**Introduction**

In our previous paper [REF: <https://accu.org/index.php/journals/c389/>] we collected measurements of latency which was later used in our analysis of the impact of cache-line aware data structures. In this paper, which can be thought of as a follow-up to our previous paper, we instead focus on throughput as a quantifiable metric. We will also be using the same hardware x86 as well as versions in our development tool chain. We will refer to some code examples, not shown here, that may require the reader to reference the code in the previous article, but we will attempt to minimize the need to reference, focusing as we investigate many ways to measure throughput and their observed behavioral changes. These changes will be measured in multiple ways to demonstrate the different aspects of throughput detailed below. We will also follow the same approach, which is to say we will first introduce our definitions, then take measurements, introduce aligned data structures (the same queue as before), and measure those observations. Keeping things simple for now so we can pin down the observations and the impact of cache-line aware data structures.

Before we get into the analysis of throughput, why does it matter? What does throughput tell us and what are the various scenarios in which we require a firm understanding of throughput and its implications? Throughput simply tells us how much work/data we can send or distribute given some constraints such as a network cable, or pipe, or wireless signal. If we have an idea of how much work or data we can send to one location to another, we can have a better understanding of the upper limits of our network.

If we can postulate some upper bound limits of our throughput, we can better model how much work can be performed, given the flow of input into our received end points. Once we have a theoretical ceiling of the received data, we can then start to see where these questions become quite interesting in many domains. For example, within a defined time window: how many frames per second can we render on screen, how many updates can we compute, how much market data can we distribute, how many orders can we route to an algorithm or broker, how many messages can we send out to subscribers. These examples all require some basic understanding of throughput, how to measure it, and implications on data load.

Taking it one step further, once ~~you~~ a firm understanding of the expected throughput is established the new insights will lend to more effective design, or even deployment of various components in a technical domain. If you abstract this even further - we can start to explore networks and control flow problems using throughput as a weight on each node in a graph, thinking of routing traffic in real time, and other interesting areas to explore, but not here.

**Definitions**

In order for us to have some way of comparing results we need to provide clear definitions of throughput and discuss some issues surrounding measurement taking.

We offer two types of throughput for our analysis. The first one can be thought of as the classical definition of throughput. This is simply a measure of data of some period of time. The data can be structured in any form and we can slice and dice the time to any unit we deem necessary. Going forward in the paper we refer to this point of observation and or operations over time as spatial throughput(ST). Often the analogy that comes to mind for ST is network bandwidth. Our formal definition for ST will be defined as the number of work units (WU) over time t. We will measure a single WU as the number of cycles observed in the TSC register. All the code found for measuring ST is shown in [Code Segment T1].

[Code Segment T1]

**// one billion is once per second**

**// works\_ is the number of work units**

**// executed this observation period.**

**uint32\_t bandwidth(uint32\_t per = 1'000'000'000)**

**{**

**// CPU is CPU speed in GHz**

**// per is observation time scale units**

**// end\_ - start\_ is the observation window.**

**return (static\_cast<float>((works\_)\*CPU\*per) /**

**(end\_ - start\_));**

**}**

[/Code Segment T1]

*Note: to scale to a time quantum, divide ST by the desired time quantum and multiply by the CPU clock speed.*

A second type of throughput we intend to observe in the paper is how much work can we do given new work to be done. This can be thought of as temporal throughput (TT). Often this is another way of looking at how frequently I am polling versus how frequently I am pulling data and processing that data. This can be thought of as saturation and often quoted as a percentage.

We will measure TT as two distinct types of value. TT ratio (TTR) will be defined as the number of WU divided by poll count and quoted as a percentage. A value of 1 would be full saturation. All the code found for measuring TTR is shown in [Code Segment T2].

[Code Segment T2]

**// polls\_ is the amount of time taken to poll for work**

**uint16\_t saturationRatio()**

**{**

**if (polls\_)**

**return static\_cast<uint16\_t>(**

**(static\_cast<float>(works\_) / polls\_));**

**else**

**return 0;**

**}**

[/Code Segment T2]

The second type of TT we will measure will be TT cycles (TTC). TTC will be defined as the total time of work divided by the total time of work plus the overhead. All the code found for measuring TTC is shown in [Code Segment T3].

[Code Segment T3]

**// saturation\_ is number of cycles spent working**

**// overhead\_ is number of cycles spent polling for work**

**uint16\_t saturationCycles()**

**{**

**if (saturation\_)**

**return static\_cast<uint16\_t>(**

**(static\_cast<float>(saturation\_) / (saturation\_**

**+ overhead\_)));**

**else**

**return 0;**

**}**

[/Code Segment T3]

There are some observational complexities that surround TT data gathering especially compared to ST. In the TT case, we should really focus on using RDTSC as an instruction and not RDTSCP, as RDTSCP basically creates a code fence and can in many ways hamper or prevent hardware optimizations such as out of order execution and execution pipelining that would affect our cycles count. Due to the aggregate nature of measuring many WU in a particular time window the inaccuracies of RDTSC do not affect our measurements in a significant way. In our previous paper each WU was measured and recorded, necessary for the purpose of statistical analysis. When many WU are recorded as an aggregate quantification the inaccuracies are tiny compared to the impact of the serialized RDTSCP instruction.

**Setup**

**ST Setup**

In our first experiment, we will focus only on measuring ST with and without cache-line awareness scaling from 1, 4, 8, and 16 threads, each thread will be pinned to a different core and producing work which will be in the simplified form of a 4-byte atomic increment. Shown below is section [Code Segment EX1] code that performs this first setup.

[Code Segment EX1]

**template <int X>  
struct DataTest  
{**

**alignas (X) std::atomic<uint32\_t> d{0};  
};  
  
void CLTest ( std::atomic<uint32\_t>& d )  
{**

**for (;;)  
 {  
 ++d;  
 }  
}  
  
template <int Align>  
int simpleTest (const std::string& pc)  
{  
 using DataType\_t = DataTest<Align>;  
 DataType\_t data[128];  
 std::vector<std::unique\_ptr<std::thread>>   
 threads;  
 threads.reserve(pc.length());**

**int32\_t core{0};  
 int32\_t idx{0};  
 for(auto i : pc)  
 {  
 if (i == 'p')  
 {  
 threads.push\_back(std::make\_unique<std::thread>**

**(CLTest, std::ref(data[idx].d)));  
 setAffinity(\*threads.rbegin(), core);  
 ++idx;  
 }  
 ++core;  
 }  
   
 auto counters = std::make\_unique<uint32\_t[]>(idx);  
  
 for (;;)  
 {**

**sleep(1);  
 int32\_t i{0};  
 for (i = 0; i < idx; ++i)  
 {**

**counters[i] = data[i].d.load();  
 data[i].d.store(0);  
 }  
  
  
 uint64\_t total{0};  
 for (int i = 0; i < idx; ++i)  
 {  
 std::cout << "d" << i << " = " <<**

**counters[i] << ", ";  
 total+= counters[i];  
 }  
 std::cout << ", total = " << total**

**<< ", avg = " << static\_cast<float>**

**(total) / idx << std::endl;  
 total = 0;**

**}  
 return 0;  
}**

[/Code Segment EX1]

Below we have captured average ST numbers for incrementing a single 4-byte atomic int. In the tests below we use thread numberings from 1 to 16 in powers of 2. In the split NUMA the treads are on different NUMA nodes. Each thread is pinned to a single core and no more than one thread per core.

[Results T1]

[/Results T1]

**TTR, TTC, and ST Setup**

In our second experiment, we will measure TTR, TTC and ST. We will focus on non-cache-line awareness and cache-line awareness for various combinations of producers and consumers. Our code to run and collect results is shown below in code section [Code Segment EX2]. Mark one, two and three methods just fetch the number of cycles using RDTSC.

[Code Segment EX2]

**template <typename T, typename Q, typename WD>  
void consumer(Q\* q, int32\_t iterations, ResultsSync& rs,**

**CycleTracker& ct, WD& wd)  
{  
 while (Thread::g\_cstart.load() == false) {}  
 T d;  
 uint64\_t start;  
 bool work = false;  
 ct.start();  
 for (;;)  
 {  
 CycleTracker::CheckPoint cp(ct, rs);  
 cp.markOne();  
 start = getcc\_ns();  
 work = q->pop(d);  
 if (!work)**

**{  
 cp.markTwo();  
 \_\_builtin\_ia32\_pause();  
 continue;  
 }  
 cp.markTwo();  
  
 // simulate work  
 for (uint32\_t k = 0; k < d.get().workIterations; k++)  
 {  
 // get a local copy of data**

**WD local\_wd(wd);  
 for (uint32\_t it = 0; it < WriteWorkData::Elem; ++it)  
 {  
 // simulate work on data  
 while (getcc\_ns() - start < d.get().workCycles){}  
 // simulate writing results  
 wd.wwd.data[it]++;  
 }  
 }**

**cp.markThree();  
 }  
}**

[/Code Segment EX2]

[Results T2]

[/Results T2]

**Analysis**

**Conclusion and future direction**

**Notes**

GCC 7.1 was used with the flags -std+c++17 -Wall -O3. Boost 1.64 was used on a 36 core Intel (R) Gold 6154 CPU @ 3GHZ. Hyper-threading was not enabled. CPU isolation is in place. Special thanks to Frances Butontempo for her early feedback and suggestions for improvement. Complete source code packages utilized in this article can be made available by contacting the authors directly.