Using an extended Roofline Model to understand data and thread affinities on NUMA systems

O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel and F. F. Rivera
Centro de Investigación en Tecnoloxías da Información (CITIUS)
Univ. of Santiago de Compostela, Spain
Email: {oscar.garcia, tf.pena, jc.cabaleiro, juancarlos.pichel, ff.rivera}@usc.es

***Abstract*—Today's microprocessors include multicores that feature a diverse set of compute cores and onboard memory subsystems connected by complex communication networks and protocols. The analysis of factors that affect performance in such complex systems is far from being an easy task. Anyway, it is clear that increasing data locality and affinity is one of the main challenges to reduce the access latency to data. As the number of cores increases, the influence of this issue on the performance of parallel codes is more and more important. Therefore, models to characterize the performance in such systems are broadly demanded. This paper shows the use of an extension of the well known Roofline Model adapted to the main features of the memory hierarchy present in most of the current multicore systems. Also the Roofline Model was extended to show the dynamic evolution of the execution of a given code. In order to reduce the overheads to get the information needed to obtain this dynamic Roofline Model, hardware counters present in most of the current microprocessors are used. To illustrate its use, two simple parallel vector operations, SAXPY and SDOT, were considered. Different access strides and initial location of vectors in memory modules were used to show the influence of different scenarios in terms of locality and affinity. The effect of thread migration were also considered. We conclude that the proposed Roofline Model is an useful tool to understand and characterise the behaviour of the execution of parallel codes in multicore systems.**

I. INTRODUCTION

Current microprocessors implement multicores that feature a diverse set of compute cores and on board memory hierarchies connected by increasingly complex communication networks and protocols with area, energy and performance implications. In multicore systems, for a parallel code to be correctly and efficiently executed, its programming must be careful, and the shared memory abstraction stands out as a sine qua non for general-purpose programming [1]. The only practical option for implementing a large cache is to physically distribute it on the chip so that every core is near some portion of the cache [2]. In particular, with exascale multicores, the question of how to efficiently support shared memory model is of paramount importance [3].

The need for models to characterize the performance of these complex systems is an open question nowadays [4]–[10]. The Berkeley Roofline Model [11] is a compact representation of the main features that affect the performance of a code when executed in a particular system. It shows in a simple plot the behaviour of this code based on information about the speed of the computations and the latency to access data.

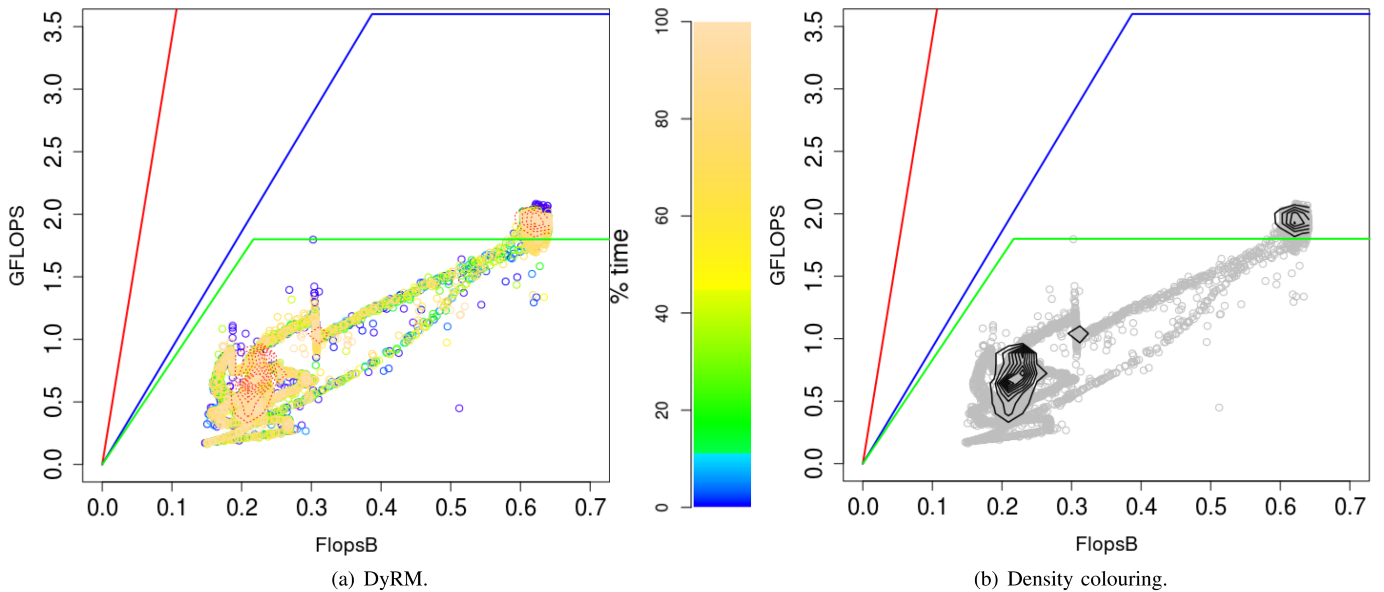
Taking into account architectural features, particularly the behaviour of memory accesses, is critical to improve locality among accesses and affinity between data and cores. Both locality and affinity are important to reduce the access latency to data. In addition, a large fraction of on-chip multicore interconnect traffic originates not from actual data transfers but from communication between cores to maintain data coherence [12]. An important impact of this overhead is the on-chip interconnect power and energy consumption [13].

In particular, performance monitoring is used to identify bottlenecks by collecting data related to how an application or system performs [14]. Characterising the nature and cause of the bottlenecks using this information allows the user to understand why a program behaves in a particular way. Some performance issues in which this information is important are, among others, data locality or load balancing. Their study may help to lead to a performance improvement [15].

Moving threads close to the place where their data reside is a strategy that can help to alleviate these issues. When threads migrate, the corresponding data and directory entries usually stay in the original memory module, and be accessed remotely by the migrating thread which is a source of inefficiencies that can be overlapped by the benefits of the migration [16].

In order to help programmers to understand the performance of their codes, on a particular system, various performance models have been proposed. In particular the Roofline Model (RM) offers a nice balance between simplicity and descriptiveness based on two important concepts: the operational intensity (OI) and the number of FLOPS. Nevertheless, its own simplicity might hide some performance bottlenecks present in modern architectures. In this work, we use extensions of this model [17], [18] to study the effects on the performance of different scenarios in terms of locality and affinity.

The rest of the paper is organized as follows. Next section presents the Roofline Model, as well as the extensions for multicore systems, for the dynamic analysis of the performance, and including latency information. In addition, an introduction to the use of the hardware counters to extract the information needed by the RM with low overhead is shown. Section III introduces a set of case studies based on the SAXPY and SDOT kernels. Section IV discusses the results obtained in the case studies. Finally, the main conclusions are summarized in Section V.



(a) DyRM.

(b) Density colouring.

Fig. 1. Examples of Dynamic Roofline Models for NPB benchmark SP.B.

II. Extensions to the Roofline Model

In this section, the Dynamic Roofline Model (DyRM) [17] and the 3DyRM [18], two extensions to the Roofline Model that have been used in this paper, are introduced.

The RM [11] is an easy-to-understand model, offering performance guidelines and information about the actual behaviour of a program when it is executed in a particular system. It offers insight on how to improve the performance of software and hardware. The RM uses a simple bound and bottleneck analysis approach, where the influence of the system bottlenecks are highlighted and quantified. In modern systems, the main bottleneck is often the connection between processor and memory. This is the reason why the RM relates processor performance to off-chip memory traffic. It uses the term operational intensity, OI, to mean operations per byte of DRAM traffic (measured in Flops/Byte, FlopsB in the Figures). Note that it measures traffic between the caches and memory rather than between the processor and the caches. Some authors have introduced cache-awareness to provide a more insightful model [19]. Thus, OI takes into account the DRAM bandwidth needed by a process on a particular computer. The RM ties together floating-point performance (measured in GFLOPS), OI, and memory performance in a 2D graph.

The Dynamic Roofline Model (DyRM) is essentially the equivalent of splitting the execution of a code in time slices, getting one RM for each slice, and then combining them in just one graph. This way, a more detailed view of the performance during the entire life of the code is obtained, showing its evolution and behaviour. As an example, Figure 1(a) shows the DyRM of a NAS application running on a multicore processor. In this figure, lineal axes are used instead of the logarithmic axes of the original RM to show more detailed differences in the behaviour. As can be seen, a colour gradient is being used to show the program evolution in time. Each point in the model is coloured according to the elapsed time since the start of the program (the same colouring schema is used in rest of figures in this paper).

The DyRM allows the detection of different execution phases or behaviours in the code. In addition, a two dimensional density estimation of the points in the extended model can be obtained (Figure 1(b)). Such an estimation allows to readily find zones in the model where the code spends more time, which are quite useful to identify performance bottlenecks. The resulting groups can be highlighted and, by changing the colour of the points in the DyRM, a better view of them can be obtained. By using both graphs, the simplicity of the RM and a detailed view of the program execution are combined in a compact and simple way.

The OI is used to model the memory performance of a program running in a specific system. As it was said before, this metric uses the number of floating point operations per byte accessed from main memory. OI takes into account the cache hierarchy, since a better use of cache memories would mean less use of main memory, and the memory bandwidth and speed, since its performance would affect GFLOPS. Yet, to characterise the performance, it may be insufficient, specially on NUMA systems. The RM sets system upper limits to performance, but on a NUMA system, distance and connection to memory cells from different cores may imply variations in the memory latency. This information is valuable in many cases. Variations in access time cause different GFLOPS for each core, even if each core performs the same number of operations. This way, the same code may perform differently depending on how different threads are scheduled. In these situations, OI may keep the same value, hiding the fact that poor performance is due to the memory subsystem. A programmer trying to increase the application performance would not know whether the differences in GFLOPS are due to memory access or other reasons, like power scaling or the execution of other processes in some cores. Extending the DyRM with a third dimension showing the mean latency of memory accesses for each point in the graph would clarify the source of the performance problem. We called this model 3DyRM. Some examples of this extension of the RM are shown along this paper.

A. Hardware Counter Monitoring

Modern Intel microprocessors use a set of hardware counters by a tool called *Precise Event-Based Sampling* (PEBS) to get on the fly information about a number of events related to the performance of the code in hand [20]. PEBS is an advanced sampling feature of the Intel Core-based processors in which the processor is directly recording samples into a designated memory region. Each sample contains the state of the processor at the time certain hardware counter reaches a set goal. A key advantage of PEBS is that it minimizes the overhead because the Linux kernel is only involved when the PEBS buffer fills up, i.e., there is no interruption until a number of samples are available. A constraint of PEBS is that it works only with certain events, but generic cache accesses and write operations are currently supported. Additionally, a minimum latency for load operations can be selected, so only load events which serve the data with high latencies are counted and sampled.

In modern Intel processors, starting with the Nehalem architecture, the PEBS record format includes detailed information about memory accesses. When sampling memory operations, the virtual address of the operation data is recorded. For load operations, the latency in which the data is served is also recorded (in cycles), as well as information about the memory level from where the data was actually read.

Intel PEBS captures the entire content of the core registers in a buffer each time it detects a certain number of hardware events. These registers include hardware counters, which can be measuring other events. The data capture tool uses two PEBS buffers. One of them captures floating point information each time a certain number of instructions has been executed. This number can be fixed by the user, determining the sampling rate. The other one captures the detailed information of a memory load event, including its latency, after certain number of memory load events. The user not only can select the memory sampling rate, but the minimum load latency that an event must have in order to be counted, allowing the user to focus only on the loads he is interested in.

The overhead from using PEBS comes from having to record in a buffer the state of the core each time it is sampled, with an extra cost for memory operations, due to latency and data source recording. As such, the overhead is mainly determined by the sampling rates: the higher the desired resolution, the larger the overhead. Since both memory events and floating point information must be sampled, there are two sampling rates. The 3DyRM is based on floating point performance, so each point in the model corresponds to a sampled event. As such, the more often floating point information is sampled, the more points per second the 3DyRM can be rendered. The memory latency assigned to that point in the model is given by the mean latency of memory events captured in the previous time interval. So, if the memory events are captured in a rate close to that of the floating point information, each point will have a close approximation of the latency in that time interval.

To obtain the information needed by the 3DyRM, the number of floating point operations executed by each core must be extracted. This means that at least ten different events must be considered in Intel Sandy/Ivybridge [21] processors. These events are in the set of FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE and FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE. Anyway, if no packed floating point operations are considered, only two of these events can be taken into account: FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE and FP_COMP_OPS_EXE:SSE_FP_SCALAR_SINGLE. Additionally, data traffic between main memory modules and caches have to be considered for each core. Therefore virtual addresses that produce cache misses have to be stored by using the OFFCORE_REQUEST: ALL_DATA_READ event. The sampling frequency is established through the number of instructions executed by each core. In this way, information about the number of instructions, the number of floating point operations and the number of data read is stored. This information is enough to define our model.

III. Case Studies

A. System

The experiments presented in this paper were carried out on a system with two Intel Xeon E5-2650L processors – 8 cores per processor, 16 total, 32 with Hyper-Threading – and 64 GB of RAM. Processor cores are named by the OS with numbers from 0 to 32. Each processor has a 20 MB shared L3 cache. The main memory is divided into two cells, each processor has 32 GB of memory closer to itself (its local memory) and 32 GB farther away, and closer to the other (remote memory). This cells are called `cell0`, which is made of the even number named cores and 32 GB of RAM, and `cell1`, which is made of the odd number named cores and the other 32 GB of RAM. All executions were carried out with 16 threads, not using the Hyper-Threading capability. The system runs a Linux Ubuntu 12.04 with kernel 3.10.1.

B. Routines

In our experiments, we have used two single precision Level 1 BLAS routines, SDOT and SAXPY.

- The SDOT operation computes the dot product of two real vectors in single precision: SDOT $\leftarrow x^\top \times y = \sum x(i) * y(i)$
- The SAXPY operation computes a constant *alpha* times a vector x plus a vector y. The result overwrites the initial values of vector y: SAXPY $\leftarrow y = \alpha x + y$

Both operations work with strided arrays. Two values, named $incx$ and $incy$, can be used to specify the increment between two consecutive elements of vector x and y (stride), respectively. Different strides are used to change the behaviour of the codes in terms of memory accesses.

C. Implementation

To be able to place segments of each vector in different memory cells, the `libnuma` library [22] has been used. Each vector has been divided into 16 segments, one for each execution thread, so each one can be allocated to a specific memory cell using `numa_alloc_onnode()`. Furthermore, each thread can be assigned to a specific core using `sched_setaffinity()`. This way, different configurations have been tested:

- Ideal: Each thread operates with the vector segments it needs in its local memory.
- Crossed: Each thread operates with the vector segments it needs in its remote memory.
- All in 0: All the segments are placed in `cell0`.
- All in 1: All the segments are placed in `cell1`.

To compare these configurations with more realistic scenarios, two other versions have been implemented. These versions use the standard `malloc()` routine for memory allocation, letting the OS choose the thread placement. Note that migration is allowed in these two scenarios. They are:

- Serial Initialisation (SI): One thread initialises the memory and sets the initial values to the vectors, then each parallel thread executes its own code.
- Parallel Initialisation (PI): Each thread initialises the memory and sets the initial values to each vector segment.

IV. Results

Results obtained for SAXPY and SDOT are shown in Table I and Table II, respectively. Results for different vector strides were considered, using the same stride for both vector x and y (that is $incx = incy$). All executions were made for vectors of 10^8 elements, that is, $4{*}10^8$ bytes, far larger than the size of L3 cache, and repeated 100 times. The `Time` columns show the execution time, in seconds, of the faster (`min` column) and the slower (`max` column) threads of the SAXPY or SDOT computations executed with 16 threads. This measured time is the mean of 3 executions, and initialisation time is not included. The `Latency` columns show the measured mean latency of memory accesses of more than 400 cycles, for cores in `cell0` and `cell1`. Column `RQ_DR/INST` shows the number of requests of data memory cache lines that were made for each 100 instructions retired. Codes were compiled with `gcc 4.6.3` and no optimisations (-O0).

A. Effect of the stride

In both codes, as the stride increases, fewer operations are performed. For $incx = 2$, for example, only half of the vector members take part in the operation. As such, when looking at the IDEAL times, both Table I and Table II show a halving of the execution time between $incx = 1$ and $incx = 2$. Nevertheless, from $incx = 2$ to $incx = 32$ in Table I, and from $incx = 4$ to $incx = 32$ in Table II, times remain the same. This is due to the management costs of the memory hierarchy. In the Sandy Bridge architecture the cache line size is of 64 bytes, which means it can hold 16 floats. Furthermore, the processor always reads two cache lines at once from main memory, meaning it can bring 32 floats at the same time from the cache. This means that from $incx = 2$ to $incx = 32$ the system will transfer from main memory to the cache the same amount of data, essentially the whole vectors x and y. For example, with $incx = 32$ only one float is needed per each two lines for each vector (in both SAXPY and SDOT), but the system will still move the full four cache lines, 256 bytes, although only 8 bytes are being used. So from $incx = 4$ to $incx = 32$ the codes are memory bound, their execution

TABLE I. Results for SAXPY.

		Time(seconds)		Latency(cycles)		
inc	code	min	max	cell0	cell1	RQ_DR/INST
inc 1	IDEAL	84.8	88.0	540	542	0.3
	Crossed	85.8	89.2	548	548	0.3
	ALL-IN-0	88.2	97.6	741	1080	0.31
	ALL-IN-1	87.8	97.6	1090	749	0.31
	PI	80.1	81.5	529	529	0.42
	SI	81.8	88.3	566	545	0.32
inc 2	IDEAL	46.0	47.6	713	714	0.61
	Crossed	58.0	60.1	1006	1006	0.63
	ALL-IN-0	67.9	102.6	1204	1616	0.62
	ALL-IN-1	67.9	100.5	1598	1219	0.61
	PI	44.9	47.1	752	752	0.61
	SI	49.7	55.4	784	807	0.65
inc 4	IDEAL	44.2	45.1	944	938	1.23
	Crossed	58.2	59.0	1153	1169	1.24
	ALL-IN-0	67.9	100.3	1248	1254	1.23
	ALL-IN-1	67.8	100.6	1208	1292	1.23
	PI	43.5	44.9	946	956	1.23
	SI	50.1	55.0	1027	940	1.23
inc 8	IDEAL	44.23	44.9	965	983	2.48
	Crossed	58.1	58.6	1184	1208	2.49
	ALL-IN-0	67.7	100.0	1281	1623	2.49
	ALL-IN-1	67.7	100.0	1621	1259	2.49
	PI	44.3	44.7	967	970	2.49
	SI	47.7	50.6	975	993	2.49
inc 16	IDEAL	44.4	44.7	962	956	4.87
	Crossed	58.0	58.4	1131	1156	4.84
	ALL-IN-0	67.8	99.9	1237	1532	4.86
	ALL-IN-1	64.1	99.9	1529	1270	4.89
	PI	44.3	44.8	959	970	4.82
	SI	48.3	51.9	1007	973	4.89
inc 32	IDEAL	44.1	44.6	978	987	8.84
	Crossed	49.6	50.1	975	972	9.02
	ALL-IN-0	69.0	95.3	1327	1541	9.02
	ALL-IN-1	69.0	95.6	1541	1363	9.03
	PI	44.0	44.6	975	984	8.85
	SI	46.6	48.2	952	993	8.95
inc 64	IDEAL	34.6	35.1	1140	1142	10.89
	Crossed	37.1	37.5	1150	1157	9.85
	ALL-IN-0	54.3	74.8	1596	1953	11.32
	ALL-IN-1	53.7	75.0	1905	1636	11.14
	PI	33.8	35.6	1105	1129	10.83
	SI	34.3	35.7	1214	1169	11.93
inc 128	IDEAL	10.8	10.9	866	851	5.21
	Crossed	11.2	11.4	826	832	5.22
	ALL-IN-0	17.3	23.2	1214	1447	5.37
	ALL-IN-1	17.3	23.4	1371	1205	5.39
	PI	10.3	11.3	842	861	5.19
	SI	10.8	11.1	801	929	6.07

TABLE II. RESULTS FOR SDOT.

		Time(seconds)		Latency(cycles)		
inc	code	min	max	cell0	cell1	RQ_DR/INST
inc 1	IDEAL	67.2	68.9	559	554	0.38
	Crossed	68.1	70.0	556	552	0.38
	ALL-IN-0	67.5	69.7	570	572	0.56
	ALL-IN-1	67.7	70.7	567	564	0.38
	PI	67.2	69.6	537	539	0.38
	SI	68.5	72.6	572	557	0.41
inc 2	IDEAL	34.6	35.7	585	577	0.74
	Crossed	38.0	38.9	714	714	0.75
	ALL-IN-0	40.2	61.3	782	1178	0.74
	ALL-IN-1	40.2	61.8	1172	786	0.74
	PI	34.4	35.1	561	563	0.74
	SI	36.6	41.4	630	614	0.77
inc 4	IDEAL	26.4	26.8	711	718	1.48
	Crossed	36.8	37.2	889	893	1.48
	ALL-IN-0	39.6	61.5	924	1255	1.48
	ALL-IN-1	39.5	61.9	1237	925	1.48
	PI	26.2	27.1	713	715	1.48
	SI	30.5	34.9	728	740	1.54
inc 8	IDEAL	26.4	26.6	760	761	3.02
	Crossed	36.8	37.4	936	938	3.01
	ALL-IN-0	39.5	61.5	968	1306	3
	ALL-IN-1	39.5	61.4	1274	965	3
	PI	26.3	26.9	757	763	2.97
	SI	29.9	33.7	810	831	3.04
inc 16	IDEAL	26.4	26.5	791	795	5.72
	Crossed	36.8	37.1	1010	1001	5.72
	ALL-IN-0	39.5	61.4	1027	1346	5.66
	ALL-IN-1	39.5	61.5	1333	1021	5.66
	PI	26.3	26.5	795	788	5.97
	SI	29.7	33.5	852	863	6.13
inc 32	IDEAL	26.3	26.5	720	729	10.92
	Crossed	36.8	37.0	962	969	10.99
	ALL-IN-0	39.9	61.2	966	1333	10.96
	ALL-IN-1	39.7	61.4	1338	962	10.97
	PI	26.1	27.1	737	737	10.92
	SI	31.3	37.8	807	806	10.91
inc 64	IDEAL	25.4	26.1	1048	1028	12.25
	Crossed	28.5	29.0	1138	1161	11.23
	ALL-IN-0	38.8	56.00	1478	1966	12.92
	ALL-IN-1	38.8	56.1	1948	1484	12.84
	PI	25.1	26.3	1078	1031	12.2
	SI	27.8	31.9	1169	1116	13.28
inc 128	IDEAL	6.6	6.7	653	648	6.13
	Crossed	6.8	6.9	680	687	6.09
	ALL-IN-0	10.3	14.3	897	1129	6.25
	ALL-IN-1	10.2	14.3	1158	879	6.22
	PI	6.3	6.6	663	649	6.14
	SI	6.5	14.2	854	998	6.74

time is limited by the memory access, and they do not gain from executing fewer operations. In SAXPY this happens at $incx = 2$ due to the store operations.

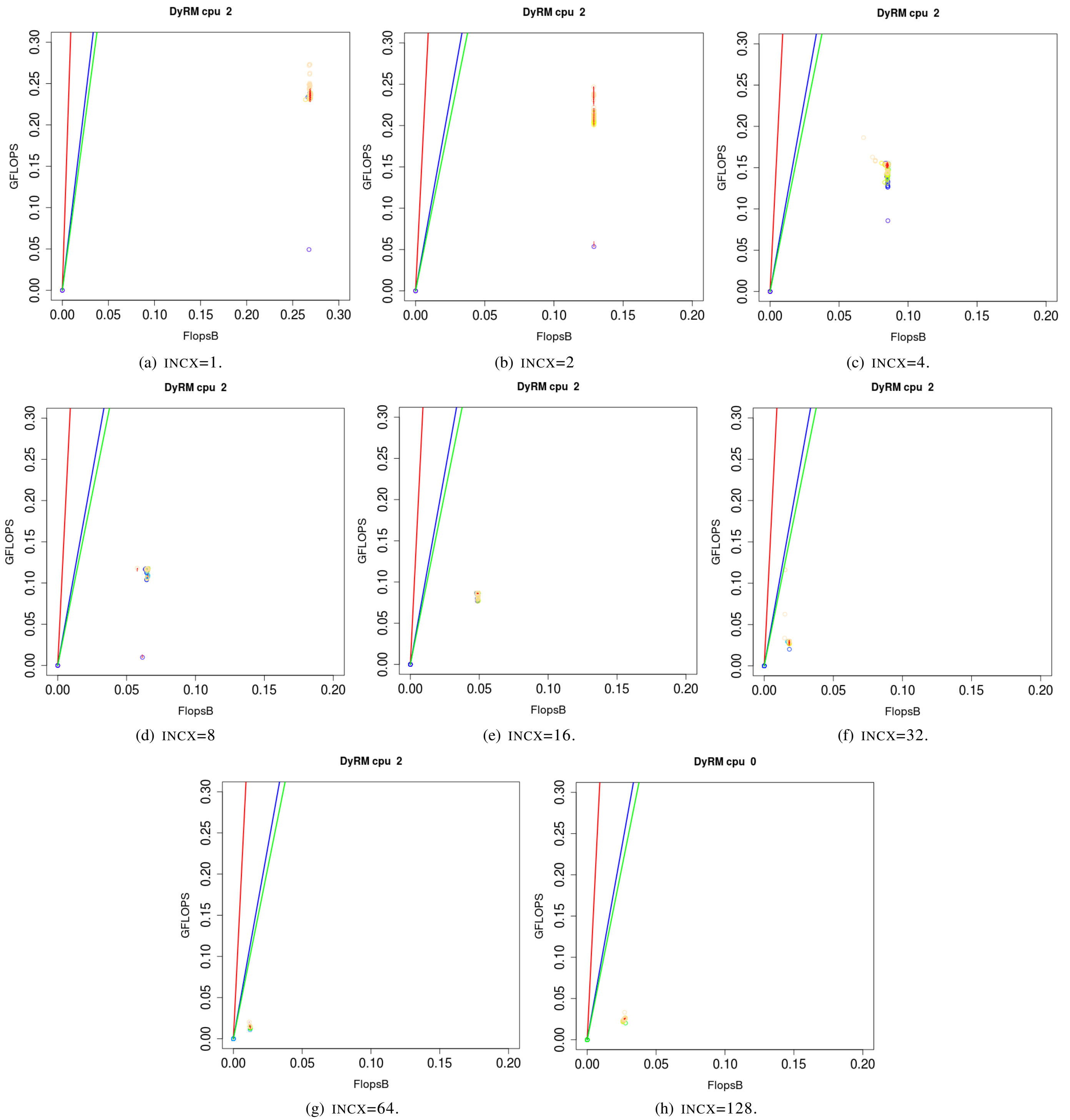
Column `RQ_DR/INST` in Tables I and II shows how as stride doubles from 1 to 32 the RQ_DR/INST ratio also doubles, since the same number of cache lines are requested, but half the instructions are executed. Memory latency also increases with the stride. This is due to the fact that, while only latencies larger than 400 cycles are measured, they can be either from main memory or from the cache. Cache loads are usually faster, and with small strides they move the mean latency closer to 400. With larger strides, memory loads make a larger share of the accesses detected, and the latency increases. This effect shows how latency can be used as a proxy of the cache hierarchy behaviour.

In Figure 2, the evolution of the GFLOPS and the OI can be seen for the SAXPY IDEAL configuration as the stride changes. It is clear that OI decreases as fewer operations are made while accessing the same memory, and GFLOPS decreases as fewer operations are performed in the same time. Figure 2(h) shows that, with $incx = 128$, this behaviour is broken, the entire vector no longer needs to be accessed, and then the OI and GFLOPS increase. Results for SDOT are similar and are not shown in this paper.

B. *Effect of the thread placement*

In the IDEAL configuration each thread is using the memory module closer to itself, which should be the best case for memory access and should present lower latencies. Tables I and II show that this is the case. In the CROSSED configuration each thread is using the memory opposite to itself, and results show higher memory latencies. As expected, the ALL-IN-0 and ALL-IN-1 configurations show the worst results. This is because all threads access the same memory cell, which produces bus conflicts and saturation. In the CROSSED configuration data has to travel more to reach its destination, but read conflicts are similar to the IDEAL configuration. In the ALL-IN configurations, the cell where the data is stored shows better behaviour than its opposite, but the overall performance is diminished. In fact, for ALL-IN configurations, threads in the same cell as the data finish their execution in the order of the minimum time, while threads in the opposite cell take a time in the order of the maximum. This states the importance of balancing the memory use.

In Figure 3 the effects of the memory imbalance are shown for SAXPY ALL-IN-1. Figures 3(a) and 3(b) show two views of the 3DyRM with data taken from all the cores (each point corresponds to one measurement in one core), `cell0` in black and `cell1` in green. In Figure 3(b) it is clear that the access to data from `cell0` results in a larger latency. This figure shows a problem with the floating point operations (FP_OPs) hardware monitoring in the Intel architectures. In the Intel Sandy Bridge architecture (and following Ivy Bridge), floating point operations counters count executed operations, not retired operations [23]. As a consequence, if a FP_OP is issued, but its operands are not in the cache or registers, it is counted as it was executed, and will be reissued until its operands appear in the cache. This means in cases like these, where main memory is accessed so aggressively, floating point operations can be



(a) INCX=1.

(b) INCX=2

(c) INCX=4.

(d) INCX=8

(e) INCX=16.

(f) INCX=32.

(g) INCX=64.

(h) INCX=128.

Fig. 2. DyRM for SAXPY IDEAL, different strides.

counted in excess, and hardware counters may not be accurate. In the case shown in Figure 3, the higher memory latency of `cell0` means its floating point operations are reissued more times than the ones in `cell1`, such producing an overcount. This makes the OI and GFLOPS counts to increase in relation with `cell1` (see Figure 3(a)).

In Figures 3(c) and 3(d) the DyRM of two different cores, one in `cell0` (Figure 3(c)) and other in `cell1` (Figure 3(d)), are shown. Since they are executing the same operation, their OI should be the same, but Figure 3(c) is displaced to the right due to the above mentioned overcount. When threads in `cell1` finish their execution, threads in `cell0` are still running. Since these threads no longer compete for the memory access with the ones in the opposite cell, their memory latency decreases and can achieve a better performance. This also means than there is a lower overcount in floating point operations. Figure 3(c) shows this, with two different execution phases, the latter with larger GFLOPS and lower OI (see Figure 1 for information on the colour gradient).

In Figure 4 a comparison between the IDEAL and the CROSSED configurations of SDOT with $incx = 8$ is shown. As in the former case, the flop overcount is clearly affecting the measurement. Threads in SDOT CROSSED with $inc = 8$ take about 37 seconds to compute, and, in SDOT IDEAL, about 26 (see Table II). Also, since they compute the same code, their OI should be almost the same. Yet, Figure 4(a), for the IDEAL case, shows lower GFLOPS and OI than Figure 4(b), for the CROSSED case, when the opposite effect should be seen. This is due to the longer latency of memory accesses of the CROSSED configuration, around 936 cycles, compared to the IDEAL one, around 760 (see Table II). This difference means floating point operations in the CROSSED case are reissued more times until their data arrives to the cache, and thus the overcount is larger. Figure 4(d) shows the higher latency detected for CROSSED compared to Figure 4(c) for IDEAL.

These two last cases show how the hardware counters on Intel Sandy Bridge are not precise enough for floating point operations counting in some cases, and how, by measuring memory latency, these cases may be identified.

C. *Effect of the OS behaviour*

Configurations PI and SI correspond to more typical usage cases. Tables I and II show that PI times and latencies are very similar to the ones of the IDEAL configuration. This is because, when each thread in PI initialises its vectors, data are stored in their cell. If there are no thread migrations during the execution, this situation is almost identical to the IDEAL case. In Figure 5 a comparison between the IDEAL and PI configurations of SDOT is shown. Only the behaviour of two cores (equivalent to the behaviour of two threads, since there are no migrations in this case) is shown, since all cores present broadly the same behaviour.

The SI configuration is a more realistic one. A single thread initialising data in a program is common. Plus, same data can be used by different threads during the execution of a program, which means they access different memory cells at different times. Tables I and II show that this configuration falls between IDEAL and CROSSED in terms of performance, but does not behave as badly as ALL-IN. This indicates that the system balances data storage between the two memory cells, and may explain the behaviour observed in Figure 6. In this figure, the execution of SDOT SI with stride 8 is shown. Figure 6(a) shows an example in which the initialising thread was executed on core 4, and did not migrate before the proper execution of SDOT, sightly after $TIME = 1 * 100ns$. Figure 6(b) shows how a compute thread starts its execution at that time, and ends its execution before the end of the program. Figure 6(c) shows the added instruction count for the entire program during its execution, including contributions for all threads. Four distinct slopes can be observed. The first corresponds to the initialisation stage, and the other three with the computation of SDOT. The third slope takes most of the computation, and the fourth one corresponds to the situation in which different threads finish their execution at different times, as such fewer instructions are executed. The second slope indicates a warm-up period during the execution of SDOT. Results are similar for SAXPY.

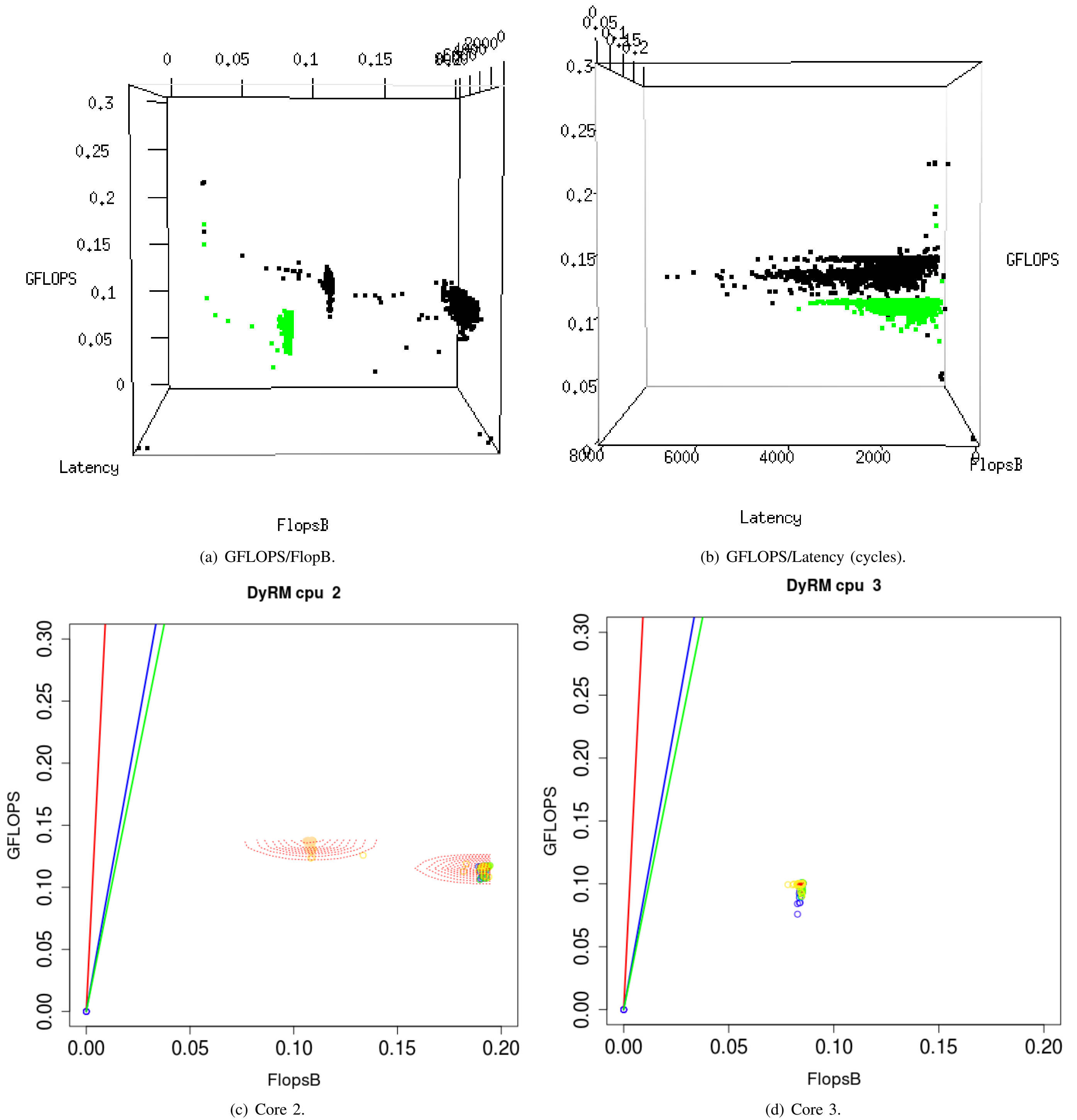
Figures 7(a) and 7(b) show the behaviour of two cores, one in each cell (results are similar for the other cores in the same cell as shown in Figure 7(c)). A warm-up phase is detected for each cell (shown in blue). Latency information (Figure 7(d)) indicates that data are in `cell0` (since cores in the other cell take longer to access memory) as expected when the initialising thread belongs to that cell. Nevertheless, data seems to be balanced between memories, which may be done by the OS during that phase. In any case, results are not as good as in the best case scenario.

V. Conclusion

Modern multicore systems present complex memory hierarchies, and make data locality and thread affinity to be important issues for obtaining good performance. In this work, extensions of the Roofline Model are used to characterise graphically the behaviour of the executions of two simple kernels in terms of data locality and thread affinity. To implement these models, advantage of the PEBS hardware counters of Intel processors was taken, allowing to gather useful information with low overhead.

Analysis of the SDOT and SAXPY routines were performed with different strides to modify their memory accesses locality, and also considering different strategies to allocate vectors in memory modules and threads to modify their affinity properties. Thread migration scenarios were also considered in the experimental study.

The results of the experiments show that the extensions of the Roofline Model, with latency and dynamic information, are useful to understand the behaviour of the execution of parallel codes in multicore systems, including the effects of data accesses locality and thread affinity. Results show that imprecisions in the Intel Sandy Bridge HC may distort measurements, and using the 3DyRM can be identified.



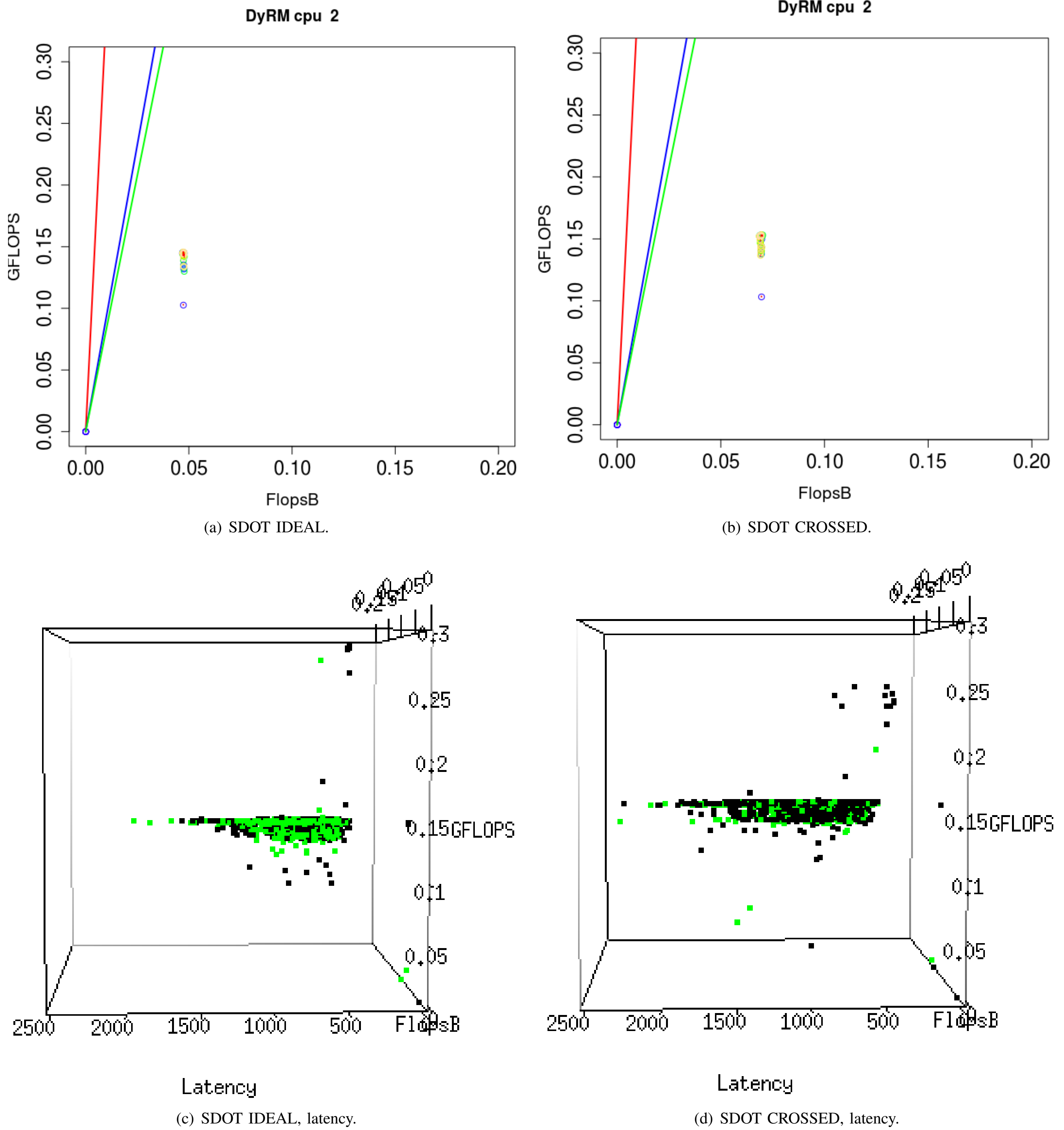
(a) GFLOPS/FlopB.

(b) GFLOPS/Latency (cycles).

(c) Core 2.

(d) Core 3.

Fig. 3. DyRM and 3DyRM of SAXPY ALL-IN-1 with incx=8 in two cores. Effect of flops overcount. Cell 0 in black, Cell 1 in green.



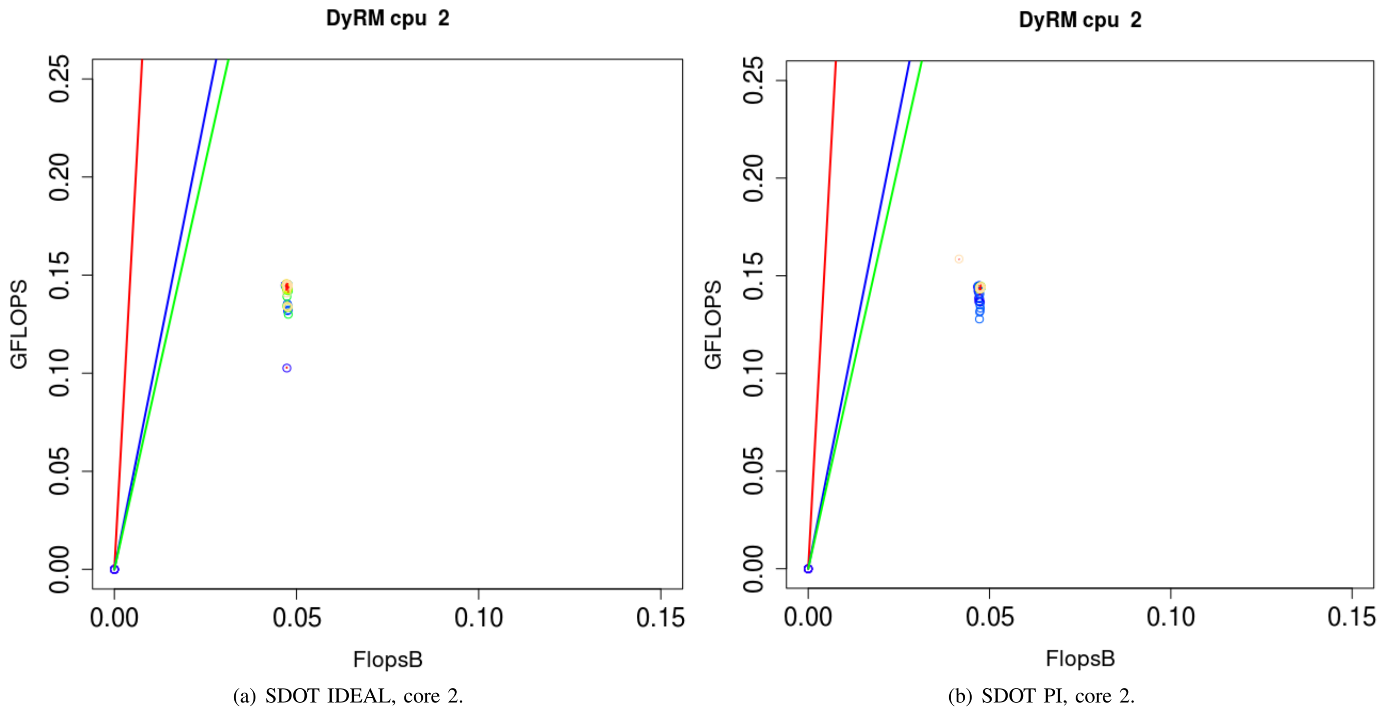
(a) SDOT IDEAL.

(b) SDOT CROSSED.

(c) SDOT IDEAL, latency.

(d) SDOT CROSSED, latency.

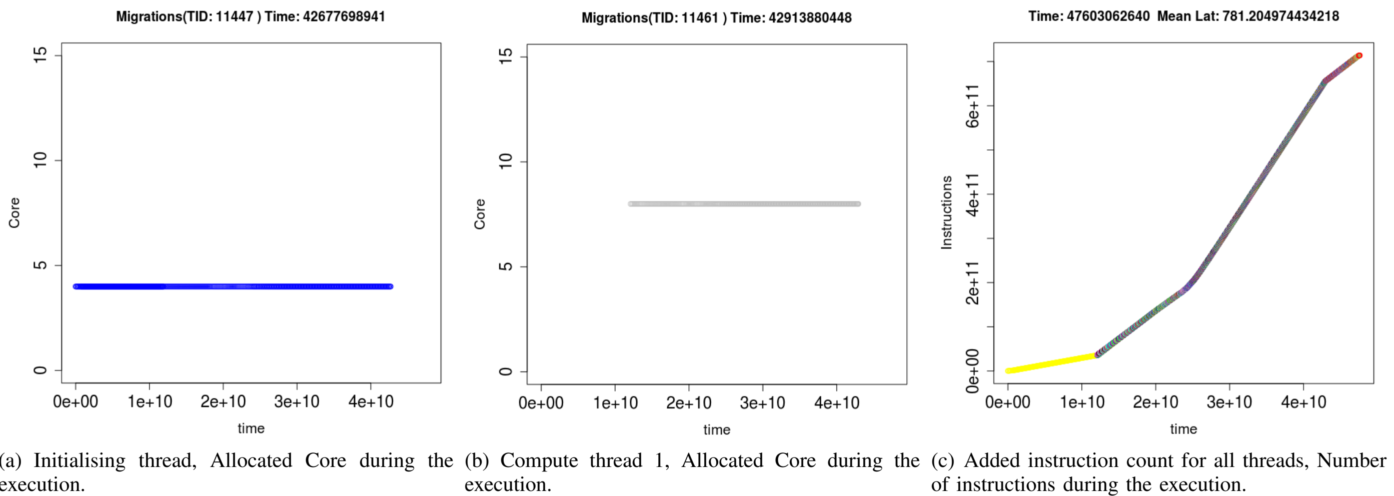
Fig. 4. DyRM and 3DyRM of SDOT with incx=8. Effect of flops overcount. Cell 0 in black, Cell 1 in green.



(a) SDOT IDEAL, core 2.

(b) SDOT PI, core 2.

Fig. 5. Good thread placement, SDOT IDEAL and SDOT PI.

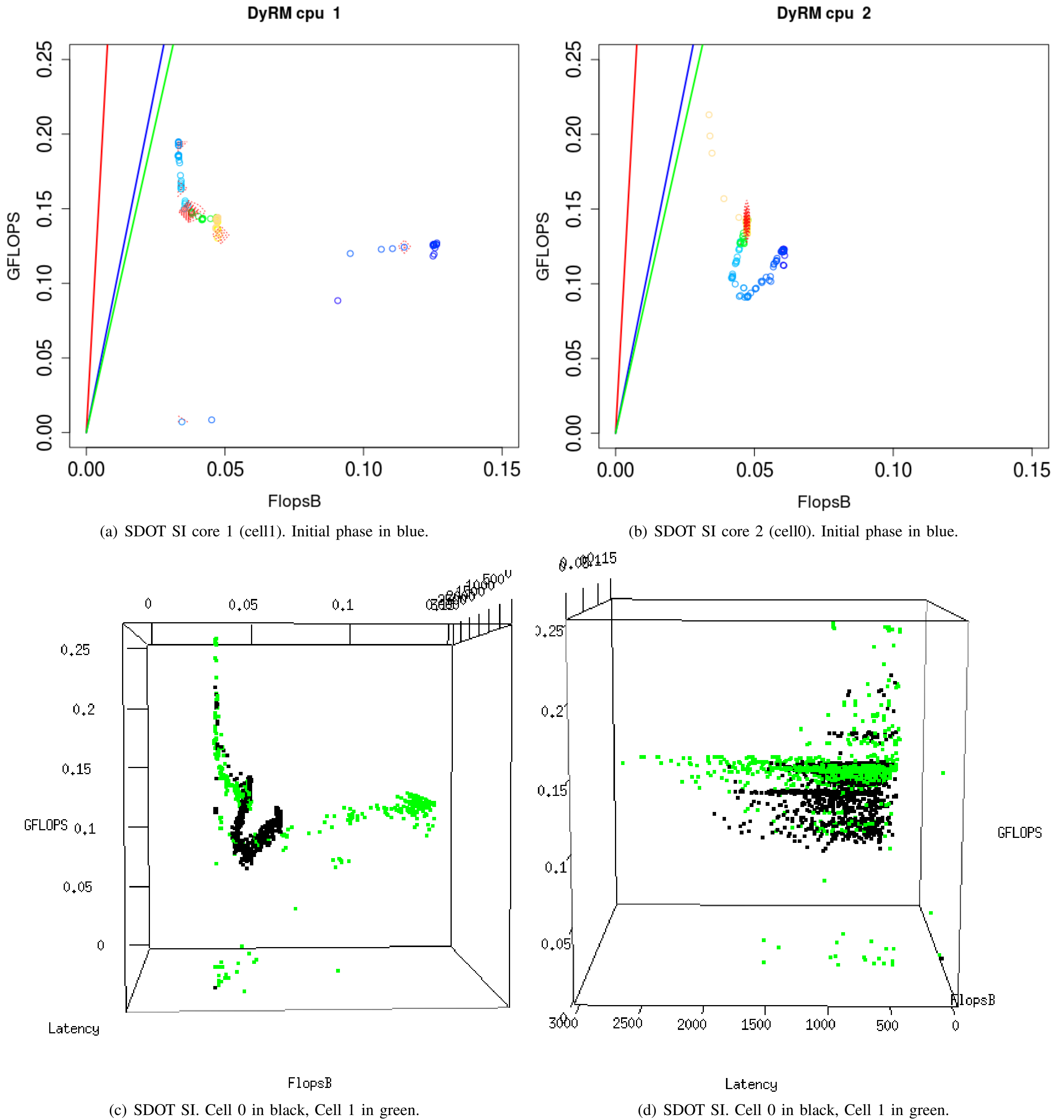


(a) Initialising thread, Allocated Core during the execution.

(b) Compute thread 1, Allocated Core during the execution.

(c) Added instruction count for all threads, Number of instructions during the execution.

Fig. 6. Migrations in the initialisation thread in SAXPY SI.



(a) SDOT SI core 1 (cell1). Initial phase in blue.

(b) SDOT SI core 2 (cell0). Initial phase in blue.

(c) SDOT SI. Cell 0 in black, Cell 1 in green.

(d) SDOT SI. Cell 0 in black, Cell 1 in green.

Fig. 7. Different behaviours for the two processors, initialisation done by core 4. SDOT SI.

Acknowledgments

This work has been partially supported by the Ministry of Education and Science of Spain, FEDER funds under contract TIN 2010-17541, and Xunta de Galicia, EM2013/041. It has been developed in the framework of the European network HiPEAC and the Spanish network CAPAP-H.

References

[1] A. Sodan, "Message-passing and shared-data programming models: Wish vs. reality," in *Proc. IEEE Int. Symp. High Performance Computing Systems Applications*, 2005, pp. 131–139.

[2] R. Hazara, "The explosion of petascale in the race to exascale," in *ACM/IEEE conference on Supercomputing*, 2012.

[3] S. Devadas, "Toward a coherence multicore memory model," *IEEE Computer*, vol. 46, no. 10, pp. 30–31, 2013.

[4] S. Moore, D. Cronk, K. London, and J. Dongarra, "Review of performance analysis tools for MPI parallel programs," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2001, pp. 241–248.

[5] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[6] A. Morris, W. Spear, A. D. Malony, and S. Shende, "Observing performance dynamics using parallel profile snapshots," in *Euro-Par 2008–Parallel Processing*. Springer, 2008, pp. 162–171.

[7] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[8] A. Cheung and S. Madden, "Performance profiling with EndoScope, an acquisitional software monitoring framework," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 42–53, 2008.

[9] B. Mohr, A. D. Malony, H. C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah, "A performance monitoring interface for OpenMP," in *Proceedings of the Fourth Workshop on OpenMP (EWOMP 2002)*, 2002.

[10] M. Schulz and B. R. de Supinski, "PN MPI tools: A whole lot greater than the sum of their parts," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 2007.

[11] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[12] M. Schuchhardt, A. Das, N. Hardavellas, G. Memik, and A. Choudhary, "The impact of dynamic directories on multicore interconnects," *IEEE Computer*, vol. 46, no. 10, pp. 32–39, 2013.

[13] K. Fürlinger, C. Klausecker, and D. Kranzlmüller, "Towards energy efficient parallel computing on consumer electronic devices," in *Information and Communication on Technology for the Fight against Global Warming*. Springer, 2011, pp. 1–9.

[14] H. Servat, G. Llort, J. Giménez, K. Huck, and J. Labarta, "Folding: detailed analysis with coarse sampling," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 105–118.

[15] O. G. Lorenzo, J. A. Lorenzo, J. C. Cabaleiro, D. B. Heras, M. Suarez, and J. C. Pichel, "A study of memory access patterns in irregular parallel codes using hardware counter-based tools," in *Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2011, pp. 920–923.

[16] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec, "Performance implications of single thread migration on a chip multicore," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 80–91, 2005.

[17] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera, "DyRM: A dynamic roofline model based on runtime information," in *2013 International Conference on Computational and Mathematical Methods in Science and Engineering,*, 2013, pp. 965–967.

[18] O. G. Lorenzo, T. F. Pena, J. C. Pichel, J. C. Cabaleiro, and F. F. Rivera, "3DyRM: A dynamic roofline model including memory latency information," *Journal of Supercomputing*, 2014, to appear.

[19] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, 2013.

[20] perfmon2. (2013, Jun.) Precise Event-Based Sampling (PEBS). [Online]. Available: http://perfmon2.sourceforge.net/pfmon_intel_core.html#pebs

[21] Intel. (2012, Jun.) Intel®64 and IA-32 architectures software developer's manual volume 3B: System programming guide, part 2. [Online]. Available: http://download.intel.com/products/processor/manual/253669.pdf

[22] A. Kleen, "A NUMA API for Linux," *Novel Inc*, 2005.

[23] Intel Developer Zone, "Fluctuating FLOP count on Sandy Bridge," http://software.intel.com/en-us/forums/topic/375320, 2014, [Online; accessed 5-February-2014].