Interference Control by Best-Effort Process Duty-Cycling in Chip Multi-Processor Systems for Real-Time Medical Image Processing

Mark Westmijze, Marco J. G. Bekooij and Gerard J. M. Smit
University of Twente, Department of EEMCS
Enschede, The Netherlands
{m.westmijze, m.j.g.bekooij, g.j.m.smit}@utwente.nl

***Abstract*—Systems with chip multi-processors are currently used for several applications that have real-time requirements. In chip multi-processor architectures, many hardware resources such as parts of the cache hierarchy are shared between cores and by using such resources, applications can significantly interfere with each other. In previous work, we showed that a single X-ray imaging streaming applications can be executed with low jitter on such systems. However, it was assumed that only one application would be running on the system, which prevents system integration where multiple real-time and best-effort applications are executing on a single chip multi-processor. In this paper, we address the limited bandwidth in the cache hierarchy, which can cause threads to interfere with each other significantly. We propose a technique that implements cache bandwidth reservation in software, by dynamically duty-cycling best-effort applications, based on their cache bandwidth usages using processor performance counters in order to control the influence of best-effort applications on real-time applications. With this technique we can control the latency increase of real-time applications that is caused by best-effort application in order to satisfy real-time requirements with a minimal reduction in best-effort performance. The results of the experiments with real-life applications indicate that we can control the increase of the latency to such an extent that we can almost completely eliminate the influence of bandwidth sharing in the cache at the cost of best-effort performance.**

Keywords-cache; bandwidth; real-time; control; jitter

I. Introduction

Commercial-off-the-shelf (COTS) systems with general purpose processors (GPPs) have become so powerful that they can be used for applications that could previously only be performed on hardware such as digital signal processors, field-programmable gate arrays (FPGAs) or by application specific integrated circuits (ASICs). However, the techniques that are employed by modern GPPs to reach their performance tend to neglect the worst case performance in order to improve the best-case and average-case performance. This seemingly does not make them a favorable target for real-time applications. A lot of work in the real-time community has been focused on the definition of architectures on which real-time applications can run predictably and modeling techniques that determine the worst case behavior of real-time applications. Applying these modeling techniques on high performance GPPs often results in loose throughput and latency bounds as a result of the GPPs and chip multi-processor (CMP) architecture. This leads to over-provisioned systems, which cannot be economically justified for most soft real-time applications.

We studied a streaming X-ray application on a CMP architecture in a previous paper [1]. In this paper, we provided evidence that it is feasible to run streaming applications in such a way that drastically reduces the occurrences of worst case behavior. The jitter | variation of latency | of the executed application is sufficiently small to satisfy the soft real-time requirements of the X-ray application. However, one important assumption was that the system was only used for the X-ray application. A consequence is that a separate system is required for additional applications while the system that runs the X-ray application has spare processing power most of the time. We therefore study whether we can relax that assumption, and run some best-effort application along with our real-time streaming application.

The problem that must be addressed is that threads that execute concurrently on different cores on a CMP architecture can degrade each others performance significantly. There are several components in a CMP that allow cores to degrade each others performance such as the cache hierarchy, memory controller and central processing unit (CPU) interconnect. In this paper, we focus on space contention and bandwidth congestion within the cache hierarchy, study how this enables applications to degrade each others performance and propose and evaluate a technique to control the degradation.

The paper is structured as follows. First, we introduce the case study | an interventional X-ray imaging system | in Section II and in Section III we briefly give an overview of the CMP architecture that we use. Related work is described in Section IV. This allows us to give a detailed explanation of why we study the space contention and bandwidth congestion in CMP architectures in Section VI and Section V. Section VII explains how we implemented a technique that reduces the bandwidth congestion and thereby controls the interference that is caused by the best-effort application. The experiments that we used for the evaluation can be found in Section VIII and the results of those experiments in Section IX. A few improvements of the control technique are described in Section X as future work. We summarize the conclusions in Section XI. Finally, we give our acknowledgments in Section XII.

II. X-RAY SYSTEM

In the X-ray system there are real-time and best-effort applications that currently run on multiple COTS systems. The real-time applications are streaming applications where images are typically processed at frame rates between 15 and 60 frames per second. For precise hand-eye coordination and to prevent fatigue, the processed X-ray images should preferably have a low latency (e.g., 120 ms) and the jitter on the latency should be low (e.g., âĽď 16 ms). These requirements enable the physician to smoothly control his equipment. Practical experience has shown that the occasional deviations from the requirements are small enough that they do not affect the patient's safety. Due to the nature of these requirements we can use modeling techniques and architectures that work most of time instead of strictly all the time. Examples of best-effort applications are the graphical user interface (GUI), the storage of processed images and the retrieval and analysis of stored images.

Our aim is to run the real-time and best-effort applications on a single system while controlling the progress of the best-effort application such that real-time requirements are satisfied and the performance of the best-effort application is maximized.

III. ARCHITECTURE

As mentioned in Section II, we consider a CMP architecture. More specifically, we focus on the Intel Nehalem architecture. In Fig. 1 a schematical overview is given for such a CMP in a dual socket system. Each chip consists of a number of cores that all have simultaneous multi-threading (SMT) capability. An inclusive cache hierarchy with three levels is used where the first and second level are local for a physical core and the third level is shared between all cores. Furthermore, a memory controller and a high speed point-to-point interface (i.e. Quick Path Interconnect (QPI)) are integrated into the chip, which improves scalability in a multi-socket system.

In this paper, we only examine the case where threads of multiple applications are mapped onto a disjoint subsets of physical cores (e.g., the real-time threads on core 0 and 1 and a best-effort application on core 2 and 3 with SMT disabled). The first hardware component for which the threads have to contend is the bandwidth to and the space in the third level of the cache. Other components in the system may also be shared, but in this paper we focus on how the cache influences the performance of multiple concurrently executing threads.

IV. RELATED WORK

In many papers cache partitioning techniques have been proposed to improve the performance and predictability of the cache. Cache partitioning splits the cache in disjoint sets that do not share cache lines in the cache hierarchy. This prevents cache thrashing between threads. Stone et al. presented and used this technique in order to minimize the miss rate of each application [2]. Chiou *et al.* presented a technique for cache partitioning based on columnization [3]. Iyer studied cache partitioning in order to improve the Quality of Service (QoS) [4]. Kannan *et al.* studied how dynamic cache partitioning can be added to an operating system in order to improve the QoS for one or more applications and propose a methodology that can dynamically alter the cache partitioning at run-time [5]. Kim *et al.* also studied cache partitioning in a CMP in [6] and optimize the cache partitioning for fairness (i.e. an application with a small amount of cache accesses is penalized less compared to an application with a lot of cache accesses).

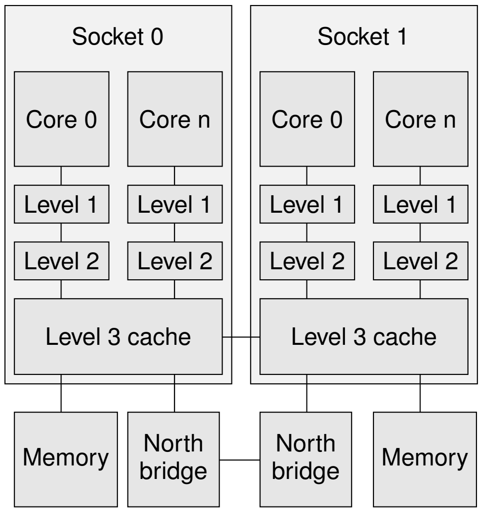

Figure 1. System architecture

Cache partitioning is a form of performance isolation that at system level is also studied. Verghese *et al.* studied the performance isolation in CMP and focus on isolation in components such as the memory controller and disk access [7]. Nesbit *et al.* introduced Virtual Private Machines (VPMs) in [8], an abstraction supported by hardware modifications in the architecture of a CMP to enable applications to reserve a certain amount of hardware resources. In the cache hierarchy space and bandwidth can be reserved in order to reduce the interference between applications. In [9], Hansson *et al.* introduced an embedded architecture that also implements performance isolation (on clock cycle level), hardware arbitration, and hard real-time scheduling techniques in order to present a virtual platform that provides complete isolation for different applications.

However, to the best of our knowledge there is no prior work that addresses the interference that is caused by bandwidth sharing in COTS CMP architectures.

V. MOTIVATION

In previous work [1], we have shown that under certain conditions COTS can be used for real-time image processing applications. The conditions were:

- **Soft real-time** Because of the complexity of modern high-performance CMP architectures (e.g., Intel Nehalem) it can be hard or even impossible to derive tight worst case execution times (WCETs). Therefore we do not formally derive the WCET, but we validate that the soft real-time constraints are met by measuring the actual jitter of the system.

- **Static streaming** The application that we examine executes static algorithms. Additionally, the application is streaming, which means that the same algorithms are executed repeatedly on new data. This allows us to map and order the algorithms and allocate the used memory of the application in such a way that less cache thrashing occurs, which is one of the main causes of jitter in CMP architectures.
- **Single application** On a system with only one application there is no contention on hardware resources with other applications.

In this paper, we want to relax the third condition in order to facilitate the execution of additional best-effort applications during the execution of the real-time applications. When two applications share hardware resources | which is the case in COTS | they can influence the execution time of each other. As mentioned before we focus on the allocation and bandwidth sharing in the cache and how that influences the interference between a real-time streaming application and other best-effort applications.

VI. Cache Partitioning

In Section IV we have mentioned that many authors have studied the partitioning of the cache as a means of performance isolation. With cache partitioning, threads do not longer evict cache lines from other threads (cache thrashing), ensuring that data remains in the cache and does not have to be retrieved from main memory. However, this technique only guarantees that other threads do not evict cache lines owned by a thread, not that the latency to retrieve that data is the same, because bandwidth to the third level of the cache is shared. Hence cache partitioning can be used to reduce some of the latency that is introduced by hardware sharing. However, it will not completely remove it. The contention can only be removed when the bandwidth to the shared cache is controlled. Which means there are two orthogonal problems: memory allocation in the cache (solved by partitioning and not studied in this paper) and bandwidth sharing to the shared level in the cache (studied in this paper).

Molka *et al.* [10] performed detailed benchmarking on the Nehalem architecture in order to determine the bandwidths between the core and different levels of the cache hierarchy. We extended those benchmarks so that we can determine how much bandwidth a single application can consume on the connection between the second and the shared third level of the cache. The most important observation from those experiments was that a single core is able to consume a significant portion of the cache bandwidth when it is writing (e.g., 15 GB/s write bandwidth for a single core that only writes where the maximal aggregated write bandwidth is 21.9 GB/s when three cores are writing at 7.3 GB/s). Application that only performs reads will only claim up to 26% of the available bandwidth on a Quad core (19.9 GB/s read bandwidth for a single core where the maximal aggregated read bandwidth is 78.8 GB/s).

With the following experiment we want to demonstrate that this is not the results of cache thrashing, but due to the bandwidth congestion. We recreated the experiments that were performed by Molka *et al* [10] with different memory configuration, namely:

A	The accessed data does not fit in the cache and the cache is not partitioned. The main memory will therefore be accessed and cores will thrash the cache.
B	The accessed data does not fit in the cache and the cache is partitioned. The main memory will be accessed, but cores will only evict their own cache lines.
C	The accessed data of the combined cores does fit in the cache and the cache is not partitioned. The main memory should only be accessed when cache thrashing occurs.
D	The accessed data does fit in the allocated cache partition. Main memory should not be accessed and cache thrashing does not occur between the cores.
E	Only the accessed data of the measured cores does fit into its allocated cache partition and the data of the other cores does not fit into the cache and they access data in one combined cache partition.

Furthermore, we run these five configuration in three different scenarios. In the three scenarios we either perform only data

TABLE I
PARTITIONING EFFECTS IN GB/S | A) DOES NOT FIT IN CACHE, NOT PARTITIONED. B) DOES NOT FIT IN CACHE, PARTITIONED. C) FITS IN CACHE, NOT PARTITIONED. D) FITS IN CACHE, PARTITIONED. E) ONLY MEASURED THREAD FITS IN CACHE, PARTITIONED

	Bandwidth (GB/s)														
	Read					Write					Mixed read (50%) and write (50%)				
Cores	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	11.5	11.5	19.9	19.9	19.8	6.3	6.3	15.0	15.0	15.1	7.1	7.1	18.2	18.2	17.5
2	6.9	7.9	19.8	19.8	16.0	3.1	3.9	10.5	10.6	8.6	4.8	4.9	16.4	16.5	12.8
3	7.0	5.6	19.7	19.8	10.4	2.7	2.5	7.3	7.3	5.1	3.3	3.3	11.5	11.5	7.4
4	4.8	4.4	17.2	19.7	8.0	2.6	1.9	5.4	5.4	4.2	3.7	2.6	8.7	8.8	5.3

read, data writes or a mix of data reads and writes. The results of this experiment can be found in Table I. From these experiments we make the following observations:

- When all the data fits in the cache and the cores are only reading (configuration C and D in the read scenario, i.e., columns 3 and 4) the read bandwidth per core is almost constant, regardless of the number of cores that are running. Only when all the cores are reading and the cache is unpartitioned we see a slightly lower read bandwidth (i.e., 17.2 GB/s for four partitioned cores versus 19.8 GB/s on average for the other cases).
- When the data does not fit in the cache (configuration A and B in all scenarios, i.e., columns 1, 2, 6, 7, 11 and 12) the read and write bandwidths are significantly degraded when more cores are running. This is caused by the limited bandwidth to main memory.
- When the data fits in the cache and the cores are writing (configuration C and D in the write scenario, i.e., columns 8 and 9) the aggregated bandwidth of the system is maximal 21.9 GB/s (7.3 GB/s per core on 3 cores) while a single thread can consume 15 GB/s essentially consuming 68% of the available write bandwidth.

Therefore we conclude that most performance degradation can be caused by the limited bandwidth, and in particular by the limited write bandwidth. Furthermore, cache partitioning does not necessarily reduce the interference, it may even degrade the performance in the situation where the performance of a memory intensive application is degraded more by the reduced cache space than by the cache thrashing of other applications.

VII. Cache Bandwidth

In order to reduce the interference of the best-effort applications on the real-time applications to an acceptable level, we have to be able to determine when this occurs. We first determine current bandwidth usage, and thereafter use this information to reduce the bandwidth of the best-effort application.

A. Bandwidth usage estimation

The system has to detect when a core uses more bandwidth than allocated. To measure this we selected two performance counters in the Nehalem architecture:

- LEVEL_2_TRANSACTIONS.LOAD: Each load transaction is counted. However, read transaction take less bandwidth than write bandwidth.
- LEVEL_2_LINES_OUT.ANY: Counts the number of lines that are written back to the third level of the cache.

The LEVEL_2_TRANSACTIONS.LOAD performance events could accurately estimate the read bandwidths as found in Table I. The LEVEL_2_LINES_OUT.ANY performance can be used to determine the total (read + write) bandwidth. We could not precisely estimating the write bandwidth in all scenarios with only one performance counter, but were able to roughly derive this value based on the total bandwidth and the read bandwidth. These counters can be used during the execution of the best-effort application to determine the currently used bandwidth and during design time to determine the worst case bandwidth usage of real-time and best-effort applications.

B. Bandwidth arbitration

There is no hardware mechanism that can allocate bandwidth within the cache hierarchy of the Nehalem architecture to specific cores. We therefore implemented a mechanism in software that can be used to achieve the same effect, but on a coarser time scale. Our mechanism to reduce the bandwidth on a coarser time scale is to repeatedly suspend the applications for a certain time on a core that uses more bandwidth than allocated, which we will now refer to as duty-cycling. This mechanism results in a much lower (average) bandwidth and reduced performance for the applications on the core that is temporarily suspended. With this technique it is not possible to completely remove the interference because the duty-cycling of cores manages the bandwidth on a coarse level and only reacts after a best-effort application has consumed too much bandwidth.

C. Duty-cycling

When the estimated read or write bandwidth of a core that is running best-effort applications reaches a certain threshold we assume that this core is degrading the performance of the real-time application and that the core that runs that offending best-effort application must be duty-cycled. We implemented this by running a thread, with real-time scheduling priorities, at a higher priority than the best-effort applications in order to temporarily suspend any best-effort application that might consume too much bandwidth. A Linux kernel with real-time patches applied allowed us to precisely sample the performance counters at regular intervals (e.g., 100 μs) and duty-cycle the best-effort applications accordingly.

D. Threshold and suspension time

The threshold for the read and write bandwidths depends on three things: available bandwidth of the specific processor, consumed bandwidth of the real-time applications and timing constraints of the real-time application. The available bandwidth of the processor in combination with the consumed bandwidths of the real-time applications determine the available bandwidth for the best-effort applications. The timing constraints (e.g., latency) of the real-time application in combination with the unobstructed performance of the real-time application determine how much performance degradation (e.g., latency increase) the real-time application can handle before a timing constraint is violated. The timing constraints and the performance of the real-time application therefore determine how long a best-effort application should be suspended in relation to the sample interval between the measurements of the performance counters.

VIII. Experiments

In this section we describe the experiments that we performed to evaluate the proposed technique. First we describe the system that we used for the experiments and then we introduce the different experiments.

A. Experimental setup

We used a Linux kernel with real-time patches applied. For each core that would be monitored and duty-cycled, a thread with real-time priorities was instantiated and mapped to that core in order to periodically sample the performance counters. When a derived bandwidth crossed a certain threshold the thread would suspend the core for a specified amount of time. Furthermore, the threads of the real-time and best-effort applications were allocated to disjoint sets of cores.

In the experiments we could influence the following parameters: the type of real-time application, type and number of applications of which the best-effort performance should be optimized, sampling interval, bandwidth threshold, suspension time. The experiments were categorized in three classes.

1) Synthetic Bandwidth Experiments: In our first set of experiments we ran the same program that we used to determine the bandwidth between the level 2 and level 3 of the cache. However, now we applied our duty-cycling technique on cores 2 through 4 and measured the bandwidth on core 1. The sampling interval was 100 μs and the suspension time 900 μs. The thresholds for duty-cycling were set at 25% of the maximal bandwidth. The results of this experiments can be found in Table II and relative to Table I in Table III.

2) Optimized Real-Time Streaming Experiments: In the second set of experiments we ran a static streaming applications on the first and second core of the processor and ran a set of best-effort applications on the third and fourth core. The X-ray application was generated and compiled with the tooling as presented in [1] with the techniques that reduces jitter on CMP when executed as a single application. In this case the interference memory allocator and the ordering thread scheduling heuristic were used. With the performance counters we estimated the read and write bandwidths of the real-time application to be between 4 and 9 GB/s, which implies that there is enough read bandwidth left, but that the write bandwidth is almost saturated. The threshold for duty-cycling were set at 10%. For the sampling interval and suspension time two sets of parameters were used: 500 μs and 500 μs; and 100 μs and 900 μs for the sampling interval and suspension time respectively. During the experiment the latency of real-time stream processing were measured in order to determine the interference between the applications.

The applications that were used as best-effort applications:

- X-ray^i: The same application that is running as real-time application, but now as background task. Generated using the interference (optimized) memory allocator. Without cache thrashing most memory accesses would hit the cache and memory bandwidth between the cache and memory is low in comparison to the bandwidth between level 2 and level 3 of the cache.
- X-ray^s: The same application that is running as real-time application, but now as background task. Generated using the simple memory allocator, which does not optimize memory accesses and the cache hit ratio is much lower in comparison with the X-ray^i application. The bandwidth between the cache and main memory is therefore also much higher.
- Syn ‘C’: An application that reads and writes to the third level of the cache as fast as possible.
- gcc: The gcc compiler while compiling an application.
- ffmpeg: A video transcoder while transcoding a video.

Table IV summarizes the results from this experiment. In the table a row is shown for each set of best-effort applications that were used in the experiment. The first two columns show which best-effort application are run on cores 3 and 4. Each set of best-effort application is run in four different configurations, namely: *baseline* where only the real-time application is run on cores 1 and 2; *congested* where the best-effort applications are executed without our proposed duty-cycle technique; and the two *duty-cycled* configuration where the duty-cycle technique is applied with two sets of parameters as explained before.

3) Unoptimized Real-Time Streaming Experiments: In this third set of experiments we reran the same set of applications as in the second set of experiments but we now ran our streaming application with another memory allocator that does not optimize level 3 cache accesses (i.e. the simple memory heuristic from [1]) and therefore has more accesses to the third level of the cache and to main memory. The results for this set of experiments can be found in Table V.

IX. Results

In Table II the results of the first set of experiments are shown. The results in this table clearly show that the maximal reachable bandwidths for the real-time thread is much higher with the duty-cycling technique enabled then without the technique (see Table I. For example, the bandwidth of configuration E under the write scenario without duty-cycling (i.e., column 10 in Table I) drops from 15.1 GB/s when only one core is writing to 4.2 GB/s when all the cores are writing. When the cores 2 through 4 are duty-cycled and all the cores are writing (i.e., the last cell in column 10 in Table II) the bandwidth only drops to 14.0 GB/s. In this case the degradation reduces 89.9% from 72.1% to 7.2%. Table III summarizes the performance degradation reduction between Table I and Table II.

The second set of experiments were used to estimate the benefits of duty-cycling in a realistic setup. The results for this set of experiments can be found in Table IV.

Although the X-ray^s application does not consume the most bandwidth in the cache hierarchy it degrades (increases) the latency of the real-time the most (i.e., the latency increases from 5867 μs to 13873 μs). The main reason that this application

TABLE II
PARTITIONING EFFECTS IN GB/S WHILE DUTY-CYCLED | A) DOES NOT FIT IN CACHE, NOT PARTITIONED. B) DOES NOT FIT IN CACHE, PARTITIONED. C) FITS IN CACHE, NOT PARTITIONED. D) FITS IN CACHE, PARTITIONED. E) ONLY MEASURED THREAD FITS IN CACHE, PARTITIONED

	Bandwidth (GB/s)														
	Read					Write					Read (50%) / Write (50%)				
Cores	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	11.4	11.5	19.9	19.9	19.8	6.3	6.4	15.0	15.0	15.1	7.3	7.3	18.2	18.2	17.6
2	11.1	11.2	19.9	19.9	19.5	6.1	6.1	14.6	15.0	14.5	7.0	7.1	18.0	18.0	17.0
3	11.0	10.8	19.9	19.9	19.1	6.0	6.0	14.2	14.2	14.1	7.0	6.8	17.7	17.6	16.6
4	10.8	10.9	19.9	19.7	18.7	5.7	5.8	14.1	14.0	14.0	6.9	6.7	17.6	17.2	16.1

TABLE III
PERFORMANCE DEGRADATION REDUCTION (%) | A) DOES NOT FIT IN CACHE, NOT PARTITIONED. B) DOES NOT FIT IN CACHE, PARTITIONED. C) FITS IN CACHE, NOT PARTITIONED. D) FITS IN CACHE, PARTITIONED. E) ONLY MEASURED THREAD FITS IN CACHE, PARTITIONED

	Bandwidth (GB/s)														
	Read					Write					Read (50%) / Write (50%)				
Cores	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
2	91.3	91.7	100.0	100.0	92.1	93.8	91.7	91.1	100.0	90.8	95.7	100.0	88.9	88.2	89.4
3	88.9	88.1	100.0	100.0	92.6	91.7	92.1	89.6	89.6	90.0	97.4	92.1	92.5	91.0	91.1
4	89.6	91.5	100.0	0.0	90.7	83.8	88.6	90.6	89.6	89.9	94.1	91.1	93.7	89.4	88.5

degrades the latency the most because it is the only tested application in our benchmark set that also uses a lot of cache space and therefore introduces more cache thrashing then the other application. Nevertheless, the duty-cycling technique also decreases the latency degradation for this application.

The X-ray^i and Syn ‘C’ application use a significantly smaller amount of the cache space and therefore inflict less cache line evictions to the real-time application, but use much more bandwidth between the level 2 and level 3 of the cache. Also for these cases (i.e., row 1 and 3) we see a significant degradation of the latency (i.e., to 9283 μs and 9219 μs for the two applications respectively). In both cases the latency degradation is reduced by the duty-cycling technique.

Gcc and ffmpeg (i.e. row 4 and 5) both exhibit much lower bandwidth usage to the third level of the cache and therefore do not degrade the latency to the same extend as the other cases.

Both sets of parameters (500/500 and 100/900 for the sampling interval and suspension time) reduced the degradation of the latency of the real-time application. The 100/900 parameters reduced the latency degradation more than the 500/500 parameters, but also results in less best-effort performance.

In the last set of experiments we see that the latency of the interfered real-time application is higher than in the second set, this is due to the fact that the unoptimized X-ray application consumes more bandwidth to the third level of the cache and to main memory is therefore more susceptible to interference. However, relative to the baseline latency of the real-time application the increase is slightly smaller. Furthermore, we observe that also in this case the latency increase could be controlled by the duty-cycling technique.

X. FUTURE WORK

The evaluated duty-cycling technique uses periodic sampling in order to estimate bandwidth usage. Furthermore, it currently uses a simple control mechanism in order to reduce bandwidth congestion. Two simple improvements that can decrease overhead and improve best-effort performance can use so-called event based bandwidth estimation and proportional suspend times.

The event based bandwidth estimation can be implemented by allowing the performance counters to make an interrupt request when a certain threshold is exceeded. This reduces the amount of times that the bandwidth is estimated in applications with low bandwidth usage.

The second improvement could be proportional suspend times instead of fixed suspend times. The amount of times the best-effort application has exceeded bandwidth usage and by how much could be used to determine the suspend time. This allows the duty-cycling technique finer control over when the best-effort should be duty-cycled and thereby increasing best-effort performance without violating real-time constraints.

XI. CONCLUSION

Best-effort applications can cause | due to bandwidth congestion in the cache hierarchy | an unacceptable amount of interference, which can results in an unacceptable latency increase of real-time applications on CMP architectures. This can make it infeasible to execute best-effort applications concurrently with real-time applications. We therefore proposed a duty-cycling technique that can throttle (duty-cycle) best-effort applications when their bandwidth consumption causes too much interference with the real-time applications. The duty-cycling technique is implemented in the Linux operating system

TABLE IV
DUTY-CYCLING RESULTS | LATENCY OF THE REAL-TIME APPLICATION

		Baseline			*Congested*			Duty-cycled 500/500			Duty-cycled 100/900		
Core 3	Core 4	Avg (μs)	Stdev	Max (μs)	Avg (μs)	Stdev	Max (μs)	Avg (μs)	Stdev	Max (μs)	Avg (μs)	Stdev	Max (μs)
X-ray^i	X-ray^i	5867	15.22	6069	9283	19.68	9470	7339	156.93	8849	6136	88.55	7037
X-ray^s	X-ray^s	5867	15.22	6069	13873	87.85	14185	8305	277.14	9474	6304	66.3	6512
Syn 'C'	Syn 'C'	5867	15.22	6069	9219	48.43	10644	7370	90.93	7912	6037	46.87	7225
Syn 'C'	gcc	5867	15.22	6069	7691	161.95	9327	6755	95.48	7256	6050	41.68	6978
ffmpeg	ffmpeg	5667	15.22	6069	6131	149.37	7855	5977	94.55	7211	5881	22.62	6010

TABLE V
DUTY-CYCLING RESULTS - NOT OPTIMIZED FOR LEVEL 3 ACCESS - LATENCY OF THE REAL-TIME APPLICATION

		Baseline			*Congested*			Duty-cycled 500/500			Duty-cycled 100/900		
Core 3	Core 4	Avg (μs)	Stdev	Max (μs)	Avg (μs)	Stdev	Max (μs)	Avg (μs)	Stdev	Max (μs)	Avg (μs)	Stdev	Max (μs)
X-ray^i	X-ray^i	10705	28.56	10918	14458	211.81	16146	13092	248.73	14704	11295	70.89	12551
X-ray^s	X-ray^s	10705	28.56	10918	21236	62.56	21513	14237	69.32	14506	11315	69.58	11670
Syn 'C'	Syn 'C'	10705	28.56	10918	12742	18.85	12805	12193	39.06	12336	11540	46.48	11754
Syn 'C'	gcc	10705	28.56	10918	11990	187.30	12773	11800	98.88	12197	11767	144.83	12368
ffmpeg	ffmpeg	10705	28.56	10918	11554	259.21	12651	11300	159.26	12260	11059	50.94	11467

and uses performance counters available in the cores of the CMP to estimate the bandwidth usage of the best-effort applications. From our experimental results we conclude that the proposed duty-cycling technique can significantly reduce the interference. For an industrial X-ray imaging application we show that the latency increase that is caused by the interference can be controlled while maintaining some best-effort performance. For synthetic benchmarks we show that there are scenarios where we the duty-cycling technique can reduce the latency degradation by 89%. We therefore conclude that best-effort core duty-cycling based on bandwidth consumption is an essential method to control the interference in CMP architectures used for real-time streaming applications in combination with best-effort applications, such as the interventional X-ray application.

XII. ACKNOWLEDGMENTS

The authors would like to thank Philips Healthcare, and in particular Marc Schrijver for their help during this research. This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs.

REFERENCES

[1] M. Westmijze, M. Bekooij, G. Smit, and M. Schrijver, "Evaluation of scheduling heuristics for jitter reduction of real-time streaming applications on multi-core general purpose hardware," pp. 140 –146, Oct. 2011.

[2] H. Stone, J. Turek, and J. Wolf, "Optimal partitioning of cache memory," *Computers, IEEE Transactions on*, vol. 41, no. 9, pp. 1054 –1068, Sept. 1992.

[3] D. Chiou, D. Chiouy, L. Rudolph, L. Rudolphy, S. Devadas, S. Devadasy, B. S. Ang, and B. S. Angz, "Dynamic cache partitioning via columnization," 2000.

[4] R. Iyer, "CQoS: a framework for enabling qos in shared caches of cmp platforms," in *Proceedings of the 18th annual international conference on Supercomputing*, ser. ICS '04. New York, NY, USA: ACM, 2004, pp. 257–266.

[5] H. Kannan, F. Guo, L. Zhao, R. Illikkal, R. Iyer, D. Newell, Y. Solihin, and C. Kozyrakis, "From chaos to qos: case studies in cmp resource management," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 21–30, Mar. 2007.

[6] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 111–122.

[7] B. Verghese, A. Gupta, and M. Rosenblum, "Performance isolation: sharing and isolation in shared-memory multiprocessors," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-VIII. New York, NY, USA: ACM, 1998, pp. 181–192.

[8] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith, "Multicore resource management," *Micro, IEEE*, vol. 28, no. 3, pp. 6 –16, May-Jun. 2008.

[9] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "Compsoc: A template for composable and predictable multi-processor system on chips," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 2:1–2:24, Jan. 2009.

[10] D. Molka, D. Hackenberg, R. Schone, and M. Muller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, Sept. 2009, pp. 261 –270.