International Journal of Applied Engineering Research
ISSN 0973-4562 Volume 10, Number 10 (2015) pp. 26979-26994

http://www.ripublication.com

Multicore Architecture Cache Framework Analysis

1. M.Vijay Albert William, 2. K.Bhaskar, 3. Hemakumar.V.S,

1,2,3, School of Electrical Engineering, Vel Tech University, Avadi, Chennai-62,
E-mail: vijayalbert@veltechuniv.edu.in,
bhaskar@veltechuniv.edu.in,chennaihk@rediffmail.com,
TamilNadu, India.
Tel: +919843696519.

Abstract

Nowadays in modern Multicore architecture processors more and more transistors are employed on chip Caches to reduce latency to data and to achieve high performance computing. Therefore On chip cache consumes significant amount of processor power and cycles but need to delivery high performance. To achieve performance cache resources must be optimized to meet the necessity. Since most modern processors support performance monitoring unit or performance counters. These performance counters are implemented to measure the events of the processor. In this paper we are exploring the different types of events which impact on L1 cache, Instruction cache, Data cache and cache TLB. It is furthermore based on per CPU monitoring instead of aggregated no of available CPU's. This could help normal application developers to self-profile their codes. Make them to understand most of the basic programming parameters which are considered for while developing applications without requiring more complicated environmental setup or programming.

Keywords: Memory Cache, Linux OS, Frame work

Introduction

The increase in nature of complexity in computing increase leads to thousands of processors in a chip by this decade. Which also implies complex memory hierarchies, interconnects etc. Moore's law drivers that the complex ability of Multicore architecture also will impact other domains like hypervisors, operating systems, compliers and applications as well. The most majority of the users are exposed to how fast their applications are running the bottom line. Now all the CPUs are manufactured with Performance Counters and with architectural registers to support it. Performance counters allow a low level analysis of the running programs on a target machine. As of November 2013 Most of the super computers in the world are running some form of Linux.

Since the inclusion (2009) of Perf_event subsystem has introduced a wide range of access to CPUs. There have been various implementations of Performance counters outside the main Linux code base expect the perf tools over all these years and require a custom patching to the kernel source code.

Background

Nowadays more number of events are been developed and these events availability are based upon the hardware architecture and even varies from manufacture to manufacture. The supported event details are mainly founded from the specification manual and doesn't guarantees for useful results. When considering Intel x86 series an access to the performance counters requires a direct control over Model Specific Registers (MSRs). These registers are further sub divided into configuration registers and counter registers. The configuration register holds the control of starting, stopping events and the event to be monitored and overflow interrupts. Generally 2 to 8 counting registers are available. Read and writing to the configuration registrations may require privilege levels. Some kernels provide additional features like event scheduling, multiplexing, Per CPU counting, Process attaching.

Most of the Linux users typically depend upon the *ps*or *top*command and look at the column *%CPU* for their running process, when the values are typically low than 100% they start investigate the reasons: Resource conflicts (more process or thread than the hardware threads), Slow IO, Virtual Memory effects, latency, etc. The CPU usage doesn't conclude how often that the processes are scheduled for execution by the OS.

Remote memory accesses are a major source for latency in multicore. This latency are due to extra cycles required to access from different levels of cache, main memory, or from another node Processor and interconnections links on the main memory. Several techniques have proposed to reduce the like contention aware scheduling with memory migration, cache coloring, cache aware scheduling. These techniques are generic and depend on experience-based technique for solving problems. They are effective as a co-scheduler and they are always not appropriate it can also reduce the performance of application adding some extra overhead to it. So these are the parameters made as to make analysis of the perf_event syscall could be an effective way for self-profiling an end developer's tool.

The global statically allocated objects and data can be found by analyzing the embedded program binary and system libraries from which the size and the precise virtual address range can be gathered by using tools like Intel VTune and Memphis.

Related Works

1. Perf tools: The inbuilt perf tool gives an overview of user application performance, since it's a command line tool it lacks the most important feature

like profiling only the programmer interested area of code especially analyzing deadlocks and different type of locks.

2. OProfile: OProfile is a system wide sampling profiler which was added to the kernel v 2.5.43. It allows user to profile based on arbitrary sampling performance counter. It also has added features like frequency graphs and stack traces. It requires a root privilege to start and it lacks aggregate event counts since it's a sampler.
3. Perfctr: The Perfctr provides interface to the kernel and opening dev/perfctr and access it using through ioctl () calls. It requires a libperfctr library which provides an abstraction to the kernel interface.
4. iv)Perform2: It's an extension of itanium-specific perform. It provides a variety of syscall and system wide configuration. The libpfm3library a list of performance events, an event scheduler and a high level interface to the kernel. It works on the user space.
5. LIKWID: It is a method which bypasses the Linux kernel and directly accesses the MSRs. It usually conflict when there is a concurrent perf_event. It is supported on x86 architecture only and it provides only system-wide profiling (MSRs value can't save at context switch).

Performance Monitoring Unit

Modern CPUs architectures like x86, x64, ARM, PowerPC provide access to hardware event counters through programmable performance monitoring unit. The PMU have control registers by configuring these registers it is possible to monitor the events which required for analysis like cache accesses or misses or branch mispredictions. The registers may be read-only or read-write. Kernel controls whether registers are directly accessible to non-privileged user applications or whether accesses must be done in privileged mode from user program code/ application. Most of the CPUs are provide with 5 or 6 registers which are typically limited than the number of events that are available, so it leaves to the user select a set of events of interest. The set of events types has to be specific to the micro architecture in which we are analyzing since they differ from architecture to architecture. Performance monitoring registers can be set up to trigger interrupts when they overflow.

Cache Framework

Cache framework is a program for the Linux environment. The familiar cache-misses cache references, branch-misses, CPU-cycles. Our framework reports L1 data cache Loads, misses, stores, prefetches and prefetch misses. Similarly for L1 instruction cache loads, load misses prefetches. Finally last level cache load, misses etc. Various methods to access hardware performance counters proposed, our framework is built on top of perf_event_open () system call added in Linux Kernel 2.6.32, which is available on many architectures including x86, x64, PowerPC, ARM. So that using perf_event_open. We can directly access the hardware performance counters and reduce the overhead.

CPU has only the limited number of performance counter registers, to overcome this problem our framework support event multiplexing. Event multiplexing is a set of the interested eventsets; kernel schedules them in Round Robin for execution, then scales theresults to extrapolate their values for the entire program.

Multiple events can be scheduled to start one after other and collects results individually. The accuracy of this mode is high when compared to Multiplexing mode but the down side the user program has to start automatically for every event and has to record the values so it consumes lot of time. The user can select between user, kernel and hypervisor mode in which they want to run profiling. Our framework supports both systems wide, Per-processor mode of profiling. Per-thread Per-Processor mode of profiling this requires privilege support. System wide events has the advantage that it can account for all threads running in all the processor, it gives the snapshot of the system resource usage. On the other hand, system wide profiling makes it difficult to separate events of interest from unrelated system activities or noise. Per-thread profiling allocates the thread with its own logical set of performance counter registers, our framework maintain a Per-thread state which updated on each CPU context switches. In Per-thread Per-Processor profiling allocate the thread with its own logical set of performance counter register on the pinned Processor and maintain the thread state and updates when it get job done.

Experimental Setup:

We have used three types of PCs in our experiment are listed in Table 1, on each system we used Linux kernel version 3.2.54. Make sure that the kernel is mounted with Cgroup under /sys/fs/cgroup. Also verify the menuconfig whether the kernel is enabled with following options under

General Setup

→ Control Group Support

→Enable perf_event per-CPU per-container group (cgroup) monitoring

Table 1: The configuration details of System used for analysis

character	Intel core 2 Duo E8300	Intel core i5 2300	Intel core i5 M450
Model	23	42	
Clock Rate	2800 MHz	3100 MHz	2400 MHz
L1 instruction Cache L1 Data Cache	2*32 KB 2*32 KB	4 *32 KB 4 *32 KB	2*32 KB 2*32 KB
L2 Cache	6 MB shared	4 *256KB	2*256 KB
L3 Cache	-	6 MB	3 MB
No of Cores/ Threads	2/2	4/4	2/4
Hyper Threading	-	-	yes

If these settings are not selected then you need to build the kernel with these changes. Per_event_open supported from Linux kernel 2.6.32 but it lacks per CPU per thread support, these features are added from kernel version 3.2.XX. Perf_event system call "one" system call to setup an event get a file descriptor back to identify event normal file sharing semantics apply.

The syntax of perf_event_open is as follow

#include<linux/perf_event.h>

int perf_event_open(structperf_event_attr **attr*, pid_t*pid*, int*cpu*, int*group_fd*unsigned long *flags*);

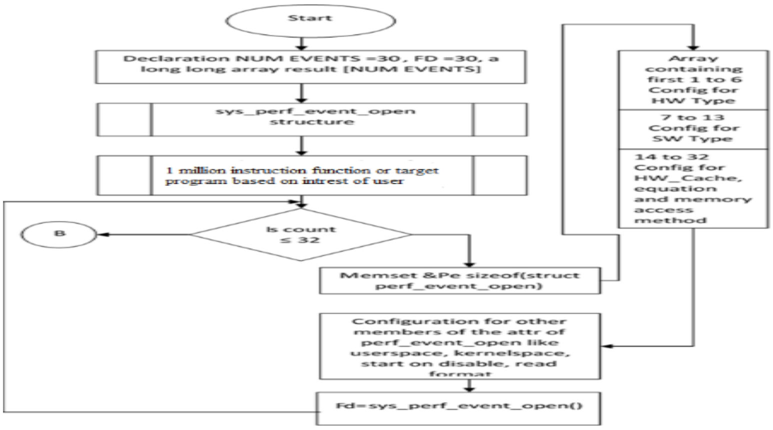

Figure 1: Successive Multiple Event configuration Steps.

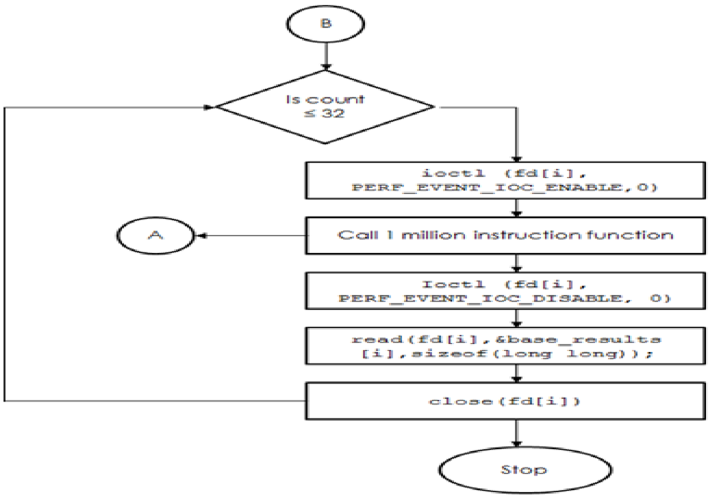

Figure 2: Successive Multiple event flow steps

The programs we used for analysis are 512*512 native matrix multiplication, a program containing 1 million instruction based on assembly language,

Fig 1 show the configuration steps for setuping the test code for profiling and getting a file descriptor for each event. This is based on Succesive Multiple events profiling (one after the aother events). For multiplexing event mode we the configuration steps are as same as shown in Fig 1. For per PID per CPU mode of

profiling we need to mount the cgoup filesystem like as folowfd=open ("/sys/fs/cgroup/test,O_RDONLY");
Event_fd[i]=sys_perf_event_open(&pe,fd,0,-1,0); close(fd); It represents to measure in cgroup test on CPU0 assuming croup is mounted.

Fig 2 shows how the events are stared and stopped for Succesive Multiple events mode. In the case of Multilpexing Events mode all the events have to start and stop at the same time so it requires a little changes. To measure the time of execution we are using the Gettimeofday API provided by Linux kernel. Apart from general configuration in order to measure the SW_CONTEX_SWITCH and SW_CPU_MIGRATION the kernel space measurement must be enable in the perf_event_structure attribute. The perf_event_structure attribute offer pinned=1, must always be on PMU, exclusive=1, only group on PMU. The newly created thread PID is acquired and monitor by configuring task=1, the location of the data's accessed are gathered by enabling mmap.

To ensure the accuracy of the performance counters we run a program containing 1,500,000 branches of which 500,000 are not taken branches. We find that most time the counter very close to the expected values. The experimental count values are just added in Table 1. Similarly we also run another programming containing 1 Million assembly instructions. The results are displayed in Table 2.

Table 2: Experimental result of 1,500,000 branch instructions

1 million instruction	Core 2 Duo	Core i5 m540	Core i5 2400
Expected	1000000	1000000	1000000
Result	1000015	1000018	1000013

Table 3: Experimental result of 1 million instructions

1 million instruction	Core 2 Duo	Core i5 m540	Core i5 2400
Expected	1000000	1000000	1000000
Result	1000015	1000018	1000013

To analysis the self performance profiling of a program we run a 512*512 native matrix multiplication. The analysis result of the program are shown in graph 1 to 9. We observed that the software context switch and software CPU migrations are occuring at the kernel level so we need to measure Kernel space also. When are measuring in the per CPU per PID mode we find that Software CPU_Migrataions are occurred even under hard CPU affinity is applied. Since they save the content of the required registers any where in the CPU during software Context switch. When ever a Hardware_CPU_Migrations ocurs in a process then the L2 cache misses are high and LLC Cache acces are high and also misses . Since the L2 caches have block of datas that are ready to excute immediately.

In the case of multiplexing events the accuray of events were not up to the level as we expected. The multiplexing can be effective only for a dual core processor with 2 levels of cache. In the case of i5 m450 the scaled count error percentage was in the range of >40% wheras in the case of i5 2300 the error rate was >50%. So it is best addviceable when you are considdering performance analysis for a heavly threaded and memory contented program it is better to go for Succusive multiple event or for only the intresed event profiling. The multiplexing analysis results are listed in Fig 3 core 2 Duo E8300 and Fig 4 intel core i5 m450.

Activities | GNOME Terminal | Fri 09:58

desd@linux-ptz9:~/core2

File Edit View Search Terminal Help

```
linux-ptz9:/home/desd/core2 # gcc ex1.c
linux-ptz9:/home/desd/core2 # ./a.out
```

Event	TotalCount	RawCount	Scale	ScaledCount	Error
HW_CPU_CYCLES	1832280636	140984219	12.93	1822925951	0.51%
HW_INSTRUCTIONS	2838489024	326883437	8.64	2821004061	0.62%
HW_CACHE_REFERENCES	152616619	17138918	8.64	147908862	3.08%
HW_CACHE_MISSES	198018	26837	8.67	232408	-17.37%
HW_BRANCH_INSTRUCTIONS	136595917	10480146	13.02	136346699	0.18%
HW_BRANCH_MISSES	287628	21830	13.02	284008	1.26%
SW_CONTEXT_SWITCHES	20	41	1.00	41	-105.00%
SW_CPU_MIGRATIONS	7	4	1.00	4	42.86%
SW_CPU_CLOCK	647916030	649848754	1.00	649848754	-0.30%
SW_TASK_CLOCK	651118123	649850387	1.00	649850387	0.19%
SW_PAGE_FAULTS	151781279	11842223	12.99	153830476	-1.35%
L1D_READ_ACCESS	1436294865	110640433	13.01	1438325629	-0.14%
L1D_READ_MISSES	136464203	10565793	13.01	137355309	-0.65%
L1D_WRITE_ACCESS	275067572	21110752	12.99	274228668	0.30%
L1D_WRITE_MISSES	524961	33405	13.01	434599	17.21%
L1D_PREFETCH_ACCESS	16573170	1268280	13.03	16513005	0.36%
L1I_READ_ACCESS	1823388363	139869328	13.01	1819699957	0.20%
L1I_READ_MISSES	198330	7545	13.00	98009	50.58%
LL_READ_ACCESS	151908012	11777561	13.00	152990517	-0.71%
LL_READ_MISSES	28252	1694	13.00	22005	22.11%
LL_WRITE_ACCESS	393088	22038	13.00	286273	27.17%
LL_WRITE_MISSES	211664	15280	13.00	198487	6.23%
DTLB_READ_ACCESS	1437333871	109894252	13.01	1428625276	0.61%
DTLB_READ_MISSES	322773	23981	13.01	311753	3.41%
DTLB_WRITE_ACCESS	277317917	21120625	13.00	274356918	1.07%
DTLB_WRITE_MISSES	18469	1241	13.00	16133	12.65%
ITLB_WRITE_ACCESS	2840411395	330054562	8.67	2858272506	-0.63%
ITLB_WRITE_MISSES	4	0	8.67	0	100.00%
BPU_ACCESS	137137268	15818241	8.67	137144149	-0.01%
BPU_MISSES	293934	22765	13.01	295945	-0.68%

```
Error on 9 events is too high.
linux-ptz9:/home/desd/core2 #
```

Figure 3: Multiplexing Events output for a Native matrix multiplication of 512*512 For Intel Core 2 Duo E8300

/home/jeeva/Documents/core2 : a.out

File Edit View Bookmarks Settings Help

```
linux-3kkp:/home/jeeva/Documents/core2 # ./a.out
```

Event	TotalCount	RawCount	Scale	ScaledCount	Error
HW_CPU_CYCLES	2087055034	317158417	6.59	2090073968	-0.14%
HW_INSTRUCTIONS	2836267377	537979890	5.24	2819014623	0.61%
HW_CACHE_REFERENCES	138240150	26334669	5.21	137203625	0.75%
HW_CACHE_MISSES	451394	87924	5.18	454567	-0.70%
HW_BRANCH_INSTRUCTIONS	136290858	26327199	5.14	135321802	0.71%
HW_BRANCH_MISSES	282340	54560	5.13	279347	1.06%
SW_CONTEXT_SWITCHES	80	79	1.00	79	1.25%
SW_CPU_MIGRATIONS	0	1	1.00	1	-inf%
SW_CPU_CLOCK	782585956	792291614	1.00	792291614	-1.24%
SW_TASK_CLOCK	783022162	792278894	1.00	792278894	-1.18%
SW_PAGE_FAULTS	137762647	21571084	6.41	138270648	-0.37%
L1D_READ_ACCESS	1214821185	190791231	6.41	1222971790	-0.67%
L1D_READ_MISSES	151587233	23854647	6.41	152908287	-0.87%
L1D_WRITE_ACCESS	271424156	42445773	6.41	272077404	-0.24%
L1D_WRITE_MISSES	339164	42940	6.41	274816	18.97%
L1D_PREFETCH_ACCESS	16910905	1327689	6.41	8497209	49.75%
L1I_READ_ACCESS	28151164	4090910	6.41	26181824	7.00%
L1I_READ_MISSES	280958	46024	6.41	294553	-4.84%
LL_READ_ACCESS	137764641	21419242	6.41	137083148	0.49%
LL_READ_MISSES	225423	28954	6.45	186753	17.15%
LL_WRITE_ACCESS	764687	53957	13.20	712232	6.86%
LL_WRITE_MISSES	268848	20683	13.24	273636	-1.78%
DTLB_READ_ACCESS	1214834633	137748482	8.83	1216319096	-0.12%
DTLB_READ_MISSES	131014	19818	6.62	131195	-0.14%
DTLB_WRITE_ACCESS	271428694	40725154	6.62	269600519	0.67%
DTLB_WRITE_MISSES	1069	154	6.62	1019	4.68%
ITLB_WRITE_ACCESS	2836227764	426995927	6.62	2826713036	0.34%
ITLB_WRITE_MISSES	2033	363	6.62	2403	-18.20%
BPU_ACCESS	136192045	20542160	6.62	135989099	0.15%
BPU_MISSES	5147699	724715	6.62	4797613	6.80%

Figure 4: Multiplexing Events Output for a Native matrix multiplication of 512*512 For Intel Core i5 m450

Figure 5: CPU_CYCLES count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

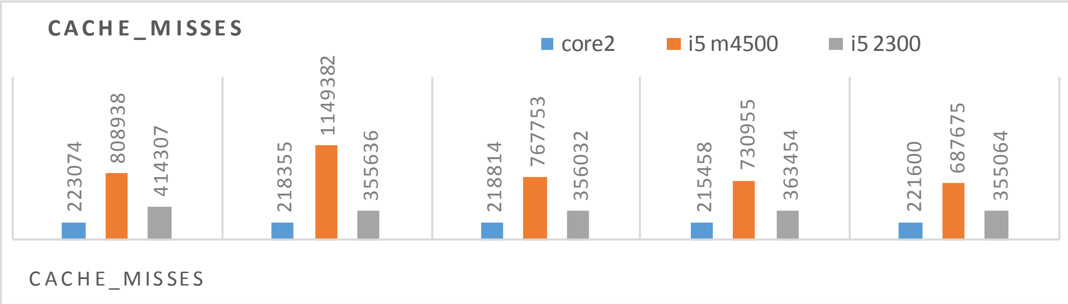

Figure 6: CACHE_MISSES count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

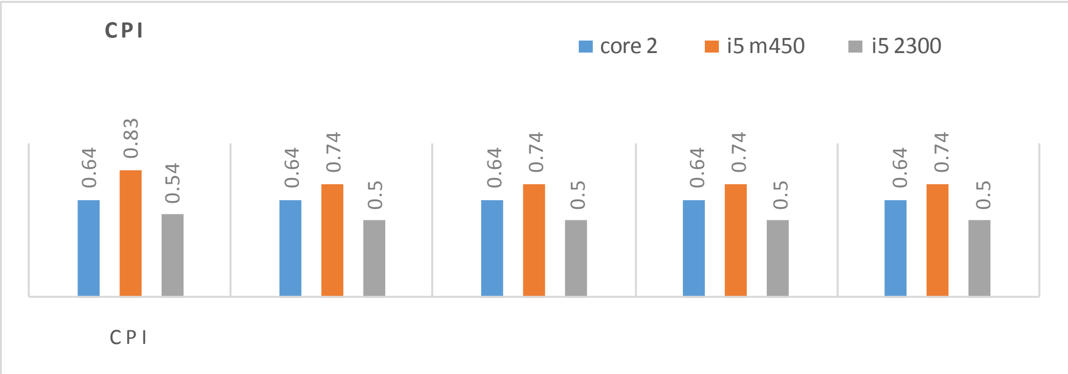

Figure 7: Cycles Per Instruction count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

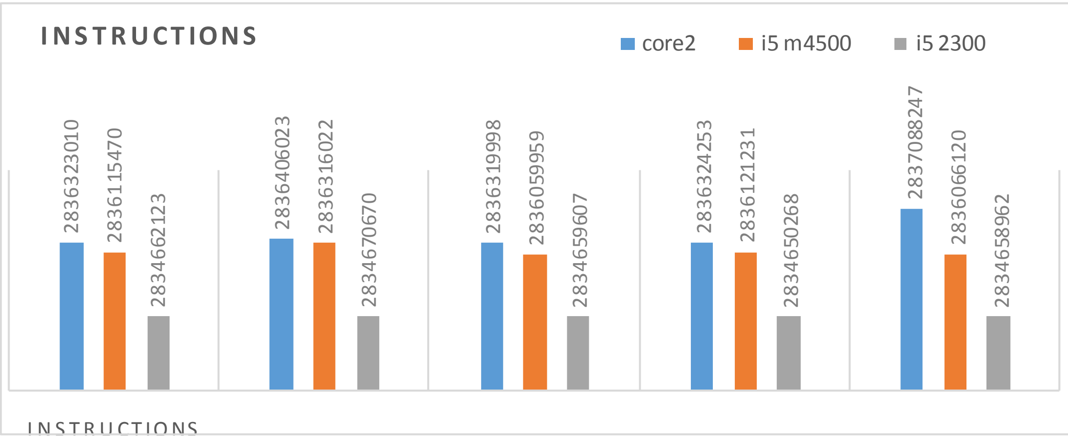

Figure 8: CPU INSTRUCTIONS executed count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

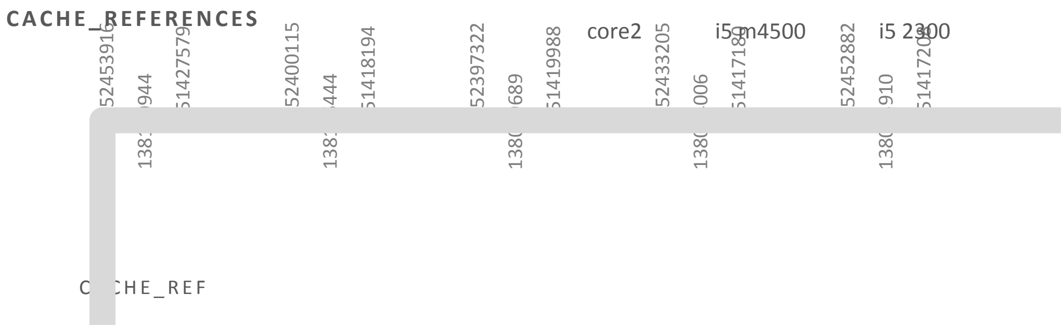

Figure 9: CACHE_REFERENCES count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

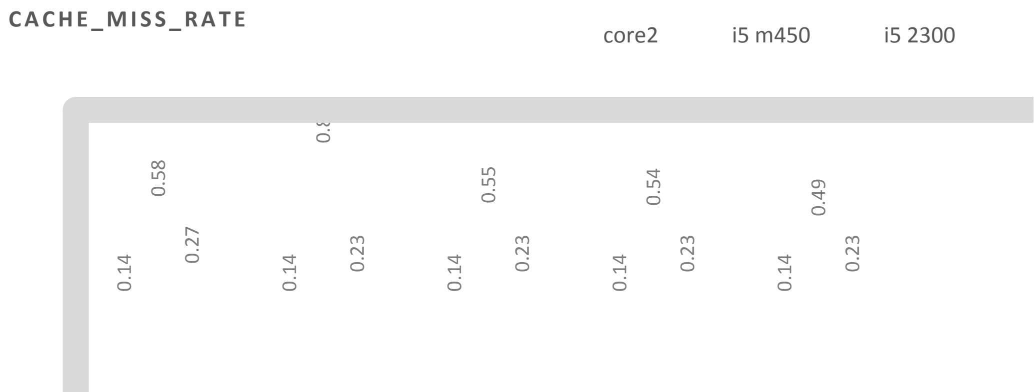

Figure 10: CACHE_MISS_RATE count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

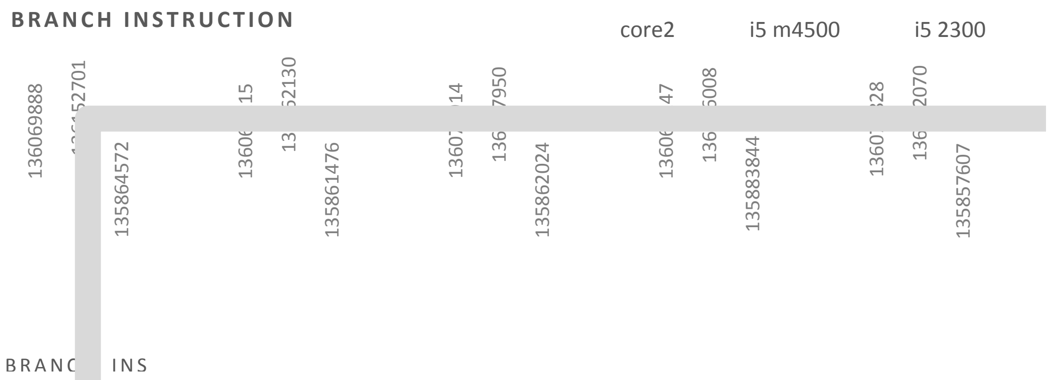

Figure 11: Branch Instructions executed count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

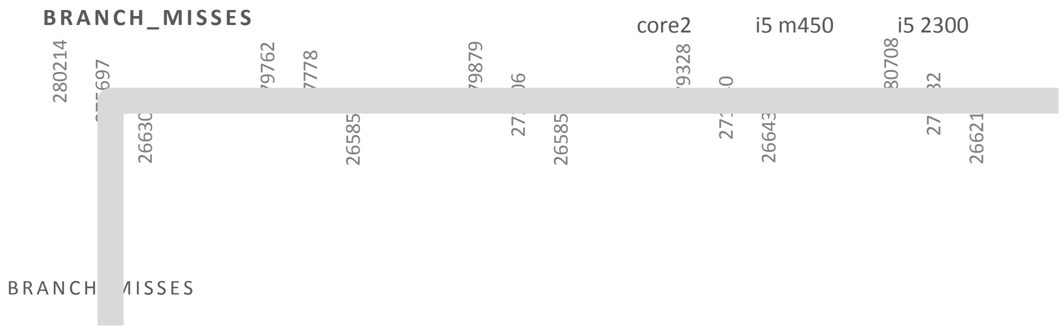

Figure 12: Branch miss prediction rate for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

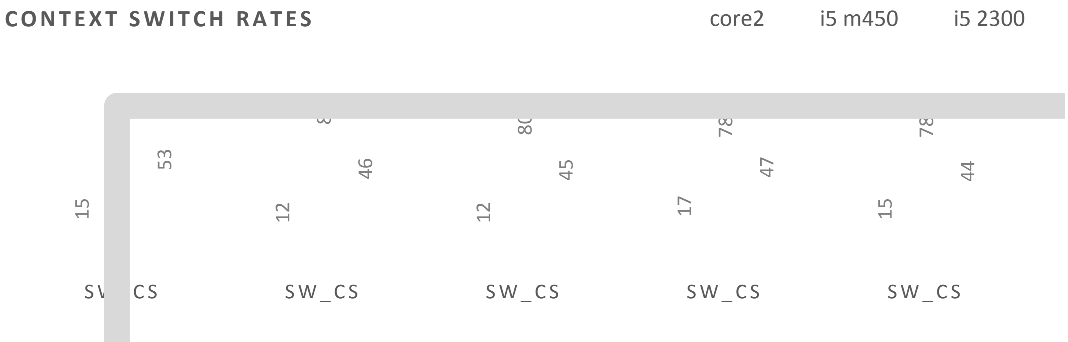

Figure 13: software cpu context switch count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

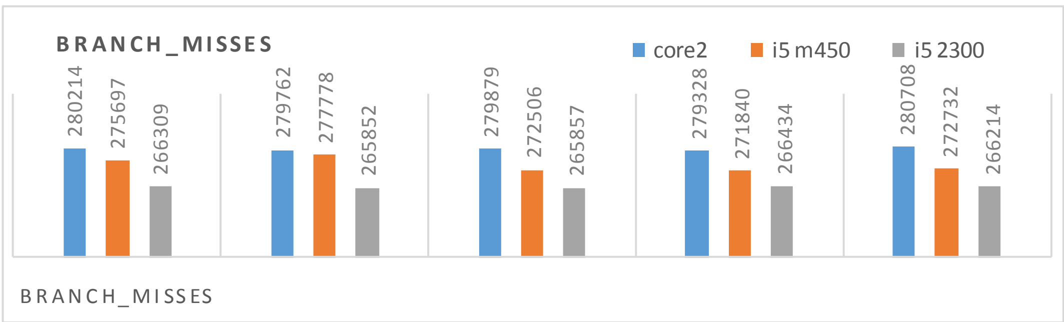

Figure 14: Branch misses count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

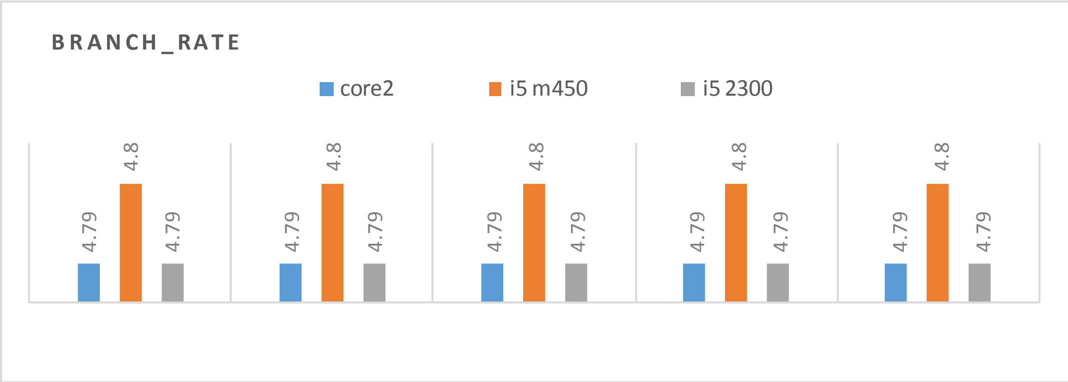

Figure 15: Branch rate count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

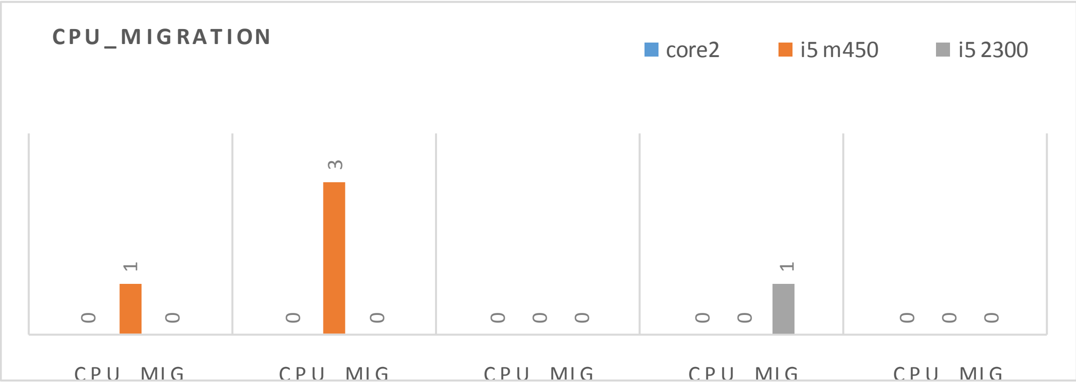

Figure 16: software CPU migration count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

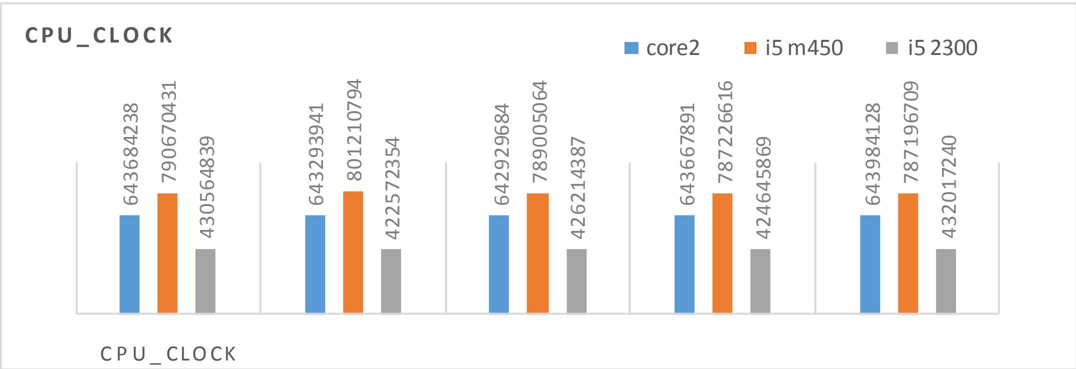

Figure 17: software CPU clock count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

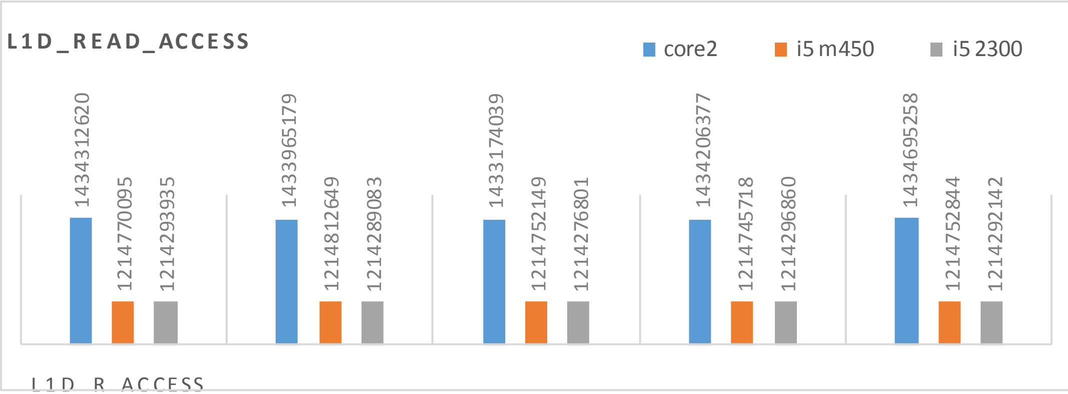

Figure 18: L1Data cache Read access count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

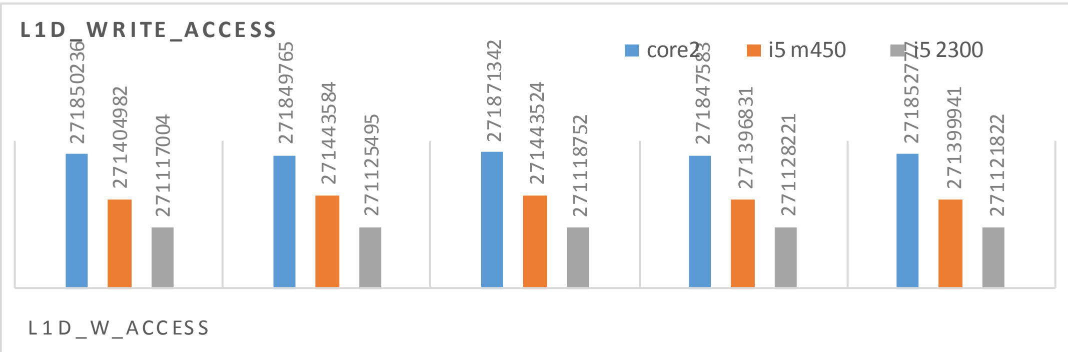

Figure 19: L1Data cache write access count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

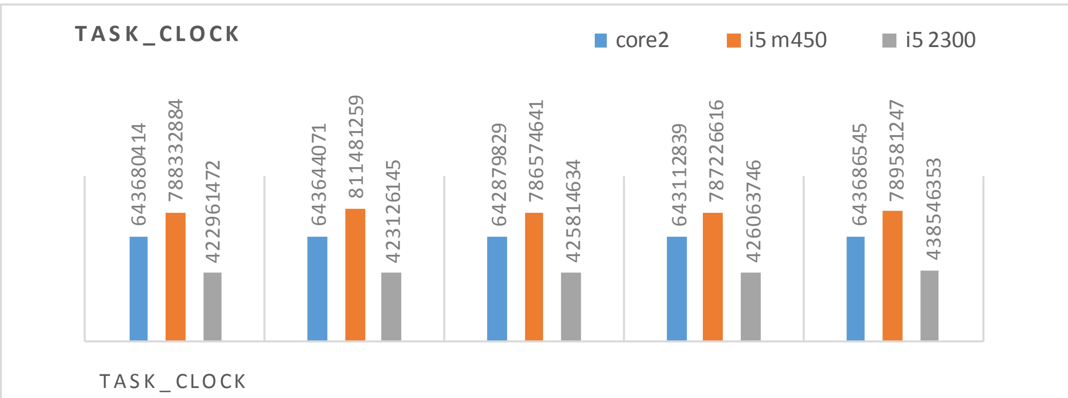

Figure 20: For Core 2 Duo E8300, i5-m450, i5-2300Fig 20: software Task clock count as measured for Native Matrix multiplication

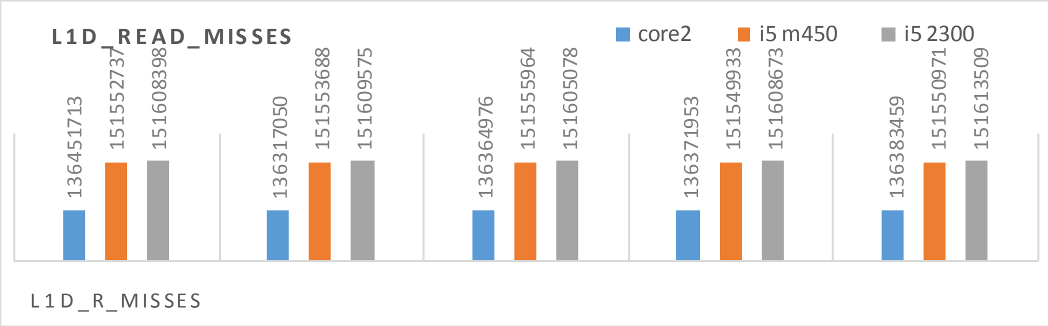

Figure 21: L1Data cache read misses count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

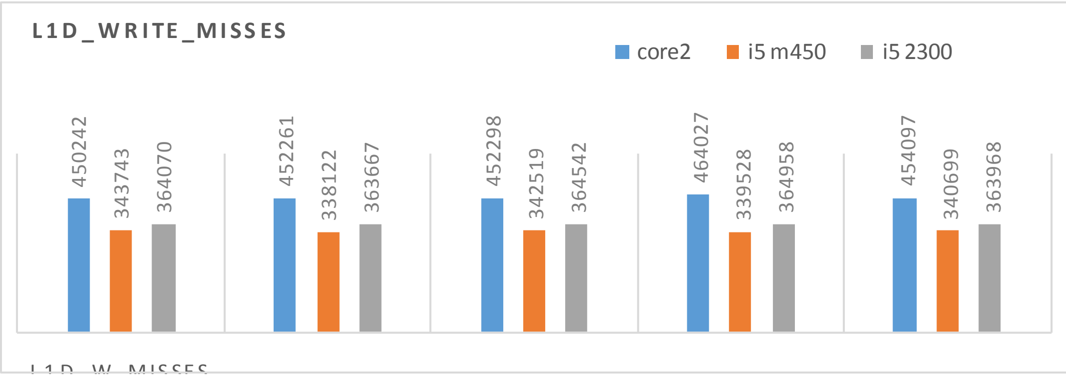

Figure 22: L1Data cache write misses count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

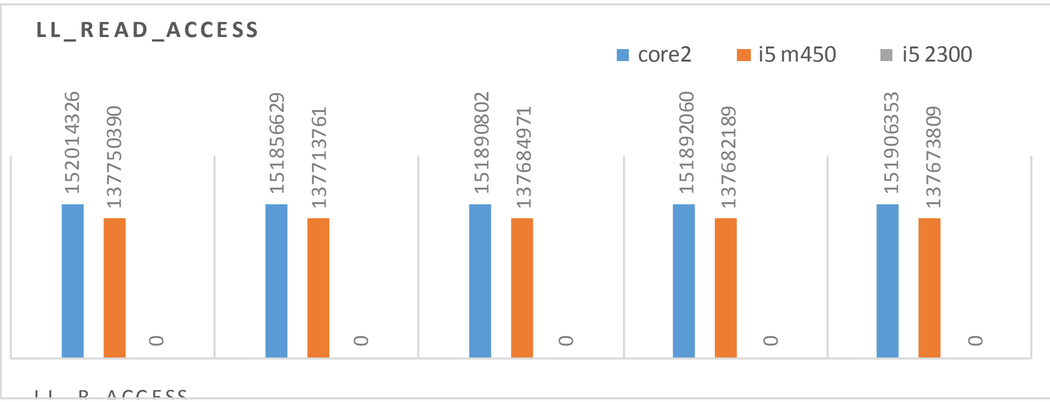

Figure 23: LL cache read access count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

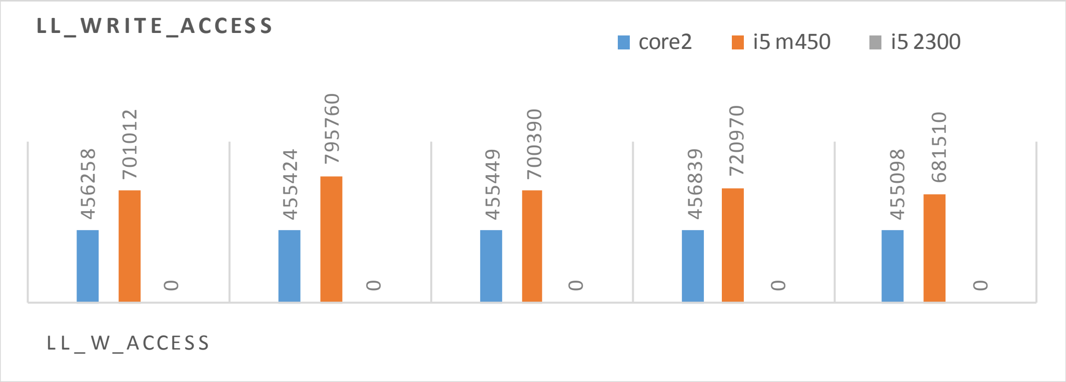

Figure 24: LL cache write acces count as measured for Native matrix multiplication for core 2 Duo E8300, i5-m450, i5 -2300

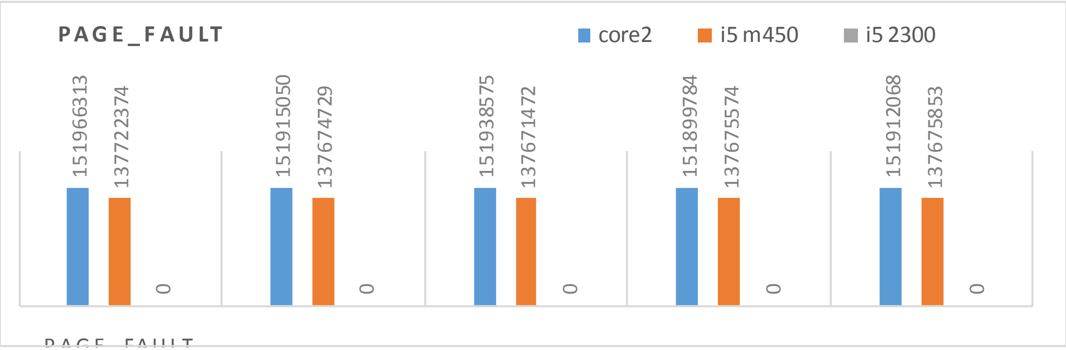

Figure 25: software page fault count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

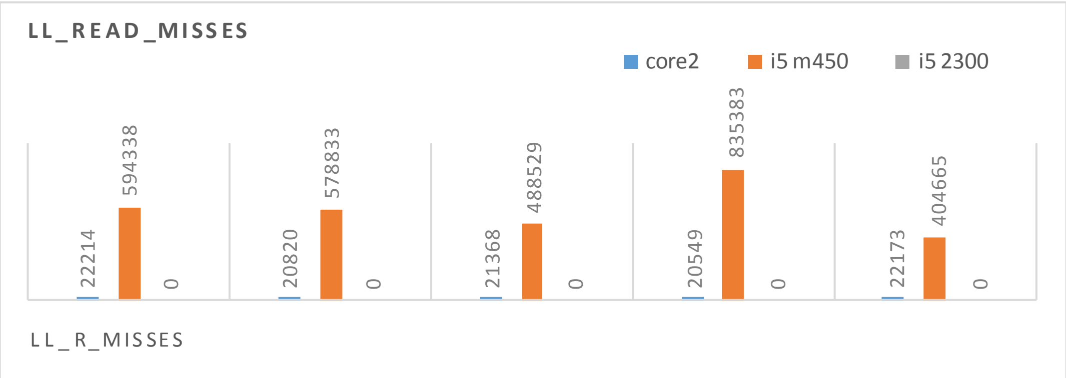

Figure 26: LL cache read misse count as measured for Native Matrix multiplication for Core 2 Duo E8300, i5-m450, i5-2300

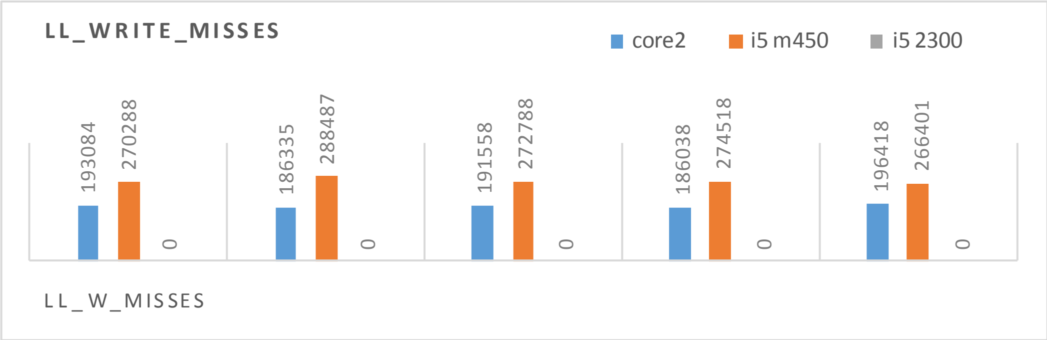

Figure 27: LL cache write Misses count as measured for Native matrix multiplication for core 2 Duo E8300, i5-m450, i5 -2300

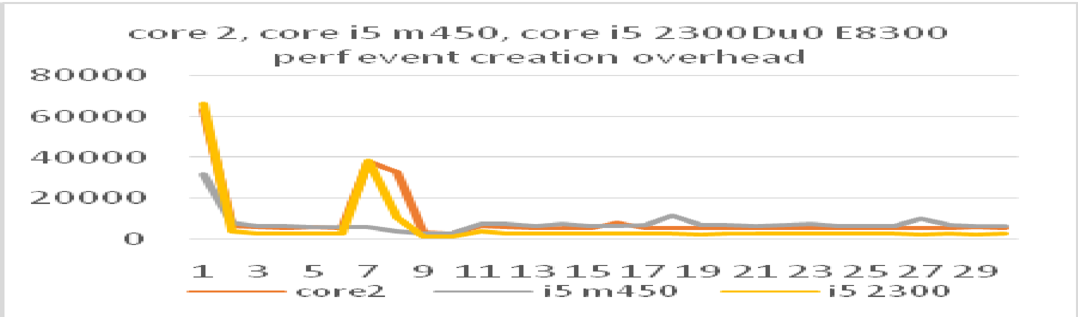

Figure 28: Perf_event creation overhead for each of the event analysied.

Analysising from fig 18 to 27 we conclude that the i5-m450 intel processor with hyperthreading has the same percentage of L1 cache hit ratio compared to off i5-2300 and have higher than core 2 Duo. But in the case of LL cache hit ratio its more closer to Core 2 Duo E8300 and much for away from i5-2300. So from this we can say that how much cycles it saved in the L1 and L2 cache levels compared to of core 2 Duo it losses it cycles at least 2times greater than Core 2 Duo. The core 2 Duo takes have advantage in the case of heavly threaded and memory content programs.

According to the memory hiearchy cpu to L1 cache access cost ~1cycles, L1 to L2 cost ~30 cycles, L2 to L3 cost ~100 cycles, L3 to main memory cost ~300 cycles. So the core 2 has only 2 levels of cache and there fore it elimanates a ~70 cycles delay due to L3 and the LLC of core 2 Duo is larger than i5-m450 hyperthreaded version of intel processors. I5-m450 can also achive the same performance when scheduled under a particular physical processor since the LLC and L2 caches are shared among the 2 logical processors.

Conclusion

We investigated the features of Linux perf_event_open sys_call for self profiling end user or application Developers code. We find that it has more advantages like it is easier to setup the enviroment without any patches to the kernel. It requires only few codes to be added to program with very less amount of overhead. It is user friendly and easily understandable by the users about the impact of parameters being measured and also where to insert in the target program. Our Cache analysis framework Doesn't require any exprince based coding. It is useful for finding deadlocks and race conditions in the user codes.

In the era of high performance computing becoming more complicated it requires performace counter based profiling as a key source to find and avoid system bottelneck.

We are looking forward to adopt this method of profiling for the Real Time Multicore systems. Since nowadays most of the Embeeded devices for remote sensings and communications are packed with Multicore of different CPU architecture to increase the performance based on the performance and available power aware Real Time Systems.

References

[1] Rob Knauerhase, Paul Brett, Tong Li, Barbara Hohlt, Scott Hahn *Intel* Using OS Observations to Improve Performance in Multicore Systems, Micro Volume:28 Issue:3

[2] Intel, 64 and IA-32 Architectures Software Developer's Manual, May 2007,http://www.intel.com/products/processor/manuals/index.htm.

[3] V.M. Weaver. "Linux perf_event Features and Overhead", FastPath 2013: The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, Austin, Texas, April 2013.

[4] V.M. Weaver, J. Dongarra. "Can Hardware Performance Counters Produce Expected, Deterministic Results?", FHPM-2010: The 3rd Workshop on Functionality of Hardware Performance Monitoring, Atlanta, Georgia, December 2010.

[5] V.M. Weaver, S.A. McKee. "Can Hardware Performance Counters be Trusted?" Technical Report CSL-TR-2008-1051, Cornell University, August 2008.

[6] S.V. Moore, D.K. Terpstra, V.M. Weaver. "Chapter 3: Software Interfaces to Hardware Counters", D.H. Bailey, R.F. Lucas (Eds.) *Performance Tuning of Scientific Applications*, Taylor and Francis, 2010.

[7] Linux perf event Features and Overhead by Vincent M. WeaverUniversity of Mainevincent.weaver@maine.edu

[8] Linux perf_events status update by Stephane Eranian, Google, Incm Petascale Tools Workshop 2013.

[9] Intel core2 Duo supported Events http://oprofile.sourceforge.net/docs/intel-core2-events.php.

[10] Intel Westmere Microarchitecture Events,http://oprofile.sourceforge.net/docs/intel-westmere-events.php.

[11] Extending the reach of computing hardware performance monitoring by Mr. Andrzej Pawel Nowak, conference at EPFL, Memento24 August 2012