Benchmark Analysis 508.namd_r SPEC CPU2017

line 1: 1st Valentinos Pariza
line 2: *Faculty of Computer Science*
line 3: *University of Cyprus*
line 4: Nicosia, Cyprus
line 5: vapriz01@cs.ucy.ac.cy
line 6: 909759

line 1: 2nd Marios Pafitis
line 2: *Faculty of Computer Science*
line 3: *University of Cyprus*
line 4: Nicosia, Cyprus
line 5: mpafit02@cs.ucy.ac.cy
line 6: 911719

Abstract - In this benchmark analysis, firstly we have checked as most as we could the capabilities and characteristics of the processors of the computers in the department of computer science in University of Cyprus Labs. After these we have taken the benchmark 508.namd_r from SPEC CPU2017 ,we have studied the use, its inputs and outputs of this benchmark program, we have tested the performance of the specific type of computers in the labs on this benchmark program (we checked the performance on a program that is strongly associated with big processing of floating point numbers) with command perf stat, we estimated the time that the program need to run and we have analyzed all these data by creating tables and graphs that show how the performance and the time of a program like this can be affected by the hardware and more important ,what are the reasons and factors that affect it and why these factors appear.

***Keywords—component, formatting, style, styling, insert** (key words)*

Analysis of program, SPEC CPU2017, 508.namd_r, Intel Core i5-4590,

I. Introduction

Many people today think that the code is the power of the programming. But at some point they ignore that all the power of the programming is derived from the hardware. All the capabilities and power that each programming language has is an association of some programming techniques and the operations that a hardware offers. This doesn't mean that the programming isn't important. We all know that with programming techniques (OOP…) we can illustrate great and efficient algorithms, build efficient data structures that can be used in many aspects of our lives and many more appliances. But all these have a conceivable upper bound of optimization. The hardware can limit you even if you have the fastest algorithm in the world. If you don't know what is happening below the abstraction of the programming language then you haven't accomplished nothing.

So, in this benchmark analysis we come to show how the hardware behaves in executing a benchmark program. This benchmark program is focused on the big number of manipulations and calculations made on floating point numbers of big precision (because this benchmark is associated with protein atoms which these elements of the world exist in a system which many of its sizes are smaller than 1 and the precision is important in this type of systems) and integer numbers. Furthermore in this benchmark program analysis we distinguish the Hardware factors that affect the execution time and performance of this program. The analysis is made on a small set of a data, but it is at some point clear to see all the factors and unexpected situations that occur in the execution of the benchmark program.

II. CPU Processor and Memory Characteristics

A. CPU Processor Characteristics

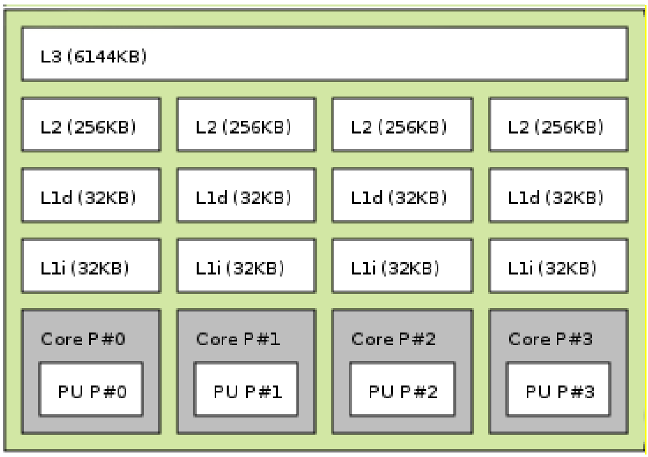


Image 1: Processor Characteristics that we have used for the benchmark tests.

Model Name:	Intel(R) Core (TM) i5-4590 CPU 3.30GHz
Announced on:	1/03/2014
Price:	$285
Lithography Process:	22nm
Architecture:	x86_64
Byte Order:	Little Endian
Core(s) per socket:	4
Socket(s):	1
CPU family:	6
Model:	60
Stepping:	3
CPU GHz:	3.3
CPU max GHz:	3.7
CPU min GHz:	0.8
Virtualization:	VT-x
Scalarity:	4-way Superscalar
TDP (Watts):	84W
Pipeline Stages:	12
Out-Of-Order execution:	Yes

B. *CPU Cores Characteristics*

Processor	0	1	2	3
Microcode	0x24			
Current CPU GHz	2.2	3.3	1.7	3.1
Physical id	0			
Siblings	4			
Core id	0	1	2	3
CPU cores	4			
Thread(s) per core:	1			
Apicid	0	2	4	6
Initial Apicid	0	2	4	6
FPU	Yes			
FPU_exception	Yes			
cpuid level	13			
Ways-of-Associativity	8			
Clflush size	64			
Cache Alignment	64			
Address Sizes	39 bits physical, 48 bits virtual			
Extensions and Technologies	MMX, SSE, SSE2, SSE3, SSSE3, SSE4 / SSE4.1 + SSE4.2, AES, AVX, AVX2			
TLB/Cache details:	64-byte Prefetching			
	Data TLB: 1-GB pages, 4-way set associative, 4 entries			
	Data TLB: 4-KB Pages, 4-way set associative, 64 entries			
	Instruction TLB: 4-KByte pages, 8-way set associative, 128 entries			
	L2 TLB: 1-MB, 4-way set associative, 64-byte line size			
	Shared 2nd-Level TLB: 4-KByte / 2-MB pages, 8-way associative, 1024 entries			

C. *Main Memory Characteristics*

Memory Size:	7897984 kB (8GB)
RAM Frequency:	1600MHz
Technology:	DDR3
DIMM slots:	4
Enable up to:	32GB
Memory Channels:	2
Memory Bandwidth:	25.6GB/s

III. Detailed Description of the CPU Processor

A. Brief Description

We have tested the SPEC in an HP EliteDesk 800 G1 TWR. The machine had a code core i5 4th generation CPU from Intel. The frequency of the CPU is 3.3GHz and 3.7GHz in turbo boost. The processor has 3 level cache. The L1 instruction and data cache and the L2 cache is unique for each core. The L3 is common for all four cores. The system is supported by an 8GB DDR3 RAM. Because the model of the CPU is i5 we have available 4 threads, one for each core. The Operating System is Linux Centos with x86_64 architecture. The machine can support either 32bit operating system or 64bit.

B. Detailed Description

The Model Name of the CPU that has been used for the Benchmark test is Intel® Core™ i5-4590 CPU 3.30GHz. The Intel® Core™ is the Brand. The i5 is the Brand Modifier. It is a 4th Generation CPU (Intel Haswell) and the 590 are the SKU Numeric Digits. It was released on 1st of March 2014 and its costs was $285. The Architecture of the CPU is x86_64 and the Scalarity of the CPU is 4-way Superscalar. The minimum frequency of the system is 0.8GHz and the maximum (turbo boot) is 3.7GHz. The CPU's frequency is 3.3GHz in idle. The endianness is Little Endian. The CPU contains 4 Cores and we have a single Thread per Core. Each Core has a unique L1 instruction cache (32 KB), L1 data cache (32KB) and L2 cache (256 KB). We have 8 Ways-Of-Associativity, more specific 8 cache lines are included in each cache set in the cache memory. Each CPU shares a common L3 cache (6144KB) for all four cores. The CPU supports Out-Of-Order Execution. It supports MMX (Single Instruction Multiple Data), SSE (Streaming SIMD Extensions), AVX & AVX2 (Advanced Vector Extensions) and AES (Advanced Encryption Standard instructions). It has FPU (Floating Point Unit) unit and FPU_exception that is handling exceptions in Floating Point Unit. Also, between the CPU and the CPU Cache we have TLBs (Translation Lookaside Buffer), either for instructions and data.

IV. SPEC CPU 2017: 508.NAMD_R

Benchmark Name:

508.namd_r

Benchmark Author:

Jim Phillips, Theoretical and Computational Biophysics Group, University of Illinois

Benchmark Description:

This Benchmark, which is known as 508.namd_r is derived-taken from the program NAMD (Basically from the data layout and inner loop of NAMD which are some structural elements that are included in software NAMD) and is a program for simulating large biomolecular systems(more specific, simulating millions of atoms).NAMD is the abbreviation for Nanoscale Molecular Dynamics ,and it is a computer software for molecular dynamics simulation, build with Charm++(parallel programming model).NAMD scales to over 200,000 cores for very large systems in order to simulate efficiently its biomolecular systems(using all the parallel capabilities that offers),but serial performance is also important to the over 50,000 users who have tried the program over the past decade. Because of the efficiency of this program, its maturing (The continuous updates of NAMD) and of the importance of the operations that this program does (calculating inter-atomic interactions in a small set of functions, most of its runtime), this program was used as a compact benchmark for CPU2017.

Input Description:

The file is named apoa1.input (APOA1 is the gene which encodes apolipoprotein A-I, which is the most important protein component of high-density lipoprotein (HDL) in plasma).This file format is created by NAMD 2.9 using a special command ("dumpbench") and it doesn't need file readers. Or any other setup code for the benchmark. At the beginning some parameters that are associated with atoms-proteins appear (PMETolerance, PMEEwaldCoefficient…) which indicate some attributes of the system where the atoms will be placed and be simulated in. After these parameters, some values-numbers follow that indicate coordination and attributes (like velocity…) for the atoms-Proteins. If we look more carefully at the format of the file, the file is broken into segments that indicate some sets of data. For example there is a data set that starts from TABLE_BEGIN and ends at word TABLE_END that is a set with some floating point numbers. More general the file consists of a lot of numbers, floating point numbers and integers that need to be analyzed. This is a great example of how a computer can be checked in efficiency in many strict factors of processing like processing floating point and integer numbers.

Output Description:

The CPU2017 uses the "—output namd.out" command –line option in order to produce a brief output file, which has name namd.out and contains various checksums of the calculations made on the forces of the proteins' atoms, in each iteration which the number of iterations is indicated by "-–iterations ".These should be consistent across platforms and work normally, within round off error, and it is used for validation.

V. Statistics Analysis: Average/Standard Deviation

Overall Observations and observations based on Mean and Standard Deviation:

From the table we observe that the time the program needed to execute is approximately the same (only 1.84 deviation) for all the different executions. The deviation may occur because of the different situations where the program ran (different processes on CPU, at the time the program run -> increased number of Context switching) and also the time multiplexing which scales the counted statistics to the execution time (not extremely precise statistics)

Also, the instructions' deviation is also very low because the number of instructions executed each time shouldn't be changed significantly. But because of time multiplexing there is a deviation. The same applies to cycles, branch instructions and branch misses which don't change in each execution with same inputs and same executable machine code. We know that our program doesn't use any random behavior in execution.We have opened the source code and we have seen that there isn't any randomness in the program which affects significantly these statistics.

Generally, we observe that approximately **1.6%** of the instructions are branch instructions and **4.44%** of the branches are predicted wrong (we have branch misses).

Standard Deviation/ Average	*Average*	*Standard Deviation*	*Diversion Percentage (%)*	*Comments*
Time:	279.70	1.84	**0.66**	The Deviation for the Execution time is <1%.
Instructions:	2598612782571	520287031	**0.02**	The amount of instructions are stable with Deviation <0%
Cycles:	1014653654177	2808275224	**0.28**	The total number of Cycles are stable with Deviation <0%
Cache-References:	449971763	36352832	**8.08**	The Deviation for the Cache-References is 8%.
Cache Misses	94312208	8367359	**8.87**	The Deviation for the Cache-Misses is 9%
Branch-Instructions:	42013192364	42036526	**0.10**	The Branch-Instructions are stable with Deviation <0%
Branch-Misses:	1867356594	1295047	**0.07**	The Branch-Misses are stable with Deviation <0%
L1-dcache-loads	707313493358	91651835	**0.01**	The L1-dcache-loads are stable with Deviation <0%
L1-dcache-load-misses	31907089229	49679082	**0.16**	The L1-dcache-load-misses are stable with Deviation <0%
L1-dcache-stores:	242144245126	40051837	**0.02**	The L1-dcache-stores are stable with Deviation <0%
L1-icache-load-misses:	18399756	336468	**1.83**	The L1-icache-load-misses are stable with Deviation <2%
LLC-loads:	381215876	36698914	**9.63**	The Deviation for the LLC-Loads is 10%.
LLC-load-misses:	76630123	4205048	**5.49**	The Deviation for the LLC-load-misses is 5%
LLC-stores:	39202624	4154680	**10.60**	The Deviation for the LLC-Stores is 11%.
LLC-store-misses:	16277009	2348003	**14.43**	The Deviation for the LLC-store-misses is 14%
dTLB-loads:	706276858458	203576469	**0.03**	The L1-dcache-stores are stable with Deviation <0%
dTLB-load-misses:	6467694	717839	**11.10**	The Deviation for the dTLB-load-misses is 11%.
dTLB-stores:	241604005925	47710291	**0.02**	The dTLB-stores are stable with Deviation <0%
dTLB-store-misses:	2118082	133914	**6.32**	The Deviation for the dTLB-store-misses is 6%.
iTLB-loads:	38456	7586	**19.73**	The Deviation for the iTLB-loads is 20%.
iTLB-load-misses:	408499	76830	**18.81**	The Deviation for the iTLB-load-misses is 19%.

This is quite interesting because we can understand that instructions don't contain to many branch instructions but although there are approximately 1.9 branch misses at approximately every 42 branch instructions.

Also there are appeared **4.5%** of all the cache loads , misses due to loads at L1-cache.But we can see that the number of loads from the cache L1 is quite big (L1 dcache-loads appear 27.2% of the instructions),something that shows that the program has used many values from memory and most of them has been found at L1-cache.For those data that haven't been found(load-misses) at L1-cache the program has searched the next levels of caches or-and the main memory for loading those data in caches.

Also, L1-dcache-loads, L1-dcahce-load-misses, L1-dcache-stores, L1-icache-load-misses normally are the same in all the identical executions (same input, same executable). We can testify this because their deviation from mean value isn't big. That's because the first level of cache (L1-cache) is a dedicated cache for each core. So, in each execution the same data and instructions will be loaded and stored in cache L1 and also approximately the same number of misses will occur at L1 cache. But there is a little deviation if some processes occur and change a little bit the number of L1-dcahce-load-misses, L1-icache-load-misses because they will be using the same L1 cache as the program uses. More specific this deviation is supposed to be a deviation due to context switching (force switch of the executed process in single CPU in order to accomplish multitasking →This leads the storing of the state (registers, etc. Data in the cache of the previous process may be lost) of the process in memory and restore of the state when its turn has arrived. This is bad because at different executions, different processes may be running on pc causing more L1-dcache load misses …

Moreover, we observe that the statistics about LLC (Last Level Cache) aren't the same for each execution and the standard deviation is big for each statistic, in association with their mean values. That's because of time multiplexing as we have mentioned and also because of the different processes that might appear and use the L3-cache (LLC) at the same time during the program runs (Context switching as was explained above). The deviation is bigger than the deviation of the corresponding statistics at L1 cache. That's because at different executions the accesses in LLC differ as explained above because of the need to restore some data in caches which have been thrown out of caches to make space for other processes. LLC is bigger than the rest caches but also is shared, and both instructions and data are stored there. After executing other processes (because of context switching) the data stored in LLC may be thrown out, because it might not be available space in the higher-level caches. Also, number of LLC cache loads may differ in different executions because it might L1 cache has more load misses than other executions (because of context switching), causing respectively the need to search more times at LLC cache (after searching L2-cache).

Respectively the number of cache-references and cache-misses in each execution is quite different because in each execution the number of references to LLC and the number of misses at LLC is different because of different number of processes that might affect the execution of the program in each execution. LLC is a shared cache for both, instructions and data. So, the possibility of a process to affect LLC is bigger than the possibility the L1-cache to be affected, by a process if we take into consideration that LLC-L3 cache is shared for all cores and both, instructions and data are stored there . For example, when a process that uses its own L1 and L2 caches which are dedicated in their core, need more space at L3 (LLC)cache (for example to bring some other data at LLC from main memory) which is shared to all cores. The current process might cause some data from another process in LLC to be changed. But when the other process tries to access its data, it will need to look in main memory, because there will be a miss. Its data doesn't exist at LLC anymore.

The interesting thing to observe is the fact that the LLC-loads are significantly smaller than the number of L1-dcache-loads. This can show us that the program at most of its loads, it found its data at L1-cache, which can show that the program has high locality (spatial and temporal).

We can observe that dTLB-stores and dTLB-loads have small deviation because the number of loads and stores instructions in the program are the same in each execution. But as we can see that the number of dTLB-load-misses and dTLB-store misses (Quite Big Deviation) are in some point significantly different in each execution. That's because of the possibility that in a dTLB, the information that is hold for the virtual-physical addresses of the program is changed by some other processes that might want to store information about their data's virtual-physical addresses inside dTLB. As a result, in different executions (which indicate different situations where the program executes), different occasions may occur, increasing or decreasing the number of dTLB-load-misses and dTLB-store misses (dTLB-store-misses are misses by store operations in TLB and dTLB-load-misses are misses in TLB by load operations). Simpler there is a big deviation because of the context switching that the Operating System performs and also because of the different physical spaces that the operating system distributes to each process every time that is executed. More specific the Operating system distributes different physical addresses for data and instructions in each execution because in each execution the available physical space in main memory (RAM) is different. Also, the Operating system distributes different virtual addresses for the program data and instructions for each execution, in order to hide the behavior of the program in memory from someone who might observe it, in order to harm the system. This is called **Address space layout randomization which is a security technique which** prevents exploitation of memory corruption vulnerabilities (Vulnerabilities from attacks). For these reasons every time the program is executed, it has different physical (Also virtual) addresses which this causes conflicts of same indexes of the addresses in dTLB (For example, this causes the action of removing one address from the dTLB that the program needs for inserting an address of another process. At the end this will cause a load or store dTLB miss because the address that was thrown away was requested by the program). A common idea applies for iTLB.

Although the dTLB-load-misses are only 8.5*10^ (-4) % of all dTLB-loads which is extremely low, showing that most of the load's operations found their desired physical address at TLB and not at the page table.

Also 8.8*10^ (-6) % of dTLB-stores are dTLB-store-misses, which is also extremely low showing the same conclusion as for dTLB-loads & dTLB-load-misses above.

We can see that iTLB-loads are less than iTLB-load-misses. But iTLB-loads represent the event of a TLB lookup that misses at first level of TLB and has a hit at second level of TLB. On the other hand, iTLB-load-misses represents the event of a TLB lookup that misses both the first level of iTLB and the unified (second level) TLB and causes a page walk at page table. So, we can observe that the number of iTLB-load-misses is the number of misses at all levels of TLBs that, forces the appliance of page walk. But the number of misses at all levels of TLBs is significantly bigger than the number of misses at first level of TLB which hits at the second level TLB. We can observe the number of iTLB-loads and iTLB-load-misses is small according to the number of branch-instructions or even instructions (iTLB is used when the physical address of an instruction is needed to fetch a new instruction from an instruction cache (L1) or from a cache for both instructions and data(L3) in the program) indicating that the deviation is big on small amount of misses at first or , first and second iTLB level caches. The deviation of these statistics is also increased as we have understood because of the different allocation of memory that the operating system does and controls, at each execution of the program and in different situations because of the different processes that run simultaneously at the CPU. More specific the Operating system distributes different physical addresses for data and instructions in each execution because in each execution the available physical space in main memory (RAM) is different (There might be some other processes that have allocated memory before the program asks for memory from OS, in main memory to load the instructions of the executable and the input file from the secondary memory).

We know that:

0 < IPC < =4 (because the processor is 4 way -superscalar processor)

The processor that the program ran on, has the power to execute four instructions (4 way -superscalar processor) per cycle. As we can see the average IPC is 2.57 which is very high. This can show that the instructions were executed without paying many cycles-penalties.

Execution Time:

From the execution of the command “time” 10 times we took some averages.

Average IPC for 10 executions: **IPC=2.57**. We know that CPI=1/IPC So **CPI= 0.389**

From the array of the next slide we took the averages of the statistics which are mentioned-used below for calculating the execution time (USER):

Execution time with Cycles_Statistic from perf stat:

For 3.3 GHz (Standard):

Cycles_Statistic*(Cycle_Time) = **307.471s**

For 3.7 GHz (Turbo Boost):

Cycles_Statistic*(Cycle_Time) = **274.231s**

Execution time with Cycles_Calculated = Instructions * CPI from perf stat: (Cycles_Calculated = CPI * Instructions = 0.389 * 2,598,612,782,571 = 1,010,860,372,420.119)

For 3.3 GHz (Standard):

Cycles_Calculated * (Cycle_Time) = **306.321s**

For 3.7 GHz (Turbo Boost):

Cycles_Calculated *(Cycle_Time) = **273.206s**

Average Time from Perf Stat: **279.7041444**

CPU frequency→ (Cycles / User_Time)/(10E+9) = 3.66GH

CPU frequency→ (Cycles / User_Time)/(10E+9) = 3.62GH

The time of the program which we calculated by the number of cycles multiplied by the time in seconds that needed for a cycle to be completed is 307.471s for CPU frequency value 3.3GHz and 274.231 for 3.7GHz. We have calculated also the clock rate of the CPU with the time and cycles estimated by the command perf stat. We calculated the fraction execution_time / cycles which represents the time in seconds that a cycle by the clock takes (we turned it into to nanoseconds and after this which represents the Period of the clock we turned it into to the frequency of the processor f=1/T where T is the period and f the frequency of the clock) and this gave us that the CPU frequency , which as we have calculated it is 3.62 GH. We can see that the pulse-frequency of processor is between the 3.3 GH and 3.7 GH, as we have seen by the hardware attributes given from the manufacturer of the CPU (INTEL). But as we explain below, the time multiplexing is a factor which changes (small although but significantly) the statistics we estimated. Because we use different statistics like the number of cycles, the time and others in our calculations for the execution time and for the processor frequency, the values taken from the command perf stat are scaled and not the actual values because of the lack of hardware counters. So, we can see that there is a deviation and the results aren’t equal because of this factor. Also, the time statistic, given by perf stat differs from the rest calculations (The other calculations with statistics of perf stat and the time calculations from the time command). We can conclude that perf stat adds extra time in the execution because it counts many statistics, one after the other or some of them simultaneously (This is explained below).

Time Multiplexing:

The kernel uses hardware counters in order to count an event. If there are more events than counters, then the system is using the Time Multiplexing method in order to estimate the final count. The time Multiplexing applies to PMU (Performance Monitoring Unit) events. Example of PMU events are the number of cycles, instructions retired, L1 cache misses, cache-references and so on. It uses a formula to scale the count based on total time enabled vs time running.

final_count = raw_count * time_enabled/time_running

The final count is an estimation of the time enabled and not the actual time running. Simpler because the hardware has limited number of counters and because the counters must be used for all the events, this method takes the first N events (where N is the number of the available counters in the hardware) and let them to use the counters for a period of time. After the time has expired for the first N events to use the counters, the next N events are taken for using the counters, and the same process continues with the method of round-robin list (List which all the elements are taken with a rational sequence and when they reach the end, the picking starts again from the beginning).This method can be applied a little more different if some events must be used together for counting. After the end of the execution of the program, the time that was used for each event (sum) is scaled in order to correspond to the time that the program was running (With the equation above). The problem that occurs with this is that the events that we ask for, are not complete counting results but are some patterned-scaled results that correspond only to segments of the execution of the program and not at the whole program. This is a big problem because we lose some calculations for some events and for some others not, which these calculations might be important (like a cache-miss might occur, a branch-miss and so on) and this can lead that our data might be a little bit inconsistent.

VI. Correlation Analysis / Graphs For 1SEC

Normalization of Statistics:

X = (Statistic * 1000) / Instructions

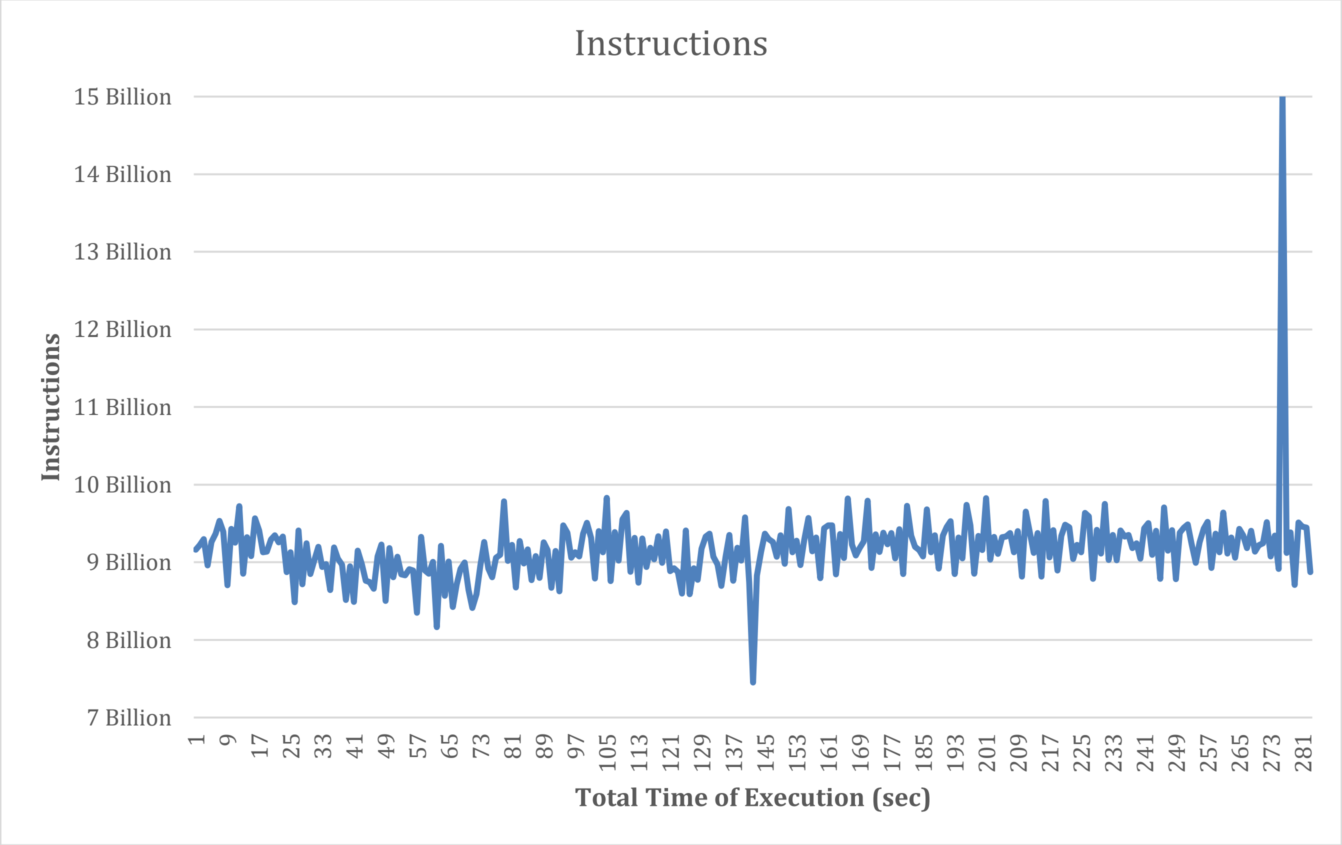
Statistic' = (X - MIN (All X)) / MAX (All X)

The following graphs are normalized with the equation above in order to limit the values of all the events-statistics in a range of values between 0 and 1.

The amount of Instructions at every second follows a stable pattern (most of the time). Its bounds are between 7.5GI to 15GI (Giga Instructions). The number of Instructions executed every second is different because different situations may occur (like branches, misses...). So, every 1 second the amount of instructions executed will not be the same. We also can see that at the end (t=276) the number of instructions executed is extremely bigger than at the rest program. There IPC is also extremely high. This might be, because at the end the program executes a different routine which is quite independent and easy executed so , it can be executed faster and more easily (due to multiscalar attribute).Also at t=142, approximately in the middle we can understand that the program follows a different routine ,combining the input data of the program with a special routine. At the same time the IPC is low. The rest program behaves almost identical, which shows that at most of the time, which the program runs the program follows constantly the same behavior, executes same routines and functions. An interesting observation is that at t=276 the dTLB-loads and dTLB-stores are low, which this is an advantage for executing many instructions. The same behavior with instructions follows the IPC. That’s expected because IPC=instructions/cycles.

From the general graph that all the statistics are appeared, we can see that our benchmark is boring. This applies because as we can see all the statistics follow a specific pattern in almost all the time of the execution.

The cycles as we can see are not as stable as they should be if we suppose that the CPU runs with a stable frequency. But this doesn’t apply because the CPU can execute instructions in a range of CPU frequency of 3.3-3.7. The processor-CPU executes instructions in a variable CPU-frequency (between



a range) which depends on the power that the computer can have at a specific time. If the computer can handle more power without causing any damage and also if the CPU can operate in higher frequency, then the CPU increases its frequency in order to optimize the cycle-speed-execution.

The dTLB-stores are following an almost same pattern as dTLB-loads showing as that at same phases in program, data are stored and loaded to/from memory in commensurate sizes. The L1-dcache-stores statistic moves in the same way as dTLB-stores and they both take approximately the same values. This is logical because at every store operation executed by CPU both the L1 cache and dTLB caches are used. We also can see that at every 1000 instructions we have approximately 240-300 load instructions and 60-110 store.

We can see that L1-dcache load misses are behaving commonly with dTLB-loads and L1-dcache-loads. Which is noticeable because as the number of loads increases at L1-cache the possibility of having a miss is increasing. This can be a factor why they have common behavior.

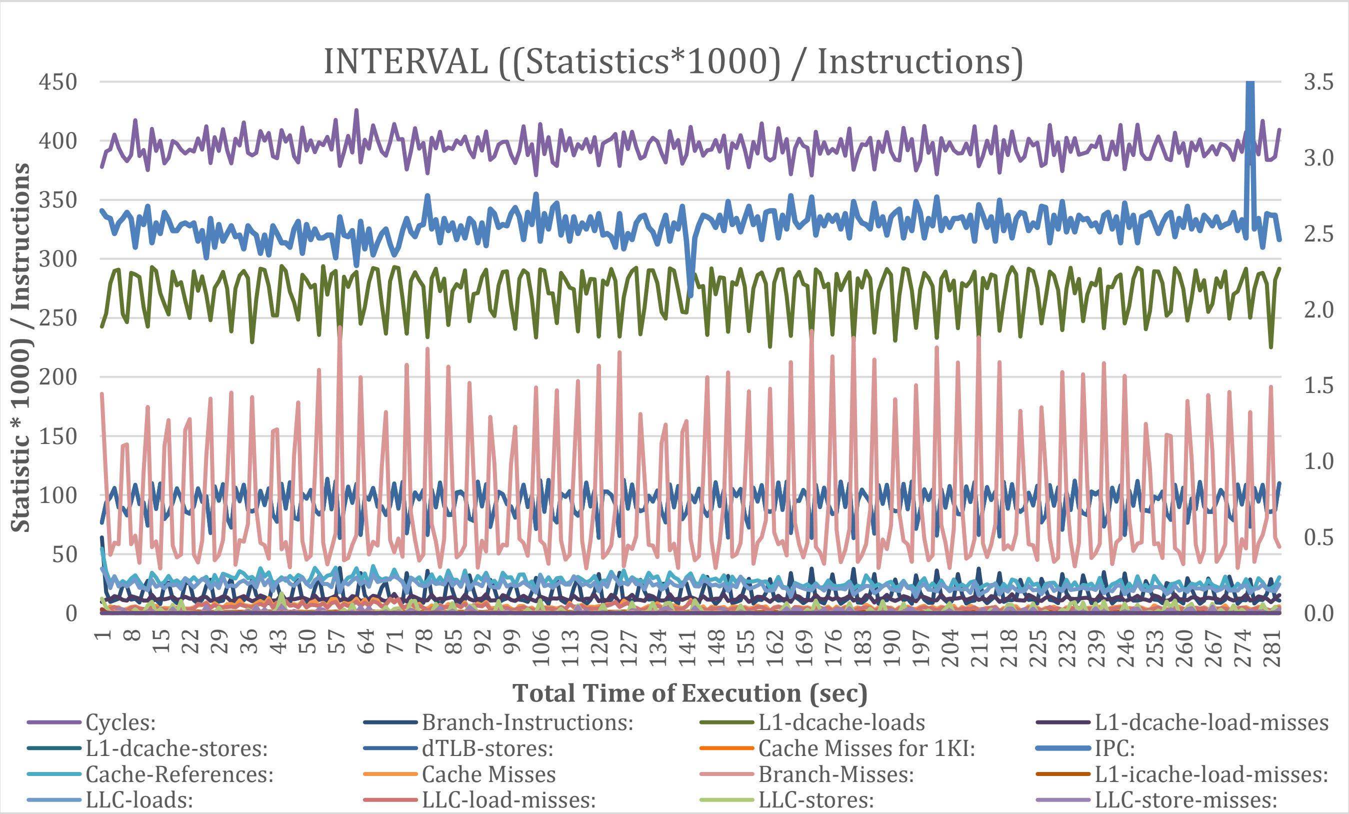
From this graph we can observe that the number of Branch-Misses is associated with the number of branch-instructions. This is normal because as the number of branch-instructions increases, the possibility of having a branch-misses increases.

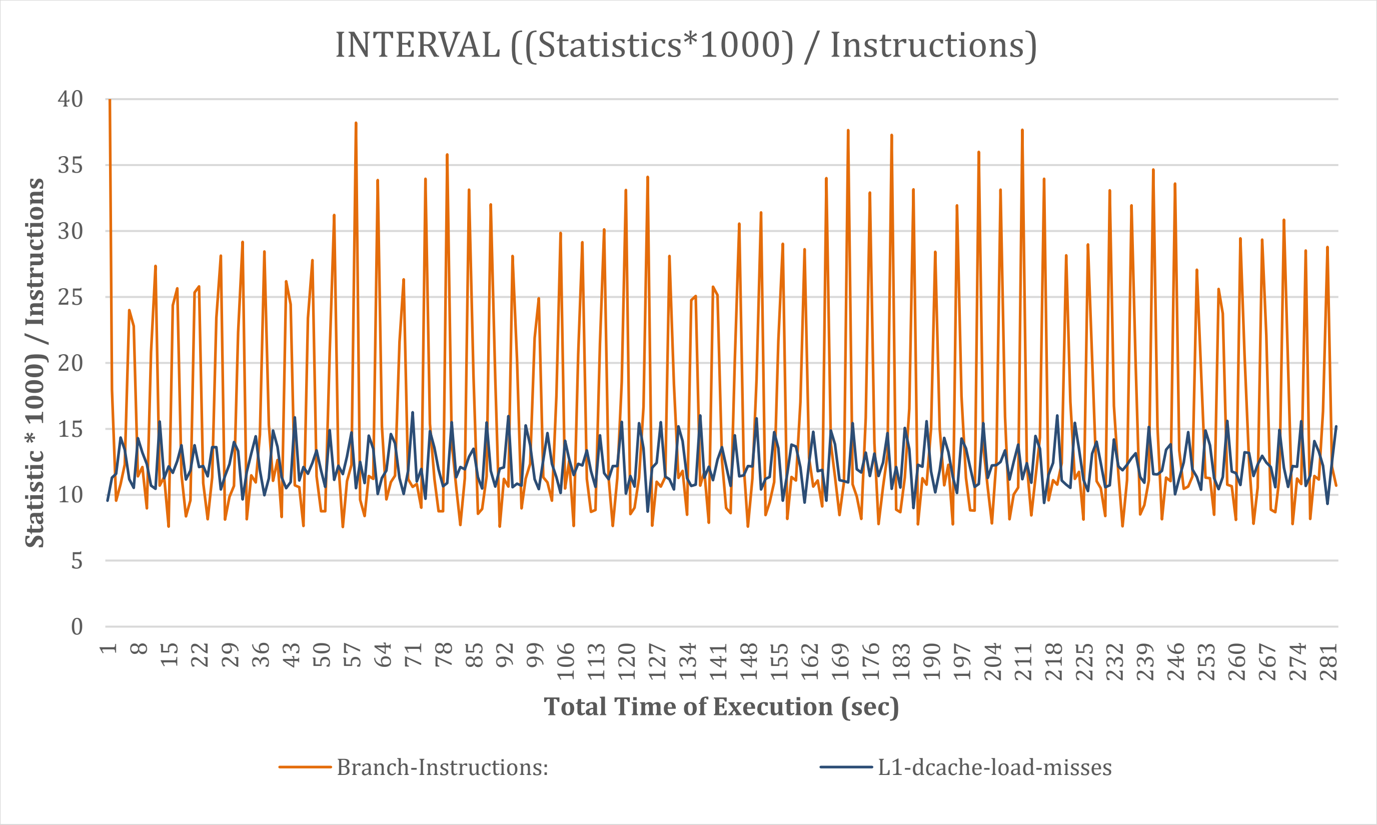
We can see that both LLC-loads and cache references behave commonly and also at the beginning the number of cache-references and LLC-loads is higher than at the rest program. We suppose that at the beginning the program loads a lot of values (has a lot of misses at caches) from the main memory in order to bring the data that it needs from memory to the caches. That part of the program is also the initialization of the program and the initialization of the caches with first data of the program. Also, we can see that the program has a lot of cache-misses at the time [22,99] which probably is the time that the program needed in order to bring all the data that it needed in the caches.

As we can see at the same time ([22,99]) the LLC-loads and the cache-references are also high which justifies that the number of LLC-load-misses and cache-misses are high. We can observe that their behavior is approximately common. As the program continues the number of the cache-references LLC-loads, cache-misses and LLC-load-misses are stabled in a pattern of behavior, until the program ends. Also the number of LLC-stores is a little bit high at the beginning because of the initialization of the program and we can observe that it takes some high values at some moments which probably are some situations where many data are stored in the memory and cannot be stored in L1-cache because they don't exist all of them there (because of lack of space at L1-cache <L2-cache < L3-cache) and at some other moments where the stores at LLC are low. Also, we observe that the number of LLC-store-misses is behaving according to LLC-store-misses. This is also normal because the number as the number of stores at LLC increases the number of LLC-store-misses increases (The possibility of not finding an element at the LLC is increasing). The other metrics like iTLB-loads, iTLB-load-misses, IPC are behaving stable.

Also, the dTLB loads are following a stable pattern. This shows us that the program loads more data from memory (caches or main memory) in a special phase of a routine-function that is repeatedly executed once in a while and at the rest times, the dTLB-loads are decreasing stably at a lower bound value and increasing again to reach the maximum upper bound. This pattern is appeared repeatedly.

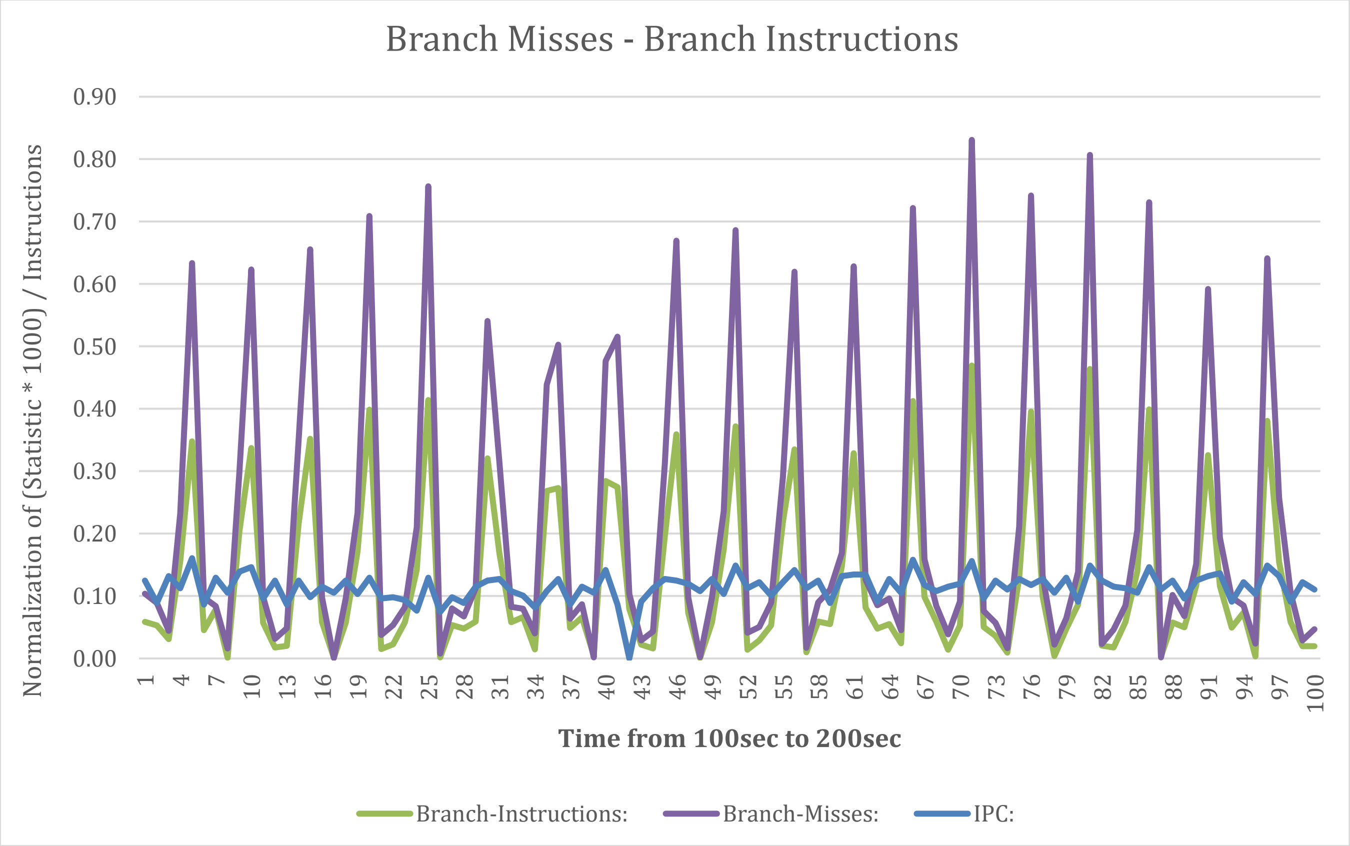
The L1-dcache-loads statistic moves in the same way as dTLB-loads and they both take approximately the same values. This is logical because at every load operation executed by CPU both the L1 cache and dTLB caches are used.

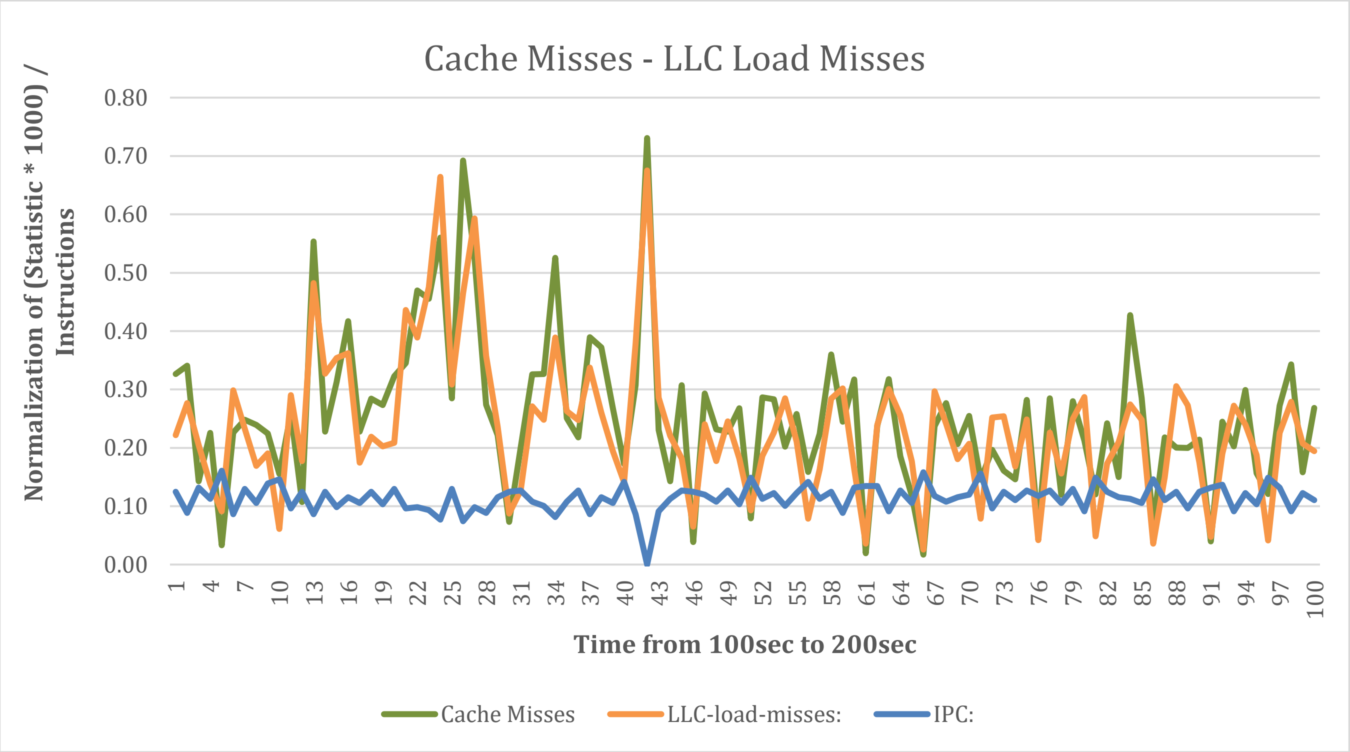




As we can see the branch instructions appear in the program in high values at the beginning, and we suppose that at the beginning there are a lot of conditions-checks for the program and after this the program follows a stable pattern of using branch instructions. This indicates that the program has a stable routine that follows, which uses approximately the same number of branches in regular time periods.

We can see that the program is constantly following the same pattern, routine. At the same time, we can see that L1-dcache load-misses are independent with branch-instructions. They behave with a different way, but we can see that it also follows a pattern in its behavior. All this behavioral functionality of our program shows that our program doesn't has anything interesting so far and the behavior of the program is boring.





Branch Misses - Branch Instructions

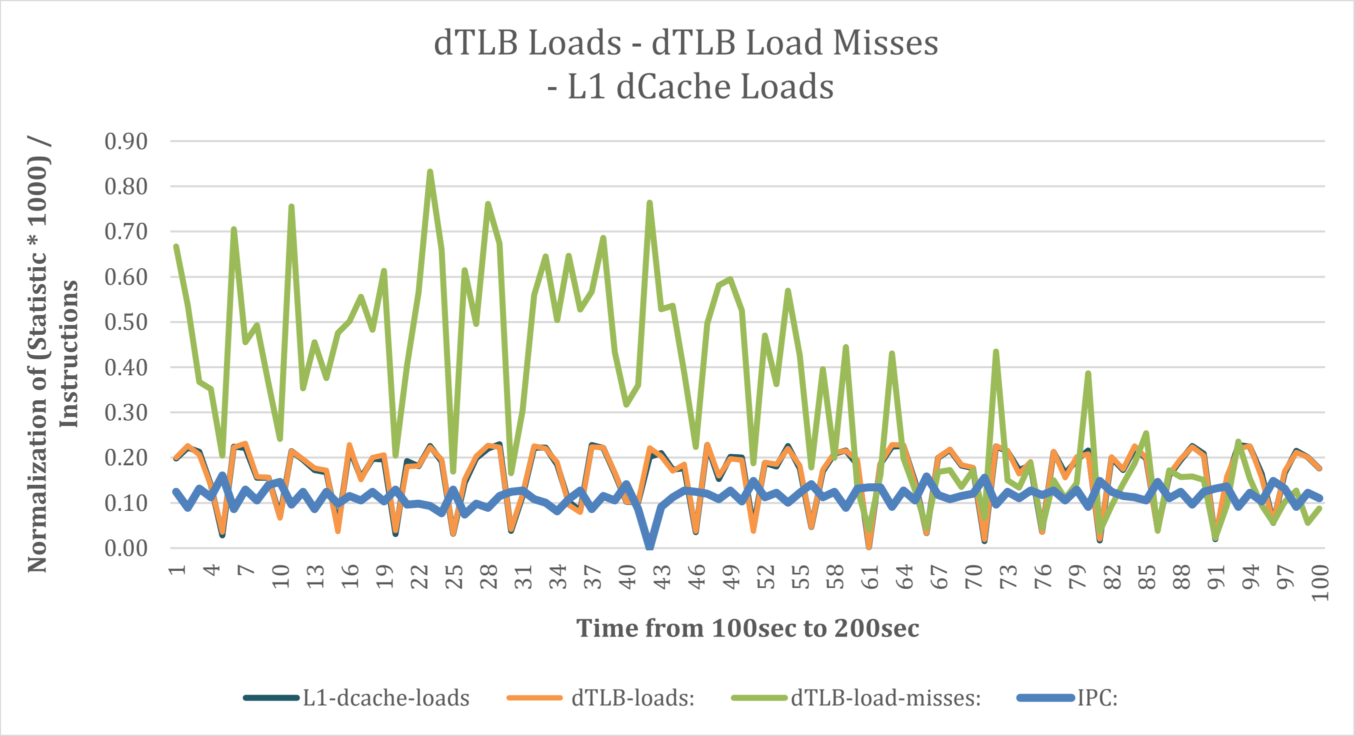
Here we do not see the values of each statistic but we emphasis on their behavior. For example, it seems weird to see branch misses to be higher than branch instructions, but this is because the number of instructions isn't less than branch misses in every second.

The rate of branch misses increases, respectively when branch instructions are increasing.

Because the rate of branch instructions appears is low (in association with instructions). The number of branch misses doesn't appear to have a negative correlation as expected with IPC. Simpler the program is predictable, we can see it from its behavior. It constantly follows the same pattern at almost all the time. The number of branch instructions at every second is big enough in association with branch misses, but we can see that as the branch instructions increase (a little bit commonly with IPC – positive correlated with IPC) the rate of branch misses is increasing.

That's mean that there is a higher appear of a branch-miss, as the branch instructions increase. Although the branch instructions aren't big enough in association with the instructions. So, at the end the effect of branch misses isn't affecting significantly enough the IPC at a specific time.

The effect of branch misses on IPC isn't as expected because might of the small number of appearances of branch instructions in the program.



Cache Misses - LLC Load Misses

Cache misses and LLC load misses are correlated between them because LLC load misses represent the misses that occur from load at the last level cache and cache misses are the misses that occur generally from all the caches (information that needed doesn't exist in none of the caches). We can see that both of them are negative correlated with IPC because as the number of cache misses and LLC-load misses increases the IPC is decreased as expected. That's because in LLC -load misses and cahce-misses the CPU has to pay a penalty of finding the information needed in the memory. That costs and increases the number of cycles executed by a specific instruction.

The above graph was derived by the main graph for 100 seconds from range [100,200]. We can observe that dTLB load-misses at the beginning they behave with a little bit unstable way. This can be seen from the beginning of the program until the 143s , when there it starts to behave more stable in

dTLB Loads - L1 dCache Loads

dTLB Loads and L1 dCache Loads are behaving commonly and with same amount of values-appears. This applies as expected because the number of dTLB Loads and L1 dCache Loads are associated extremely high. Whenever the CPU wants something from memory. The reference to L1-dcache for load presupposes the use of dTLB for load of a physical address (convert physical from virtual)

We observe that the dTLB-load misses are correlated a little bit positive with both L1 dCache Loads and dTLB Loads, because as the number of dTLB-loads and L1-dcache-loads increases the dTLB-load-misses is increasing. That's happening at the first 43 s of the sample (More specific this happens from 0 until 143). We suppose that the program uses at the first 143 seconds addresses of data that haven't been loaded all yet to the TLB caches. This can cause penalties and can increase the rate of dTLB-load misses. But at the rest seconds 44-100(More specific from 144-288) the rate of dTLB load misses is decreased and there we can see that the rate of dTLB load misses is corresponding to the rate of having dTLB-load. We suppose that, this happens because of the misses that might have been occurred more frequently at the beginning in order to load at dTLBs all the necessary addresses of data and at the end because of the lack of misses which leads to the decreasing of dTLB-load misses. Moreover we can see that there is a positive correlation between IPC and dTLB loads .At the beginning because of the high rate of dTLB-load-misses the correlation wasn't big enough between them but at after the middle of the execution of the program , when the rate of dTLB-load-misses was decreased , IPC was behaving more commonly with dTLB-loads because of the low rate of dTLB-load-misses. That's because the dTLB-loads is positive correlated with IPC. We can justify that the behavior of IPC after the 43ths (143s basically in the hole program) is corresponding to dTLB-loads. After the middle of the execution as the number of instructions was increased, the number of dTLB loads has been increasing and because of low dTLB-load-miss rate the IPC seemed to behave commonly with dTLB-loads.

Correlation	*IPC:*
Time:	**0.144**
Instructions:	**0.996**
Cycles:	**0.907**
Cache-References:	**-0.019**
Cache Misses	**-0.286**
Branch-Instructions:	**0.444**
Branch-Misses:	**0.432**
L1-dcache-loads	**0.518**
L1-dcache-load-misses	**0.177**
L1-dcache-stores:	**0.009**
L1-icache-load-misses:	**0.091**
LLC-loads:	**-0.077**
LLC-load-misses:	**-0.359**
LLC-stores:	**-0.023**
LLC-store-misses:	**0.039**
dTLB-loads:	**0.518**
dTLB-load-misses:	**-0.292**
dTLB-stores:	**0.010**
dTLB-store-misses:	**0.023**
iTLB-loads:	**0.052**
iTLB-load-misses:	**0.089**
Cache References for 1KI:	**-0.342**
Cache Misses for 1KI:	**-0.397**

- The **time** is independent from IPC because as we can see the correlation of time and IPC is near to 0.
- **Instructions** are correlated with IPC because as the instructions increase the IPC increases. Its correlation with IPC is the highest between all statistics.
- Cycles
- **Cache-references, L1-dcache-stores, L1-icache-load-misses, LLC loads, LLC stores** we can see that they are independent from IPC because their correlation value with IPC is near to zero.
- We can see that **cache-misses** are inverse correlated with IPC because cache-misses can be described as the misses that occur at all cache levels and forces the hardware to look for the requested data in main memory. Something which is very low and it costs. Because of this as the number of cache-misses increases the IPC at some point starts to decrease. Not straightly but sensibly.
- We can observe that because there aren't many **LLC-stores**, neither many **LLC-stores misses** their correlation with IPC is near to 0.
- We also can see that as the number of instructions increases the number of **branch-instructions** is increased in a common behavior, because branch-**instructions** are correlated with IPC at value 0.5.
- **L1-dcache-stores** have a correlation approximately to 0 with IPC because the values it takes are behaving in a different manner than IPC.
- **LLC-load-misses** are inverse associated with IPC. Its correlation value with IPC is -0.359 which is logical because the misses (from loads) that occur at last level

cache will force the hardware to look for the data in memory that costs (penalty that increases the cycles needed to complete a set of instructions). So as the IPC increases the LLC-load-misses should start to decrease at some point and opposite.

- We observe that as the number of IPC increases at some point the number of L1-dcache-loads also starts to increase. They have correlation 0.518. This shows that their behavior is common. It is important to mention also that this can be done because of the out-of-order and multiscalar CPU that the PC which this program ran, has.
- Also, it is interesting the fact that both **branch-instructions** and **branch-Misses** behave similarly (It can be seen from the graphs) but also their correlation with IPC is 0.44 and 0.43 respectively. This is not as we might expect. At first glance we think that it can't be because as the branch-instructions increase, the branch-misses at some point might also increase and this can cost to the CPU. But as we can see this doesn't applies always. Because of the good hardware of CPU, and its capabilities, like out-of-out execution, technique of reorder-buffer, superscalar CPU and many more, the cost of penalty is significantly minimized. Also the small number of branch instructions make the effect of the penalty of branch miss weak in order to decrease the IPC.
- Also, **L1-dcache-loads** have 0.51 correlation with IPC. This might happen because as the loads from L1-cache increase because of the small number of **L1-dcache-load-misses** the IPC increases at some point. Also because of some optimizations and capabilities of the processor, a significant number of penalties caused from **L1-dcache-load-misses** can be skipped (Escape from the cost on the performance of the processor).

- Also, the **L1-dcache-load-misses has** a small correlation with IPC (= 0.177), something that also should move inversely with IPC and not at some point behave commonly.
- **dTLB-loads** have the third biggest correlation with IPC. Its correlation with IPC is 0.518. That's because the loads from dTLB are behaving quite similarly with IPC. This applies as we see, also to **L1-dcache-loads**. Both of them have a big correlation with IPC. This might apply generally because of the efficient hardware of the CPU.
- We observe that the correlation between **dTLB-load-misses** and IPC is negative (= -0.29). This is logical because we can understand that the occurrence of many misses (from load operation) at dTLB is decreasing the IPC because more and more penalty will be having to be paid, delaying the execution of a set of instructions (increasing the number of cycles needed for the execution of the instructions).
- We can see that **dTLB-stores** are behaving with a different way Than IPC because their correlation is approximately.
- Also, **dTLB-store-misses** has got correlation 0.02 (approximately 0) because **dTLB-store-misses** takes small values.
- **iTLB-loads** is independent from IPC because we can see that its correlation with IPC is near to 0.
- **iTLB-load-misses** isn't correlated with IPC because the values that it takes are extremely low. This is a factor that doesn't affect the IPC at all.

VII. ITRACE / CODE DISASSEMBLE

We have tried to analyze the code of our benchmark. We have used iTrace software in order to find the Instructions that are executed most of the time. We have changed the code of the iTrace program in order to store in two arrays the instruction id and the counter of the repetitions. We have used bash commands in order to sort the array and find the most instructions that are executed most. We found that 850 unique x86_64 assembly instructions are executed 1264975759 times each one of them in our program. Those instructions represent 6 functions of our SPEC. The 41% of our execution time is taken by those 6 functions. We have calculated this number by using this formula:

Time Top Instructions (sec) = (Top Instructions / Instructions) * Time

Top Unique Instructions	850
Repeats per Instruction	1264975759
Top Instructions	1.08E+12
Total Instructions	2.60E+12
Time	279.7041444
Time Top Instructions (sec)	**115.1973318**
Percentage	**41.30%**

Top 6 functions: (ComputeNonbondedUtil)

- calc_pair(nonbonded*)
- calc_pair_energy(nonbonded*)
- calc_pair_fullelect(nonbonded*)
- calc_pair_energy_fullelect(nonbonded*)
- calc_pair_merge_fullelect(nonbonded*)
- calc_pair_energy_merge_fullelect(nonbonded*)

Caller: (Compute)

doWork(PatchList *patchList)

Each function was taken as argument a struct of type nonbonded which represents a molecule. The functions where not implemented in our source code and they are imported from another library. We could not find the source code of the functions and we have to make some hypothesis about there functionality from the code we had. As we observed from the assembly code calc_pair() is a recursive method that is calculating actions between to molecules. In general, our SPEC almost, all of the runtime is spent calculating inter-atomic interactions in a small set of functions. Our SPEC uses the 41.3% of the execution time in those 6 methods. Each Instruction in those methods repeats 1264975759 times. In total of assembly instructions, we have 850 unique commands. Those methods are very important for our program as they used in order to calculate the energy between a pair of molecules. Due to the fact that most of our time is spent in just mathematic equations our benchmark is very boring ang stable with IPC in average at 2.57. From our assembly code and C++ code we could see that our program is not using too much comparison statements. Most of the time is calculating behaviors and interactions between two molecules. Bellow you can see the assembly code of the method calc_pair(). We have searched for other functions to see which one is the caller of those 6 functions. The function doWork() is the caller of those 6 Top Functions and the source code is at Compute.C. We decided to compare this function's C++ code with the assembly in x86_64. Bellow you can see the two codes.

Assembly x86_64 (91 Instructions)

```
sub $0xc8,%rsp
movslq (%rdi),%rax
mov 0xc(%rsi),%r8d
movsd 0x2fb84a(%rip),%xmm0 # 0x6fd7e0
<_ZN20ComputeNonbondedUtil6cutoffE>
mov (%rsi),%ecx
movq $0x0,0x58(%rsp)
movq $0x0,0x60(%rsp)
movq $0x0,0x68(%rsp)
lea 0x0(,%rax,8),%rdx
shl $0x6,%rax
sub %rdx,%rax
add 0x18(%rsi),%rax
test %r8d,%r8d
mov 0x8(%rax),%rdx
mov %rdx,(%rsp)
mov %rdx,0x8(%rsp)
mov 0x10(%rax),%rdx
.
.
.
je 0x4020e0 <_ZN11SelfCompute6doWorkEP9PatchList+352>
mov 0x20(%rax),%rax
mov %rsp,%rdi
mov %rax,0x40(%rsp)
mov %rax,0x48(%rsp)
mov 0x8(%rsi),%eax
test %eax,%eax
je 0x4020f8 <_ZN11SelfCompute6doWorkEP9PatchList+376>
test %ecx,%ecx
je 0x402120 <_ZN11SelfCompute6doWorkEP9PatchList+416>
callq *0x2fb28b(%rip) # 0x6fd330
<_ZN20ComputeNonbondedUtil19calcMergeSelfEnergyE>
add $0xc8,%rsp
retq
nopl (%rax)
movsd 0x2fb708(%rip),%xmm0 # 0x6fd7c0
<_ZN20ComputeNonbondedUtil12pairlistdistE>
movl $0x1,0x18(%rdi)
movl $0x1,0x90(%rsp)
mov $0x1,%edx
movsd %xmm0,0x98(%rsp)
jmp 0x402060 <_ZN11SelfCompute6doWorkEP9PatchList+224>
nopw 0x0(%rax,%rax,1)
test %ecx,%ecx
mov %rsp,%rdi
jne 0x402110 <_ZN11SelfCompute6doWorkEP9PatchList+400>
callq *0x2fb2d3(%rip) # 0x6fd3c0
<_ZN20ComputeNonbondedUtil8calcSelfE>
add $0xc8,%rsp
retq
nopl (%rax)
test %ecx,%ecx
je 0x402130 <_ZN11SelfCompute6doWorkEP9PatchList+432>
callq *0x2fb26e(%rip) # 0x6fd370
<_ZN20ComputeNonbondedUtil18calcFullSelfEnergyE>
add $0xc8,%rsp
retq
nopw 0x0(%rax,%rax,1)
callq *0x2fb29a(%rip) # 0x6fd3b0
<_ZN20ComputeNonbondedUtil14calcSelfEnergyE>
add $0xc8,%rsp
retq
xchg %ax,%ax
callq *0x2fb21a(%rip) # 0x6fd340
<_ZN20ComputeNonbondedUtil13calcMergeSelfE>
add $0xc8,%rsp
retq
xchg %ax,%ax
callq *0x2fb24a(%rip) # 0x6fd380
<_ZN20ComputeNonbondedUtil12calcFullSelfE>
add $0xc8,%rsp
retq
```

C++ Source Code (50 Lines of Code)

```
void PairCompute::doWork(PatchList *patchList) {

  Patch *p1 = &(patchList->patches[patchId1]);
  Patch *p2 = &(patchList->patches[patchId2]);
  int doEnergy = patchList->doEnergy;
  nonbonded params;
  const Lattice &lattice = patchList->lattice;
  params.offset = lattice.offset(image1) - lattice.offset(image2);
  params.p[0] = p1->atoms;
  params.p[1] = p2->atoms;
  params.pExt[0] = p1->atomsExt;
  params.pExt[1] = p2->atomsExt;
  params.ff[0] = p1->f_nbond;
  params.ff[1] = p2->f_nbond;
  params.numAtoms[0] = p1->numAtoms;
  params.numAtoms[1] = p2->numAtoms;
  params.reduction = patchList->reductionData;
  params.pressureProfileReduction = 0;

  params.minPart = 0; // minPart;
  params.maxPart = 1; // maxPart;
  params.numParts = 1; // numParts;

  params.workArrays = &workArrays;

  params.pairlists = &pairlists;
  params.savePairlists = 0;
  params.plcutoff = cutoff;
  if ( patchList->savePairlists ) {
    params.plcutoff = pairlistdist;
    pairlistsValid = 1;
    params.savePairlists = 1;
  }
  params.usePairlists = pairlistsValid;
  //  params.groupplcutoff = cutoff + 2. * patch-
>flags.maxGroupRadius;
  params.groupplcutoff = params.plcutoff + hgroupcutoff;

  if ( patchList->doFull ) {
    params.fullf[0] = p1->f_slow;
    params.fullf[1] = p2->f_slow;
    if ( patchList->doMerge ) {
      if ( doEnergy ) calcMergePairEnergy(&params);
      else calcMergePair(&params);
    } else {
      if ( doEnergy ) calcFullPairEnergy(&params);
      else calcFullPair(&params);
    }
  }
  else
    if ( doEnergy ) calcPairEnergy(&params);
    else calcPair(&params);

}
```

VIII. CONCLUSION

As a conclusion we observe that our program benchmark is boring. We can see it from the stable behavior of the program, from the high IPC and from the patterns that each statistic follows. Except the fact that our benchmark program is boring, something more interesting to say is that this program is a great example on how a program can break into segments-parts which each part can repeat a routine. This can be seen from the patterns of the behavior of the statistics in the graphs. Also, something interesting to see is that our program at the first half of its execution time, the rate of dTLB-load misses is big and as the time passes after the middle of the execution of the program the rate of dTLB-load-misses is decreasing, helping the IPC to increase. Moreover, the small number of branch instructions doesn't affect the quite big rate of branch misses to affect the IPC.

All these factors which we have seen above aren't affecting the program at its execution. This helps the program to have big IPC. We can see this, because our IPC is 2.5 (average). this is greater than the half of the ideal IPC.Because we have a superscalar CPU with 4-way superscalarity the ideal IPC is 4. Ours is 2.5 which is extremely high.

Also, some general conclusions:

We can observe the differences of some values in different executions and that this depends on the hardware and on software, its initial and general state, and also, we can see how the performance of the software changes in some hardware capabilities. Also, the analysis that has been made above has clarified the fact that the programs take advantage of some capabilities of the hardware, scalability, out of order execution and more. Moreover, we have observed and realized that we can recognize some phases of the program (like functions with many branches, phases handling a lot of data) by observing how the program behaves during the its execution and last, we have realized that many factors that we think affect the hardware, at the end might not affect it, as we think, but with a different way and with not as much weight as we think. So, we come at the conclusion that the processor can optimize quite well the behavior of the program only and only if we know a little deeper than the surface of abstraction of the Programming principles. Sometimes the only thing that remains is to look a little bit deeper or and from a different sight of view.

We would like to thank our Professors Dr. Petros Panagi and Dr. Giannos Sazzeidis for their support in analyzing this benchmark and for providing us with the knowledge which we needed in order to start researching.

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. (references)

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, “Title of paper if known,” unpublished.

[5] R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[8] https://www.ks.uiuc.edu/Research/namd/

[9] https://en.wikipedia.org/wiki/NAMD

[10] https://ark.intel.com/products/80815/Intel-Core-i5-4590-Processor-6M-Cache-up-to-3-70-GHz-

[11] https://www.intel.com/content/www/us/en/processors/processor-numbers.htm

[12] https://www.spec.org/cpu2017/

[13] https://perf.wiki.kernel.org/index.php/Tutorial#multiplexing_and_scaling_events

[14] https://en.wikipedia.org/wiki/Intel_Core

[15] https://twiki.cern.ch/twiki/bin/view/LCG/VIHugePages

[16] http://oprofile.sourceforge.net/docs/intel-core2-events.php

[17] https://www.ks.uiuc.edu/Research/namd/doxygen/index.html

[18] https://docs.oracle.com/cd/E19641-01/802-1948/802-1948.pdf