Low-level Scheduling Implications for Data-intensive Cyclic Workloads on Modern Microarchitectures

Håvard Espeland, Preben N. Olsen, Pål Halvorsen, Carsten Griwodz
Simula Research Laboratory, Norway IFI, University of Oslo, Norway
{haavares, prebenno, paalh, griff}@ifi.uio.no

***Abstract*—Processing data intensive multimedia workloads is challenging, and scheduling and resource management are vitally important for the best possible utilization of machine resources. In earlier work, we have used work-stealing, which is frequently used today, and proposed improvements. We found already then that no singular work-stealing variant is ideally suited for all workloads. Therefore, we investigate in more detail in this paper how workloads consisting of various multimedia filter sequences should be scheduled on a variety of modern processor architectures to maximize performance. Our results show that a low-level scheduler additionally cannot achieve optimal performance without taking the specific micro-architecture, the placement of dependent tasks and cache sizes into account. These details are not generally available for application developers and they differ between deployments. Our proposal is therefore to use performance monitoring and dynamic adaption for the cyclic workloads of our target multimedia scenario, where operations are repeated cyclically on a stream of data.**

I. Introduction

There is an ever-growing demand for processing resources, and the trend is to move large parallel and distributed computations to huge data centers or cloud computing. For example, Internet users uploaded one hour of video to YouTube every second in January 2012 [19], an increase of 25% in the last 8 months. Each of these is encoded using several filters that are arranged as vertices in dependency graphs and where each filter implements a particular algorithm representing one stage of a processing pipeline. In our research, we focus on utilizing available resources in the best possible manner for this kind of time-dependent cyclic workloads. The kind of workload is typical for multimedia processing, where large amounts of data are processed through various filters organized in a data-intensive processing pipeline (or graph). Such workloads are supported by for example the OpenCV framework [4].

To process large data sets in general, several frameworks have emerged that aim at making distributed application development and processing easier, such as Google's MapReduce [6], IBM's System S [11] and Microsoft's Dryad [13]. However, these frameworks are limited by their design for batch processing with few dependencies across a large cluster of machines. We are therefore currently working on a framework aimed for distributed real-time multimedia processing called P2G [7]. In this work, we have identified several challenges with respect to low level scheduling. The de facto standard is a variant of *work-stealing scheduling* [2], for which we have earlier proposed modifications [17]. We see challenges that have not been addressed by this, and in our work of designing an efficient scheduler, we investigate in this paper how to structure parallel execution of multimedia filters to maximize the performance on several modern architectures.

Performance implications for scheduling decisions on various microarchitectures have been studied for a long time. Moreover, since the architectures are constantly being revised, scheduling decisions that were preferred in the past can harm performance on later generations or competing microarchitectures. We look at implications for four current microarchitectures and how order of execution affects performance. Of recent work, Kazempour et al. [14] looked at performance implications for cache affinity on Clowertown generation processors. It differs from the latest microarchitectures by only having two layers of cache, where only the 32 KB L1 D-cache is private to the cores on a chip multiprocessor (CMP). In their results, affinity had no effect on performance on a single chip since reloading L1 is cheap. When using multiple CMPs, on the other hand, they found significant differences meriting affinity awareness. With the latest generation CMPs having significantly larger private caches (e.g., 256 KB on Sandy Bridge, 2 MB on Bulldozer), we can expect different behavior than on Clowertown. In terms of schedulers that take advantage of cache affinity, several improvements have been proposed to the Work Stealing model. Acar et al. [1] have shown that the randomized stealing of tasks is cache unfriendly and suggest a model that prefers stealing tasks where the worker thread has affinity with that task and gains increased performance.

In this paper, we present how a processing pipeline of real-world multimedia filters runs on state-of-the-art processors. That there are large performance differences between different architectures is to be expected, but we found so large differences between modern microarchitectures even within the same family of computers (x86) making it hard to make scheduling decisions for efficient execution. For example, our experiments shows that there are huge gains to be earned looking into cache usage and task dependencies. There are also huge differences in the configuration of the processing stages, e.g., when changing the amount of rotation in an image, giving

completely different resource requirements. Based our these observations, it is obvious that the low-level scheduling should not only depend on the processing pipeline with the dependencies between tasks, but also the specific micro-architecture and the size of the caches. We discuss why scheduling approaches such as standard work stealing models do not result in an optimal performance, and we try to give some insights that a future scheduler should follow.

II. Design and Implementation

Inspired by prevalent execution systems such as StreamIT [16] and Cilk [3], we look at ways to execute data-intensive streaming media workloads better and examine how different processing schemes affect performance. We also want to investigate how these behave on different processors. Because we are processing continous data streams, we are not able to exploit task parallelism, e.g., by processing independent frames of a video in parallel and therefore seek to parallelize within the data domain. A scenario with embarrassingly parallel workloads, i.e., where every data element can be processed independently and without synchronization, is video stream processing. We believe that processing a continous flow of video frames is a reasonable example for several embarrassingly parallel data bound workloads.

Our **sequential approach** is a straight-forward execution structure in which a number of filters are processed sequentially in a pipeline, each frame is processed independently by one or more threads by dividing the frame spatially. In each pipeline stage, the worker threads are created, started, and eventually joined when finished. This would be the natural way of structuring the execution of a multithreaded media pipeline in a standalone application. Such processing pattern has natural barriers between each stage of the pipeline. For a execution system such as Cilk [10], Multicore Haskell [15], Threading Building Blocks [12] and others that use a work stealing model [2], the pattern of execution is similar to the sequential approach, but this depends very much on that way in which work units are assigned to worker threads, the workloads that are running simultaneously and scheduling order. Nevertheless, sequential execution is the baseline of our evaluation since it processes each filter in the pipeline in a *natural* order.

As an alternative execution structure, we propose using **backward dependencies** (BD) to execute workloads. This approach only considers the last stage of the pipeline for a spatial division among threads and such avoids the barriers between each pipeline stage. Furthermore, for each pixel in the output frame, the filter backtracks dependencies and acquires the necessary pixel(s) from the previous filters. This is done recursively and does not require intermediate pixels to be stored to memory. Figure 1 illustrates dependencies between three frames connected by two filters. The pixels in frame 2 are generated when needed by filter B using filter A. The BD approach has the advantage of only computing the pixels that are needed by subsequent filters. The drawback, however, is that intermediate data must be re-computed if they are accessed multiple times because intermediate results are not stored. These re-computations can be mitigated by using the intermediate frames as buffer caches between filters, although the overhead of managing and checking this buffer cache can be large, which we see later in the paper.

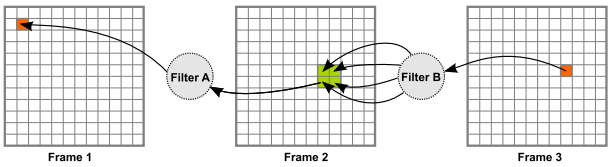


Figure 1. Backward dependencies (BD) example for two filters processing frames in a pipeline. The arrows indicate which pixels are required from the previous frame to generate the current pixel. Only one pixel's dependencies per frame are illustrated, other pixels have similar dependencies.

The different approaches incur varying cache access patterns. Depending on memory access patterns, execution structure and chosen CPU microarchitecture, we expect the performance to change. The sequential approach accesses the buffers within a filter in sequential order, and the prefetch unit is thus able to predict the access pattern. A drawback of this approach is that data moved between filters do not necessarily reside in the cache of the core using the data last. First, this includes data whose cache line has been evicted and written back to a cache level with increased access time or memory. This may happen because the data size processed by a filter is larger than the amount of cache available, forcing write-back. Other reasons include context switches and shared caches. Second, output from a previous filter may not have been generated on the same core as the one that accesses the data, resulting in accesses to dirty cache lines on other cores. Given the spatial division of a frame within a filter, this sounds easy to avoid, but an area of input to a filter may result in output to a different spatial area, which the processor's prefetcher may not be able to predict. Thus, re-using the same core for the same part of a frame for multiple filters in a pipeline only increases cache locality for filters whose source pixels map spatially to the destination pixels.

To increase cache locality between filters, we also evaluate the BD approach, where data is accessed in the order needed to satisfy dependencies for the next filter in the pipeline. This ensures that pixels are accessed in a manner where data in between filters are likely to reside in the core's cache. That is to say, if the access pattern is not random, one can expect BD execution to always access spatially close memory addresses.

III. Experimental Setup

To evaluate the approaches and find which performs best in a low-level scheduler, we built an experimental framework supporting the proposed execution structures and wrote a set of image processing filters as a case study working on real-world data. The filters were arranged in different pipelines to induce behaviour differences that can impact performance.

All experiments measure exclusively computation time, i.e., the wall clock time of the parallel execution, excluding I/O and setup time. We use this instead of CPU time to further measurements on how good performance is actually possible,

Microarchitecture	CPU	Cores (SMT)	Private Cache	Shared Cache
Nehalem	Intel i5-750	4	64 kB L1 256 kB L2	8 MB L3
Sandy Bridge	Intel i7-2600	4 (8)	64 kB L1, 256 kB L2	8 MB L3
Sandy Bridge-E	Intel i7-3930K	6 (12)	64 kB L1, 256 kB L2	12 MB L3
Bulldozer	AMD FX 8192	8 (4x2)	64 kB L1[1], 2 MB L2[1]	8 MB L3

[1] Shared between two modules each having a separate integer unit while sharing an FPU.

Table I
MICROARCHITECTURES USED IN EXPERIMENTS.

since having used only half of the CPU time available does not mean that only half of the CPU's resources are utilized (cache, memory bandwidth, etc.). Also, by not counting I/O time, we remove a constant factor present in all execution structures, which we believe better captures the results of this study. Each experiment is run 30 times, and the reported computation time is the average All filters use 32-bit float computations, and overhead during execution, such as removing function calls in the inner-loop and redundant calculations, has been removed. Our data set for experiments consists of the two standard video test sequences *foreman* and *tractor* [18]. The former has a 352x288 pixel (CIF, 4:2:0) resolution with 300 frames of YUV data, the latter has 1920x1080 pixels (HD, 4:2:0) with 690 frames of YUV data.

The experiments have been performed on a set of **modern microarchitectures** as listed in table I. The CPUs have 4 to 8 cores, and have rather different cache hierarchies: While the Nehalem has an L3 cache shared by all cores, operates at a different clock frequency than the cores and is called the *uncore*, the Sandy Bridge(-E) has a slice of the L3 cache assigned to each core and accesses the other parts using a ring interconnect running at core speed. Our specimen of the Bulldozer architecture consists of four modules, each of which containing two cores. On each module, L1 and L2 are shared between the two cores with separate integer units but a single shared FPU. We expected that these very different microarchitectures found and used in modern computing would produce very different program behaviour, and we have investigated how media workloads should be structured for execution on each of them to achieve the best performance.

We have developed a set of image processing **filters** for evaluating the execution structures. The filters are all data-intensive, but vary in terms of the number of input pixels needed to produce a single output pixel. The filters are later combined in various configurations referred to as pipelines. A short summary of the filters and their dependencies is given in table II.

The filters are combined in various configurations into **pipelines** (as in figure 1). The tested pipelines are listed in table III. The pipelines combine the filters in manners that induce different amounts of work per pixel, as seen in the table. For some filters, not all intermediate data are used by later filters and are unnecessary to produce the final output.

Blur convolves the source frame with a Gaussian kernel to remove pixel noise.
Sobel X and Y are two filters that also convolve the input frame, but these filters apply the Sobel operator used in edge detection.
Sobel Magnitude calculates the approximate gradient magnitude using the results from Sobel X and Sobel Y.
Threshold unset every pixel value in a frame below or above a specified threshold.
Undistort removes barrel distortion in frames captured with wide-angle lenses. Uses bilinear interpolation to create a smooth end result.
Crop removes 20% of the source frame's height and width, e.g., a frame with a 1920x1080 resolution would be reduced to 1536x864.
Rotation rotates the source frame by a specified number of degrees. Bilinear interpolation is used to interpolate subpixel coordinates.
Discrete discretizes the source frame by reducing the number of color representations.
Binary creates a binary (two-colored) frame from the source. Every source pixel that is different from or above zero is set, and every source pixel that equals zero or less is unset.

Table II
IMAGE PROCESSING FILTERS USED.

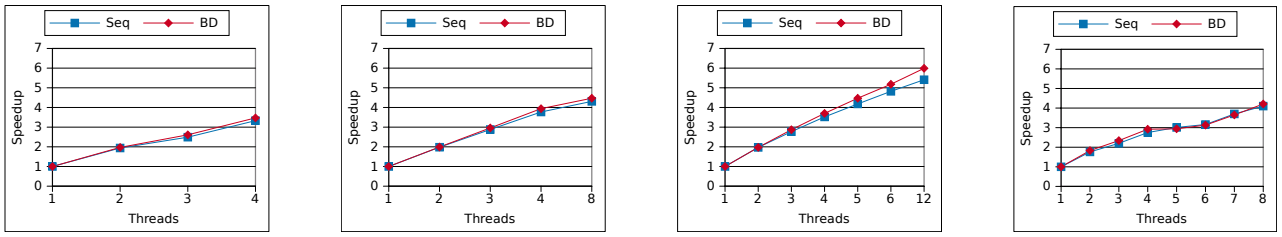
Pipeline	Filter	Seq	BD	BD-CACHED
A	Blur	9.00	162.00	9.03
	Sobel X	9.00	9.00	9.00
	Sobel Y	9.00	9.00	9.00
	Sobel Magnitude	2.00	2.00	2.00
	Threshold	1.00	1.00	1.00
B	Undistort	4.00	10.24	2.57
	Rotate 6°	3.78	2.56	2.56
	Crop	1.00	1.00	1.00
C	Undistort	4.00	8.15	2.04
	Rotate 60°	2.59	2.04	2.04
	Crop	1.00	1.00	1.00
D	Discrete	1.00	1.00	1.00
	Threshold	1.00	1.00	1.00
	Binary	1.00	1.00	1.00
E	Threshold	1.00	3.19	0.80
	Binary	1.00	3.19	0.80
	Rotate 30°	3.19	3.19	3.19
F	Threshold	1.00	3.19	0.80
	Rotate 30°	3.19	3.19	3.19
	Binary	1.00	1.00	1.00
G	Rotate 30°	3.19	3.19	3.19
	Threshold	1.00	1.00	1.00
	Binary	1.00	1.00	1.00

Table III
EVALUATED PIPELINES AND THE AVERAGE NUMBER OF OPERATIONS PERFORMED PER PIXEL WITH DIFFERENT EXECUTION STRUCTURES.

The BD approach will not produce these, e.g., a crop filter as seen in pipeline B will not require earlier filters to produce unused data. Another aspect that we expect to influence the results is cache prefetching. This means that by having filters that emit data in a different spatial position relative to its input, e.g. the rotation filter, we expect the prefetcher to contend fetching the relevant data.

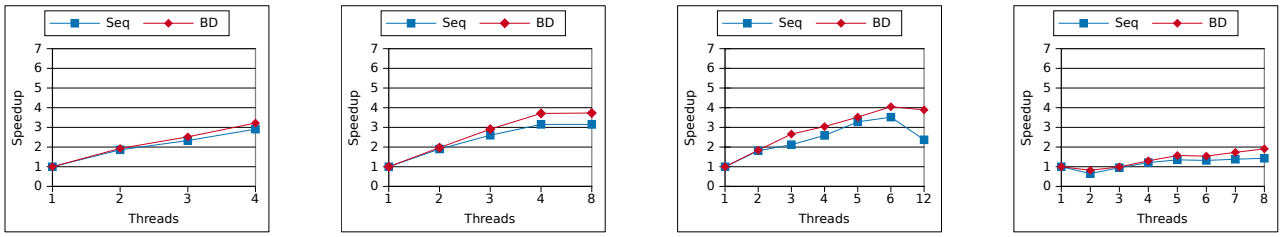
IV. SCALABILITY

The pipelines are embarrassingly parallel, i.e., no locking is needed and they should therefore scale linearly with the number of cores used. For example, using four cores is expected to yield a 4x execution speedup. The threads created



(a) Nehalem (4) (b) Sandy Bridge (4) (c) Sandy Bridge-E (6) (d) Bulldozer (8)

Figure 2. Individually normalized scalability of pipeline B running the tractor test sequence. Number of physical cores in parenthesis.



(a) Nehalem (4) (b) Sandy Bridge (4) (c) Sandy Bridge-E (6) (d) Bulldozer (8)

Figure 3. Individually normalized scalability of pipeline B running foreman test sequence. Number of cores in parenthesis.

are handled by the Linux scheduler, which decides which thread to execute on which CPU (no affinity). Each new thread created works on its own frame-segment, but the last frame-segment is always processed by the thread that performs the I/O operations. In the single-threaded execution, both the I/O and processing operations are performed in the same thread, and therefore also on the same CPU core. Using two threads, we created one additional thread that is assigned the first half of a frame while the I/O thread processes the last half of the frame. We do this to minimize fetching source data from another core's private cache.

Figure 2 shows the relative speedup for the Nehalem, Bulldozer, Sandy Bridge and Sandy Bridge-Extreme microarchitectures that process pipeline B with the *tractor* sequence as input, comparing the sequential (Seq) and BD executions. Note that the running times of sequential and BD are individually normalized to their respective performance on one core, i.e., a higher value for BD does not necessarily imply better absolute performance than sequential. Looking at each architecture individually, we see that there is little difference in how the two execution methods scale on physical cores, but comparing architectures, we see that SB (figure 2(b)) is the only architecture that is able to scale pipeline B perfectly with the number of physical cores, as one could expect. SB-E (figure 2(c)) performs a little below perfect linear scaling achieved by its predecessor for this workload, and we see slightly better scalability results for BD execution than for sequential execution. On Nehalem (figure 2(a)), this pipeline doubles its performance with two threads, but after this, the increase in speedup per core diminishes. Bulldozer results (figure 2(d)) are even worse; from one to four threads we gain a 3x speedup, but there is not much to gain from using five to eight threads as we only manage to achieve a 4x speedup using the maximum number of cores. Bulldozer has four modules, each with two cores, but each module has only one FPU, and it is likely that this clamps performance to 4x. To summarize, the scalability results of pipeline B with the tractor sequence as input scales relatively well on Nehalem, Sandy Bridge and Sandy Bridge-Extreme using both execution modes. However, Bulldozer scales poorly as it only manages a 4x speedup using all of its eight cores.

Looking at scalability when testing pipeline B with the *foreman* sequence, the results become far more interesting as seen in figure 3. None of the four microarchitectures are able to achieve perfect scalability, and standing out is the Bulldozer plot in figure 3(d), where we see very little or no speedup at all using more threads. In fact, the performance worsens going from one to two threads. We look more closely into the behaviour of Bulldozer in section VII. Furthermore, we can see that the BD mode scales better than sequential for the other architectures. From one to two threads, we get close to twice the performance, but with more threads, the difference between sequential and BD increases.

To explain why pipeline B scales so much worse with foreman as input than tractor, one must take the differences into account. The foreman sequence's resolution is 352x288 (YUV, 4:2:0), which leads to frame sizes of 594 kB stored as floats. A large chunk of this can fit in private cache on the Intel architectures, and a full frame on Bulldozer. A frame in the tractor sequence has a resolution of 1920x1080 which requires almost 12 MB of data, exceeding even L3 size. In all pipeline stages except the last, data produced in one stage are referenced in the next, and if the source data does not reside in a core's cache, it leads to high inter-core traffic. Segmenting the frames per thread, as done when executing the pipelines in parallel, reduces the segments' sizes enough to fit into each core's private cache for the foreman frames, but not the larger tractor frames. Not being able to keep a large part of the frame in a core's private cache require continuous fetching from shared cache and/or memory. Although this takes time,

it is the normal mode of operation. When an large part of a frame such as foreman fits in private cache after I/O, other cores that shall process this will have to either directly access the other core's private cache or request eviction to last-level cache on the other core, both resulting in high inter-core traffic. We were unable to find detailed information on this behaviour for the microarchitectures, but we can observe that scalability of this behaviour is much worse than that of the HD frame experiment. Moreover, since the BD mode only create one set of worker threads which are used throughout the lifetime of that pipeline cycle, and it does its computations in a recursive manner, the input data is likely to reside in the core's private cache. Also, since the BD mode does not require storing intermediate results and as such does not pollute the caches, we can see it scales better than sequential for the foreman tests.

With respect to the other pipelines (table III), we observed similar scalability across all architectures for both execution modes and inputs as seen in figure 2 and 3. In summary, we have shown that the BD mode provides slightly better scaling than sequential execution for our data-intensive pipelines on all architectures when the working unit to be processed (in this case a segment of a video frame) is small enough to a large extent reside in a core's private cache. Although scalability is beneficial, in the next section, we will look at the performance relative to sequential execution and see how BD and sequential perform against each other.

V. Performance

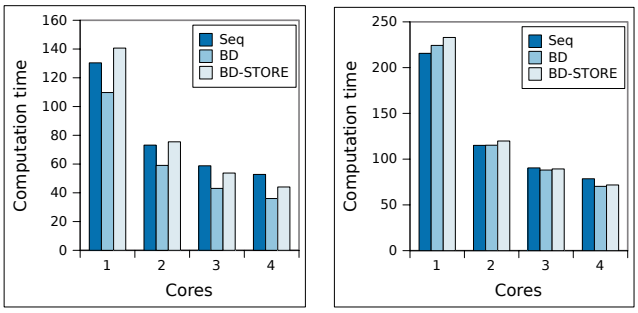
Our experiments have shown that achieving linear scaling on our data-bound filters is not trivial, and we have seen that it is especially hard for smaller units of work that mostly fit into a core's private cache and needs to be accessed on multiple cores. In addition, there are large microarchitectual differences visible. In this section, we look at the various pipelines as defined in section III to see how the sequential execution structure compares to backward dependency on various microarchitectures.

To compare sequential and BD execution, we have plotted computation time for pipeline A to G relative to their individual single-threaded sequential execution time using the foreman test sequence in figure 4. The plot shows that sequential execution provides best performance in most cases, but with some notable exceptions. The BD execution structure does not store and reuse intermediate pixels in any stage of the pipeline. Thus, when a pixel is accessed multiple times, it must be regenerated from the source. For example, if a filter in the last stage of a pipeline needs to generate the values of a pixel twice, every computation in every stage in the current pipeline involved in creating this output pixel must be executed twice. Obviously, this leads to a lot of extra computation as can be seen from table III, where for instance the source pixels are accessed 162 times per pixel for the BD approach in pipeline A, but only 9 times for the sequential approach. This is reflected in the much higher BD computation time than sequential for pipeline A for all plots in figure 4.

When looking at the other pipelines, we can see some very significant architectural differences. Again, Bulldozer stands out showing that for pipeline D, F, and G, the BD approach performs considerably better than sequential execution using all eight cores. Pipeline D and G perform better with BD execution, but it is rather unexpected for pipeline F, which does require re-computation of many intermediate source pixels. Still, the scalability achieved on Bulldozer for the foreman sequence were miniscule, as we saw in section IV.

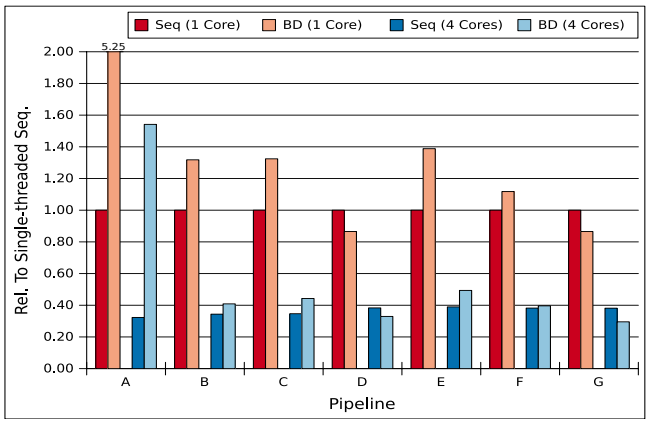
The next observation we show is the performance of pipeline D. This pipeline performs the same amount of work regardless of execution structure, since every intermediate pixel is accessed only once. Most architectures show better performance for the BD approach, both for one and all cores. The only exception is the Sandy Bridge-E which performs slightly worse than sequential when using 6 cores. A similar behaviour as pipeline D is to be expected from pipeline G since it requires the same amount of work for both modes. This turns out not to be the case; for single-threaded execution on Nehalem and Bulldozer, pipeline G is faster using BD. Using all cores, also Sandy Bridge performs better using the BD approach. Sandy Bridge-E runs somewhat faster with sequential execution. We note that Sandy Bridge and Sandy Bridge-E are very similar architectures which ought to behave the same way. This turns out not to be the case, and even this small iteration of the microarchitecture may require different low-level schedules to get the highest level of performance - even for an embarrassingly parallel workload.

To find out if the performance gain seen with the BD approach is caused by better cache usage by keeping intermediate data in private caches or by the reduction of cache and memory pollution resulting from not storing intermediate results, we looked at pipeline D and G using BD while storing intermediate data. This mimics the sequential approach, although data is not referenced again later on, resulting in less cache pressure. Looking at figure 5, which shows pipeline D and G for Sandy Bridge, we can see that for a single core, the overhead of writing back intermediate results using BD-STORE results in worse performance than sequential execution, whereas this overhead diminishes when the number of threads is increased. Here, the BD-STORE structure outperforms sequential execution significantly. Accordingly, we ascribe the performance gain for the BD approach in this case to better private cache locality and usage than sequential execution.

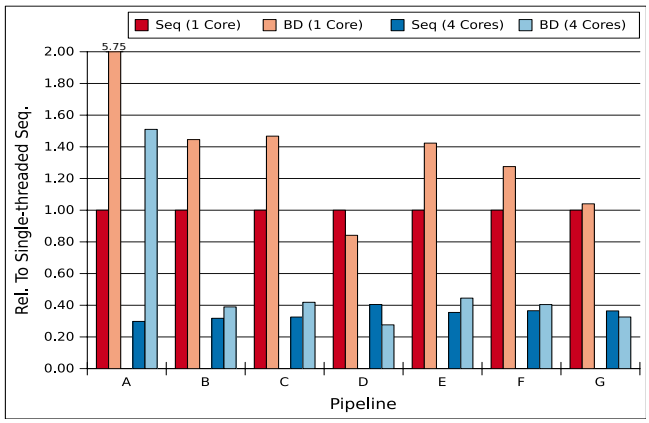


(a) Pipeline D (b) Pipeline G

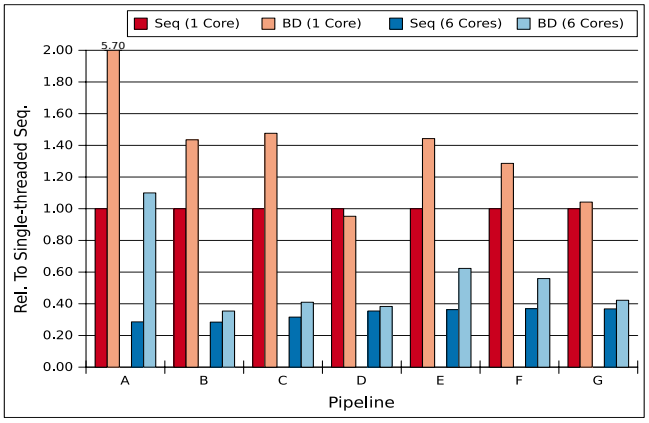
Figure 5. Computation time (ms) for foreman on Sandy Bridge.



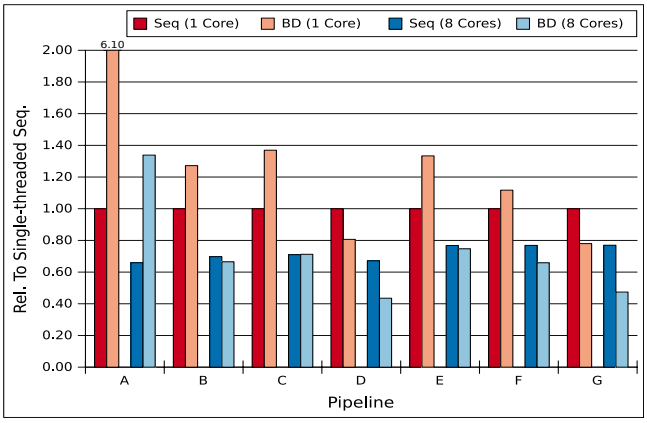
(a) Nehalem



(b) Sandy Bridge



(c) Sandy Bridge-E



(d) Bulldozer

Figure 4. Execution structure performance using the foreman dataset. Running times are relative to each pipeline's 1-core sequential time. Lower is better.

VI. Backward Dependency with Buffer Cache

Having shown that backward dependency execution structure performs better than sequential execution in some cases, we look at ways to improve the performance further. The main drawback of BD execution is, as we saw, that intermediate pixels must be recomputed when accessed multiple times, evident by pipeline A results. We have also shown that, when doing the same amount of work as the sequential approach, BD can perform better such as with pipeline D and G. In this section, we experiment with adding a buffer cache that keeps intermediate data for later reuse to mitigate this issue.

Instead of recursively generating all prior intermediate data when a pixel is accessed, the system checks a data structure to see if it has been accessed earlier. To do this without locking, we set a bit in a bitfield atomically to mark a valid entry in the buffer cache. It is worth noting that this bitfield consume a considerably amount cache space by using one bit for every pixel, e.g. about 37 kB for every stage in a pipeline for the foreman sequence. Other approaches for this buffer cache are possible, including a set of ranges and tree structures, but this is left for further work.

The cached results for pipelines A to G (BD-CACHED) using four threads on Sandy Bridge and processing foreman are shown in figure 6. We can see that the BD-CACHED approach provides better performance than sequential on pipeline B, D and G, but only B performs better than BD. Pipeline B has rotation and crop stages that reduce the amount of intermediate pixels that must be produced from the early filters compared to sequential execution (see table III). In comparison, pipeline C has also a significant reduction of intermediate pixel requirements, but we do not see a similar reduction in computation time. The only difference between pipeline B and C is the amount of rotation, with 6° for B and 60° for C. The skew from rotation for the pipelines causes long strides over cache lines when accessing neighbouring pixels from previous stages, which can be hard to prefetch efficiently. Also, parts of an image segment processed on a core may cross boundaries between segments, such that high inter-core traffic is required with sequential execution even though the same core processes the same segment for each stage. This is avoided with BD and BD-CACHED modes, but we can not predict which mode performs better in advance.

VII. Affinity

In section IV, we saw the diminishing performance when processing the foreman sequence with two and three threads on the Bulldozer architecture. When processing our pipelines in a single thread, we did not create a separate thread for processing, but processed in the thread that handled the I/O operations. Further, this lack of scaling was only observed

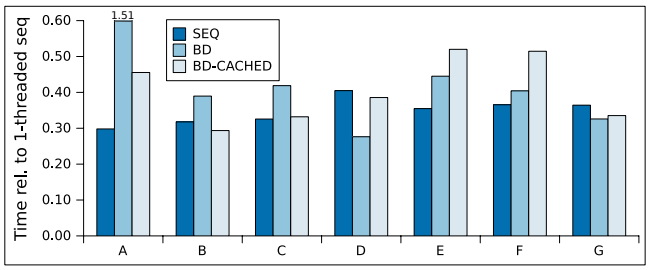


Figure 6. Computation time for Backward Dependency with buffer cache measured relative to single threaded sequential and tested with Sandy Bridge using 4 threads and foreman as input sequence.

when processing the low-resolution foreman sequence, for a large part of the frame could fit inside a core's private L2 cache. To investigate this further, we did an experiment where we manually specified each thread's affinity, which implies that we use separate threads doing I/O and processing, even in single-threaded execution.

Even though we only noticed the lack of scaling on the Bulldozer architecture, we tested the impact of affinity on the Nehalem and Sandy Bridge architectures as well (we omitted SB-E, which is very similar to Sandy Bridge). In figure 7, we have plotted the results for sequential processing of pipeline B while varying the number of threads. On Nehalem and Sandy Bridge, we tested two different execution modes, one mode where the processing of a frame was done on the same core that executed the I/O thread (IOSAME), while the second mode processed the frame on a different core than the I/O thread (IODIFF). Since the Bulldozer architecture has CPU modules, we added an additional mode for this architecture, in which processing threads were executed on different CPU modules than the thread handling I/O (MODULEDIFF).

We expected that processing on another core than I/O would increase the processing time, which was found to be only partially correct: From figure 7(a) we see that there are not any significant difference in processing on the same core as the I/O thread versus a different core on Sandy Bridge. Much of the same behaviour is seen on Nehalem using one thread, but when using two threads there is a large penalty of pinning these to different cores than the one doing I/O operations. In this case, using two threads and executing them on different cores than the I/O operations adds so much to the computation time that it is slower than the single threaded execution.

Looking at the Bulldozer architecture in figure 7(c), using only one thread we see that there are not any significant difference in which core or CPU module the processing thread is placed on. However as with Nehalem, when using two or more threads the IODIFF and MODULEDIFF execution modes have a huge negative impact on the performance. Even more unexpected, IOSAME and IODIFF using three threads are actually slower than IOSAME using two threads, and the fastest execution using eight threads completes in 1174 ms (omitted in plot), not much faster than what we can achieve with two threads.

In summary, we have seen that thread affinity has an enormous impact on Nehalem and Bulldozer when a large part of the data fits into the private cache. This can cause the two threads to have worse performance than a single thread when the data reside in another core's private cache. On Sandy Bridge, this limitation has been lifted and we do not see any difference due to affinity. Bulldozer is unable to scale to much more than two threads when the dataset fits into the private cache, presumably because the private cache is shared between two cores and the cost of doing inter-module cache access is too high.

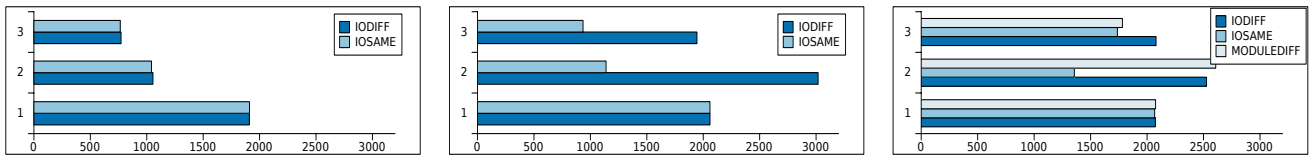
VIII. Discussion

The long-running data-intensive filters described in this paper deviate from typical workloads when looking at performance in terms of the relatively little computation required per unit of data. The filters used are primitive, but we are able to show large performance impacts by varying the order of execution and determining which core should do what. For such simple and embarrassingly parallel workloads, it is interesting to see the significant differences on how these filters perform on modern microarchitectures. As a case study, we chose image processing algorithms, which can intuitively be connected in pipelines. Real-world examples of such pipelines can be found in OpenCV [4], node-based software such as the Nuke [9] compositing tool and various VJ software. Other signal processing domains such as sound processing are applicable as well.

There is a big difference between batch processing and processing a stream of data, where in the former we can spread out independent jobs to a large number of cores, while in the latter we can only work on a limited set of data at a time. If we batch-processed the pipelines, we could instantiate multiple pipelines, each processing frames independently while avoiding the scalability issues that we experienced with foreman. In a streaming scenario, however, this is not possible.

The results shown in this paper are both unexpected and confusing. We had not anticipated such huge differences in how these modern processors perform with our workloads. With the standard parallel approach for such applications with either sequential execution of the pipeline or a work stealing approach, it is apparent that there is much performance to be gained by optimizing the order of operations for better cache usage. After all, we observe that the performance of the execution modes varies a lot with the behavior of a filter, e.g, amount of rotation applied by a filter. Moreover, it varies inconsistently with the number of threads, e.g., performance halved with two threads and sub-optimal processor affinity compared to a single thread. Thus, scheduling these optimally based on a-priori knowledge, we conjecture, is next to impossible, and profiling and feedback to the scheduler must be used to find the best configuration.

One option is to use profile-driven optimizations at compile time [5], where one or more configurations are evaluated and later used during runtime. This approach does, however, not work with dynamic content or tasks, i.e., if the parameters such as the rotation in pipeline B changes, the system must adapt to a new configuration. Further, the preferred configuration



(a) Sandy Bridge (b) Nehalem (c) Bulldozer

Figure 7. Execution times (ms) of pipeline B for the foreman video using different thread affinity strategies (lower is better).

may even change based on interactions between co-scheduled tasks, e.g., a CPU-bound and memory-bound task may perform better when co-scheduled than two memory bound tasks [8].

The preferred option then is to have a low-level scheduler that adapts dynamically to varying tasks, data, architectural limitations and shared resources. We propose an approach that uses instrumentation and allows a scheduler to gauge computation times of specific filters at execution time. Since the workloads we look at are periodic and long-running, the low-level scheduler can react to the observations and make appropriate adjustments, e.g., try different execution modes, data granularity, affinity and co-running competing workloads.

The prevalent approaches used in execution systems today are work stealing variants. Here, the system typically use a heuristics to increase cache locality such as spawning new tasks in the same queue; or stealing tasks from the back of another queue instead of the front to reduce cache contention, assuming tasks that are near in the queue are also near in terms of data. Although work stealing in its simplicity provides great flexibility, we have shown in this paper that execution order has a large performance impact on filter operations. For workloads that are long-running and periodic in nature, we can expect that an adaptive low-level scheduler will outperform the simple heuristics of a work stealing scheduler. An adaptive work stealing approach is feasible though, where work units are enqueued to the worker threads based on the preferred execution mode, data granularity and affinity, while still retaining the flexibility of the work stealing approach. Such a low-level scheduler is considered for further work. Another direction that we want to pursue is building synthetic benchmarks and looking at performance counters to pinpoint the cause of some of the effects that we are seeing, in particular with the Bulldozer microarchitecture.

IX. Conclusion

In this paper, we have looked at run-time considerations for executing (cyclic) streaming pipelines consisting of a data-intensive filters for media processing. A number of different filters and pipelines have been evaluated on a set of modern microarchitectures, and we have found several unexpected performance implications, within the well-known x86-family of microprocessors, like increased performance by backtracking pixel dependencies to increase cache locality for data sets that can fit in private cache; huge differences in performance when I/O is performed on one core and accessed on others; and different execution modes perform better depending on minor parameter variations in filters (or stages in the processing pipeline) such as the number of degrees to rotate an image. The implication of the very different behaviors observed on the different microarchitectures is a demand for scheduling that can adapt to varying conditions using instrumentation data collected at runtime. Our next step is therefore to design such a low-level scheduler in the context of our P2G processing framework [7].

Acknowledgments

We would like to thank Håkon Stensland for providing machines and insights to the differences observed. This work has been performed in the context of the *iAD* center for Research-based Innovation funded by the Norwegian Research Council.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proc. of SPAA*, pages 1–12, 2000.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. of SPAA*, pages 119–129, 1998.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of PPOPP*, pages 207–216, 1995.

[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[5] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, 1991.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[7] H. Espeland, P. B. Beskow, H. K. Stensland, P. N. Olsen, S. Kristoffersen, C. Griwodz, and P. Halvorsen. P2G: A framework for distributed real-time processing of multimedia data. In *Proc. of ICPPW*, pages 416–426, 2011.

[8] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):49–57, Feb. 2010.

[9] T. Foundry. Nuke. http://www.thefoundry.co.uk/products/nuke/.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of PLDI*, pages 212–223, Montreal, Quebec, Canada, June 1998.

[11] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proc. of SIGMOD*, pages 1123–1134, 2008.

[12] Intel Corporation. Threading building blocks. http://www.threadingbuildingblocks.org.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys*, pages 59–72, 2007.

[14] V. Kazempour, A. Fedorova, and P. Alagheband. Performance implications of cache affinity on multicore processors. In *Proc. of Euro-Par 2008*, pages 151–161. 2008.

[15] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. *SIGPLAN Not.*, 44(9):65–78, Aug. 2009.

[16] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proc. of CC*, pages 179–196, 2002.

[17] Ž. Vrba, P. Halvorsen, and C. Griwodz. A simple improvement of the work-stealing scheduling algorithm. In *Proc. of MuCoCoS*, pages 925–930, 2010.

[18] XIPH. *XIPH.org Test Media*, 2012. http://media.xiph.org/video/derf/.

[19] YouTube. Holy nyans! 60 hours per minute and 4 billion views a day on youtube. http://youtube-global.blogspot.com/2012/01/holy-nyans-60-hours-per-minute-and-4.html, Jan. 2012.