Index

1. Hardware Specifications
2. Micro-architecture Implementation
3. Oprofiler commands
4. Oprofiler Internals
5. Profiling strategy
6. Hardware events
7. Algorithm details
8. Results and analysis
9. Future work to be done
10. Key take-away

Foreword

This document can be used as an introductory material for a student who wants to profile his/her code on Dr.R’s server using oprofile. I have also included some micro-architecture specific details that I found interesting while I was working on the project. For all my output tables, please refer to another word document “Results.doc” and for code dump refer to “code\_(parameters).doc”. If any intuition discussed here is completely wrong, please feel free to mail me as I am always looking forward to learn and understand various concepts better.

In the days to come , I plan to update this document with syntaxes and use-cases of other profilers and simulators too and do a comparative study on each of their positives and negatives.

Push Relabel Profiling Project

**1.Hardware specifications:**

Model name:  Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz

**Detailed model specific data can be found at:**

<http://ark.intel.com/products/75269/Intel-Xeon-Processor-E5-2650-v2-20M-Cache-2_60-GHz>

[http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2650%20v2.html](http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon E5-2650 v2.html)

**Important information for us:**

In our server, we have a 2 :  Intel(R) Xeon(R) CPU E5-2650 v2 CPUs. Each of these processors have 8 cores with hyper-threading facility. Essentially, the optimal number of software threads that can be supported at any point in time by the hardware is 32(8\*2(no. of processors)\*2(Hyperthreading)). It’s a lot of threads, it’s paramount that the software takes advantage of the available hardware resources.

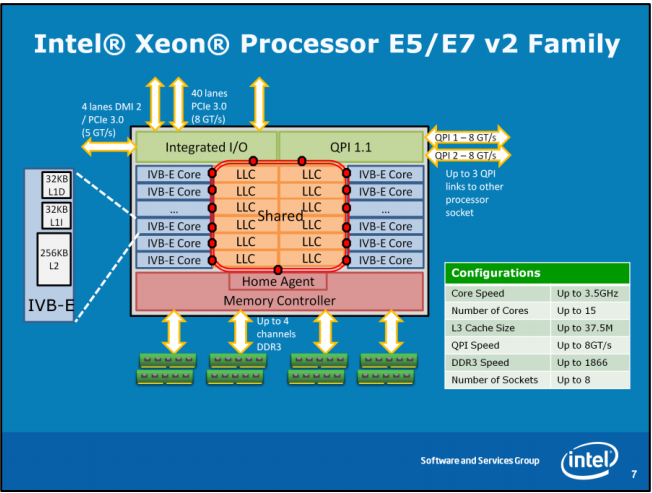
Sizes and organization of caches:

L1 cache size = 32 KB (8 of them in number : one for each core)

L2 cache size = 256 KB (8 of them in number : one for each core)

L3 cache size = 20 MB (1 common for all 8 cores)

Cache organization: Nowadays there aren’t any north-bridges. The caches are a part of the core’s die.



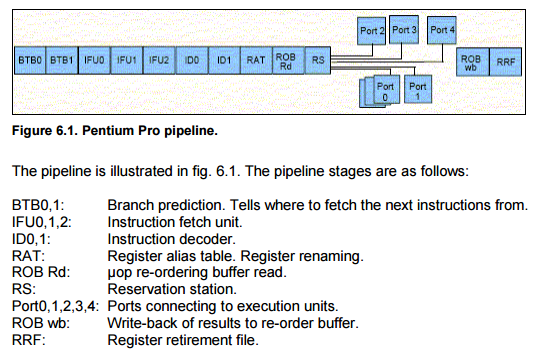
 Source : <https://software.intel.com/sites/default/files/Using_Intel_VTune_Amplifier_XE_on_Xeon_E5v2_or_E7v2_Family_1.0.pdf>

<http://www.techpowerup.com/164677/intel-announces-3rd-generation-core-ivy-bridge-processor-family>

**2.Micro-architecture Implementation:**

**Data path and instruction execution:**

Basic CISC pipeline implementation:



Source: <http://www.agner.org/optimize/microarchitecture.pdf>

pg-46

The above diagram shows the schematic of the Intel-Pentium Pro-Pipeline. The contemporary architectures have a more deep pipelines but the stages and their intuitions remain the same.

Some major intuitions that are different in a CISC or semi-CISC pipeline when compared to a RISC pipeline:

**Instruction Length Decoder:**

* In CISC architecture the length of each instruction is different and can vary from 2 Bytes to 19 bytes. Hence instruction fetch is non-trivial problem.
* RISC pipelines have a constant instruction length hence instruction fetch happens ideally in one clock cycle.
* To solve this problem we have a hardware component called the instruction length decoder, that is at the IFU1 stage. IFU0 fetches instructions as 16 byte chunks, IFU1 decodes the length of the instructions, sometimes we have only parts of the instruction that come in. It buffers the part it has till the next 16 byte block arrives from IFU0 in the next clock cycle. Hence by the time instructions go to IFU2, they are in the form of complete instructions or a set of complete instructions.

16 Byte blocks   complete instructions.

Instruction Length Decoder

For more detailed information about instruction fetch unit and how instruction length decoders behaves during various scenarios such as jumps pls refer <http://www.agner.org/optimize/microarchitecture.pdf>

pg-46.

**The 4-1-1 rule** :

* After the instruction length decoder comes the instruction decoders which translate instructions into µops. There are three decoders working in parallel so that up to three instructions can be decoded in every clock cycle.
* A group of up to three instructions that are decoded in the same clock cycle is called a decode group. The three decoders are called D0, D1, and D2. D0 can handle all instructions and can generate up to 4 µops per clock cycle. D1 and D2 can only handle simple instructions that generate no more than one µop each and are no more than 8 bytes long. The first instruction in an IFETCH block always goes to D0. The next two instructions go to D1 and D2 if possible.
* If an instruction that would go into D1 or D2 cannot be handled by these decoders because it generates more than one µop or because it is more than 8 bytes long, then it has to wait until D0 is vacant.

**Instruction prefixes** :

* Can also incur penalties in the decoders. Instructions can have several kinds of prefixes. An operand size prefix gives a penalty of a few clocks if the instruction has an immediate operand of 16 or 32 bits because the length of the operand is changed by the prefix.

RAT  Register alias table

ROB  Re-order Buffer

RS  Reservation stations

Ports are execution units. We have separate execution units for stack operations. For more information about various aspects of the pipeline pls refer : <http://www.agner.org/optimize/microarchitecture.pdf>

pg-46.

Hence CISC or semi-CISC pipeline is a very deep complicated piece of pure hardware, which is in the heart of high-performance processors. This complicated hardware has been designed with only performance in mind. A design level decision that is in the heart of Intel’s philosophy that is very performance-centric. Later they will realize power is a critical parameter to be considered too.

Hence to take the best of RISC, they introduced a scheme where-by the break(decode) a complicated instruction into its constituent micro-operations right after the decode-stage.

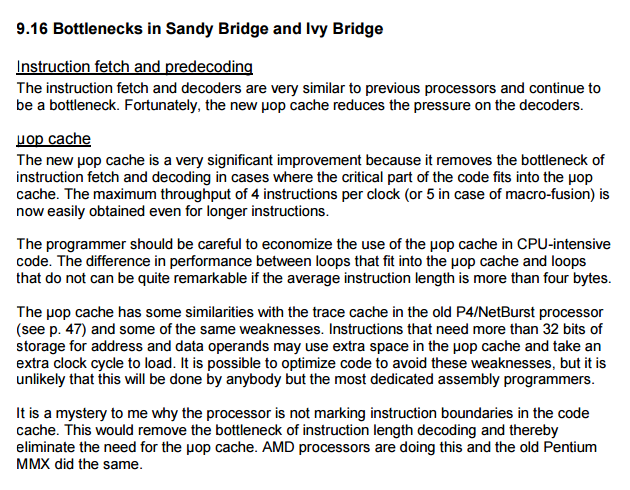
<https://en.wikipedia.org/wiki/Micro-operation>

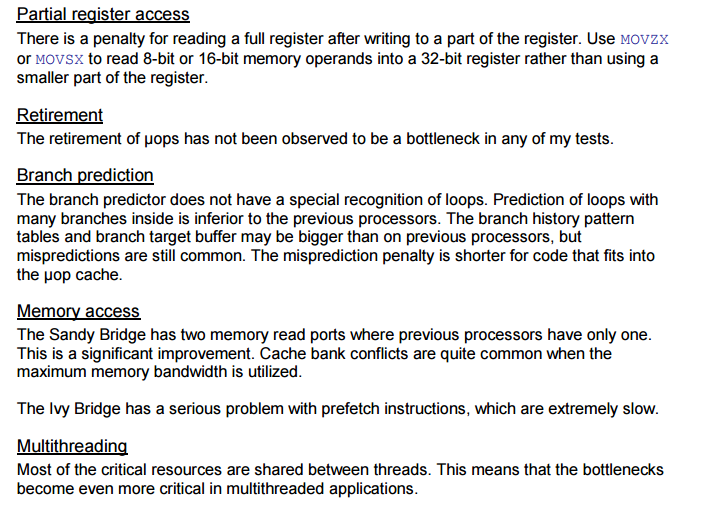
**Ivy-Bridge specific features :**

Contemporary architectures such as Ivy-Bridge has a whole lot of optimizations in this phase of the pipeline. Like

* Micro-operations caching
* Micro-operation fusion
* Loop buffer (holds decoded code of a loop or similar iterative section of the code, hence we don’t have to decode every time once the pattern is recognized).
* We have 4 decoders. One complex instructions decoder and 3 simple instruction decoders.
* Stack engine that provides hardware support for stack operations.

The µop cache is organized as 32 sets  8 ways  6 µops, totaling a maximum capacity of 1536 µops. It can allocate a maximum of 3 lines of 6 µops each for each aligned and contiguous 32-bytes block of code. Code that runs out of the µop cache are not subject to the limitations of the fetch and decode units. It can deliver a throughput of 4 (possibly fused) µops or the equivalent of 32 bytes of code per clock cycle. The µop cache is rarely used to the maximum capacity of 1536 µops. The utilization is often less than optimal.

Source : Source: <http://www.agner.org/optimize/microarchitecture.pdf> pg-133



**How hardware resources are shared by in a Multi-threaded environment.**

3 Kinds of sharing :

Competitively shared : If first thread takes more of a resource, the next thread gets less of it.

Evenly shared : The resource is held alternately by each thread in different clock cycles.

One resource per thread (Duplication).

Caches  Competitively shared.

Branch target buffer and global history tables  competitively shared

Instruction fetch and decode unit  evenly shared

Loop buffer  There is one loop buffer per each thread.

RAT and ROB  Evenly shared between 2 hardware threads.

Execution Ports and Execution units  competitively shared

Permanent Register file  One for each thread.

Read and Write buffers  shared competitively.

It is clear that there is no advantage to running two threads per core if any of the shared resources are limiting factors for the performance. In many cases, however, the execution resources are more than sufficient for a single thread. It can be particularly advantageous to run two threads per core in situations where a large fraction of the time goes to cache misses and branch misprediction. However, if any of the shared resources are bottlenecks then it is not advantageous to run two threads per core. On the contrary, each thread is likely to run at less than half the single-thread speed because of evictions in the cache and branch target buffers and other resource conflicts. There is no way to give one thread higher priority than the other in the CPU.

Source : Source: <http://www.agner.org/optimize/microarchitecture.pdf>

pg-134

**3.Oprofile Commands:**

* Oprofile spawns a daemon process and the daemon process gets attached to it’s parent process and it interrupts the parent process in a (user) specified sampling frequency and checks for the hardware counters that are related to the specific hardware events specified by us.
* Running the oprofile daemon slows down the parent process quite a bit due to interrupts at a given sampling frequency. In our experiments while running the profiling daemon process the application slows down at least by three times or more.
* There is a minimum sampling frequency for each event. When you want to monitor more than one event, we have to keep the sampling frequency of all the events monitored as the highest out of minimum frequencies of the available events.

Let’s say we want to monitor the following events:

Lock\_cycles which has a min count of 2000000

Tlb\_access which has a min count of 100000

CPU\_CLK\_UNHALTED has a min count of 6000

We have to configure all these events to have a minimum frequency as 2000000.

**Command** : sudo opcontrol --help

**Output** : list’s various options

-l/--list-events list event types and unit masks  
   -?/--help        this message  
   -v/--version     show version  
   --init           loads the oprofile module and oprofilefs  
   --setup          give setup arguments (may be omitted)  
   --status         show configuration  
   --start-daemon   start daemon without starting profiling  
   -s/--start       start data collection  
   -d/--dump        flush the collected profiling data  
   -t/--stop        stop data collection  
   -h/--shutdown    stop data collection and kill daemon  
   -V/--verbose[=all,sfile,arcs,samples,module,misc,ext]  
                    be verbose in the daemon log  
   --reset          clears out data from current session  
   --save=name      save data from current session to session\_name  
   --deinit         unload the oprofile module and oprofilefs  
  
   -e/--event=eventspec  
  
      Choose an event. May be specified multiple times. Of the form  
      "default" or "name:count:unitmask:kernel:user", where :  
  
      name:     event name, e.g. CPU\_CLK\_UNHALTED or RTC\_INTERRUPTS  
      count:    reset counter value e.g. 100000  
      unitmask: hardware unit mask e.g. 0x0f  
      kernel:   whether to profile kernel: 0 or 1  
      user:     whether to profile userspace: 0 or 1  
  
   -p/--separate=type,[types]  
  
       Separate profiles as follows :  
  
       none:     no profile separation  
       library:  separate shared library profiles per-application  
       kernel:   same as library, plus kernel profiles  
       thread:   per-thread/process profiles  
       cpu:      per CPU profiles  
       all:      all of the above  
  
   -c/--callgraph=#depth         enable callgraph sample collection with a  
                                 maximum depth. Use '0' to disable callgraph  
                                 profiling.  
   --session-dir=dir             place sample database in dir instead of  
                               default location (/var/lib/oprofile)  
   -i/--image=name[,names]       list of binaries to profile (default is "all")  
   --vmlinux=file                vmlinux kernel image  
   --no-vmlinux                  no kernel image (vmlinux) available  
   --kernel-range=start,end      kernel range vma address in hexadecimal  
   --buffer-size=num             kernel buffer size in sample units.  
                                 Rules: A non-zero value goes into effect after  
                                 a '--shutdown/start' sequence.  A value of  
                                 zero sets this parameter back to default value  
                                 but does not go into effect until after a  
                                 '--deinit/init' sequence.  
   --buffer-watershed            kernel buffer watershed in sample units (2.6  
                                 kernel). Same rules as defined for  
                                 buffer-size.  
   --cpu-buffer-size=num         per-cpu buffer size in units (2.6 kernel)  
                                 Same rules as defined for buffer-size.  
   --xen                         Xen image (for Xen only)  
   --active-domains=<list>       List of domains in profiling session (for Xen)  
                                 (list contains domain ids separated by commas)  
  
  System z specific options  
  
  --s390hwsampbufsize=num        Number of 2MB areas used per CPU for storing sample data.

**Command** : sudo opcontrol –list-events

**Output** : gives the list of all events supported by our micro architecture. I have attached the complete list as an annexure to this document.

**Command** : sudo opcontrol --status

**Output** : very important command, helps us check various settings we have already configured before starting the scheduling daemon.

Session-dir: /var/lib/oprofile  
Event 0: mem\_load\_uops\_retired:2000003:16:1:1  
Separate options: library kernel thread cpu  
vmlinux file: /usr/lib/debug/boot/vmlinux-3.19.0-51-generic  
Image filter: /home/srameshb/profiling/oprof/final  
Call-graph depth: 0

List of options in the output of “status” command :

**sessions-dir**: link to where each session’s profiling information is stashed.

**Event** : List of events that we have configured to monitor the performance of the application on the underlying hardware.

**Separate option**: different parameters we want to profile out of the library,kernel,thread or cpu.

In our runs we use the option ‘all’, which collects data about all of the above parameters.

**vmlinux file** : link to compiled kernel image with debug symbols.

Run the following commands in Ubuntu to generate a kernel image with debug symbols.

echo "deb http://ddebs.ubuntu.com $(lsb\_release -cs) main restricted universe multiverse" |

sudo tee -a /etc/apt/sources.list.d/ddebs.list

sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 428D7C01

sudo apt-get update

sudo apt-get install linux-image-$(uname -r)-dbgsym

Once we run the following commands, we can find the kernel image (vmlinux) at

/usr/lib/debug/boot/vmlinux-3.19.0-51-generic

**Image filter :** link to the image we are planning to profile.

If the binary image is in the current directory. We just have to specify the name of the executable.

**Call-graph depth**: Specify the call graph depth, default value is 0. We can have any integer as the call graph depth.

**Command** : sudo opcontrol --start

**Output** : starts a daemon process,   
Using 2.6+ OProfile kernel interface.  
Reading module info.  
Using log file /var/lib/oprofile/samples/oprofiled.log  
Daemon started.

Profiler running.

**Command :** sudo opcontrol --shutdown

**Output** : Kills the daemon process.   
Stopping profiling.  
Killing daemon.

**Command** : sudo opcontrol --dump

**Output** : dumps the collected data.

**Command :** opcontrol--save=session\_name

**Output :** saves the session data in the sessions directory.

**Typical use-case:**

sudo opcontrol --status   check the status of the opcontrol

// configure the following parameters.

Session-dir: /var/lib/oprofile  
Event 0: mem\_load\_uops\_retired:2000003:16:1:1  
Separate options: library kernel thread cpu  
vmlinux file: /usr/lib/debug/boot/vmlinux-3.19.0-51-generic  
Image filter: /home/srameshb/profiling/oprof/final  
Call-graph depth: 0  
sudo opcontrol --start    starts the daemon process   
./final      run our executable and the daemon collects profiling data.   
sudo opcontrol --status    
sudo opcontrol --dump   dumps the collected data in the session\_dir

opcontrol --save=session\_final1  save the session with session name : session\_final1

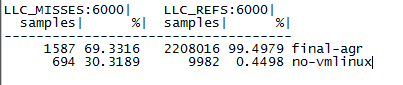
**Commands used to generate profiling report:**

**Command :** opreport

**Use case :** opreport --threshold=1 session:session\_name

**Output** : gives us a top level report of various event details we have collected during the time we were running the profiler.

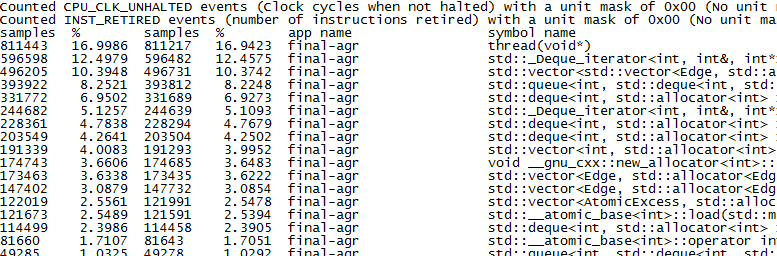
**Example output :**



Final-agr is the name of our executable and no-vmlinux shows the data collected on library functions. Had we had a compiled binary of vmlinux with debug symbols we would have had the data about time or hardware resource spent running kernel code.

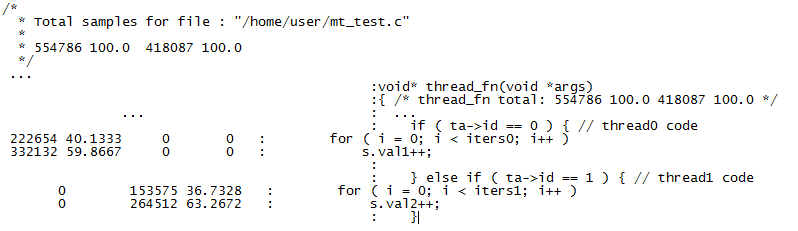
**Use case :** opreport --threshold=1 session:session\_name image:mt\_test --symbols

**Output :** Detailed report of various events with respect to various symbols in our user code.



**Command**: opannotate --source session:session\_name image:mt\_test

**Output :** amount of time spent or resources utilized in different parts of our code.



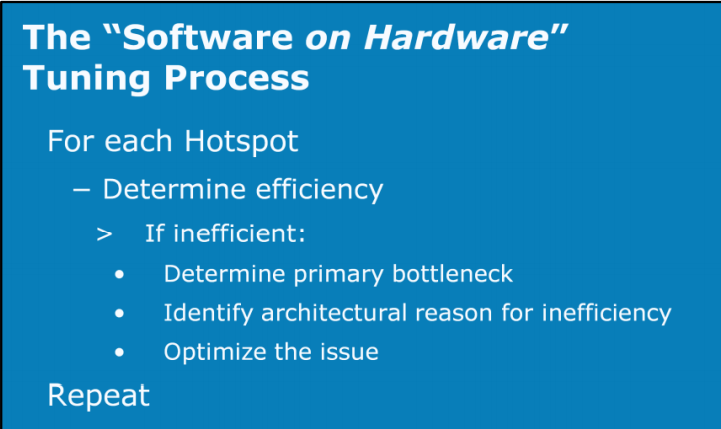
Sources : [http://hpseewiki.ipb.ac.rs/index.php/Optimization\_techniques\_for\_scalability#Oprofile](http://hpseewiki.ipb.ac.rs/index.php/Optimization_techniques_for_scalability" \l "Oprofile)

**4.Oprofiler internals**

<http://oprofile.sourceforge.net/doc/internals/>

**5.Profiling Strategy**

The profiling strategy is by running a basic hotspot analysis first and find out parts of the code where we spend maximum time.



Source : <https://software.intel.com/sites/default/files/Using_Intel_VTune_Amplifier_XE_on_Xeon_E5v2_or_E7v2_Family_1.0.pdf>

To run a basic hot spot analysis, we have to configure the following 2 events and run the executable and collect profiling information.

* CPU\_CLK\_UNHALTED  gives the number of clock-cycle spent
* INST\_RETIRED  gives the number of instructions retired in different parts of the code.

We can find CPI(Cycles per instruction) in different parts of our code. Parts that has high CPI is the hotspot.

For results, please refer results sections.

We also have generated results using “opannotate” tool and there by we can see where exactly the bottlenecks in our code is.

To monitor cache performance we configure the following hardware events.

* LLC\_MISSES  L3 misses
* LLC\_REFS  L3 references

To monitor the number of cycles the program spends while holding L1 data cache block under lock.

* Lock\_cycles  number of cycles L1Data block is under lock.

There are many other straightforward counters to find performance data like TLB\_flushes, Branch mis-predictions and pipeline stalls.

Smart Cache is a level 2 or level 3 caching method for multiple execution cores, developed by Intel.

**Intel smart cache technology :**

Smart Cache shares the actual cache memory between the cores of a multi-core processor. In comparison to a dedicated per-core cache, the overall cache miss rate decreases when not all cores need equal parts of the cache space. Consequently, a single core can use the full level 2 or level 3 cache, if the other cores are inactive.Furthermore, the shared cache makes it faster to share memory among different execution cores.

**Intel’s quick path inter connect:**

<https://en.wikipedia.org/wiki/Intel_QuickPath_Interconnect>

**7.Algorithm Specific issues:**

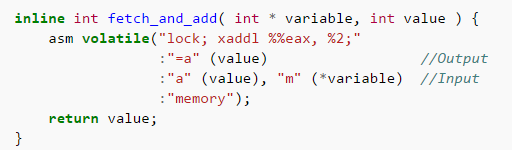
**Problem with graphical loads:** The amount of computation posed by a graph algorithm on any hardware in not typically dependent on a single parameter. Like in the case of a matrix multiplication, amount of computation(computational load) is directly dependent on matrix size. As larger matrices involve more number of arithmetic operations. When it comes to graphical loads, the density of a given graph is one important parameter. Likewise number of nodes in the graph is another important parameter. A lot depends on the random graph generator that is used to generate the input graph.

**Atomic Variables** : The algorithm by itself is asynchronous. In other words, there is no explicit global synchronization point. This avoids a lot of contention between multiple threads. Where as the algorithm uses “Atomic Variables” when it needs synchronization. A short digression on atomic variables shall follow.

**Work-sharing between threads:** The algorithm is asynchronous and waits on a global exit condition and each of the threads keep checking this exit condition in a while loop. There is a possibility of some threads emptying their local queues(finishing all of the work they can find). Then the threads would wait over a global exit condition, while other threads are busy doing all the weight-lifting. We have implemented a global work-sharing strategy by which we have a global work-sharing buffer in which any one of the threads that is still working will copy half of its local queue. The thread that is waiting will take the vertices from this global work-sharing buffer and starts working all over again.

**Asynchronous Global and Gap Relabeling heuristics:** 2 popular heuristics used in sequential max-flow algorithm has been made asynchronous.

**Atomic Variables :** Function used here apart from having individual atomic counters, is the atomic\_fetch\_add routine. That is implemented as follows:



We execute a “lock” instruction,

1. LOCK is not an instruction itself: it is an instruction prefix, which applies to the following instruction. That instruction must be something that does a read-modify-write on memory (INC, XCHG, CMPXCHG etc.) --- in this case it is the incl (%ecx) instruction which increments the long word at the address held in the ecx register.

The LOCK prefix ensures that the CPU has exclusive ownership of the appropriate cache line for the duration of the operation, and provides certain additional ordering guarantees. This may be achieved by asserting a bus lock, but the CPU will avoid this where possible. If the bus is locked then it is only for the duration of the locked instruction.

1. This code copies the address of the variable to be incremented off the stack into the ecxregister, then it does lock incl (%ecx) to atomically increment that variable by 1. The next two instructions set the eax register (which holds the return value from the function) to 0 if the new value of the variable is 0, and 1 otherwise. The operation is an **increment**, not an add (hence the name).

<http://x86.renejeschke.de/html/file_module_x86_id_159.html>

This lock instruction invokes exclusive access to the associated cache block, we ran oprofiler to find the number of cycles L1 data blocks are held by a lock.

**8.Results and analysis**

In the basic hotspot analysis that we performed, it can be observed that the application spends majority of its time executing STL routines (around 35% STL\_Dequeue and around 23 % STL\_Vectors and so on). Only 17 % of the time the program executes the code that I had written.

We have increased the working set size to some extent, but larger working sets take a very long time to get processed. Due to limited time, I did not try filling up the L3 cache (20MB) by a huge graph. As such the executable will take a long time to terminate, more over with profiling daemon it takes even longer. So I keep this aside for future work to be done. I want to work with different working set sizes and observe the CPI.

For our 3 sizes (smaller than L1, larger than L1 and larger than L2) CPI mostly remained the same. But we could find hotspots in our application, We even checked with an “Intel V tune” from my computer the hotspot analysis is spot on.

In the part of the program that we had written, we have also run test cases to monitor L1 data blocks that are under lock.

* Most cache locking is done by the memory allocator(Malloc and Free Library around 22%).
* Around 21 % is done by STL methods.
* 6% by new and delete library.
* 3% by atomic\_fetch\_add routine.

We had also run test cases to check the performance of the program in L3 cache. As Intel has inclusive caches. Out of 18567 accesses in L3 we had 203 misses. We can also view specific places in the program where we have many misses.

 Total samples for file : "/home/satya/push\_relabel/profiling/agr-statsh-final.cpp"  
 \*   
 \*    203  8.8685 18567  0.8367  
 \*/  
  
  
 /\* AtomicCounter::get() total:      0       0  3822  0.1722 \*/  
 /\* AtomicCap::AtomicCap(AtomicCap const&) total:      0       0     1 4.5e-05 \*/  
 /\* AtomicCap::AtomicCap(int) total:      0       0     3 1.4e-04 \*/  
 /\* Edge::Edge(Edge&&) total:      0       0    10 4.5e-04 \*/  
 /\* Edge::Edge(Edge const&) total:      0       0     3 1.4e-04 \*/  
 /\* Edge::Edge(int, int, int, int, int) total:      1  0.0437     3 1.4e-04 \*/  
 /\* PushRelabel::AddEdge(int, int, int) total:      0       0     6 2.7e-04 \*/  
 /\* thread(void\*) total:    202  8.8248 14708  0.6628 \*/  
 /\* agr\_thread(void\*) total:      0       0     1 4.5e-05 \*/  
 /\* main total:      0       0    10 4.5e-04 \*/  
/\*   
 \* Total samples for file : "/usr/include/c++/4.8/bits/stl\_queue.h"  
 \*   
 \*    121  5.2862856414 38.591

Given above is the summary of LLC performance in our application. The first column is number of misses and the 3rd column gives us the number of references,

**9.Future work to be done :**

* We can run test results by monitoring more hardware events for L1 and L2 caches.
* We can also run straight-forward events for instruction decoder performance and branch prediction units.
* Use GPROF or similar tools to get a top-level idea on memory use age.
* Compile and attach vmlinux to profile system calls and interrupt performance.

**10.Key take-away:**

* We used oprofile on an executable that ran on Dr.R’s server. Oprofile requires a lot of time to go through micro-architecture specific events list and choose events that are relevant to our study.
* A combination of CPU\_CLK\_UNHALTED and INST\_RETIRED events which are supported by all the contemporary micro-architectures can be used to find the hotspots in our application.
* We don’t get reports or graphs in extremely reader-friendly manner like we get out of GPROF, Kcachegrind or Vtune.
* We don’t get any analysis or trends, we get only raw data from performance counters.
* Can be useful if we are developing and testing for performance side by side. In my honest opinion if performance is our primary goal and all the development work is over, we should go for vtune.