

Evaluation of Scheduling Heuristics for Jitter Reduction of Real-Time Streaming Applications on Multi-core General Purpose Hardware

M. Westmijze, M.J.G. Bekooij and G.J.M. Smit
University of Twente, Department of EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
{m.westmijze, m.j.g.bekooij, g.j.m.smit}@utwente.nl

M. Schrijver
Philips Healthcare
Veenpluis 6, 5684 PC Best, The Netherlands
marc.schrijver@philips.com

Abstract—The real-time system research community has paid a lot of attention to the design of safety critical hard real-time systems for which the use of non-standard hardware and operating systems can be justified. However, stream processing applications like medical imaging systems are often not considered safety critical enough to justify the use of hard real-time techniques that would increase the cost of these systems significantly. Instead commercial off the shelf (COTS) hardware and OS are used, and techniques at the application level are employed to reduce the variation in the end-to-end latency of these imaging processing systems.

In this paper, we study the effectiveness of a number of scheduling heuristics that are intended to reduce the latency and the jitter of stream processing applications that are executed on COTS multiprocessor systems. The proposed scheduling heuristics take the execution times of tasks into account as well as dependencies between the tasks, the data structures accessed by the tasks, and the memory hierarchy.

Experiments were carried out on a quad core symmetric multiprocessing (SMP) Intel processor. These experiments show that the proposed heuristics can reduce the end-to-end latency with almost 60%, and reduce the variation in the latency with more than 90% when compared with a naive scheduling heuristic that does not consider execution times, dependencies and the memory hierarchy.

I. INTRODUCTION

Nowadays, COTS hardware is often used for real-time (medical) image processing applications, of which an interventional X-ray application is an illustrative example.

With an interventional X-ray system, a physician makes use of images captured with an X-ray imaging device to perform delicate medical procedures inside a patient, where the only visual feedback is provided by the images captured by the X-ray device. It is therefore desirable that the latency between the capturing of an image and displaying it is low enough (< 200 ms) to provide sufficient eye-hand coordination. Furthermore, the variation of the latency, which is called jitter, must be sufficiently low such that the physician experiences a constant delay which improves the eye-hand coordination and prevents fatigue.

Due to low radiation limits advanced image processing is necessary to obtain sufficient image quality. In an interventional X-ray application, only a fraction of the latency budget is available for image processing due to the latency that the

detector and display introduce. Therefore, the image processing used to be performed on architectures like application specific integrated circuits (ASICs), digital signal processors (DSPs) and field-programmable gate arrays (FPGAs). However, high performance SMP COTS hardware has become performance-wise so powerful and cost-effective, that the trend is to perform the processing on this type of hardware despite the increased temporal uncertainty that this hardware may introduce. The use of COTS hardware seems to be acceptable as long as temporal constraints are rarely violated. Therefore, it is a valid approach to use heuristics for these systems during the design process, after which the systems performance is validated by means of extensive testing.

In this paper we present a number of scheduling heuristics that are intended to reduce the latency as well as the jitter of streaming applications, such as the interventional X-ray application described above. We implemented these heuristics in a tool flow that can synthesize an application from a high level description of an image processing chain. The tool flow was used to evaluate the scheduling heuristics on one image processing chain from the interventional X-ray application and on a set of synthetically generated image processing chains. The synthesized applications were executed on COTS hardware with Intel Nehalem central processing units (CPUs).

This paper is structured as follows. First we discuss related work in Section II, after which we elaborate in Section III what components influence the latency and jitter. In ?? we describe the hardware platform that we use as platform for our application and in Section IV we present how our tool flow synthesizes a high level description of an image processing chain into an application. The experiments are described in Section V. The results of the experimental evaluation can be found in Section VI. Finally we discuss the conclusions in Section VII.

II. RELATED WORK

In [1], Wilhelm et al. discuss the components in an embedded system that affect the tightness of the computed worst-case execution times bounds by means of static timing analysis. The authors conclude that static timing analysis of systems with shared caches is very complex and that the computed bounds

are often not tight. As our objective is to improve the typical behavior instead of the worst-case behavior of an application, we do not need to use formal timing analysis to derive the worst case behavior. Instead we measure the execution times and employ techniques to use the architecture in a way that reduces jitter.

Extensive measurements on a similar multiprocessor system as we consider in this paper, are presented by Molka et al in [2]. However, only results are presented for a synthetic benchmark set, while we study the behavior of complete application, which may provide other insights than a set of synthetic benchmarks.

An approach for improving the temporal behavior of a multiprocessor system with a shared cache by means of locking of cache lines, is presented by Suhendra et al. in [3]. For the machine we consider in this paper, this approach is not applicable because cache line locking is not supported.

In [4], [5], Anderson et al. and Kim et al. analyze the influence of thread scheduling on the behavior of the cache. However, the focus of the papers is mainly on the interaction of different applications, while we focus on the case that only one application is executed on the system.

Papers [6], [7], [8] by Chakraborty et al, Schlieker et al. and Yan et al. introduce analysis methods to take the effect of caches and shared resources into account. However, these papers consider either the case of a single processor system without a shared cache, or consider systems in which only the instruction cache is shared.

In [9], Albers et al. use another model for the mapping and partitioning of computation to threads, but the scheduling order of the application is not taken into account. Furthermore, the focus is on the reduction of latency and not primarily on the reduction of jitter.

III. SOURCES OF JITTER

This section discusses the hardware and software components and features that typically introduce a significant variation in the execution times of the tasks. The variation in execution times result in a variation in the moment that the output results of the application are produced. This variation in production moment is called the jitter. In the following paragraphs we first discuss the hardware components and features that have typically a large effect on the jitter and then the influence of the software.

A. Hardware

The systems that we consider in this paper are systems with similar characteristics as the Intel Nehalem microarchitecture. We consider the following hardware features: functional units, caches, buses, simultaneous multi-threading (SMT), dynamic frequency scaling and dynamic overlocking.

1) *Functional units*: When an instruction is executed it is placed in one of the execution engines, that can perform the specific instruction. These execution engines are deeply pipelined and because of the out-of-order execution of the instructions the latency between the start of an instruction and the end of it depends on several factors like the current status of the pipeline, data dependencies, etc. It is therefore not always

feasible [1] to give accurate upper bounds on the execution times of each individual instruction. However, we are only interested in the execution of large numbers of instructions and therefore assume that the effects of the pipeline averages out.

2) *Caches*: Due to the difference of the clock frequency between the processor and main memory a cache hierarchy is used. The cache hierarchy of the Nehalem microarchitecture consists of three levels. The last level of the cache – the level directly connected to the main memory – is shared between all the cores on the die [10]. When the accessed data by a core is only available locally (e.g. in a register, the first or second level of the cache) the latency of the access is not influenced by other cores and only depends on where it is available locally. The access to the data that is stored in the third level of the cache could be influenced by other cores if the total bandwidth to the third level of the cache is saturated [2], but due to the large size of the second level of the cache and the locality of reference of most streaming applications this is usually not the case. We therefore assume that accessing data that is available in the local cache hierarchy introduces neglectable jitter. When this is not the case some jitter will be introduced, because data has to be loaded from main memory over a shared connected or has to be retrieved from another part of the cache hierarchy.

Reducing the communication between the cache and main memory will therefore mitigate some of the temporal effects of the cache. Allocating the data such that at each moment the actively used data fits in the last level of the cache reduces said communication. We therefore want to use data level parallelism as much as possible, because with functional level parallelism more data is used at the same time, which may result in cache trashing when the amount of accessed data exceeds the capacity of the cache.

Another technique for reducing the communication between the cache and main memory is preventing cache evictions of old data. Such cache evictions can be prevented by reusing the memory location where old data (i.e. data that was used, but is never accessed again) resides for new data. The computation has to be scheduled in an order that would reuse the memory locations of old data before it is evicted from the cache.

Some jitter could also be introduced when data has to be retrieved from non local parts of the cache hierarchy (e.g. from the level 2 cache from another core). A reduction of non-local cache access would therefore reduce the amount of jitter that is introduced by this kind of access. This can be achieved by scheduling computation on cores where the required data was produced.

3) *Buses*: The use of a single bus has been replaced by a interconnect called QuickPath [11] in the Nehalem architecture. Where in a typical system that employs a single bus all cores can influence each other, the QuickPath interconnect limits the influence to cores that share a QuickPath connection. In a multi-die system the communication between the level 3 caches of the cache hierarchy is routed through the QuickPath interconnect. However, a side effect of the cache aware scheduling that reduces the on die communication also reduces the amount of data that is moved through the QuickPath interconnect, e.g. cache coherency traffic, and therefore we assume that effects

introduced by the QuickPath interconnect can be neglected.

4) *Simultaneous multi threading*: With SMT [12], [13], multiple threads use the same execution engines. The use of SMT can have a significant influence on the execution time of a task. In our experiments we evaluate the system with SMT enabled and disabled in order to compare the effects of this hardware feature.

5) *Dynamic Frequency Scaling and Dynamic Overclocking*: The clock frequency of a core in the Nehalem architecture can be scaled dynamically in order to reduce energy usage. Running a core on different clock frequencies introduces jitter, this technique is therefore disabled.

An additional technique (TurboBoost [14], [2]) can dynamically overclock the clock frequency of a core when certain conditions and thresholds (e.g. temperature) are not violated. This technique also introduces jitter and it is therefore disabled.

B. Software

An operating system (OS) can have a significant influence on the jitter of an application because it is responsible for thread scheduling. Because the operating system decides where and when to execute threads, it is important to map the data in such a way that it is likely that a thread is executed on a core where the data is already available. We want to achieve this by replacing the OS scheduler by statically mapping the computations to a limited number of threads so that the operating system can schedule these threads efficiently and that there is only minimal data movement between the caches.

IV. TOOL FLOW

In this section we describe our tool flow that we use to synthesize an application from a high level description of the image processing chain. The tool flow and associated high level description language are designed in such a way that they can vary the usage of the two components that influence the jitter the most, namely, the cache hierarchy and the OS. Our tool flow takes a high level description that only describes functional level parallelism and will then perform the following steps:

- a) Introduction of data parallelism
- b) Scheduling computational steps to threads
- c) Allocate memory
- d) Introduce synchronization
- e) Generate code

In the following paragraphs we will describe the steps in our tool flow in more detail. The tool flow takes as input a high level description in which the functional behavior (e.g. code, functions, etc) is encapsulated in a *box* and connect boxes together to describe the structure of the image processing chain, we will refer to this description as the structural description. Hence, the structural description exposes the functional level parallelism. Each box has a number of associated input and output ports that can be used to connect boxes together. A connection between an output and input port is associated with a memory buffer in order to store the data between the execution of the connected boxes. The applications that we describe with our high level description are *streaming* applications. In our

description it means that the source boxes (i.e. boxes without inputs) are triggered periodically or are triggered at some external event (i.e. arrival of input data). Each execution of an (sub) box takes places in an *iteration*. Depending on the scheduling and mapping of the application it is possible that (sub) boxes from multiple iterations are executing at the same time. In this context we also define the *current* iteration as the *oldest* iterations that still has (sub) boxes to execute. See Fig. 1(a) for a simple example where the image processing chain first applies a gain filter and secondly a convolution on the image that is produced by the source. Each edge that connects two boxes together represents a memory buffer.

A. Introduction of data parallelism

The structural description, that only contains functional level parallelism, is transformed into another description, which we call the instantiated description. This description also incorporates data level parallelisms, where the tool flow has instantiated data parallelism by splitting the boxes into *sub boxes*. Under most circumstances, which we do not elaborate, it is not necessary to introduce more data parallelism than processors available in the hardware platform. Our tool therefore splits each box into as many sub boxes as there are processors. The box is annotated with additional information that is used by the compiler to split the box into sub boxes.

Each sub box performs a part of the computation from the original box where it was instantiated from; the tool annotates each sub box with the part of the computation that it has to perform. In our structural description we have also annotated each box with additional information that can be used to derive fine grained dependencies between sub boxes. Without this information we would have to instantiate dependencies between all sub boxes of subsequent boxes and this would limit the freedom during the scheduling step and thereby would introduce unnecessary synchronization.

See Fig. 1(b) for an example of how the tool flow transforms the structural description from Fig. 1(a) and derives the fine grained dependencies. In this example, there is a gain filter, where each pixel only depends on one pixel and therefore needs the minimal amount of dependencies. This is in contrast to the convolution filter, where each pixels depends on a region of pixels, and each sub box of the convolution therefore depends on multiple sub boxes of the gain filter. Lastly, we can see that our tool could not split the output box into multiple sub boxes because the implementation of that box could not be parallelized.

B. Scheduling computational steps to threads

At this step, we have a description of our image processing chain with data and functional level parallelism, and fine grained dependencies. We can now schedule and order the sub boxes of this description to threads. In our tool we implemented several scheduling heuristics so that we can evaluate the influence of each scheduling heuristic on latency and jitter.

1) *One-to-One*: The One-to-One is the simplest scheduling heuristics where each sub box is given its own thread. We will refer to this mapping method as the *simple* method. In this

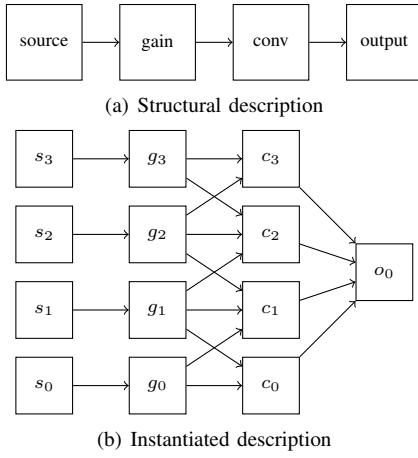


Fig. 1. Example description

method it is possible that some sub boxes of the subsequent iteration execute before the end of a complete iteration, because there is no synchronization between iteration.

2) *One-to-One without pipelining*: Pipelining can significantly increase the amount of sub boxes that can execute and because the OS does not have a notion of which sub boxes belong to the current iteration it may execute sub boxes of subsequent iterations and thereby increase the latency. Pipelining can be prevented by adding a barrier between the last sub box(es) of the current iteration and the first sub box(es) of the next iteration. This method will be called the *barrier* method.

3) *Many-to-One*: The Many-to-One scheduling technique schedules and orders all sub boxes to a configurable amount of threads, we call it the *fixed* method. For each processor that is available for the execution of the application one thread is instantiated. Each of these threads can be fixed to a specific core or to a subset of cores that share a cache level in order to prevent the OS from moving the thread and thereby trashing the cache. The scheduling and ordering is performed by first constructing a homogeneous synchronous dataflow graph (HSDFG) [15] that models the temporal behavior of the application. For each sub box in the application an actor will be instantiated in the HSDFG. The dependencies are translated into the edges of the HSDFG. This HSDFG graph is used to construct a static schedule for each thread.

The advantage of a static schedule is that the application controls the order in which the boxes are executed instead of the scheduler of the operating system. A disadvantage is that this technique does not take into account the state of the cache, which might result in a lot of cache trashing.

4) *Many-to-One cache aware*: This scheduling technique works almost in the same way as the fixed method, but it tries to schedule boxes to a thread taking the state of the cache into account in order to reduce cache misses. During the scheduling of sub boxes this technique gives priority to sub boxes of which the input data is most likely to be in the cache. We will refer to this mapping method as the *predictable* method.

5) *Many-to-One cache aware reduced*: Furthermore, a heuristic can be used to reduce the amount of active data (i.e. data that will be used in this or subsequent iteration) throughout an iteration that was generated using the predictable method. A static order schedule (SOS) is constructed of the structural graph where actors of which execution results in the least amount of active data to be stored in memory are scheduled first. Then a back-tracking algorithm is used to check whether some choices of actor ordering would have resulted in a schedule with a smaller amount of active data during the execution of one iteration. Because the exponential complexity of this back-tracking algorithm, it is stopped when a specific amount of tokens is reached or when it takes too long to explore the complete state space to find the optimal solution. This method will be referred to as the *reduced* method.

C. Allocate memory

When the thread mapping has completed, the memory allocation for the application can be computed. We have examined two memory allocation methods.

1) *Simple*: The *simple* memory allocation scheme allocates a separate memory range for each memory buffer.

2) *Reuse*: The *reuse* memory allocation schemes tries to reuse memory buffers from actors that have already finished their execution (within an iteration). First, an *interference graph* [16] is derived from the thread schedules. Secondly, a first fit heuristic is used to allocate memory for all memory buffers.

D. Introduce synchronization

After the third step all sub boxes are mapped to threads and the required memory has been allocated. The next step is to instantiate the synchronization between the execution of the instantiated boxes. The box level dependencies are instantiated using signals. After a thread has executed a box it will send signals for all box level dependencies of which it is the source. Vice versa before a thread can execute a box it has to wait for all box level dependencies of which it is the destination.

E. Generate code

The final step of the application synthesis is the code generation. Four important pieces of code will be generated that are used to construct the complete application. Firstly, the *initialization code* will be generated. During the initialization memory will be allocated, synchronization primitives will be created and configured and other data structures that configure the application will be configured. Secondly, the *thread code* will be generated. In each thread the call to the sub boxes and synchronization statements are inserted. Thirdly, the *code* is generated that is responsible for starting the execution of all threads. Lastly, the *cleanup code* is generated that stops the threads, deallocates memory and cleans up all the used resources.

V. EXPERIMENTS

In this section, we present the experiments that we have run for the evaluation of the described techniques. Firstly, we describe the platform that we executed the experiments on in

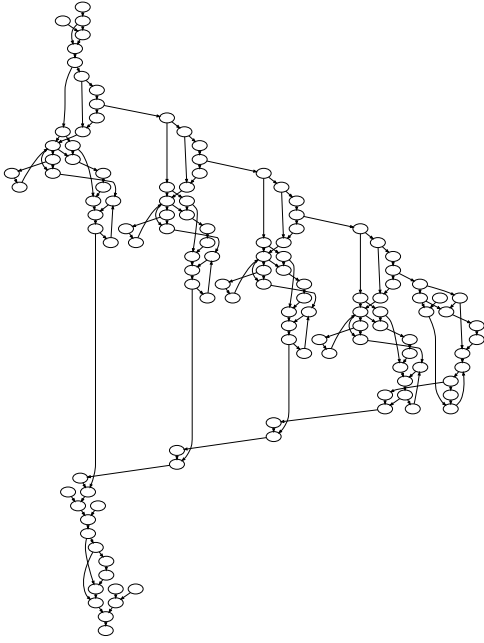


Fig. 2. Topology for the X-ray image processing chain

Section V-A. Secondly, the applications that have been used as input for the experiments are elaborated in Section V-B. Thirdly, we define the experiments that we have run in Section V-C.

A. Experimental setup

The experiments were performed on a quad-core Core i7 860 from Intel. The four cores share the third level of the cache that has a size of 8 MiB. A minimal Ubuntu installation [17] with the 2.6.32 Linux kernel was used as operating system, the system was running without a graphical user interface and unnecessary services were shutdown. Furthermore, the processors were run at 2.8 GHz with TurboBoost disabled.

In each experiment we measured the end-to-end latency of each iteration by collecting time stamps before and after its execution.

B. Experimental input

Figure 2 shows the topology of the structural description of an image processing chain from the interventional X-ray application. Due to the limited amount of image processing chains in the real-life interventional X-ray system we chose to generate additional image processing chains. A tool was created that could generate random graphs that have similar characteristics as the actual image processing chains in the interventional X-ray system. Each graph was created with roughly 100 boxes with a topology that resembles the topology of the actual image processing chain. Furthermore, the number of input and output boxes was chosen to match with some of the interventional X-ray scenarios. After the graphs were generated each box was associated with a random image processing algorithm such as: averaging, addition and convolution. In total 250 image processing chains were generated.

C. Experiments

In each experiment the application was run for 100.000 iterations.

1) *Execution on four physical cores:* In this experiment we applied all the scheduling techniques and where applicable both memory allocation techniques to all the input applications. For the fixed, predictable and reduced techniques we instantiated four threads so that each of those threads could be mapped to one physical core (i.e. the affinity of the thread was reduced to one physical core). Hence, our application did not use SMT but we did not disable SMT altogether, so that the OS could still perform computation on the empty virtual core.

2) *Execution on eight physical cores:* For this experiment we only applied the reduced technique with both memory allocation techniques on one of the synthetic image processing chains. Furthermore we applied the techniques thrice. We instantiated four, eight and sixteen threads respectively in each test. Each thread was mapped onto a single logical core.

VI. RESULTS

A. Execution on four physical cores

In Table I and Table II detailed results can be found from the real image processing chain and one of the synthetic chains, respectively.

First of all, the simple mapping technique does have a significant variation in latency. This is due to pipelining, as explained in Section IV-B2. Pipelining is prevented in the barrier technique and we can see that this reduces the jitter significantly.

The mapping techniques fixed, predictable and reduce result in almost the same jitter regardless of the memory allocation method.

When these heuristics are used in conjunction with the memory reuse technique, a reduction in end-to-end latency is observed. The jitter, however, does not seem to reduce significantly when the reuse heuristic is applied.

When we compare these techniques when the memory reuse heuristic is used we can see a significant difference between the fixed scheduling heuristic and the predictable and reduced scheduling heuristic. The memory reduction algorithm does not immediately seem to impact the average execution length or jitter, but this is due to the fact that in both cases all the accessed memory will fit in the cache. However, the reduction in memory usage might have additional benefits such as a high hit rate on the second level of the cache and it could be more robust against the cache thrashing that might be the result of other applications running on the same system. These possible benefits have not been explored in this paper.

We also see a difference in the effectiveness of the techniques between the actual image processing chain and the synthetic one. On the actual image processing chain the predictable and reduced techniques do not decrease the latency as much as we observe in the synthetic image processing chains. Although we have not shown the box plots of all the synthetic chains, this observations holds for the complete set of synthetics chains.

Thread scheduling	Memory allocation	Memory size (KiB)	\bar{x} (μs)	σ (μs)	
simple	naive	30635	49494	3400	
barrier	naive	30635	4940	37	
fixed	naive	30635	5439	29	
predictable	naive	30635	4561	24	
reduced	naive	30635	4117	30	
fixed	reuse	18250	3527	14	
predictable	reuse	14074	2823	18	
reduced	reuse	7486	1871	8	

TABLE I
RESULTS FOR X-RAY IMAGE PROCESSING CHAIN

Thread scheduling	Memory allocation	Memory size (KiB)	\bar{x} (μs)	σ (μs)	
simple	naive	75456	271618	23726	
barrier	naive	75456	12118	207	
fixed	naive	75456	11088	50	
predictable	naive	75456	10965	62	
reduced	naive	75456	10831	57	
fixed	reuse	11520	5271	11	
predictable	reuse	9792	4622	38	
reduced	reuse	5760	4309	25	

TABLE II
RESULTS FOR SYNTHETIC IMAGE PROCESSING CHAIN

In Fig. 3 we have stacked a compact representation of the box plots of all the executions of the synthetic chains when they were scheduled with the reduced technique and the reuse memory allocation technique was applied. For clarity we have sorted the graphs on the median latency. In the figure we see that the typically observed latency is roughly the same for all the graphs and more importantly that the maximum observed latency is relative to the median.

B. Execution on eight physical cores

In this experiment SMT was evaluated on the synthetic image processing chains. In Table III we see that using the logical cores does increase the latency, most likely due to additional synchronization. However, there is no significant change in the jitter. We expected that there would be an increase in jitter because the execution of the threads on logical cores are highly dependent on each other, but we speculate that since both threads execute similar code they both receive a fair amount of time on the execution engines.

VII. CONCLUSIONS

In this paper we have presented eight scheduling heuristics for the scheduling of static streaming applications on a general purpose multiprocessor system. The scheduling heuristics are intended to reduce the variation in the execution times of the tasks and thereby the jitter of an application that is described as

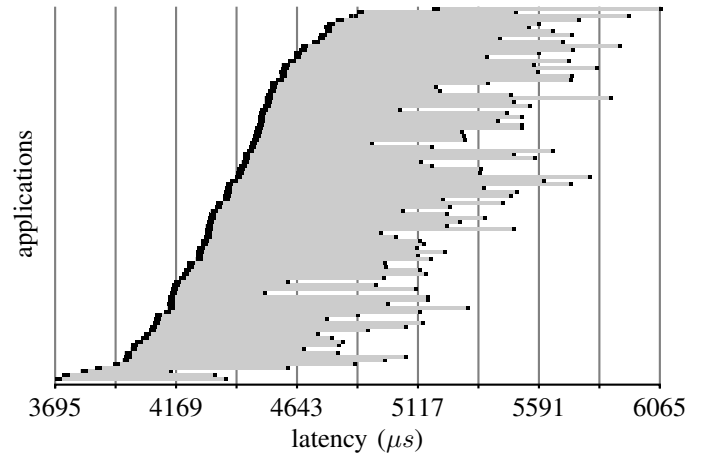


Fig. 3. Detailed results for run with reduce reuse on 4 cores

a high level streaming application. Furthermore, it is desirable that these heuristics reduced the end-to-end latency of the application. The scheduling heuristics have been evaluated using a Quad core SMP Intel machine.

From the experimentally obtained results we observe that if the scheduler in the OS is given the maximum scheduling freedom, the variation of the end-to-end latency, i.e. the jitter, is typically large. If the freedom of the OS scheduler is reduced

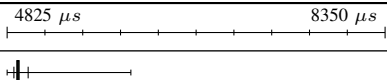


Threads	Memory	\bar{x} (μs)	σ (μs)	
4	Interference	4929	38	
8	Interference	5452	37	
16	Interference	7942	76	

TABLE III
HYPER THREADING ENABLED ON 4 CORES

by using at compile time computed static-order schedules, the jitter is reduced drastically. Our experimental results indicate that the jitter is reduced by roughly 90% by making use of the fixed, predictable and reduced techniques. Another interesting result is that once the influence of the OS scheduler has been removed the jitter is almost equal for all the techniques. We think this a surprising result, we expected more influence of the cache on the jitter.

Furthermore, from the experimentally obtained results we observe that the average latency is mostly dependent on the total data set that is alive at any point in time during the execution of the application. The reuse memory allocation technique decreases the size of this data set because it takes care that memory buffers are reused within one iteration. In the case that the data set fits in the cache, most of the memory accesses will result in cache hits so that the end-to-end latency as well as the jitter is reduced significantly.

In our experiments, we found that using cache aware scheduling techniques that reduce the memory footprint can reduce the average latency by roughly 60% compared to the case that the memory buffers does not fit into the cache.

The fixed, predictable and reduce scheduling heuristics further improve the locality of the memory accesses, such that more – ideally all – accessed data that fits in the cache and that more data is closer to the core. As expected, this results in a further decrease of the end-to-end latency while the jitter remains roughly the same.

Given these observations we conclude that a reduction of the scheduling freedom of the operating system scheduler by applying scheduling heuristics, can reduce the latency and jitter of stream processing applications significantly, and can therefore be a valuable technique for the design of e.g. medical image processing applications that are executed on general purpose multiprocessor systems.

REFERENCES

- [1] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, pp. 966–978, July 2009. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1669804.1669808>
- [2] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 261–270. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1636712.1637764>
- [3] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Proceedings of the 45th annual Design Automation Conference*, ser. DAC '08. New York, NY, USA: ACM, 2008, pp. 300–303. [Online]. Available: <http://doi.acm.org/10.1145/1391469.1391545>
- [4] J. H. Anderson, J. M. Calandrino, and U. C. Devi, "Real-time scheduling on multicore platforms," pp. 179–190, 2006. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1128017.1128438>
- [5] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," pp. 111–122, 2004. [Online]. Available: <http://dx.doi.org/10.1109/PACT.2004.15>
- [6] S. Chakraborty, T. Mitra, A. Roychoudhury, and L. Thiele, "Cache-aware timing analysis of streaming applications," *Real-Time Syst.*, vol. 41, pp. 52–85, January 2009. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1485069.1485080>
- [7] S. Schliecker, M. Negrean, and R. Ernst, "Bounding the shared resource load for the performance analysis of multiprocessor systems," *Proc. of Design, Automation, and Test in Europe (DATE)*, March 2010.
- [8] J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared l2 instruction caches," *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 80–89, 2008. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1440456.1440579>
- [9] R. Albers, E. Suijs, and P. H. N. de With, "Optimization model for memory bandwidth usage in X-ray image enhancement," in *SPIE Electronic Imaging*, 2008, pp. 6811–04.
- [10] (2011) Smart cache. [Online]. Available: <http://www.intel.com>
- [11] (2011, Apr.) Intel quickpath architecture. [Online]. Available: <http://www.intel.com/technology/quickpath/whitepaper.pdf>
- [12] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1995, pp. 392–403.
- [13] D. Koufaty and D. Marr, "Hyperthreading technology in the netburst microarchitecture," *Micro, IEEE*, vol. 23, no. 2, pp. 56 – 65, march-april 2003.
- [14] (2011, April) Turbo boost. [Online]. Available: <http://www.intel.com>
- [15] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, pp. 24–35, January 1987. [Online]. Available: <http://dx.doi.org/10.1109/TC.1987.5009446>
- [16] G. J. Chaitin, "Register allocation & spilling via graph coloring," *SIGPLAN Not.*, vol. 17, pp. 98–101, June 1982. [Online]. Available: <http://doi.acm.org/10.1145/872726.806984>
- [17] (2011, July) Ubuntu. [Online]. Available: <http://www.ubuntu.com>