on a flag that read "optimised" – perhaps that holds the key?

But the language in that file is *not* C++. It is *not* assembly either. It is GIMPLE – the intermediate representation of gcc (and hence g++). It can be thought of as pseudocode C++ - I would think that it is easier to read than assembly, but harder than C++. But I found confusing having just come from assembly.



```
<bb 14> [local count: 1605787]:
# k.1_135 = PHI <k.1_41(46), k.1_133(48)>
x_lsm.30_178 = x;
y_lsm.32_180 = y;
k.1_2 = k;
powcheck_lsm.26_188 = powcheck;

<bb 15> [local count: 14598064]:
# b.0_117 = PHI <1.0e+0(14), _16(26)>
# k.1_118 = PHI <k.1_135(14), k.1_2(26)>
# powcheck_lsm.26_125 = PHI <powcheck_lsm.26_188(14), powcheck_lsm.26_110(26)>
# powcheck_lsm.27_129 = PHI <0(14), powcheck_lsm.27_113(26)>
# d_lsm.28_88 = PHI <1.0e+0(14), d_lsm.28_156(26)>
# d_lsm.29_157 = PHI <0(14), d_lsm.29_162(26)>
# x_lsm.30_163 = PHI <x_lsm.30_178(14), x_lsm.30_164(26)>
# x_lsm.31_165 = PHI <0(14), x_lsm.31_166(26)>
# y_lsm.32_167 = PHI <y_lsm.32_180(14), y_lsm.32_168(26)>
# y_lsm.33_169 = PHI <0(14), y_lsm.33_170(26)>
# c_lsm.34_171 = PHI <1.0e+0(14), c_lsm.34_172(26)>
# c_lsm.35_173 = PHI <0(14), c_lsm.35_174(26)>
if (k.1_118 >= 1.0e+0)
  goto <bb 16>; [89.00%]
else
  goto <bb 26>; [11.00%]
```

Now, this is in SSA form. The confusion that related to this was – what is k.1_2 and how is it different from, say, k.1_118? And what on earth is `PHI <a,b>` - surely it cannot be $\frac{1+\sqrt{5}}{2}$?

Turns out that `PHI <a,b>` can be thought as the union of all possible values of the variable up to that point. For example, if we have a conditional statement that resembles like the following:

```
(a > b)?(c = 2):(c = 3)
```

Then the compiler needs to keep track of both possible values of c, which is done using `c.1_1` (c_1) and `c.1_2` (c_2). But now if we have the next statement as

```
d = c + 2
```

Now, which c would the compiler take? c_1 or c_2? It can't decide one without finishing the statement (otherwise we get a hazard). Hence it uses the PHI statement to keep track of this fact. So the above statement could be written in GIMPLE as

$$d = \phi(c_1, c_2)$$

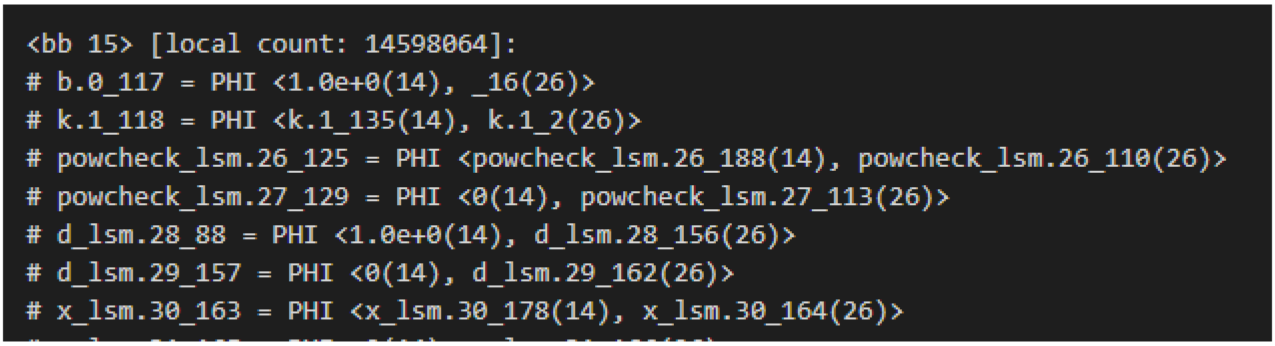
We could go a bit mathematical and also define the PHI function as

$$\phi(a_1, a_2, a_3, \dots, a_n) = \left\{\bigcup_{i=1}^{n} a_i\right\}$$

GIMPLE also likes to represent numbers in the form $(a)e + b$, which translates to $a(10^b)$ (not ae^b).

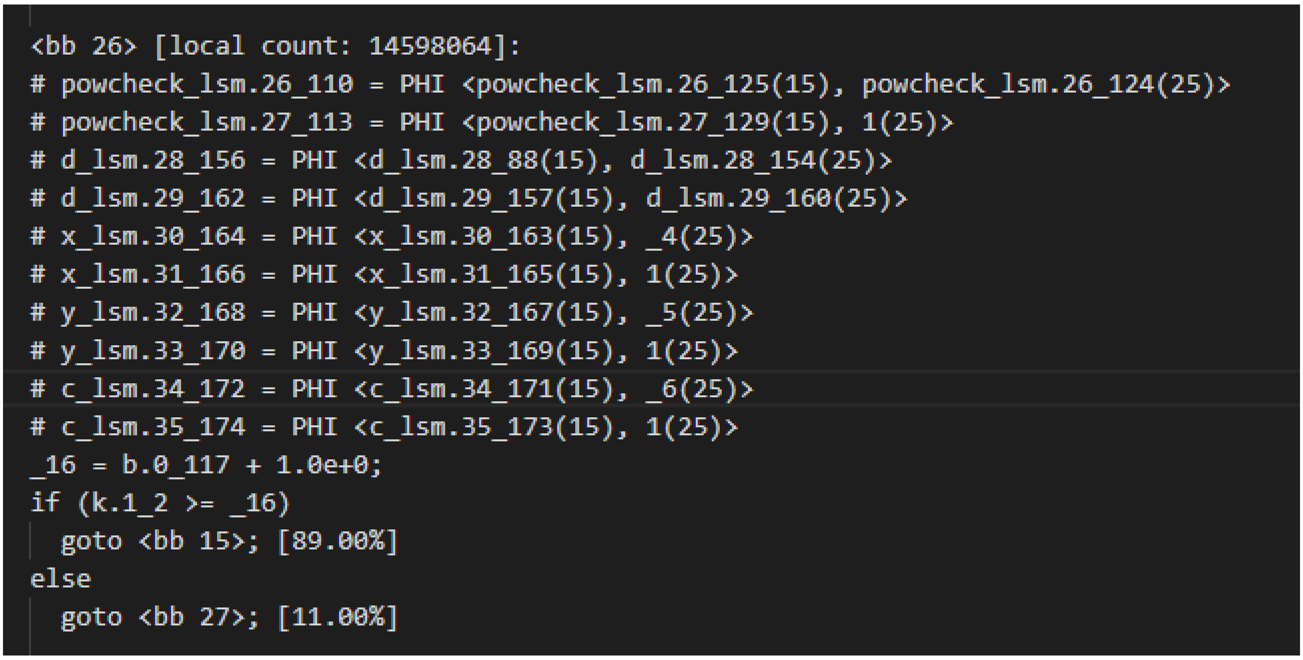
Another notable thing is the inclusion of goto statements, with some percentage that appears to be g++'s interpretation of the probability of moving to that block of code, but I'm not sure on that.

But how does the flag affect the output? There is only one segment of code which varies between the two versions (one straight `/O1` and the other `/O1` without the flag). Even then, making clear conclusions about what is going on is difficult.



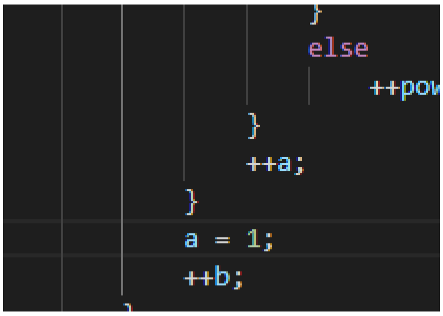
```
<bb 15> [local count: 14598064]:
# b.0_117 = PHI <1.0e+0(14), _16(26)>
# k.1_118 = PHI <k.1_135(14), k.1_2(26)>
# powcheck_lsm.26_125 = PHI <powcheck_lsm.26_188(14), powcheck_lsm.26_110(26)>
# powcheck_lsm.27_129 = PHI <0(14), powcheck_lsm.27_113(26)>
# d_lsm.28_88 = PHI <1.0e+0(14), d_lsm.28_156(26)>
# d_lsm.29_157 = PHI <0(14), d_lsm.29_162(26)>
# x_lsm.30_163 = PHI <x_lsm.30_178(14), x_lsm.30_164(26)>
```

_16(26) can be found much further inside the program:



```
<bb 26> [local count: 14598064]:
# powcheck_lsm.26_110 = PHI <powcheck_lsm.26_125(15), powcheck_lsm.26_124(25)>
# powcheck_lsm.27_113 = PHI <powcheck_lsm.27_129(15), 1(25)>
# d_lsm.28_156 = PHI <d_lsm.28_88(15), d_lsm.28_154(25)>
# d_lsm.29_162 = PHI <d_lsm.29_157(15), d_lsm.29_160(25)>
# x_lsm.30_164 = PHI <x_lsm.30_163(15), _4(25)>
# x_lsm.31_166 = PHI <x_lsm.31_165(15), 1(25)>
# y_lsm.32_168 = PHI <y_lsm.32_167(15), _5(25)>
# y_lsm.33_170 = PHI <y_lsm.33_169(15), 1(25)>
# c_lsm.34_172 = PHI <c_lsm.34_171(15), _6(25)>
# c_lsm.35_174 = PHI <c_lsm.35_173(15), 1(25)>
_16 = b.0_117 + 1.0e+0;
if (k.1_2 >= _16)
  goto <bb 15>; [89.00%]
else
  goto <bb 27>; [11.00%]
```

This actually corresponds towards the end of the main function:



```
                }
                else
                    ++pow
            }
            ++a;
        }
        a = 1;
        ++b;
```

As we are towards the end of the loop – if that condition passes we go back. This is also why the PHI function is required before.

However, when we consider the /O1 version *without* the flag, this only happens for b. In /O1, even the other variables are optimised under multiple PHI values that carefully consider the possibility of applying loop invariant code motion to values in the inner loops (i.e, the while statements). This is harder to prove by changing the code in the program, and hence was not done.

Summary

TLDR;

It took four weeks for me to realise that one line was the reason for a program being three time slower.

But that would not explain the *work* I did in realising that. The idea of this project was to take a deeper look into *what* in a program could affect performance (so drastically as I showed in Linpack and my benchmark program). When one compiles a C program, there are *optimisation flags* which when set, improve the performance in a variety of ways (for example, by iterating recursive functions). To the programmer, the code does not change. The output (the executable) runs faster as it is optimised. But what exactly happens between these flags?

This required me to look up the documentation (and often with less than desirable results), and then write separate programs to try running the program with each of the flags that make up the difference between say `/O0` (slow) and `/O1` (fast) code. However, I was disappointed, as I was unable to identify a flag that could do so (and I found out later that the compiler ignores them (!!) in some cases). Then I decided to try it the other way around: start with `/O1` (fast) code and then *disable* (rather than enable) each flag which are collectively responsible for the increase in performance. I managed to get some interesting results and was actually able to explain the difference between `/O1` (fast) and `/O2` (faster) code in Clang, and `/O2` (faster) and `/O3` (fastest) code in g++ (with Linpack). That was nice, but I was still looking for a solution to why `/O1` (fast) code is that much faster than `/O0` (slow).

To do that, I had to basically disassemble the result that comes when compiled with `/O0` and `/O1` and look for clues. When doing that, to my surprise, all the flags that get enabled were listed – and they weren't what was given in the official documentation! So I had to rerun all the tests again, and I found out that two flags were responsible for the difference between `/O0` and `/O1` in my benchmark program! But how could they be making such a huge impact (we're talking over 6 times!) in performance? It was time to disassemble the program again, and to my shock, I solved the mystery of one flag by writing x^2 as $x * x$; the library function (provided by the language I used) to find the power of a number is badly inefficient for a square! Just that nearly tripled the performance of the program!

We then finally try to solve the mystery of the other flag, which takes us into trees (not the datatype which most would have been familiar), and try to understand some GIMPLE code to figure out how the loop invariant code motion could possibly take place inside the program.

Acknowledgements

While this is not a normal feature of my reports, it is appropriate to acknowledge the following, amongst others:

- Lord Laidlaw and the Laidlaw Foundation – for giving us the opportunity to take on this research scholarship to start with. Even though I do not consider this as a major factor, they pay us fairly well.
- My supervisor Dr Susmit Sarkar – for helping me work on the project, giving me valuable pointers (and being supportive of my ideas in the first place even though I had never done anything on that scale before).
- CAPOD – as they manage this scholarship for St Andrews and they've done good work on that, with the leadership weekends, lunches and general support.

References

The nature of this work meant that I consulted many online sources, from the official documentation of Clang, g++ and Microsoft (which was the better one to me), to research papers and StackOverflow posts (which often did a better job than the official documentations). Where possible, they have been linked to directly in this report, and hence there is no separate section listing *all* the websites that were consulted.

Appendix: The research proposal

This is the research idea/proposal that I sent to the supervisor. This was before I applied for the scholarship and is significantly different from what I actually did. In particular, I am not doing research on multiple threads for now.

Analysing the factors which could affect speed of a task and their effects on performance

Abstract and research context:

In the 1990s, when computers were limited by their clock speed, how 'fast' a computer was could be determined simply by looking at its clock speed (at least for the same microarchitecture of CPUs) – the higher the clockspeed, the faster the computer would become.

Then in 1998, Intel's Celeron was launched. It was like the Pentium but didn't have any secondary cache at all [1]. That meant that even though the clockspeed and core architecture of the Pentium and Celeron were the same, the Pentium would be significantly better than the latter since the lack of a cache hit it hard on things like office processing. That meant that a different metric had to be used for the comparison of performance – since some applications would be impacted by the lack of a cache than others.

As computers became faster, things like IPC started to come into play; the Pentium 3 was actually faster than the Pentium 4 when compared against the same clockspeed [2]. This meant that simply looking at the clockspeed wouldn't tell the full story. AMD in particular did not use a clockspeed to measure its performance; an Athlon XP 1500+ actually had a clockspeed of 1.33 GHz.

The advent of multi-core processors in the late 2000s complicated things further. Suddenly there were multiple metrics by which a CPU (let alone a computer) could be compared. It's not as simple as whether a 6 core without SMP (i.e, Intel's Hyperthreading) is as fast as a quad-core *with* SMP. For instance, Geekbench gives a slender lead to the i7, but Cinebench thinks otherwise. Many quad-cores would lose to dual-cores when gaming for instance, as those applications were optimised for only one core back then (which was combated with Intel's Turbo Boost feature in 2008). And it gets only worse on mobile devices (eg: what about devices with the *big-little* config?).

My proposal involves analysing the performance differences in running the same task over different compilers, languages and machines and determine why that could be the case – and how they effect comparisons between devices.

I have personally worked on a C++ benchmark program which uses the time taken to find the first n Pythagorean triplets as the metric. As I have shown, the scores scale abnormally on the latest compilers (XCode and VS 2017, but *not* VS2008 or earlier) with the theoretical CPU performance. Why does that happen? IPC improvements do not seem to be considered at all irrespective of the generation. It might be that the math coprocessor is pretty static with generations, but then why is this anomality inconsistent with compilers?

As part of an extension for my CS2101 MP2 practical, I built a simple multi-threaded benchmark around the Hadoop framework. How does the performance vary if it was C++ instead of Java? What about different Java versions or even different operating systems? My limited testing shows that it is a more robust than my C++ program, but is there a catch to it? What about with high thread counts?

When I push 25 threads at once on my multi-threaded C++ benchmark program, some computers show a spectacular decrease in performance (my Ryzen 7 laptop showed a 10x decrease despite being a quad-core/octa-thread CPU; the lab machines shave off 60% of performance). In fact, this has been reported. Is this due to the scheduler of the operating system, or is it something deeper?

Potential outcomes

- Measuring the increase in performance with increase in cores for different CPUs. This can be more than the number of threads of the CPU.
- Data-supported explanations of trends observed from the same, including a potential explanation for the possible reasons for any anomaly observed
- Analysing different common frameworks (like Hadoop), languages (like Java, C++; including different versions) and compilers and determining the possible reasons for any unusual variations on performance (like the C++ scaling problem described earlier). Determining the potential differences in performance for upcoming versions (eg: C++ 19/VS 2019).
- Testing specific parts of a CPU (where possible) and reporting any parts (i.e, a component which seems to be static in performance irrespective of the processor) which could explain the mis-scaling problem describer earlier

Potential results

- Some components not scaling with IPC improvements – might mean that there is no actual improvement to that component
- A scheduler, operating system or (critically) a CPU having issues handling multiple processing cores or on certain tasks
- (A) framework(s) which specifically scale very well with core count compared to others

References

- https://www.tomshardware.com/reviews/cpu-performance-socket-7-slot-1,63-5.html
- https://arstechnica.com/civis/viewtopic.php?f=8&t=313654