Online ISSN 1848-3380, Print ISSN 0005-1144
ATKAFF 55(3), 359–371(2014)

Josip Knezović, Igor Čavrak, Daniel Hofman

Parallelizing MPEG Decoder with Scalable Streaming Computation Kernels

DOI 10.7305/automatika.2014.12.617
UDK 621.397.13:004.272.43.042
IFAC 2.8.1

Original scientific paper

In this paper, we describe a scalable and portable parallelized implementation of a MPEG decoder using a streaming computation paradigm, tailored to new generations of multi–core systems. A novel, hybrid approach towards parallelization of both new and legacy applications is described, where only data–intensive and performance–critical parts are implemented in the streaming domain. An architecture–independent StreamIt language is used for design, optimization and implementation of parallelized segments, while the developed STREAMGATE interface provides a communication mechanism between the implementation domains. The proposed hybrid approach was employed in re–factoring of a reference MPEG video decoder implementation; identifying the most performance–critical segments and re–implementing them in StreamIt language, with STREAMGATE interface as a communication mechanism between the host and streaming kernel. We evaluated the scalability of the decoder with respect to the number of cores, video frame formats, sizes and decomposition. Decoder performance was examined in the presence of different processor load configurations and with respect to the number of simultaneously processed frames.

Key words: data streams, multicore, multimedia, parallel systems, stream computing, video decoding

Skalabilna implementacija dekodera po normi MPEG korištenjem tokovnog programskog jezika. U ovom radu opisujemo skalabilnu i prenosivu implementaciju dekodera po normi MPEG ostvarenu korištenjem paradigme tokovnog računarstva, prilagođenu novim generacijama višejezgrenih računala. Opisan je novi, hibridni pristup paralelizaciji novih ili postojećih aplikacija, gdje se samo podatkovno intenzivni i računski zahtjevni dijelovi implementiraju u tokovnoj domeni. Arhitekturno neovisni jezik StreamIt koristi se za oblikovanje, optimiranje i izvedbu paraleliziranih segmenata aplikacije, dok razvijeno sučelje STREAMGATE omogućava komunikaciju između domena implementacije. Predloženi hibridni pristup razvoju paraleliziranih aplikacija iskorišten je u preoblikovanju referentnog dekodera video zapisa po normi MPEG; identificirani su računski zahtjevni segmenti aplikacije i ponovno implementirani u jeziku StreamIt, sa sučeljem STREAMGATE kao poveznicom između slijedne i tokovne domene. Ispitivana su svojstva skalabilnosti s obzirom na ciljani broj jezgri, format video zapisa i veličinu okvira te dekompoziciju ulaznih podataka. Svojstva dekodera su praćena u prisustvu različitih opterećenja ispitnog računala, i s obzirom na broj istovremeno obrađivanih okvira.

Ključne riječi: podatkovni tokovi, višejezgrena računala, multimedija, paralelni sustavi, tokovno računarstvo, dekodiranje video zapisa

1 INTRODUCTION

Multimedia has permeated a multitude of application domains, ranging from mobile applications, entertainment systems, on–line multimedia processing environments and services, gaming, up to the highly specialized domains such as surveillance and security systems, medical imaging and analysis, remote imaging etc. Such diversity of domains dictates the implementation of multimedia algorithms on heterogeneous computing platforms of varying processing and storage capabilities; from smart phones and embedded systems with severely limited resources, to large clusters of servers with abundance of processing power. The constant advancement of applications, formats and algorithms poses two main challenges to hardware and software implementation aspects of multimedia systems: required processing power of underlying platforms and quick adoption of changes in existing or novel implementations of standards/algorithms. Current trends in addressing the processing power challenge utilize two approaches; the first approach relies on increasing raw processing power by employing multi– and many–core architectures, while

the second approach aims to maximize efficiency of algorithm implementations by realizing performance critical segments in platform specific assembly language.

Due to heterogeneity of existing and emerging computing platforms, portability and scalability of algorithm implementations are crucial for quick adoption and dissemination of new multimedia standards and algorithms. However, current programming practices of sequential execution approach and low–level algorithm optimization are in sharp contrast to required scalability and portability requirements [1] [2], effectively restricting each algorithm implementation to specific target architecture and failing to utilize processing power of novel multicore and many–core architectures. In addition, emergence and fast development of highly parallel computing systems require a shift in the programming model that can scale with the computational resources. To exploit those resources, it is necessary to utilize a mechanism enabling the programmer to explicitly express parallelism and to embed it into the abstract model of computation. The automatic extraction of parallelism, either by hardware or software, without a direct assistance of programmer and programming model has been proven as infeasible task [3] [4]. Explicit parallel programming models, languages and run–times [5] allow for load balancing, synchronization and partitioning to be automated by run–time systems. Those models enable more aggressive automated analysis and tailoring of programs to target computational infrastructure.

The streaming model of computation exposes parallelism and data dependencies by representing data processing in a form of a computational graph, with vertices (filters) representing atomic computations and arcs denoting data streams between computations. Raising the design abstraction level enables the exploitation of abundant parallelism in image and video processing applications [6], taking into account distributed address spaces and multiple control flows, as well as enabling programmers to explicitly expose communication and parallelism in programs. Portability and scalability properties are ensured by both high abstraction level of design and implementation specification (consequently hiding target architecture details), and by the ability to conduct automatic model analysis, transformation and synthesis of application code in order to optimally utilize target architecture resources.

In this paper we propose a novel approach towards building scalable and portable parallelized applications for multicore and many–core systems, and employ it in re–implementation of a MPEG–2 decoder application. Our approach is based on a functional decomposition of existing (non–parallelized) application or a newly designed application, where the most compute–intensive elements are (re)designed and (re)implemented in the streaming domain, whereas non–critical elements are left/implemented in the sequential domain. The streaming domain part of the application is implemented using StreamIt language [7], while the bridge between the streaming and the sequential domain is provided and managed by STREAMGATE interface and run–time [8].

The rest of this paper is organized as follows: Section 2 describes the current state of streaming–based computation models and run–time implementations for multicore and many–core systems, as well as their usage in multimedia domain. Section 3 describes the STREAMGATE, our modification of StreamIt language, compiler and run–time system for interfacing and integration of stream–based computation into general–purpose imperative domain. Section 4 presents our approach for implementation of a parallelized MPEG–2 decoder using StreamIt and STREAMGATE. Evaluation results and detailed analysis of the employed approach are given in Section 5. We conclude the paper with Section 6.

2 RELATED WORK

The idea of streaming model of computation has been present in computer science for quite a long time [9] [10]. Previous research on the streaming model was oriented towards scheduling of data flow graphs for digital signal processing [11], reactive systems [12] [13] [14] [15], or system modeling and prototyping [16] [17]. The idea of empowering the model's advantages for efficient and scalable programming of new multicore processors is, however, quite new [18] [19] [20]. This has led to the emergence of new streaming languages; supporting models for efficient mapping of data–driven parallel applications to novel computing platforms, ranging from graphics processing units to multicore processors, cluster of workstations or even embedded multiprocessor systems.

Brook [21], CUDA [22] and Sh [23] are oriented towards graphical processing units and thus designed with the intricacies that GPUs currently have, such as fixed pipelines, programming model issues etc. A program developed using one of the listed tools/models is tied to the target GPU platform and must be carefully designed to follow the appropriate data layout in the GPU memory.

OpenMP platform supports multithreaded programming with pragma lines introduced in serial C/C++ or Fortran code [24]. It is bound with the shared memory model and therefore suffers from scalability issues. Message Passing Interface (MPI) [25] is another concurrency platform based on message passing and aimed at large clusters of machines with local address spaces.

Accelerator, on the other hand, addresses data parallelism with data–parallel arrays and by transferring data–intensive tasks from C# to GPU. Commercial solutions such as RapidMind and PeakStream also of-

fer just–in–time based compilation strategy for offloading data–parallel tasks to graphical processing units in a streaming manner [26].

StreamIt [18] [7] [27] is a platform–independent programming language and compiler infrastructure that allows the design in a streaming model and exploitation of task, data and pipeline parallelism. The constructs in the language are especially crafted to allow structural representation of the stream graph with useful options of managing data, such as peeking on input channels. The theoretical foundation of the model assumes some very important invariants such as infinite stream of data to be processed and a relatively stable computational pattern expressed with stream graphs. These assumptions are not always met in real–world applications of the model; a more likely scenario is that only parts of a large application can be modeled in accordance with such strict prerequisites. Thus, the method for a seamless integration of the streaming model into other programming paradigms and with other models could prove beneficial; domain–specific models require efficient interfaces to more general models in order to be widely applicable.

MPEG–2 is a widely used video compression standard for progressive and interlaced coding, spanning a wide application area ranging from DVD video to video signal broadcasting [28] [29]. It is characterized with high complexity and incorporates hierarchical image data structure organized in several layers: video sequence, group of pictures (GOP), frame, slice, macro block and block layer.

Significant effort has been invested in parallelization of MPEG–2 encoder and decoder by various groups [30] [31] [32] [33]. It has been concluded that parallelism should be exploited at all available levels, from the GOP (group of pictures) level to the finer–grained macro block and block level. GOP level parallelization is trivial in case of independent GOPs, with issues arising due to load imbalance and random access problems for large GOPs. A coarse–grained parallelization approach at the slice level is desirable, with drawbacks such as excessive communication thus hindering the performance improvements.

Drake et al. experimented with the implementation of a complete MPEG–2 decoder in StreamIt language, primarily to demonstrate malleability of the proposed stream model to different application domains [34]. Their experience has shown that parts of the decoder algorithm with high dynamism, control–oriented data flow and irregular events exchanged between algorithm stages prevent usage of a "pure" data–flow streaming model. To alleviate the problem, a teleport messaging mechanism had been introduced, allowing out–of–band exchange of control messages between arbitrary stream graph filters [35].

Despite the fact that the MPEG–2 standard has been replaced by novel, upcoming standards such as H.264, the concepts for parallelization remain valid. Moreover, recent standardization efforts have increased computational complexity in order to achieve higher video quality under similar bitrate constraints. Therefore the research interest to map compute intensive parts of video compression standards onto parallel architectures [6,36,37]. The importance of parallelization is also evident in upcoming video coding standards such as HEVC, carefully designed with the parallelism in mind [38].

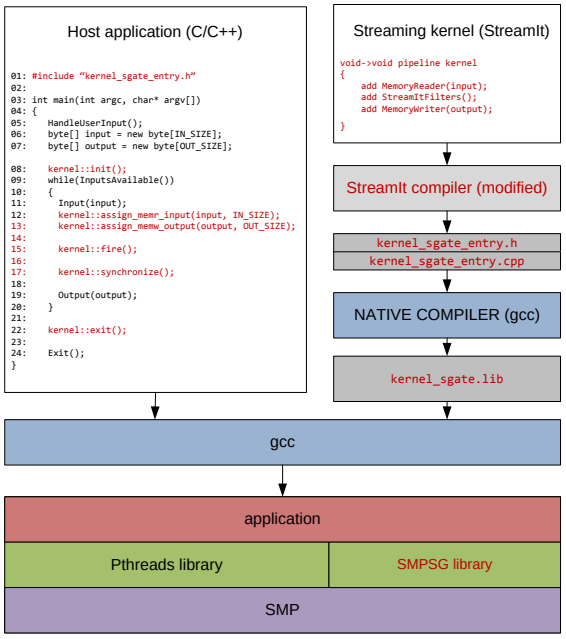

Fig. 1. STREAMGATE *overview: Hybrid host–kernel design and implementation process*

3 STREAMGATE INTERFACE

Most of the data–intensive applications prove inadequate for complete transformation into a pure streaming model, thus requiring parts of the implementation to retain a standard, sequential model of execution. Resulting in a hybrid approach, transformation of such applications mandates their careful functional decomposition and identification of functional components most amenable to stream based modeling and parallelization. Such components, in general, exhibit easily identifiable task, data and pipeline level parallelism, clear and stable data flow among functional sub–elements, and absence of upstream or downstream command flow. In order to justify the transformation effort with the increase of application performance, parallelized components should encapsulate only the most computationally demanding parts of the application.

To allow such a hybrid approach using StreamIt language, we have devised an interaction mechanism between

the sequential and the streaming parts of applications in the form of a STREAMGATE interface. The STREAMGATE interface defines syntax, mechanisms and semantics of interactions between the host and kernel segment of computation. In this approach, an application is decomposed into the sequential host part (main application code developed in a general purpose programming language) and one or more kernels (parallelized execution application segments implemented using StreamIt language) explicitly invoked from the host in an asynchronous manner.

Fig. 1 depicts the methodology and implementation steps of a hybrid single kernel C++/StreamIt application using the STREAMGATE interface. Implementation decomposition follows the functional decomposition of the streaming application: host (employing sequential execution paradigm) is implemented in the target implementation language (C/C++), while the kernel functionality is specified in StreamIt language. Explicit interaction between the host and the kernel part is expressed by library function invocations (red lines in host application code) and the STREAMGATE's *MemoryReader* and *MemoryWriter* filters (the first and last filter in the StreamIt code example). StreamIt kernel is compiled in the form of a static library, produced as a result of our modification of the StreamIt compilation tool chain. The generated library exposes a standard set of functions to be used in the host application code, providing mechanisms to initialize parallel computation resources (threads on the underlying multicore), exchange data between host and kernels, start the computation, synchronize execution of host and kernels, and to release allocated resources, all implemented through our SMPSG library.

The high–level StreamIt code is compiled using a modified StreamIt compiler into a backend–specific lower–level language with all the necessary synchronization boilerplate code, highly optimized with respect to available computing cores, reducing scheduling overhead in a purely Synchronous Data Flow model. Compilation and optimization process subjects the original stream model to multiple optimizations such as filter fusion, filter fission, pipeline and data parallelism identification. Following this device–independent process is a process of static scheduling and allocation of tasks to available target architecture processing units. The resulting C++ source code is generated and the static STREAMGATE kernel library (SMPSG), accompanied with StreamIt backend library (part of the original StreamIt tool chain), is assembled for inclusion in the main application executable code. The STREAMGATE and StreamIt backend libraries rely on low–level system resources available at run–time, such as low–level threading mechanisms (pthreads library), hiding the implementation details from system designers and developers.

While StreamIt assumes a single firing of stream computation as a monolithic application, the STREAMGATE encapsulates such computation in one or more kernels, allowing their multiple activations over the application life cycle. In a simplified view, the STREAMGATE defines three thread states (Fig. 2) and transitions between those states: *blocked*, *running* and *exiting*. Upon kernel initialization (*kernel_init()* function invoked by the host code), all kernel threads are created and placed in *blocked* state, minimizing inactive state kernel overhead. Invocation of *kernel_fire()* kernel function transitions the threads into *running* state where they remain until all the stream input data is consumed and stream computation completed (threads resume *blocked* state). Multiple firings of kernel processing during the host application life cycle are allowed, and all the allocated resources can be discarded by calling *kernel_exit()* function, when all the threads enter *exiting* state prior to their termination. Firing of streaming kernel is non–blocking but the kernel and host execution can be synchronized by using *kernel_synchronize()* function which blocks the host's thread execution until all the kernel's threads transition into *blocked* state, i.e. streaming kernel processes all available inputs.

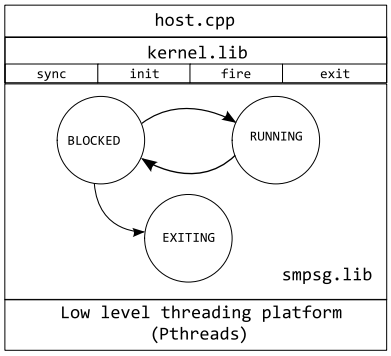

Fig. 2. STREAMGATE *kernel thread states and transitions*

Pipeline–level parallelism between the host and the kernel stages can be easily incorporated into the proposed host–kernel partitioning model. *MemoryReader* and *MemoryWriter* stream filters can be observed as boundaries between host and streaming application stages, forming a 3–stage pipeline at the application level (as illustrated in Fig.3). To exploit the application–level pipelining and even further increase the application performance, system designer is allowed to introduce additional processing in the main host thread, bounded by *kernel_fire()* and *kernel_synchronize()* calls to kernel functions, feeding the kernel with data in wavefronts. While one wavefront is being prepared for kernel processing (1), kernel computation can be performed of the previous data wavefront (2) and previous kernel processing results can be collected by the host (3) (Fig. 3). We support this high–level pipelining in

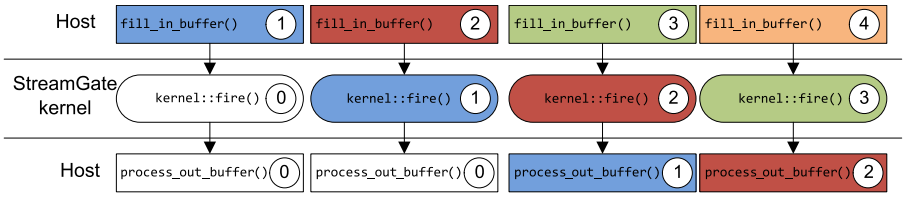

Fig. 3. Host–kernel pipelining

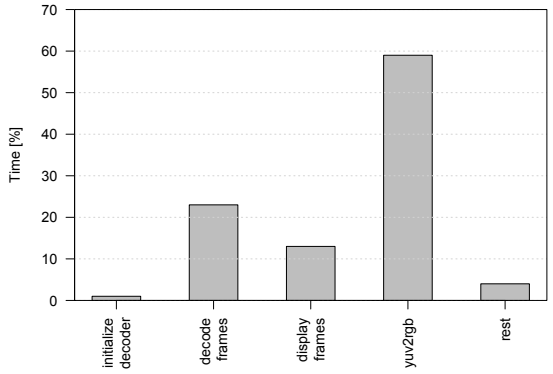

Fig. 4. Execution time distribution of MPEG–2 reference decoder implementation

the interface by double–buffering the *MemoryReader* and *MemoryWriter* exchange filters.

4 MPEG–2 DECODER IMPLEMENTATION

MPEG–2 is a widely used video compression standard for progressive and interlaced coding, and as such presents a typical multimedia application suitable for the hybrid approach. MPEG–2 encoder and decoder are hard or practically impossible to implement in the streaming domain using only data channels and strict producer–consumer relationships due to the dynamism, control oriented data flows and irregular events exchanged among some, usually independent stages of the algorithm [34].

Fig. 4 presents the execution time profiling results of the modified MPEG–2 decoder reference implementation (non–parallelized) from MediaBench benchmark suite [39]. Our modification of the reference implementation consisted of configuring the benchmark to use highly optimized integer based IDCT and 4:2:0 chroma format. The profile identified the yuv2rgb functional element to be the most computationally demanding — consuming around 60% of the processor time on chrominance components upsampling and YCrCb to RGB color space conversion. To increase the application performance, a hybrid re–factoring approach was selected, preserving the majority of functional components in sequential implementation domain, while transforming upsampling and conversion component (yuv2rgb) into streaming domain using StreamIt language and the STREAMGATE as an interface between the host and the kernel application segments.

The simplified pseudo–code in Listing 1 represents the implementation of the main MPEG–2 decoder application (host) using the hybrid approach. The STREAMGATE is used for managing data exchange and synchronization between the host application and the yuv2rgb functional element implemented in the streaming domain using the StreamIt language and run–time. The streaming subsystem kernel is initialized and resources allocated at line 10 of the example, lines 15 and 16 contain input and output shared buffer declarations paired with *MemoryReader* and *MemoryWriter* data exchange filters used in the streaming domain. Parallelized processing is asynchronously started at line 17, while line 18 blocks the main application thread until all processing within the kernel is finished. At line 22 all the resources allocated by the kernel are released.

Listing 1. Pseudo–code of host's main decoder function

```
1  #include "yuv2rgb_sgate_entry.h"
2
3  int mpeg2decode(int width,int height,int
       frames)
4  {
5    initialize_decoder();
6    int in_size = frames*width*height*3/2;
7    int out_size = frames*width*height*3;
8    byte* buf_in = new byte[in_size];
9    int*  buf_out = new byte[out_size];
10   yuv2rgb::init();
11
12   while(frames_available)
13   {
14     decode_frames(in_buffer);
15     yuv2rgb::assign_memr_buf_in(in_size)
16     yuv2rgb::assign_memw_buf_out(out_size)
17     yuv2rgb::fire();
18     yuv2rgb::sync();
19     display_frames();
20   }
21
22   yuv2rgb::exit();
23   delete[] buf_in;
24   delete[] buf_out;
25   exit_decoder();
26 }
```

The original stream graph of the color channel upsampling and conversion streaming kernel integrated into the

MPEG–2 decoder application, as perceived and designed by the developer, is shown in Fig. 5. The computation is performed at a coarse level, by processing individual video sequence frames. Data exchange between the main body of the decoder (host code) and the streaming kernel is established through the StreamGate's *MemoryReader* and *MemoryWriter* StreamIt filters. Data on the input side of the graph are raw pixels in 4:2:0 chroma format and YCrCb space, and the output is a stream of pixels in the RGB space and 4:4:4 chroma format. Data are delivered to the streaming module in chunks of varying size, parametrized with the number of frames, and the processing is repetitively fired to perform the required transformations.

The main body of the streaming conversion and upsampling kernel consists of two split–join constructs arranged in a producer–consumer relationship. The first split–join construct distributes incoming data in a round–robin fashion to a single forwarding luminance component and two components dedicated to upsampling chrominance components of decoded frames from 4:2:0 to 4:4:4 spatial resolution. The second split–join construct performs a color–space conversion from YUV to RGB by distributing duplicated input data to R, G and B converters. The preceding streaming model exhibits pipeline–level parallelism (sequential arrangement of input, two split–join and output functional blocks, sequential arrangements of 4:2:0 to 4:4:2 and 4:4:2 to 4:4:4 upsampling blocks), task–level parallelism among upsampling sub–modules and data–level parallelism among *stateless* filters which compiler freely exploits in order to utilize all the available cores.

Fig. 5 annotates filters with colors and their statically determined work estimates (performed as a step during StreamIt compilation). Filters colored in red perform significantly more work than blue colored filters. Work estimates are further used by the StreamIt compiler in order to decide on the transformations performed on the original stream graph such as filter fusion and fission, with the goal of producing a well balanced graph for deployment on target architecture.

One of the key properties of multimedia applications is the achievable constant data throughput, determining the performance of an application on a particular hardware platform. The streaming model allows designers and programmers to focus on high–level aspects of application or algorithm parallelism, abstracting away underlying processing infrastructure and implementation details. However, such initial design falls far from optimal usage of target architecture resources, in particular from achieving optimal processing granularity and allocation of tasks to available processing elements.

The task of supporting infrastructure is to analyze the original stream graph, identify types of parallelism present

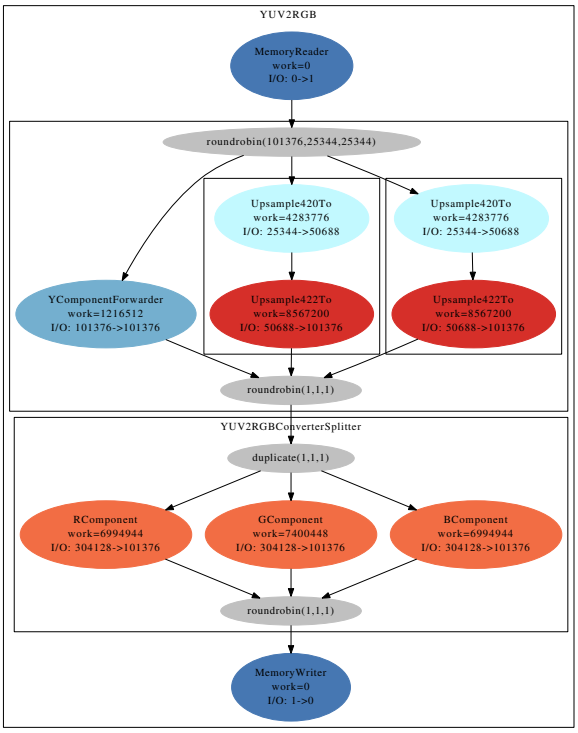

Fig. 5. StreamGate *MPEG–2 decoder implementation of color conversion and upsampler loop*

among filters, filter groups and branches, and conduct a series of graph and filter transformations. Resulting stream graph, together with generated source code, represents the optimal partitioning/grouping of filters and mapping of resulting tasks onto a target processing infrastructure, guaranteeing maximal data throughput for a particular application or application segment. Load balancing of transformed filters is particularly important when a significant amount of pipeline–level and task–level parallelism is present in the stream graph. For stream programs, the theoretically maximal obtainable speedup in throughput is given by Equation 1:

$$S_{max} = \frac{\sum_{i=1}^{N} W_i}{MAX_{i=1}^{N}(W_i)} \tag{1}$$

where W_i denotes the amount of processing performed by filter i, given there are N resulting filters in the transformed streaming graph and that all pipelined filters can be simultaneously mapped onto separate processing elements (processors, processor cores or virtual processing elements). If R_i denotes the relative amount of work of filter W_i, compared to total work by all filters in a stream, then:

$$R_i = \frac{W_i}{\sum_{i=1}^{N} W_i} \tag{2}$$

then the maximal achievable speedup S_{max} is:

$$S_{max} = \frac{1}{MAX_{i=1}^{N}(R_i)} \tag{3}$$

Parallelized multimedia algorithms predominantly contain two types of parallelism: data–level and pipeline–level. Data–level parallelism is present in various types of stateless data filtering and transformation operations, while pipeline–level parallelism is a result of a series of inherently sequential, possibly interdependent operations realized in a form of a chain of stateful filters in a producer–consumer relationship. While data–level parallelism inherently ensures balanced workload among replicated filters, pipeline–level parallelism, at both application scope and individual stream branch scope, must be carefully managed by the supporting infrastructure.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.164 & 0 & 1.596 \\ 1.164 & -0.391 & -0.813 \\ 1.164 & 2.018 & 0 \end{bmatrix} \begin{bmatrix} Y - 16 \\ U - 128 \\ V - 128 \end{bmatrix} \tag{4}$$

Listing 2. YUV2RGB StreamIt filter

```
1  int->int filter YUV2RGB
2  {
3    work pop 3 push 3
4    {
5      int Y = pop()-16;
6      int U = pop()-128;
7      int V = pop()-128;
8
9      int R = (76309*Y+104597*V)>>16;
10     int G = (76309*Y-25675*U-53279*V)>>16;
11     int B = (76309*Y-132201*U)>>16;
12
13     push(R);
14     push(G);
15     push(B);
16   }
17 }
```

5 EXPERIMENTAL RESULTS

Listing 2 presents a simplified specification of the parallelized YUV to RGB color space conversion filter defined by Equation 4, and is a fraction of the complete StreamIt specification of the MPEG–2 decoder kernel. The processing is defined in a form of a stateless StreamIt filter consuming three integers from input and producing three integers on its output in one filter firing (lines 1 and 3). Such a filter is an ideal candidate for fission (utilizing stateless nature of computation and inherent data parallelism between calculations of R, G and B values) and fusion with more computation–demanding filters. As long as filter fusion results in stateless filter, streaming compiler can data–parallelize resulting coarsened fused filter in order to utilize available processing cores.

Test	Resolution	Chroma	Frame rate
QCIF	176x144	4:2:0	25
CIF	352x288	4:2:0	30
SD	720x576	4:2:0	25
HD	1280x720	4:2:0	25

Table 1. Properties of test video sequences

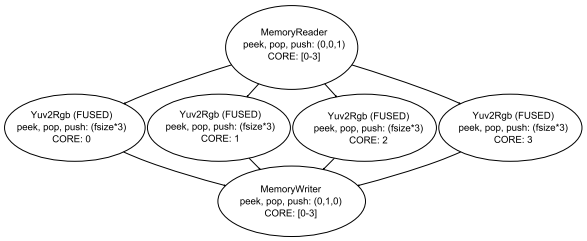

Fig. 6. Partitioned stream graph for StreamGate *kernel in MPEG–2 decoder (SMP = 4)*

To test the viability of our approach, we evaluated the performance of a parallelized MPEG–2 decoder on a set of test video sequences. In particular, we were interested in investigating the portability and scalability of the hybrid (host–kernel decomposition) approach to different processor configurations and the possible performance limitations of StreamIt based MPEG–2 decoder with regards to core utilization, overall system load and the number of frames processed by the decoder at one kernel firing.

The primary indicator of performance used in the evaluation was the decoder throughput, i.e. the amount time required by the decoder to fully decode a video sequence of known resolution and size. The set of test video sequences, along with their key properties, is presented in Table 1. The first two test videos (QCIF and CIF formats) resembled the typical video resolutions used on embedded and constrained devices featuring low–power multicore CPUs. The remaining two test videos represented a wider image formats suited for high–resolution devices. The evaluation environment consisted of a personal computer with a single 2.4 GHz Intel Core 2 Quad Q6600 processor and 4 GB of main memory, Linux kernel version 3.9. Additional experiments were conducted using the Sniper simulator – a multicore simulator based on the interval core model [40].

5.1 Scalability

Fig. 6 contains the resulting, partitioned stream graph of the *yuv2rgb* StreamGate kernel targeted to four cores. Compared to the original, programmer perceived stream graph from Fig. 5, the streaming compiler performed fusion (filter coalescing) and fission (replication of

balanced filters) transformations in order to produce four well–balanced filters encapsulated in the top–level split–join construct with the *MemoryReader* as the source and *MemoryWriter* as the final stream consumer.

Since a coarse–grained processing strategy was employed in the kernel by processing video at the frame level, we evaluated the scalability of our approach for different frame sizes. We also measured and compared the throughput of the decoder parametrized with the running number of cores. Other domain specific streaming optimizations incorporated in the StreamIt compiler were not used in our experiments. All throughput tests were run 20 times and resulting mean values were recorded.

The set of tested StreamIt kernels included four *SMP* (symmetric multiprocessing) kernels including stream graph transformations and optimizations and utilizing 1–4 processing cores, as denoted by the number in the kernel name. In order to test the effect of increased processor load on performance of generated n–core StreamIt kernels, two processor load types were used in the experiments: *LL* postfix denoting light processor load configuration (only OS–related processes active in the system, all cores below 5% utilization) and *HL* postfix denoting heavy load (at least 15% and at most 25% utilization of at least two cores prior to decoder run). *SMP1* kernel performance tests were always conducted in the *LL* configuration.

Table 2 presents the average decoding speedups for a set of test video sequences, normalized to the SMP1LL kernel performance, where decoding performance is measured for a set of generated StreamIt kernels and two configurations of processor loads. If comparing decoder performances under *LL* configuration, speedup is almost linear with respect to the number of cores utilized by kernels (Fig. 7), independent of the test video sequence used. A more detailed analysis shows that the mean speedup gain between *SMP3* and *SMP4* is higher than speedup gain between *SMP2* and *SMP3* kernels (0.40 : 0.23). On the other hand, *HL* configuration reveals an impact of high processor utilization on performance of highly parallelized kernels (Fig. 8) where linear increase in performance with respect to the number of cores is no longer present. This effect is especially visible for *SMP4* kernel (Fig. 9), whose performance under *HL* configuration is lower than *SMP3*'s, and in the case of CIF video sample, lower than *SMP2*'s performance. The mean speedup gain between kernels *SMP2* and *SMP3* remains the same as for the *LL* configuration, but the gain between *SMP3* and *SMP4* becomes negative (0.23 : -0.1).

Relatively minor effects of *HL* configuration on performances of *SMP2* and *SMP3* kernels can be explained by the number of processing cores available on the test system and the low–level details of StreamIt SMP backend implementation. *SMP2* and *SMP3* kernel implementations, as

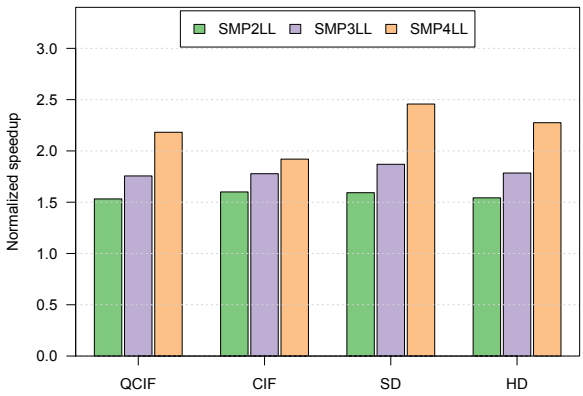

Fig. 7. Kernel speedups under LL configuration, normalized to SMP1 kernel performance

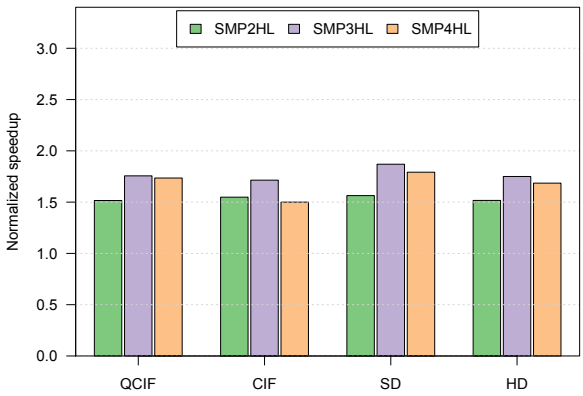

Fig. 8. Kernel speedups under HL configuration, normalized to SMP1 kernel performance

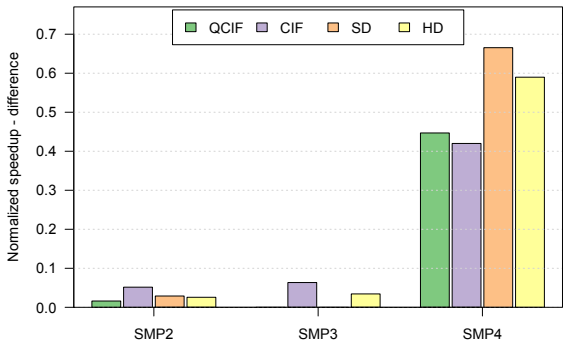

Fig. 9. Differences between LL and HL normalized speedups

Test	SMP2LL	SMP2HL	SMP3LL	SMP3HL	SMP4LL	SMP4HL
QCIF	1.53	1.52	1.75	1.76	2.18	1.73
CIF	1.60	1.55	1.78	1.71	1.92	1.50
SD	1.59	1.56	1.87	1.87	2.46	1.79
HD	1.54	1.52	1.78	1.75	2.28	1.69
MEAN	1.57	1.54	1.80	1.77	2.20	1.67

Table 2. Kernel speedups, normalized to SMP1LL kernel performance

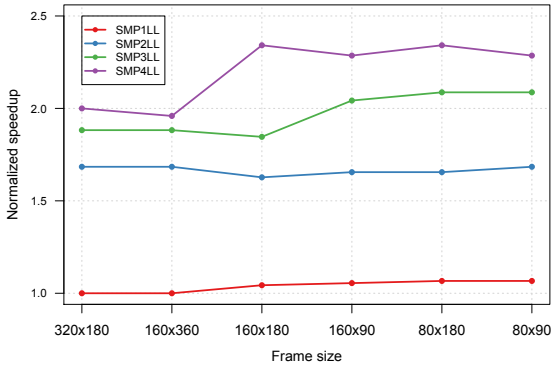

Fig. 10. Simulated kernel speedups, normalized to SMP1LL kernel performance on 320×180 *HD subframe*

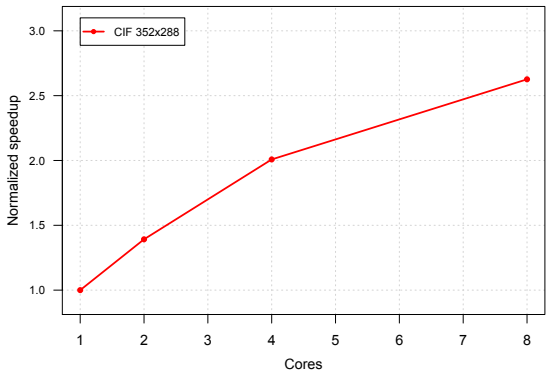

Fig. 11. Kernel speedups on 2,4 and 8 cores (Sniper), CIF video sequence, normalized to SMP1 kernel performance

currently mandated by StreamIt compiler and runtime, utilize two or three threads to parallelize tasks. Under significant load, operating system can still allocate SMP kernel threads to available cores in a quad–core processor, ensuring that the processing time wasted at the thread synchronization barrier is minimal. However, *SMP4* kernel implementation requires allocation of four threads to individual cores, which leads to allocation of at least one thread to a significantly loaded core, delayed execution due to preemption and significant time wasted by fast–performing threads to synchronize with the slow–performing thread at the *spinlock*–based synchronization barrier.

Additional simulations using the Sniper simulator were conducted to further evaluate decoder scalability on more than 4 cores [40]. CIF test video sequence from previous evaluation was used, and the simulated system was 2/4/8–core Pentium M 2.66GHz, 4GB RAM, 32KB L1 cache, 32KB L2 cache, 8MB L3 cache, no operating system or applications present. The results, normalized to SMP1 kernel performance, show constant positive speedups on systems up to 8 simulated cores (Fig. 11), proving good scalability of the MPEG–2 decoder kernel implementation.

5.2 Data Decomposition

In the previous experiment, during testing of SD and HD videos, internal buffering in the compiled stream graph exhausted the process stack space. To avoid the issue, input frame data had been further decomposed into subframes: QCIF subframes for SD and CIF subframes for HD test sequence. Using this approach, the buffer sizes among stream filters were reduced, but the more fine–grained approach also resulted in improved performance of streaming kernels. Decreasing required buffering among filters lowered the memory requirements, most notably the program stack used by internal buffers. In theory, placing large internal buffers on heap could allow more general usage thus lowering the overall performance. However, the experiments revealed performance degradation in some cases reaching as high as 30% compared to implementations using stack space, proving the approach useless.

Further experimental study of the effect of data decomposition on decoder performance was conducted by using a 1280×720 HD video sequence, partitioned into different subframe sizes, as input data. Table 3 presents the subframe sizes and achieved speedups for different versions of decoder kernels, relative to SMP1 kernel performance on 320×180 subframe. Kernel performances were tested only under the *LL* configuration. A significant performance difference was visible for all parallelized kernels compared to the non–parallelized kernel *SMP1*. *SMP3* and *SMP4* kernels exhibited visible performance improvements with the decrease of subframe size, higher than expected given the Amdahl's law (Fig. 10). Although a more thorough analysis of the underlying mechanisms is needed for detailed

	320 × 180	160 × 360	160 × 180	160 × 90	80 × 180	80 × 90
SMP1LL	1.00	1.00	1.04	1.05	1.07	1.07
SMP2LL	1.68	1.68	1.63	1.66	1.66	1.68
SMP3LL	1.88	1.88	1.85	2.04	2.09	2.09
SMP4LL	2.00	1.96	2.34	2.29	2.34	2.29

Table 3. Kernel speedups, normalized to SMP1LL kernel performance on 320 × 180 *HD video subframe*

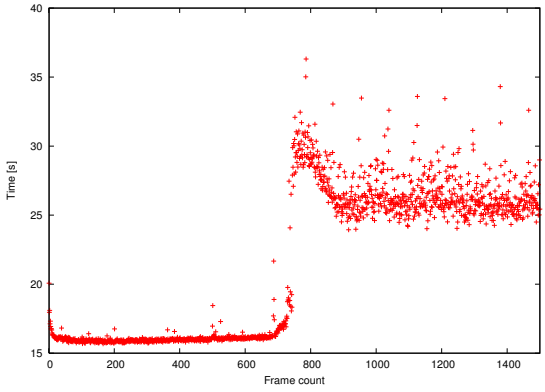

Fig. 12. Execution time vs. STREAMGATE *buffer size (in number of frames)*

explanation of the effect, initial findings suggest that, in addition to high–level parallelization of processing, clever data decomposition can also benefit decoder performance.

5.3 Frame group sizes

Fig. 12 presents the results of the experimental evaluation of the total CIF video sample decoding time versus the number of frames decoded on one kernel firing. The size of frame groups provided to one kernel firing varied from 1 to 1500. For small groups (1–10 frames per group), the large number of kernel invocations introduced a significant overhead and increased the total decoding time, compared to mid–sized groups. For the group sizes above 700, decoding times increased sharply as a result of increased time spent by the OS in performing housekeeping operations. The minimal decoding times for this experiment were obtained for groups sized around 150 frames, accounting for approx. 60 MB of the total memory allocated on data exchange buffers. Moreover, execution times exhibited small variations for group sizes smaller than optimal, down to 10 frames per group, accounting for approx. 4 MB of the data exchange buffer size.

While the smaller frame groups require significantly less memory for implementation of intra–filter private buffers, the number of kernel invocations is proportionally higher. On the other hand, large frame groups require less kernel invocations, but present significantly higher memory demands, resulting in far more frequent occurrences of cache misses and page faults, thus drastically reducing application performance. Performance degradation threshold is system specific and dependent on the available system memory. For resource constrained systems with small amount of main memory, StreamIt implementation of MPEG–2 decoder would not be suitable, as the performance would be bounded by both the number of kernel firings and the amount of memory available for buffering.

6 CONCLUSION

Current and upcoming processor architectures are characterized with the inherent parallelization of processing, predominantly due to increased number of processing cores. Another trend in computing landscape is the rising importance of data–intensive applications, requiring high processing power and presenting natural targets for various types of performance optimizations.

In this paper we demonstrated the benefits of using streaming programming model for high–level re–factoring of data–intensive parts of legacy programs. We proposed an interface and a tool which enable integration of streaming kernels, providing portable and scalable performance with the increase of processing cores. Kernels are expressed in a high–level stream programming language, allowing programmers to explicitly model computation and communication dependencies. In this way, streaming compiler is able to perform aggressive parallelization and efficiently map kernels to target multicore architectures. Tedious tasks of partitioning and load balancing are left to the compiler who has all the information available (computation and communication dependencies) to perform it in a scalable and efficient manner.

Our interface enables incremental parallelization of legacy programs by isolating data–intensive parts into streaming kernels. The interface allows iterative firing of streaming kernels, executed in parallel using available processing cores. The viability of the approach was tested on a parallelized MPEG decoder implementation with a single streaming kernel encapsulating color conversion and upsampling computations.

Current interface implementation allows only static declarations of input and output data rates in filter defi-

nitions (number of consumed and produced tokens in single kernel firing), restricting usability of the proposed approach in real–world programs. Introduction of dynamic data rate support, thus enabling filters to consume and produce a number of tokens and control messages on single kernel firing unknown at compile–time, would offer flexibility and more general use of stream programming model. However, such approach would restrict the effectiveness of static filter analysis and resulting optimizations of streaming computation graph. An approach to overcome the drawbacks of dynamic data rates is proposed in [41], based on separation of static and dynamic domains by partitioning the stream program into static subgraphs separated by dynamic boundaries. Each subgraph then undergoes aggressive compile–time optimization and is allocated to a dedicated run–time thread during kernel firing.

REFERENCES

[1] D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe, "A lightweight streaming layer for multicore execution," *SIGARCH Comput. Archit. News*, vol. 36, pp. 18–27, May 2008.

[2] A. D. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin, "Soc-c: efficient programming abstractions for heterogeneous multicore systems on chip," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, (New York, NY, USA), pp. 95–104, ACM, 2008.

[3] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," *SIGPLAN Not.*, vol. 44, pp. 177–187, June 2009.

[4] S. Rul, H. Vandierendonck, and K. De Bosschere, "A profile-based tool for finding pipeline parallelism in sequential programs," *Parallel Comput.*, vol. 36, pp. 531–551, Sept. 2010.

[5] D. B. Skillicorn and D. Talia, "Models and Languages for Parallel Computation," *ACM Computing Surveys*, vol. 30, pp. 123–169, 1996.

[6] C. Meenderinck, A. Azevedo, B. Juurlink, M. Alvarez Mesa, and A. Ramirez, "Parallel scalability of video decoders," *J. Signal Process. Syst.*, vol. 57, pp. 173–194, Nov. 2009.

[7] W. Thies, *Language and Compiler Support for Stream Programs*. Phd thesis, Massachusetts Institute Of Technology, Cambrige, MA, 2009.

[8] J. Knezović, M. Kovač, and H. Mlinarić, "Integrating streaming computations for efficient execution on novel multicore architectures," *AUTOMATIKA: Journal for Control, Measurement, Electronics, Computing and Communications*, vol. 51, pp. 387–396, March 2011.

[9] R. Stephens, "A Survey of Stream Processing," *Acta Informatica*, vol. 34, no. 7, 1997.

[10] W. M. Johnston, P. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, pp. 1–34, March 2004.

[11] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[12] P. Guernic, A. Benveniste, P. Bournai, and T. Gautier, "Signal – a data flow–oriented language for signal processing," in *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 34(2), pp. 362–374, 1986.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," in *Proceedings of the IEEE*, pp. 1305–1320, 1991.

[14] G. Berry, G. Gonthier, A. B. G. Gonthier, and P. S. Laltte, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," 1992.

[15] G. Delaval, A. Girault, and M. Pouzet, "A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs," in *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, (Tucson, Arizona), June 2008.

[16] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Tamming Heterogeneity – the Ptolemy Approach," in *Proceedings of the IEEE*, vol. 91, pp. 127–144, 2003.

[17] P. K. Murthy, E. G. Cohen, and S. Rowland, "System canvas: a new design environment for embedded dsp and telecommunication systems," in *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, (New York, NY, USA), pp. 54–59, ACM, 2001.

[18] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings 2002 International Conference on Compiler Construction*, 2002.

[19] S.-w. Liao, Z. Du, G. Wu, and G.-Y. Lueh, "Data and Computation Transformations for Brook Streaming Applications on Multiprocessors," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, (Washington, DC, USA), pp. 196–207, IEEE Computer Society, 2006.

[20] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, May 2008.

[21] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Mike, and H. Pat, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics*, vol. 23, pp. 777–786, August 2004.

[22] J. Nickolls and I. Buck, "CUDA Software and GPU Parallel Computing Architecture," *Microprocessor Forum*, May 2007.

[23] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, (New York, NY, USA), pp. 787–795, ACM, 2004.

[24] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*. Cambridge, MA, USA: MIT Press, 2nd. (revised) ed., 1998.

[26] M. D. McCool, "Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform," in *GSPx Multi-core Applications Conference*, 2006.

[27] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies, "Language and compiler design for streaming applications," *Int. J. Parallel Program.*, vol. 33, no. 2, pp. 261–278, 2005.

[28] *ISO/IEC 13818: Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s*. International Organization for Standardization, 1999.

[29] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards*. Kluwer Academic Publishers, 1995.

[30] I. Assayad, P. Gerner, S. Yovine, and V. Bertin, "Modelling, analysis and parallel implementation of an on-line video encoder," in *1st International Conference on Distributed Frameworks for Multimedia Applications (DFMA 2005), 6-9 February 2005, Besançon, France*, pp. 295–302, 2005.

[31] T. Jacobs, V. Chouliaras, and D. Mulvaney, "Thread-parallel MPEG-2, MPEG-4 and H.264 video encoders for SoC multi-processor architectures," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 1, pp. 269–275, 2006.

[32] E. Iwata and K. Olukotun, "Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm," Tech. Rep. CSL–TR–98–771, Stanford University, 1998.

[33] A. Bilas, J. Fritts, and J. P. Singh, "Real–Time Parallel MPEG–2 Decoding in Software," in *In Proceedings of the 11th International Parallel Processing Symposium*, pp. 197–203, 1997.

[34] M. Drake, H. Hoffman, R. Rabbah, and S. Amarasinghe, "MPEG-2 decoding in a stream programming language," in *Proc. International Parallel and Distributed Processing Symposium*, (Rhodes Island, Greece), 2006.

[35] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, "Teleport messaging for distributed stream programs," in *Symposium on Principles and Practice of Parallel Programming*, (Chicago, Illinois), Jun 2005.

[36] B. Pieters, C.-F. Hollemeersch, J. De Cock, P. Lambert, W. De Neve, and R. Van De Walle, "Parallel deblocking filtering in mpeg-4 avc/h.264 on massively parallel architectures," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 21, no. 1, pp. 96–100, 2011.

[37] K. Choi and E. S. Jang, "Leveraging parallel computing in modern video coding standards," *IEEE MultiMedia*, vol. 19, no. 3, pp. 7–11, 2012.

[38] G. J. Sullivan, J.-R. Ohm, W. Han, and T. Wiegand, "Overview of the high efficiency video coding (hevc) standard," *IEEE Trans. Circuits Syst. Video Techn.*, vol. 22, no. 12, pp. 1649–1668, 2012.

[39] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO 30: 30th annual ACM/IEEE international symposium on Microarchitecture*, (Washington, DC, USA), pp. 330–335, IEEE Computer Society, 1997.

[40] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and

accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[41] R. Soule, M. I. Gordon, S. Amarasinghe, R. Grimm, and M. Hirzel, "Dynamic expressivity with static optimization for streaming languages," in *The 7th ACM International Conference on Distributed Event-Based Systems*, (Arlington, TX), June 2013.

Josip Knezović received his B.Sc., M.Sc. and Ph.D. degree in Computer Science from the Faculty of Electrical Engineering and Computing, University of Zagreb in 2001, 2005 and 2009, respectively. Since 2001 he has been affiliated with Faculty of Electrical Engineering and Computing. He is assistant professor at the Department of Control and Computer Engineering. His research interests include programming models for parallel systems in multimedia, image and signal processing. He is a member of IEEE and ACM.

Igor Čavrak is an assistant professor at the Department of Control and Computer Engineering, University of Zagreb, Faculty of Electrical Engineering and Computing (FER). He received his M.Sc. and Ph.D. degrees in computer science from FER in 2001 and 2006. He joined University of Zagreb in 1996, and since then has been involved in various educational and research activities. His research interests include pervasive computing, distributed intelligent systems and software engineering. He published more than 20 papers in journals and conference proceedings, and is a member of ACM, IEEE and KES.

Daniel Hofman graduated electrical engineering at University of Zagreb, Faculty of Electrical Engineering and Computing (FER) in 2008. Since then he has been working as a research assistant at the Department of Control and Computer Engineering at FER and pursuing his PhD at the same Faculty. His research focus is on Network Algorithms for Video Coding on Multi-core Architectures. He published numerous papers in journals and proceedings. He is an active member of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) and Institute of Electrical and Electronics Engineers (IEEE).

AUTHORS' ADDRESSES

Assist. Prof. Josip Knezović, Ph.D.
Assist. Prof. Igor Čavrak, Ph.D.
Daniel Hofman, B.Sc.
Department of Control and Computer Engineering,
Faculty of Electrical Engineering and Computing,
University of Zagreb,
Unska 3, HR-10000 Zagreb, Croatia
email: {josip.knezovic, igor.cavrak, daniel.hofman}@fer.hr

Received: 2013-07-17
Accepted: 2014-04-30