Optimization

The kind of software we need to write when working with automated systems and with robotic systems often requires us to think carefully about ***optimization***. There are many reasons for this, but among the most important ones:

a) Automated and robotic systems often run on devices that have limited hardware resources. This often means a smaller memory capacity, reduced or slower processing capacity, and power supply limitations (e.g. battery powered devices with limited battery life which must be carefully preserved).

b) Automated and robotic systems often have to perform tasks that have strict timing requirements placed on them. For example, the anti-skid technology now available on many consumer cars has to detect a skid, determine the appropriate corrective action, and apply that corrective action within a very short time interval – failure to do so will result in failure of the system to prevent a car from skidding.

While the problem of designing, implementing, testing, and validating systems that have to perform under strict timing constraints is a wide ranging and complex topic (such systems are called ***real-time systems*** and constitute an entire field of study in computer science and engineering); one important aspect of writing software for automated systems is that of ***profiling*** and ***optimizing*** software to make the best use of available resources.

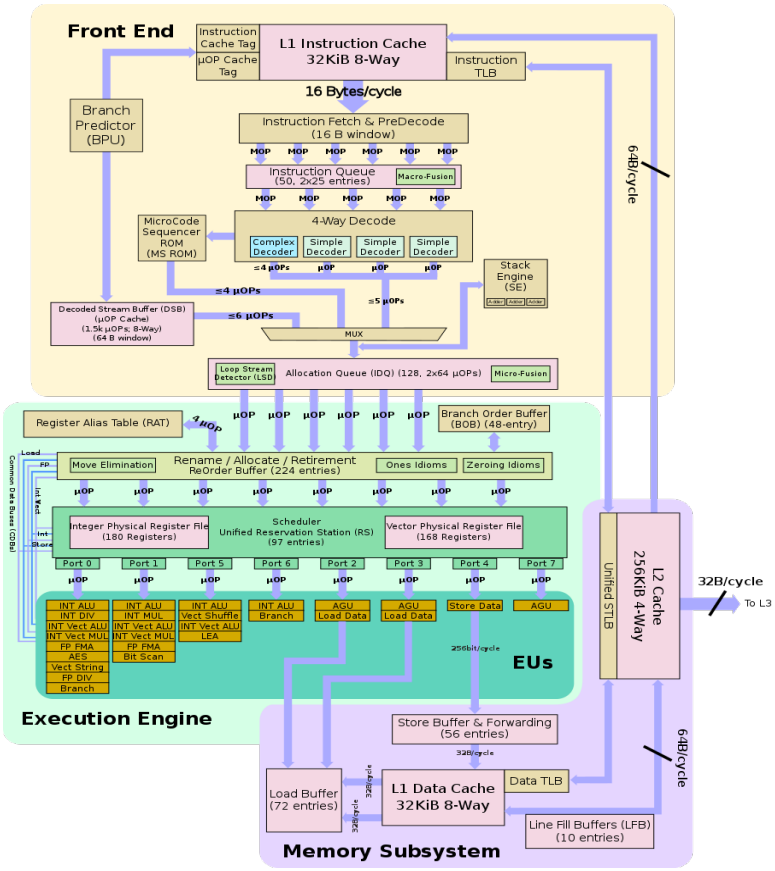
Here we will focus on the problem of optimizing software to make the most of the processing resources available. This is relevant for the implementation of computationally intensive tasks such as ***robot localization***, ***robot perception, task planning (A.I.), control,*** and ***signal processing and denoising***. These are just a few of the topics we have covered in the course, and all of them involve a computational component the software for which should be carefully optimized.

Since our goal is to make the most of available computational resources, we must begin by reviewing how modern CPUs are organized, what their components and features are, and how each of these components contributes to the CPUs processing capability. We will then consider how to write software that makes the best use of the available CPU features in order to achieve the best possible performance.

A short review of CPU architecture

In order to be able to design and implement a program that makes the most of the computational resources provided by modern CPUs, we need to consider the blocks that make up a modern CPU. The specific arrangement and particular details of these blocks will change from one CPU to the next, and from one generation of processors to the next, but the functional components and their place in the context of CPU performance remain consistent.

The figure below shows a block diagram for a modern CPU (Intel’s Skylake architecture, produced from 2015 to 2019).



Intel Skylake architecture (showing a single core). Source: Wikimedia commons, author: Chipwikia, CC-SA4.0

The figure shows the main components of a modern CPU, and allows us to note three types of components present in evert modern CPU:

CPU control and sequencing components: Yellow, beige, and green
Processing and computation: Orange
Memory (storage) and buffers: Pink

The main data pathways and connections are shown in purple.

The general principles that contribute to the performance of modern CPU, and which we have to consider when developing programs that require high performance are:

1) Instruction-level parallelism

All modern CPUs support a significant amount of parallel computation. In the diagram above, you can see a number of independent ***execution units*** in orange, each can perform a ***a specific type of instructions*** and is optimized to carry out these instructions quickly. What matters for us is to understand that ***each of these execution units can be working on instructions concurrently***. In other

words, the CPU is able to execute a large number of instructions in parallel – ***provided the instructions are of a type that can be assigned to an available execution unit.***

Normally, when we are writing a program, we do not think of parallel computations being carried out at the CPU instruction level. We may plan for particular sections of our program to be parallelized – maybe across multiple cores available in a CPU, or maybe within a cluster of computers. But, each individual CPU core is in effect a parallel computation unit.

Fortunately, modern compilers, including those for languages such as C and C++ are quite good at optimizing the instruction stream so that it takes advantage of the available hardware. For instance, it may reorganize the sequence of mathematical operations so that integer computations take advantage of the multiple integer execution units in the CPU, or it may reorganize computations into blocks that can be carried out much faster in vector execution units.

Program code that works on array data, and performs the same operations on the data in the array can often be optimized to some degree to take advantage of CPU hardware. One reason why this kind of optimization is often left to the compiler is that it is highly dependent on the specific CPU the program is being compiled for – writing code that is highly optimized for a specific CPU would make it difficult to re-compile it on a different architecture and achieve good performance.

2) Pipelining

Every CPU performs a continuous loop of instruction execution whose basic steps are:

- ***Fecth:*** Get the next instruction in the program's instruction stream from memory
- ***Decode:*** Determine what the instruction does, which operands (if any) are required, and what components in the CPU will be involved in carrying it out. Set things up for the instruction to be carried out
- ***Execute:*** Carry out the instruction, carry out any computation required, and update any registers as needed.
- ***Writeback:*** Update results in memory as required by the instruction

The ***Fetch-Decode-Execute-Writbeack*** cycle happens continuosly as long as the CPU has power, and every instruction has to go through the four stages of execution. Everything is synchronized by the CPU clock. Each of the stages of the FDEW cycle requires 1 clock cycle – therefore, each instruction requires 4 clock cycles to complete, and the CPU can complete one instruction every 4 cycles.

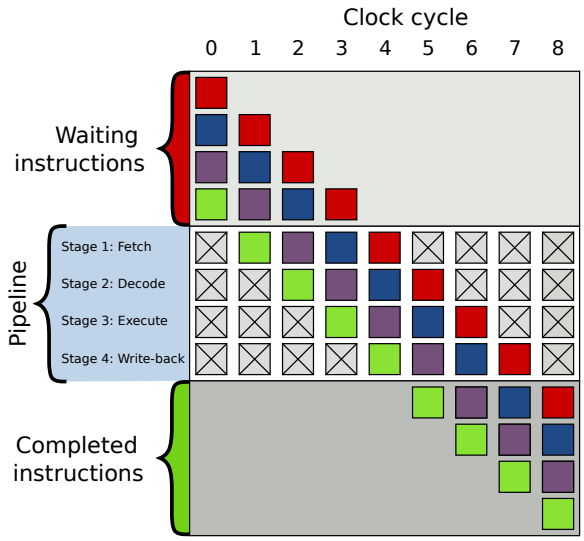
The idea behind pipelining is that we can take advantage of the fact that each of the stages of instruction execution uses separate components of the CPU. In a simple CPU design (without a pipeline), while an instruction is being *fetched*, the circuitry that does the decoding, the execution, and the updating of memory locations remains unused. While the instruction is being *decoded*, the circuits that fetch, execute, and update memory are unused; and so on.

This is not a good use of the CPU's components – imagine for instance a car assembly line that has 20 or 30 steps, each of which performs some part of the process of building a car. Except that we set it

up so that only one car can be inside the entire assembly line at any given point in time. Clearly we are wasting the assembly line by keeping most of its stations empty most of the time.

A pipelined assembly line would have cars at different stages of completion moving through the line at all times. No station is ever empty, and as a result, the pipelined assembly line puts out cars at a much higher rate than the non-pipelined one.

A pipelined CPU works in the same way. In a pipelined CPU, there is one instruction at each of the four steps of execution at (ideally) all times. Thereby we don't have unused CPU components. This is illustrated in the figure below:



A 4-stage pipelined CPU. Source: Wikimedia, author: Cburnett, CC-SA3.0

Note the following: The first instruction (green) takes 4 cyles to complete, as expected since it has to go through 4 stages and each takes one CPU clock. However, once the pipeline is full, the CPU appears to be completing one instruction at each CPU cycle.

A non-pipelined CPU would take 16 clock cycles to complete the 4 instructions shown above. The pipelined CPU takes only 8. If the instruction stream was much longer, and the pipeline remained full, the performance of the pipelined CPU would approach 4x the performance of the non-pipelined one (in terms of completing instructions in any case).

All modern CPUs are pipelined. However, they do not have only 4-stage pipelines like the simple one shown above. This is because of the following:

- Since each stage of execution must have enough time to do its work. The maximum clock rate is determined by the stage in the pipeline that takes the longest time to complete
- However, we could *break-up* each of the stages in the pipeline into smaller steps that can be completed in less time – thereby allowing for a higher clock rate

And so modern CPUs have pipelines between 5 and 19 steps in length – and in effect, there are pipelines of different lengths within the same CPU as different ***execution units*** will take more or fewer steps to carry out when working on their instructions.

The length of the pipeline is not very important, and is CPU dependent. What is important is to understand ***what happens when the pipeline can't be kept full.***

For a pipeline to have the expected effect on performance, the pipeline must be kept full of instructions at all times. If the pipeline is not full, then the performance of the CPU decreases (in the limit it goes back to the performance of a non-pipelined CPU). Keeping the pipeline full is not a trivial problem because of the following common programming constructions:

Computations that require results from one another

For instance, consider the sequence:

```
a=x+5
b=a+3
c=b*2
```

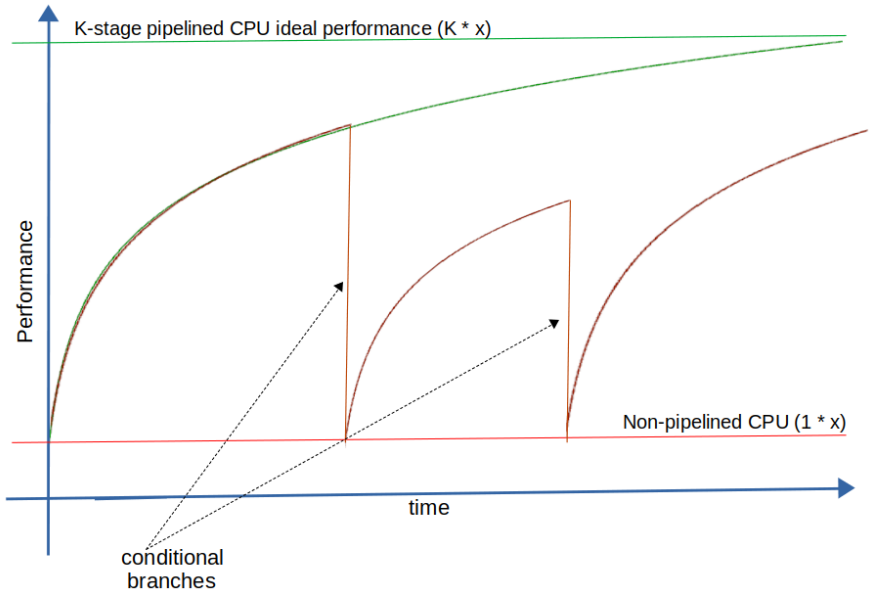
Once the instructions for the computations above enter the pipeline we will find that ***we can't complete b=a+3 until a=x+5 has been completed***. Therefore at some point the instructions that compute the result for ***b=a+3*** get ***stuck*** in the pipeline until the results they require are available. This does not mean that the pipeline goes empty (as it may with conditional branches), but it creates a ***pipeline bubble***, which means a hole somewhere in the pipeline whereby some of the stages are empty while results are being computed.

Pipeline bubbles are bad for performance. Modern CPUs implement a few techniques within their architecture to reduce the impact of pipeline bubbles, but we can certainly help by carefully reviewing and where possible re-organizing computations so as to introduce as few dependencies as possible between nearby computations. Modern compilers also do a good job of reorganizing code so that the resulting instruction stream so as to avoid bubbles as much as possible.

Conditional statements (branches)

These correspond to any *if-then-else* instructions or as part of the *loop termination condition*. These are a problem for pipelined processors because depending on the result of the condition, ***one of two possible streams of instructions must be executed***, but the CPU doesn't know which until the instructions that evaluate the condition have been fully executed.

Therefore, *once the instructions that evaluate the condition enter the pipeline*, we have to wait until the CPU has completed executing them. Meanwhile, the pipeline goes empty as the CPU does not know which instructions to fetch next. Branches can have a ***fairly significant impact on performance***.



The figure above illustrates the problem. A pipelined CPU with ***K*** stages should asymptotically approach a performance that is ***K-times*** that of an equivalent, non-pipelined CPU. The green curve above shows the peformance of the pipelined CPU ***when the pipeline remains full.*** As expected, this approaches the ideal performance asymptotically.

However, conditional branches cause the pipeline to go empty (because the CPU does not know which instructions to execute next until the condition has been fully evaluated). As a result, the ***actual performance*** of the pipelined CPU looks as shown by the dark-red line. The ***average performance*** of the CPU when the program contains many conditional statements can be much, much lower than the ideal maximum.

It should be clear that we should make every effort ***to keep the pipeline full at all times***. This means reducing as much as possible the effect of conditional branches. All modern CPUs provide support for this in the form of a ***branch prediction unit (BPU)*** - you can see it in the CPU block diagram above. The goal of the BPU is to ***enable the CPU to guess which branch will be taken*** and to start fetching and processing the corresponding instructions ***even before the branch condition has been actually evaluated***. This technique is called ***speculative execution*** and is essential to help sustain the performance of the CPU in the presence of conditional statements.

How does the BPU work?

- It creates a table with entries for the conditional statements found in the code recently executed.
- It keeps statistics of ***how often each of the branches is taken***
- It guesses the branch that will be taken based on these statistics

The exact details of how this is done vary from one CPU to the next, but in general the BPU can do a reasonably good job provided the programmer is aware of what it does ***and is careful about how conditionals in the program will interact with the BPU***.

Here are a few guidelines on how to reduce performance problems due to conditionals:

- ***for loops*** are good: the loop condition is always ***false*** except at the end of the final iteration in the loop. This makes the condition easy to guess by the BPU.

- ***Avoid complicated conditions***: you want to make your conditionals as ***predictable*** as possible, in the sense of having one result being significantly more common than the other. This allows the BPU to guess correctly often. A conditional with no clear common case will likely cause problems. Therefore, avoid complicated conditional statements consisting of multiple boolean conditions joined by logical operators. Simplify logical expressions as much as possible.

- ***Nested conditionals are ingerently problematic:*** This will translated into a sequence of guesses the BPU needs to get right. Suppose you have 3 nested conditional statements, and the BPU can guess right 90% of the time for each of them. The chance it will guess right for a sequence that requires all three statements to be evaluated is $.9^3 = .72$ – not wonderful. Avoid nested conditionals if possible, and where they are required, ***order the conditionals so that the common case is first***.

- ***Consider carefully the patterns in your data:*** If there is a predictability to the data you are working with (that affects the results of conditions in your code), carefully think about how to ***craft the conditions in your code so that they take advantage of these patterns to become more predictable***. Alternately, ***consider doing some extra work to organize data into predictable patterns.***

A classic example of how patterns in the data can affect the run-time of a program, consider the following program:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define ARR_SIZE 250000000
#define TIMES 100
#define DO_SORT 0

int compareI(const void *a, const void *b)
{
  if (*(int *)a<*(int *)b) return -1;
  if (*(int *)a>*(int *)b) return 1;
  else return 0;
}

int main()
{
 int *array;
 time_t start,end;
```

```
 double cputime=0;
 double sumtime=0;
 double tottime=0;
 double sum=0;

 array=(int *)calloc(ARR_SIZE,sizeof(int));
 printf("Initializing array\n");
 for (int i=0; i<ARR_SIZE; i++)
     *(array+i)=(int)(100.0*drand48());

 if (DO_SORT)
 {
  printf("Initialized array - calling quicksort!\n");
  start=clock();
  qsort(array,ARR_SIZE,sizeof(int),compareI);
  end=clock();
  cputime=((double) (end - start)) / CLOCKS_PER_SEC;
  printf("Time taken by quicksort %lf\n",cputime);
 }

 printf("Computing the sum!\n");
 start=clock();

 for (int i=0;i<TIMES;i++)
    for (int j=0; j<ARR_SIZE; j++)


 end=clock();
 sumtime=((double) (end - start)) / CLOCKS_PER_SEC;
 printf("Time taken adding up entries in the array %lf\n",sumtime);

 tottime=cputime+sumtime;
 printf("Total time=%lf\n",tottime);
 printf("The sum is %lf\n",sum/TIMES);

 free(array);
 return 0;
}
```

The program creates an array of 250 million integers and fills the array with random integers in [0,100). The program then computes ***the sum of all entries in the array that are >= 50***. This is done ***100*** times so that the time it takes to do the sum is in the range of seconds (otherwise, our current computers are simply too fast and timing results are strongly affected by other things hapenning inthe CPU).

The key part of the program is the conditional statement:

```
if (*(array+j)>=50) sum+=(*(array+j));
```

When the array is ***not sorted***, consecutive entries in the array are equally likely to be less than 50, or equal/greater than 50. This makes the chance we have to add to the ***sum*** 50-50. The BPU can not be expected to guess whether or not ***sum*** will be updated at each entry in the array, and as a result, the program causes a large number of ***branch mispredictions*** – which are costly: the CPU pipeline will have been filled with ***instructions from the wrong program branch*** which have to be ***flushed***, and then the pipeline has to be ***re-filled*** with the instructions from the correct branch.

With an ***un-sorted array***, the run-time for the program above is:

```
 ./a.out
Initializing array
Computing the sum!
Time taken adding up entries in the array 96.484386
Total time=96.484386
The sum is 9313156556.000000
```

Now, notice the program has an optional pre-processing step controlled by the ***DO_DEBUG*** flag. When set, the program will first call ***qsort()*** to sort the array before proceeding to the section that computes the sum of elements >= 50.

Wth a sorted array, the ***branch that updates 'sum'*** will be taken for approximately one-half of the array, and the ***branch that does not update 'sum'*** will be taken for the other half. So the BPU can guess which branch will be taken almost all the time – except for the part near the middle of the array where the decision to take one branch or the other changes.

With the block to sort the array enabled, the program's runtime is as shown below:

```
./a.out
Initializing array
Initialized array – calling quicksort!
Time taken by quicksort 25.671746
Computing the sum!
Time taken adding up entries in the array 25.671746
Total time=46.520335
The sum is 9313156556.000000
```

Note that it took ***half the time*** to compute the ***sum*** with the sorted array (this includes the time to do the sorting). This is surprising: ***in terms of computation*** the program that uses ***qsort()*** is doing a lot more work! But in terms of actual ***runtime***, the CPU is able to complete the task much faster when we are able to ***remove most of the branch mis-predictions***.

This clearly shows just how large an impact ***branch mis-predictions*** can have on CPU performance when running programs. This is one facctor that should be paid close attention when optimizing programs for computational performance.

3) Caching

The third and possibly most important feature of modern CPUs, one with possibly the greatest effect on performance, is ***caching***. One key limitation of current computer architectures is that the large memory banks used to hold both the programs and the data they work with are *too slow* compared to the rate at which the CPU can process information.

Recall that there are two different types of RAM technology: ***static ram*** which is built with flip-flops, and which can *change state* (we can write a value to it) incredibly fast – but because it consists of multiple transistors it is comparatively expensive to fabricate. The alternate technology is called

dynamic ram, and is built with *capacitors* instead of flip-flops. Because of that, it is much cheaper to fabricate, but it is also significantly slower – capacitors take time to charge and discharge, so a change of state is much slower for ***d-ram*** than for ***s-ram***.

Because of the price difference, a computer's main memory bank is made using ***d-ram***. Typical capacities are in the order of ***gigabytes*** which is enough for storing both complex programs and large amounts of data these programs need to work on. But this becomes a problem for the CPU – recall that we can only expect the CPU to achieve its full performance if ***we can keep the CPU pipeline full at all times***. This is not possible if we have to wait for the slow dynamic memory.

In effect, the large bank of *comparatibely slow dynamic ram* can become a bottleneck, the CPU spends significant amounts of time idle, waiting for data from the main storage in RAM.

To reduce this problem, CPUs implement ***internal data caches***. These are visible in the CPU diagram above. There are ***several levels of caching*** with increasing capacities, and each CPU implements a slightly different policy for how data in the caches is managed. However, the general principles are consistent:

- The program instructions and the data that the CPU is working on will be ***pre-fetched*** into the cache in advance of when the CPU will need them. This is done by dedicated memory management circuitry. ***This will be a key point*** for us to consider when designing our programs for performance.

- There are multiple levels of caching, the smallest and fastest is L1, and the largest (in terms of capacity) but also slowest is L3.

- Why is there a speed difference? Because the cache is *much smaller* than the main memory bank, if the CPU requires specific data or instructions it first *needs to check whether those are stored in the cache*. This is done by consulting an index called the ***TLB*** – *Translation Look-aside Buffer* (you can see it in the CPU diagram above). Checking the index takes time, so the larger the cache, the longer it takes to search the TLB to determine if what the CPU wants is in the cache, and where in the cache it is. To optimize performance, the cache is divided so that most look-ups take place in the smallest L1 cache, and are thus less time-consuming.

- The resulting *memory hierarchy* consists of different levels of storage with different access speeds. Typical *latencies* for accessing information are:

    L1 cache: 1-2 nanoseconds
    L2 cache: 5-10 nanoseconds
    L3 cache: 10-20 nanosecods
    Main RAM: 60-100 nanoseconds

You can see that main RAM can be up to *two orders of magnitude slower* than the L1 cache. When running a program, the CPU must fetch *instructions* and *data* it needs at each particular time. The CPU looks for the instructions and data using the following process:

- Search L1, if the data and code are there, access them – this is the optimal case
  - If data or code not in L1, search in L2, if found, load them into L1 and use them.
    - If data or code not in L2, search in L3, if found, load into L2 and L2 and use them.
      - If data or code not in L3, fetch from RAM and update L3, L2, and L1 then use them.

From the above it should be clear that if the data or code that the CPU needs are not in the cache, there will be a *significant wait* while we check the entire memory hierarchy looking for them, and then fetch them from main memory. This will have *a much larger effect on CPU performance than pipeline bubbles or branch prediction errors*.

Considerations for Writing Cache-Friendly Code

Your program must be ***predictable***.

- In terms of the ***instruction stream***, a continuous, linear stream of instructions is the best. *Long instruction branches* (blocks of code whose execution depends on a conditional statement), and *function calls* can be a problem since they interrupt the linear flow of the instruction stream.

- In terms of the ***data*** the program is working with – write code that accesses data in a *predictable pattern*. For example, a program that works with a huge array, but does so using a *for loop* should have good performance because the CPU memory systems can easily figure out what data will be needed next and where that data is – so it can be *pre-fetched* into the cache. Conversely, accessing data without a fixed, easily predictable pattern, using complex data structures with multiple fields only some of which may be needed at any given time, and dynamic data structures in which data is stored not contiguously are likely to cause performance issues.

In short, when designing your program you must be keenly aware of ***data locality*** and organize your program's data so that it will be accessed in a predictable pattern, and so that data that will be used at about the same time, will be stored close together in memory. You must also consider writing code that avoids wherever possible breaking from a linear stream of instructions.

General advice for writing well-performing code in C

1) Carefully organize data to achieve good ***temporal locality***. Use simpler data structures such as arrays, rather than dynamic data structures such as linked lists or trees. Store data if possible so that it is roughly organized in terms of the order in which it will be accessed.

2) Avoid repeated/frequent dynamic allocation (and de-allocation) – pre-allocate data where possible.

3) Organize ***if-else*** statements so that the ***common case is first***. This increases the chance that the code for the common case is already in the cache. Make each conditional block as short as possible.

4) ***Reduce*** or ***eliminate*** function calls. For short functions that are called often, ***inline*** these functions.

5) Move loops ***into function calls*** rather than having ***a function call inside a loop***. Use early loop termination.

6) Avoid un-necessary cata type conversions (casting).

7) Reduce the number of function parameters, pass by reference (using pointers), not by value, in general try to reduce the amount of stack management that needs to be done for a function call

Code Profiling

Often, once we have carefully thought of how to organize our code and data keeping in mind how it will interact with the CPUs different components, we will still end up with a program whose performance can be significantly improved. But any further improvement requires us to analyze how the program behaves and measure which parts of the program may be causing a performance problem.

We can do this by ***profiling*** our program. There are several tooks we can use for this, but a fairly powerful tool that is available on most Linux distributions is ***valgrind*** – which we also use to check for, and remove memory management bugs.

To profile a program with ***valgrind***:

- Compile your program as you normally would (including compiler optimizations), but add the '-g' flag so the executable contains debugging information

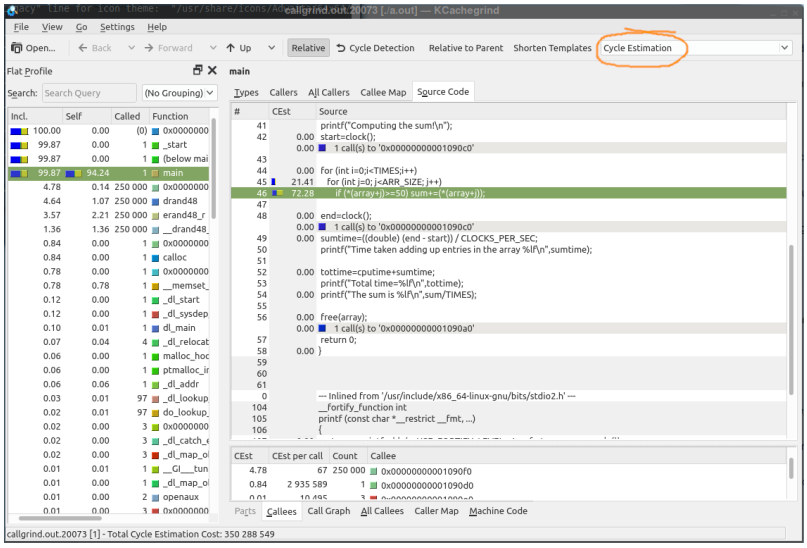
- Run the program using ***valgrind*** to create the profile:

```
valgrind --tool=callgrind --branch-sim=yes --cache-sim=yes a.out [a.out arguments if any]
```

The call to ***valgrind*** will produce a profile in the form of ***callgrind.out.NNNN*** where the last bit is a number related to the process-ID of the walgrind process we just ran. Several runs of the profiled will create different output files, make sure you're looking at the correct one. To visualize the results of the profiling process, we use a tool called ***kcachegrind***.

```
kcachegrind callgrind.out.NNNN
```

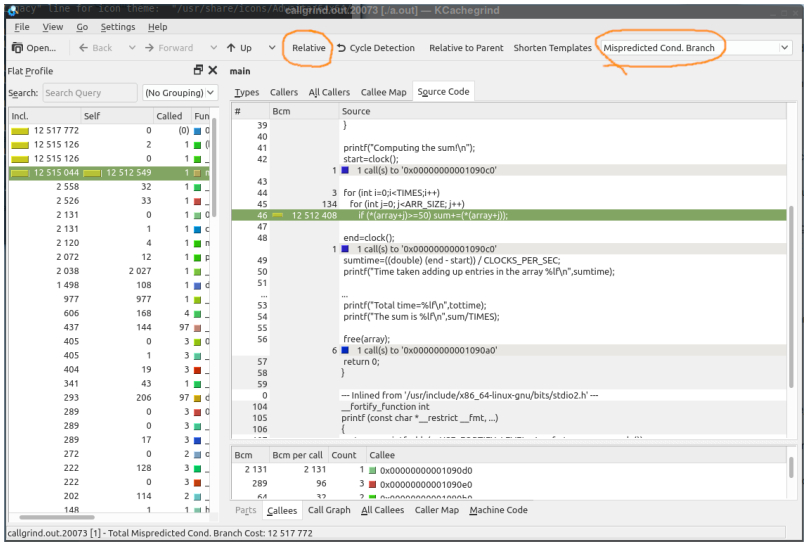
As an example of what this looks like for a real program, the images below correspond to the profiling results for the program described above (in the branch prediction section) that computes the sum of a large number of random integers.



The image above shows a typical visualization from ***kcachegrind***. Notice that we have selected the ***'Source Code'*** tab – so we can see each line of our program. Note the pull-down menu with the ***'Cycle Estimation'*** choice currently selected (circled in orange). This gives us an ***estimate of the percentage of the program's runtime*** caused by each of the lines in the code.

You can see that (as we had expected given our discussion in the pipelining section), the single ***conditional statement*** that decides whether or not to accumulate an entry in the array is taking up most of the program's runtime.

This instruction ***is likely a bottleneck*** but we need to find out for sure, and we need to determine ***the reason why this is a bottleneck***. The same pull-down menu we used to select ***'Cycle Estimation'*** can be used to give us information regarding ***branch mis-predictions***, or ***cache misses***. Let's have a look at the results for ***branch mis-predictions***.



In the image above, notice we have selected ***'Mispredicted cond. Branch'*** and we have toggled the ***'Relative'*** button (both circled in orange). We can now see that the conditional statement we were worried about has caused over 12 million mis-predicted branches (the program was working with a 25 million entry array, so this is roughly 50% mis-prediction rate, as expected).

We now know that this line is indeed a bottleneck and should be causing a significant performance problem, and we can think of ways to reduce or remove the problem (in this example, we already know that pre-sorting the array will remove most branch-prediction problems and reduce the overall run-time).

A similar process can be used to find bottlenecks due to ***problems with caching***. The profiler provides information for the different levels of cache, as well as for the ***instructions cache*** and the ***data cache*** separately for the ***L1*** cache.

In conclusion: When optimizing a program for performance, we need to carefully design the code to account for ***pipelining, branch prediction, and caching,*** and carefully build our code to take advante of these CPU features. We then should run a profiler and spend time carefully studying the results to detect and reduce or eliminate bottlenecks.

The process is work and thought intensive, but can make the difference between a program that runs smoothly even on reduced-capability hardware, and a program that performs too slowly to be of practical use for robotics or automated applications.