Online ISSN 1848-3380, Print ISSN 0005-1144
ATKAFF 53(3), 272–283(2012)

Josip Knezović, Hrvoje Mlinarić, Martin Žagar

Lossless Image Compression Exploiting Streaming Model for Efficient Execution on Multicores

DOI 10.7305/automatika.53-3.265
UDK 004.415.3.032.6:004.272.43.032.24
IFAC 2.8.3; 4.0.5

Original scientific paper

Image and video coding play a critical role in present multimedia systems ranging from entertainment to specialized applications such as telemedicine. Usually, they are hand–customized for every intended architecture in order to meet performance requirements. This approach is neither portable nor scalable. With the advent of multicores new challenges emerged for programmers related to both efficient utilization of additional resources and scalable performance. For image and video processing applications, streaming model of computation showed to be effective in tackling these challenges. In this paper, we report the efforts to improve the execution performance of the CBPC, our compute intensive lossless image compression algorithm described in [1]. The algorithm is based on highly adaptive and predictive modeling, outperforming many other methods in compression efficiency, although with increased complexity. We employ a high–level performance optimization approach which exploits streaming model for scalability and portability. We obtain this by detecting computationally demanding parts of the algorithm and implementing them in StreamIt, an architecture–independent stream language which goal is to improve programming productivity and parallelization efficiency by exposing the parallelism and communication pattern. We developed an interface that enables the integration and hosting of streaming kernels into the host application developed in general–purpose language.

Key words: Lossless image compression, Image coding, Stream programming, Parallel programming, Multicores

Kompresija slika bez gubitaka uz iskorištavanje tokovnog modela za izvođenje na višejezgrenim računalima. Postupci obrade slikovnih podataka su iznimno zastupljeni u postojećim multimedijskim sustavima, počev od zabavnih sustava pa do specijaliziranih aplikacija u telemedicini. Vrlo često, zbog svojih računskih zahtjeva, ovi programski odsječci su iznimno optimirani i to na niskoj razini, što predstavlja poteškoće u prenosivosti i skalabilnosti konačnog rješenja. Nadolaskom višejezgrenih računala pojavljuju se novi izazovi kao što su učinkovito iskorištavanje računskih jezgri i postizanje skalabilnosti rješenja obzirom na povećanje broja jezgri. U ovom radu prikazan je novi pristup poboljšanja izvedbenih performansi metode za kompresiju slika bez gubitaka CBPC koja se odlikuje adaptivnim modelom predviđanja koji omogućuje postizanje boljih stupnjeva kompresije uz povećanje računske složenosti [1]. Pristup koji je primjenjen sastoji se u implementaciji računski zahtjevnog predikcijskog modela u tokovnom programskom jeziku koji omogućuje paralelizaciju izvornog programa. Ovako projektiran predikcijski model može se iskoristiti kroz sučelje koje smo razvili a koje omogućuje pozivanje tokovnih računskih modula i njihovo paralelno izvođenje uz iskorištavanje više jezgri.

Ključne riječi: Kompresija slika bez gubitaka, kodiranje slikovnih podataka, tokovni računalni model, paralelni sustavi, višejezgrena računala

1 INTRODUCTION

Many present–day applications rely on acquisition, processing, archival and retrieval of visual data. Those data, when transmitted and stored, require a vast amount of storage space or communication bandwidth if processed uncompressed. Compression is therefore desirable, especially in the embedded devices with sparse storage as well as in internet–enabled devices with limited connectivity. Many applications that deal with visual data, such as medical imaging and diagnostics, geological imaging, satellite and remote sensing, pre–press imaging, image archival and biometrics based on visual data, require no loss in the compression phase, thus utilizing lossless compression methods.

Past and recent research on lossless compression for images has focused on the predictive contextual modeling in order to extract redundancy present in typical images [1–5]. Image is first treated by a prediction phase

where pixels are predicted based on some information about source, such as correlation among neighboring pixels, edges, planar structures, textures etc. Prediction accuracy is crucial in image decorrelation, and therefore has been the main subject of extensive research in the field. Some algorithms, such as CALIC [3], or JPEG–LS [6], use simple static or switching predictors and therefore lack the robustness in the cases of abrupt changes in image region properties, or more complex image structures such as nontrivial edges, textures with complex properties etc. Other proposals deploy highly adaptive predictors such as the least squares based predictors [5, 7], neural networks [8], two pass linear prediction [9] etc. These approaches offer improved compression efficiency with the compromise in extremely high computational requirements. Predictive modeling has also found applications in other image processing areas such as edge detection [10], high dynamic range image compression [11], image archival [12] etc.

Nowadays, computer–architecture has gone to the revolutionary shift into the multicore and manycore era. The reason for this change is due to the fact that the instruction–level parallelism exploited thus far has reached the point of diminishing returns increasing the power/performance ratio beyond the acceptable level. The multicore and manycore path brings a new set of challenges to the computing system designer that were not present before: the parallelism and efficient parallel implementation of programs. Since the number of cores on the chip is going to raise as mandated by Moore's law, automatic extraction of parallelism, either by hardware or software, without a direct assistance of the programmer and programming model will become a daunting and infeasible task. Therefore, an increased interest in explicit parallel programming models, languages and runtimes has emerged. From the viewpoint of predictive lossless image compression, increased computational capability of multicores can be efficiently used by parallel execution of parts of algorithms that are parallelizable and computationally demanding, notably adaptive predictor modeling. Moreover, streaming nature of these computations enable their implementation in a high–level streaming language such as StreamIt which exposes task, data and pipeline parallelism and enables portability and scalability, features not possible with old–fashioned programming practices which optimize in low level architecture–specific languages [13, 14].

In this paper we concentrate on performance improvements of our proposed lossless image compression algorithm named CBPC after the term Classification and Blending Predictive Coding algorithm [1]. While our previous publications on CBPC compression concentrate on the problem of highly adaptive predictive coding of images from the viewpoint of achieving high compression ratio, here we address the execution time of the algorithm. Moreover, our goal is to use a high–level streaming language in the performance optimization in order to make a portable and scalable implementation of the codec which would scale well with the increase of number of the cores. Additionally, our intention was to enable the portability among different architectures, from symmetric multicores with shared memory to heterogeneous multiprocessor architectures which rapidly emerge in the embedded computing devices. In order to end up with a scalable and portable implementation we exploit streaming compilation infrastructure that we described previously in [15], which we further extended in order to generate reusable streaming kernels in the form of static library. This approach enabled us to obtain scalable performance improvement in regard to increasing the number of processing cores without any modifications to existing source. The only step required for the whole program to be sped up was eventual recompilation of the program targeting different number of processing threads.

The rest of this paper is organized as follows. Section 2 describes the CBPC compression algorithm. Section 3 presents our approach where a high–level performance aware implementation of the most demanding parts of the algorithm in the StreamIt programming language is described. Results are given in Section 4. Finally, we draw the conclusions in Section 5.

2 CBPC COMPRESSION METHOD

In this section we briefly describe our proposed CBPC compression method, details can be found elsewhere [1, 16]. Fig. 1 shows the basic steps of CBPC compression algorithm. Image is treated as two-dimensional array $I(x, y)$ of pixel grey intensity values with the width W and the height H, where $0 \leq x < W$ and $0 \leq y < H$. Pixels are observed sample by sample in raster scan order, from top to bottom, left to right. In assumed backward adaptive approach, the encoder is allowed to use only past information that is also available to decoder. This means that for forming the prediction only previously observed pixels are used. In fact, only a subset of previously encoded pixels is used to form the causal template. In order to efficiently model different image structures we propose adaptive predictor based on the idea of predictor blends [17]. The blending predictor is extended with dynamic determination of blending context on a pixel–by–pixel basis. The set of predictors to be blended $\mathcal{F} = \{f_1, f_2, \cdots, f_N\}$ is composed of N static predictors adjusted to predict well in the presence of specific property. For example simple predictor $f_W = I(x-1, y)$ is known to predict well in the presence of sharp horizontal edge. The classification process determines the set of neighboring pixels on which the blending of $\mathcal{F}$ is performed. It is similar to initial step of

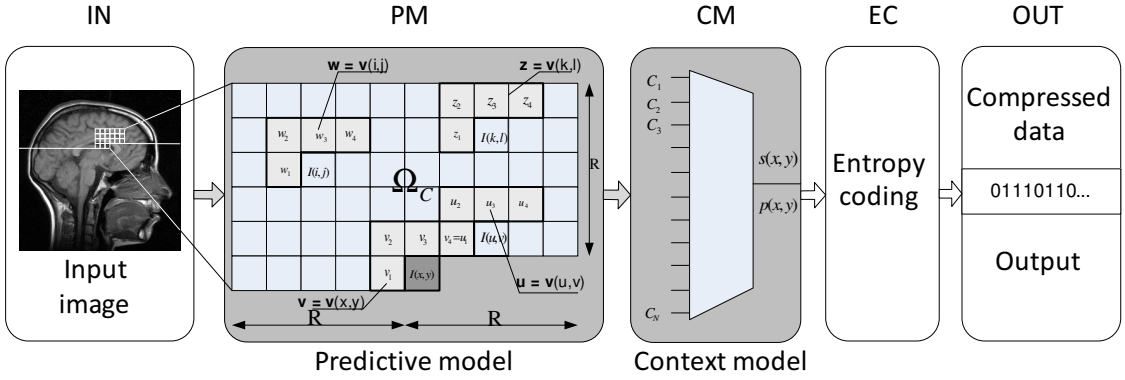


Fig. 1. Basic blocks of the CBPC compression method

VQ design substantially simplified in order to be usable in symmetric, backward adaptive algorithm [18].

As illustrated in Fig. 1 pixels are first processed by a predictive model denoted by PM. As already said, the goal of this step is to decorrelate image and thus remove statistical redundancies in order to allow more efficient entropy codes for image data, i.e. prediction errors. After prediction step, remaining redundancies are removed by a contextual modeling CM step in which, a state, i.e. context for probability estimation of prediction error is computed. The state and the error are further sent to entropy coder step denoted by EC in order to produce a variable length code for the current pixel. Finally, the code is sent out in the compressed data stream. Predictor used in PM step uses causal context of surrounding pixels for the prediction of the current pixel:

$$\hat{I}(x,y) = f(\Omega(x,y)) \tag{1}$$

PM block in Fig. 1 depicts basic elements of proposed predictor employed in CBPC. Ω_C denotes the causal context of surrounding pixels which used by the predictor. We also call this context the *search window*. It is a rectangular window of radius *R* composed of previously encoded pixels on which the search procedure for classification is performed. Current, unknown pixel $I(x,y)$ and each pixel from the Ω_C have their own vector template $\mathbf{v}(x,y)$ composed of *d* closest causal neighboring pixels as shown in the figure where vectors of size $d = 4$ ($\mathbf{v}_4(x,y)$) are used. Euclidean distance between associated vectors will be used for classifications of pixels into cells of *similar* elements similar to vector quantization. In order to reduce the complexity of proposed scheme some basic simplifications are introduced. First, as shown by Slyz and Neuhoff, only the *current* cell in which the current pixel lies needs to be calculated [18]. Next, the cell population, i.e. number of pixels that go into the cell together with the current pixel's vector, is set as constant *M* at the beginning of the coding process. Proposed prediction scheme operates as follows:

1. **IN – Image Input:** Currently our implementation of CBPC supports grayscale or color images in RGB color space. In case of color images, every component is independently processed by each subsequent step.

2. **PM – Predictive Modeling:** The predictor is described in following steps:

2.1.*Classification:* For each pixel $I(i,j) \in \Omega_C$ compute the Euclidean distance $D(i,j)$ between its corresponding vector $\mathbf{v}(i,j)$=$\mathbf{w}$ and the current pixel's vector $\mathbf{v}(x,y)$=$\mathbf{v}$:

$$\begin{aligned} D(i,j) &= ||\mathbf{v}(i,j) - \mathbf{v}(x,y)|| \\ &= ||\mathbf{w} - \mathbf{v}|| \\ &= \textstyle\sum_{k=1}^{d} |w_k - v_k|^2 . \end{aligned} \tag{2}$$

Based on the computed distances, determine *M* pixels from Ω_C that belong to the current cell, i.e. with the smallest vector distances from the current pixel's vector. The current cell will be used as blending context Ω_B for set of static predictors $\mathcal{F}$. This step is similar to nearest neighbor selection in VQ design.

2.2.*Blending:* On the blending context Ω_B perform the blending of the set $\mathcal{F} = \{f_1, f_2, \cdots, f_N\}$ of static predictors, as described in [17]. For every predictor f_k the penalty G_k is calculated by the following equation:

$$G_k = \sum_{I(i,j)\in\Omega_B} (\hat{I}_k(i,j) - I(i,j))^2, \tag{3}$$

where $\hat{I}_k = f_k(i,j)$ is the prediction of $f_k \in \mathcal{F}$ for the pixel $I(i,j) \in \Omega_B$. Based on the penalties we form the prediction for the current pixel $\hat{I}(x,y)$ as:

$$\hat{I}(x,y) = F(x,y) = \left\lfloor \frac{\left(\sum_{k=1}^{N} \frac{1}{G_k} \cdot \hat{I}_k(x,y)\right)}{\sum_{k=1}^{N} 1/G_k} \right\rfloor . \tag{4}$$

The prediction for the current pixel is the weighted sum of predictions of all the predictors from $\mathcal{F}$ with weights

inversely proportional to corresponding penalties. The penalty of predictor reflects its accuracy on the blending context. If the predictor predicts well, its contribution in the final prediction will be higher. The predictors that do not predict well on the current blending context will eventually be blended out by associated large penalties. The denominator in (4) normalizes the final prediction so that the sum of weights equals to 1. Our exhaustive tests had shown that the set $\mathcal{F}$ should contain simple static predictors that are suited to various regions, such as oriented edges, planar regions, smooth regions etc [1]. Based on our experiments, we fix the set $\mathcal{F}$ to be:

$$\mathcal{F} = \{f_N, f_W, f_{NW}, f_{NE}, , f_{GW}, f_{GN}, f_{PL}\}, \quad (5)$$

where:

$$\begin{array}{ll} f_N & = I(x, y-1) \\ f_W & = I(x-1, y) \\ f_{NW} & = I(x-1, y-1) \\ f_{NE} & = I(x+1, y-1) \\ f_{GW} & = 2 \cdot I(x, y-1) - I(x, y-2) \\ f_{GN} & = 2 \cdot I(x-1, y) - I(x-2, y) \\ f_{PL} & = I(x, y-1) + I(x-1, y) - I(x-1, y-1). \end{array} \quad (6)$$

$\mathcal{F}$ includes simple edge predictors (f_N, f_W, f_{NW}, f_{NE}), predictors with horizontal and vertical gradient modeling (f_{GW}, f_{GN}), and one planar predictor (f_{PL}). Border pixels for which the Ω_C is not defined are predicted using one of the simple predictors from $\mathcal{F}$ that has defined prediction context, i.e. f_W for the first row, f_N for the first column except for the $I(0,0)$ pixel which is PCM coded.

2.3.*Error correction:* On the blending context Ω_B calculate *typical* error of the final predictor as:

$$\bar{e}(\Omega_B) = \frac{1}{M} \sum_{I(i,j) \in \Omega_B} (F(i,j) - I(i,j)). \quad (7)$$

Based on the typical error of blending predictor F the final prediction for the current pixel is further refined as:

$$\dot{I}(x,y) = F(x,y) + \bar{e} = \hat{I}(x,y) + \bar{e}. \quad (8)$$

This final step of proposed predictor captures typical bias of the blending predictor F on the classified set of pixels Ω_B that are the part of similar structure as current pixel.

Through the classification and blending process, proposed predictor adjusts itself to the dominant local property. The blending allows other non–dominant properties to be modeled in the prediction, although with less contribution. This is crucial difference compared with switching predictors that don't have the capability to model nontrivial image structures with mixture of properties. Note that pixels from the search window that do not belong to the region with the same dominant property as the region in which current pixel resides will not be included in the current cell and thus they will not be part of the blending context.

3. **CM – Contextual Modeling:** Although the prediction step removes statistical redundancies within image data, there are remaining structures in the error image which cannot be completely removed using only previously applied prediction step [2]. Those structures are removed using contextual modeling of prediction error, where the context or the state is the function of previously observed pixels, errors or any other relevant variables. As reported by Wu, the heuristic method that uses both, previous pixels template and causal error energy estimate is best suited for this purpose [3]. Wu's contextual model is composed of two different submodels: (1) Model with large number of states that is used for prediction error feedback; and (2) Model with low number of states used for error probability estimation. On the other hand, Wu's predictor is a heuristic predictor with low degree of adaptation and our proposal is highly adaptive predictor with already built in error feedback mechanism (error correction step in the prediction). This implies that our mechanism needs smaller and less complex contextual model for estimation of symbol probabilities. Therefore it is built as follows: Besides of the high correlation with texture pattern, current prediction error is also highly correlated with the errors on neighboring pixels. This is modeled with the error discriminant

$$\Delta = d_h + d_v + 2|e_w|, \quad (9)$$

where

$$d_h = |W - WW| + |N - NN| + |N - NE|, \quad (10)$$

and

$$d_v = |W - NW| + |N - NN| + |NE - NNE|, \quad (11)$$

are horizontal and vertical gradients around the current pixel, and e_w is the prediction error on the west pixel W from the current pixel. Δ is uniformly quantized into eight levels to produce the state of the model [3]. Every state contains the histogram table which is used for probability estimation of the prediction error in the current state. Because of the context dilution effect, this context is required to have small number of states.

4. **EC – Entropy Coding:** The final step of proposed image compression algorithm is entropy coding of the resulting prediction error. For the given error symbol and given probability estimate computed from the contextual state, the codeword is computed by the entropy coder. This codeword is output as the final result of pixel compression process. Our proposal uses highly efficient implementation of arithmetic coding [19].

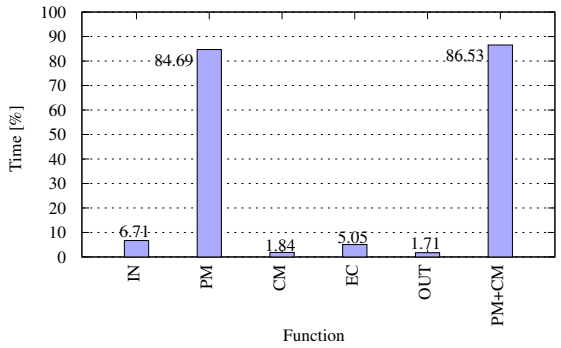


Fig. 2. Execution time breakdown of CBPC encoder

5. **OUT – Compressed Data Output:** Variable length code generated by entropy coder is put into compressed bitstream.

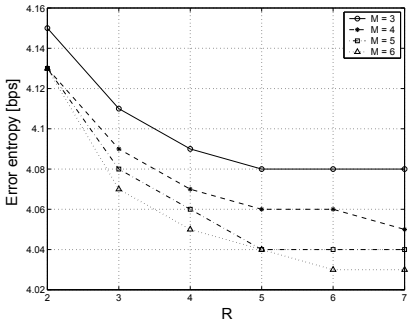
2.1 Computational Complexity

In order to investigate our algorithm performance, we profiled its execution with the resulting breakdown shown in Fig. 2. Each bar shows the contribution of the block (in percentage) from the Fig. 1 to the overall time. Predictive modeling (PM) block is computationally most demanding occupying almost 85% of total execution time. This step is tightly connected to the contextual modeling step CM in the algorithm. Therefore we show their joint contribution as the last bar in the figure. Therefore, PM and CM will be our target for performance improvements using the streaming model due to their computational requirements. In addition, they are characterized with the properties that make them a good fit for streaming abstraction: independent and dataflow–like operations on a large stream of data. The details of streaming optimizations of these two blocks are given in Section 3.

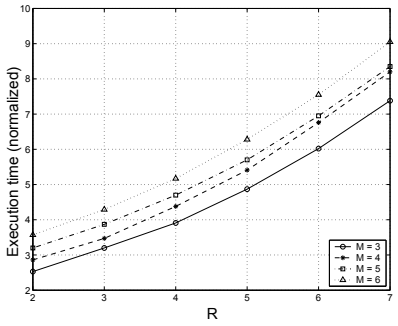
On a pixel–by-pixel basis, it takes

$$N_{op} = C_1 \cdot (2R^2 + R) \cdot d + C_2 \cdot M \qquad (12)$$

arithmetic operations to compute the predicted value. Constants C_1 and C_2 depend on the size of the vector template $v(x, y)$ and the size and complexity of the set of predictors $\mathcal{F}$, respectively. In order to more thoroughly understand the sensitivity of our PM step we measured the bitrate and the execution time when varying radius R and cell size M. Vector size d was fixed to five. Summary of the results are shown in Fig. 3 where we show the average bitrate and execution time obtained on the set of popular 8–bit grayscale test images. The set will also be used to demonstrate the compression efficiency of our algorithm (Table 2). Fig. 3(a) shows the entropy of the prediction error vs. the radius of search window R with the cell size M as a parameter, while Fig. 3(b) shows the execution time



(a) Entropy of prediction error



(b) Execution time

Fig. 3. Entropy and execution time versus algorithm parameters

Table 1. CBPC algorithm parameters

Parameter	Radius R	Vector size d	Cell size M
Value	5	5	6

measurements. We see near quadratic dependency of the execution time with radius of the search window, while increasing R above 5 and M above 6 reached the point of diminishing returns in terms of minimizing the entropy. Following the insights from our experiments we fixed the parameters of the PM step to values shown in Table 1. These values will be further used for the rest of the paper, where we report the compression efficiency of our proposal.

In terms of computational complexity, our predictor lies somewhere between simple predictors and highly adaptive LS–based predictors. However, in order to make our proposal more practical, we investigated and report an approach to improve the whole program execution time. Reasonably, we chose the predictive modeling step PM as the main target for performance optimization. Therefore we will concentrate on it and try to scalably improve its execution on the multicore machine in order to improve the overall program response time. We will leverage its properties and show its implementation in the streaming domain as a stream graph in the StreamIt programming language.

Table 2. Bit-rate of a complete coder (bps)

Image	JP–LS	JP2KR	CALIC	TMW	CBPC
Balloon	2.90	3.04	2.83	2.60	2.78
Barb1	4.69	4.61	4.41	3.83	4.13
Barb2	4.69	4.80	4.53	4.24	4.47
Board	3.68	3.78	3.56	3.27	3.49
Boats	3.93	4.07	3.83	3.53	3.77
Girl	3.93	4.07	3.77	3.47	3.69
Gold	4.48	4.61	4.39	4.22	4.39
Hotel	4.39	4.59	4.25	4.01	4.24
Zelda	4.01	4.00	3.86	3.50	3.72
Average (bps)	**4.08**	**4.17**	**3.94**	**3.63**	**3.85**

2.2 Compression Efficiency

In this section we report the compression results obtained with our proposed scheme. Table 2 compares the compression efficiency of the complete CBPC encoder with other popular lossless compression schemes. Aside for already described predictive modeling PM, our encoder employs contextual modeling CM and integer implementation of the arithmetic entropy encoder, as previously described. We present the results as the bit–rates obtained with the algorithm on the set of popular test images used in JPEG standard benchmarking. First column shows the results of a JPEG–LS standard coder [6], second column shows the results of the JPEG 2000 coder with the reversible wavelet transform [20], next two columns gives the results of the CALIC scheme [3] and a TMW scheme [9] respectively. The last column shows the results of our proposed CBPC scheme. Compared to JPEG–LS, JPEG 2000 and CALIC, our CBPC obtains lower bit–rates. Compared to the TMW compression scheme, our coder is outperformed. However, extreme complexity of the TMW scheme requiring two passes over image and hours to encode it, prohibits is practical use. Our CBPC scheme is on the other hand in the range of seconds to encode the image, albeit substantially more complex than other schemes such as JPEG–LS or CALIC which employ heuristic–based switching among several static prediction functions.

As previously said, the improvements in the compression efficiency of proposed scheme are paired with the increase on the computational demands. The goal of this paper is to show the approaches we took in order to improve the execution performance of the algorithm in order to make it more usable and luring for real applications. These efforts are described in the rest of the paper.

3 STREAMING IMPLEMENTATION OF CBPC

In this section we describe a portable and scalable performance optimization on the CBPC compression method that we call SCBPC. We improve the execution performance of CBPC utilizing the streaming model of computation and expressing compute intensive part of the algorithm in the StreamIt programming language. Following the observations from our previous work on integrating streaming computations we modified our experimental tool reported in [15], which resulted in StreamGate interface and runtime. The main purpose of the interface is to provide existing applications the easy way to integrate high–level and optimized streaming computations as reusable kernels. In this sense, the interface also provides a runtime which manages threads used in parallelized execution of streaming kernels, their creation, execution and finally their termination. Moreover, encapsulation of the streaming kernel through the interface fosters portability and scalability which will be demonstrated in Section 4.

We have been motivated by the following two observations:

1. CBPC encoder and decoder are impossible to implement entirely as a set of streaming computations. This is because the streaming is a domain-specific model requiring some specific properties of the computations to be expressed such as regular, data–oriented processing. Some parts of the algorithm exhibit properties that are better described in other abstractions. Examples are user input handling, error handling, control–oriented flows etc. Additionally, aggressive streaming compiler optimizations are possible only for static–rate filters, i.e. filters that have their rates defined at the compile time.

2. Stream–oriented parts are usually computationally the most demanding parts of typical image compression algorithms. Therefore, they are naturally selected for performance optimizations. This fact holds true for CBPC as well, an execution profile shown in Fig. 2 is dominated by the PM and CM parts which could be efficiently and naturally expressed as streaming computations.

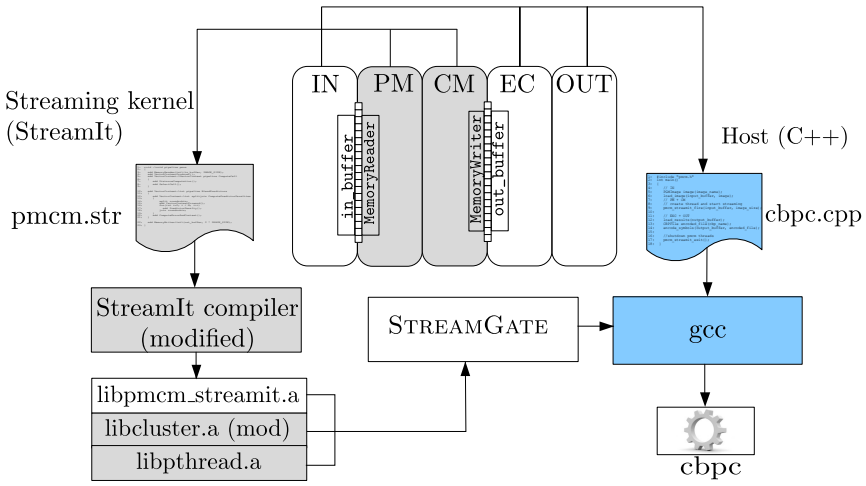


Fig. 4. Implementation flow of CBPC encoder

Fig. 4 shows the design flow with the StreamGate interface we developed in order to integrate the streaming computations into the CBPC algorithm. First, based on our profile from Fig. 2, we selected PM (predictive modeling) and CM (contextual modeling) steps to be implemented in StreamIt. Those steps exhibit streaming nature and contribute nearly 87% of total encoder execution time. StreamIt source file `pmcm.str` containing streaming description of these two steps is compiled with the high–level StreamIt compiler which we modified in order to compile its input into a library (`libpmcm_streamit.a`) and connect to the StreamGate interface. This library exposes, among the routines that manage interface, an entry routine for our streaming kernel `pmcm_streamit_fire()`. The kernel is asynchronously called from the host application written in C/C++ language (`cbpc.cpp`) and it performs specified streaming computations in parallel. Host application code in `cpbc.cpp` can require to eventually synchronize with spawned streaming kernel which is also provided through the StreamGate interface. The final CBPC encoder executable is the result of the compilation and linkage of the host application code `cbpc.cpp` and the `libpmcm_streamit.a` library which is further linked with `cluster` and `pthreads` libraries. The former is the part of StreamIt infrastructure for parallel execution of StreamIt programs on multicores and clusters, while the latter is a standard POSIX threading library. Resulting executable `cbpc` performs PM and CM steps in multiple threads as streaming kernel, while the rest of the application consisting of IN, EC and OUT steps is serial, i.e. performed in single, main program thread implemented in the host code.

The source code of PM and CM steps in `pmcm.str` file with the high–level overview is shown in Fig. 5. At the boundaries of the top–level `pmcm` *pipeline* are the `MemoryReader` and `MemoryWriter` filters that we integrated into the StreamIt language and compiler infrastructure, lines 3 and 22. These filters are reusable, built–in filters which perform data exchange between streaming kernel and the host application. They are parameterized with the data–type they exchange, the name and the size of the external host buffer used to get from or fill the data. Filters can exchange primitive and user–defined types. As shown in Fig. 4, `MemoryReader` pops the input pixels from the `in_buffer` array, while `MemoryWriter` pushes the data into the `out_buffer` array. The exchange arrays are allocated in the host program `cbpc.cpp` source outlined in Fig. 6. After the pixels are read and pushed forward, `VectorContextProducer` stream generates the vector template $v(x, y)$ for each of them, as illustrated in Fig. 1. Next, generated vector templates are processed by the `ComputeCell` pipeline that classifies the set of M pixels from the Ω into a current cell (lines 5 through 9). A set of static predictors is blended in the `BlendPredictors` pipeline (lines 10 through 19) in order to produce the prediction value for the current pixel. Finally, `ComputeErrorAndContext` filter (line 20) calculates the prediction error together with the context in which the error probability is estimated, a work done in the CM step of the algorithm. Generated symbol data are placed in the `out_buffer` by `MemoryWriter`. These data are ready for entropy coding by the EC step, which is implemented in the host code.

Original stream graph of the `pmcm.str` is shown in Fig. 5. This graph is a high–level view of the algorithm as perceived by the programmer. The resulting stream graph, as generated by streaming compiler is usually not iden-

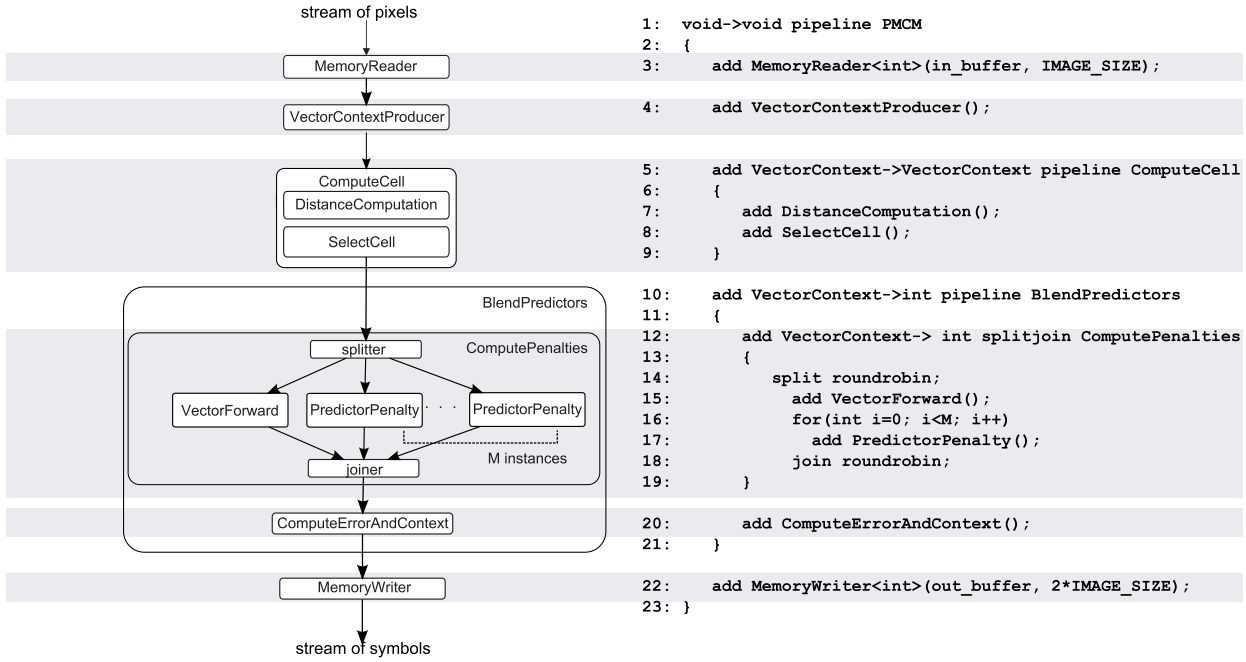


Fig. 5. High–level source of PM and CM steps in StreamIt

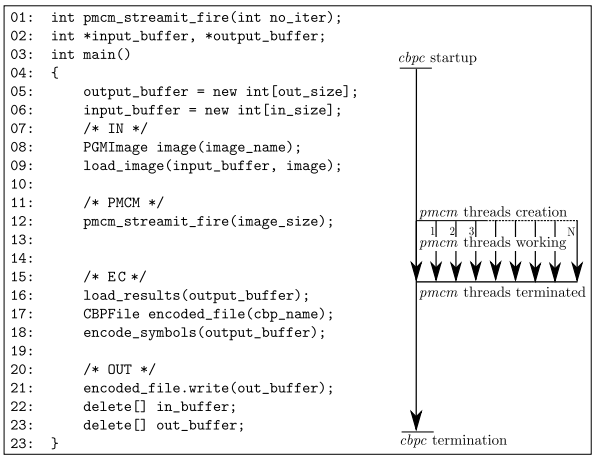


Fig. 6. Host–application code and thread activity

tical to the original, as the compiler performs transformations in order to target the underlying cores to obtain a balanced set of computational threads. These transformations are transparent and automated into the streaming compiler, thus freeing the programmer from the exhaustive task of load balancing. Additionally, automation of the stream graph transformations into the compiler helps the portability across both, increasing number of cores and different computer architectures.

Fig. 6 illustrates the host source code in `cbpc.cpp`. Additionally, right side of the figure illustrates the general thread activity associated with the execution of the final *cbpc* executable. Line 1 declares the *fire* routine which invokes parallel execution of the streaming kernel *pmcm*. Lines 2, 5 and 6 allocate the space for exchange buffers that link the host application and the streaming kernel using `MemoryReader` and `MemoryWriter` filters. The host application also reclaims the buffers when they are no longer needed on lines 22 and 23. Lines 8 and 9 load the image from the file and fill the `in_buffer` with the pixels which `MemoryReader` filter from Fig. 5 passes to the streaming kernel. Line 12 invokes the `pmcm_streamit_fire` routine. At this point, the number of threads determined by the streaming compiler are created and streaming computations start the execution in parallel. The call to the routine is synchronous in terms that the code after the call will continue only after the streaming threads complete. We also allow asynchronous calls of streaming kernels. When the kernel is executed, the threads terminate and exit reducing the execution to single thread of the host application. Threading support which includes thread creation, execution and termination, is realized with the *pthreads* library [21] and is encapsulated in the StreamGate interface as well as synchronization routine which is not shown. This removes the burden of tedious low–level multithreaded programming from the programmer. Line 16 loads the resulting symbols from the `output_buffer` array into statistical encoder which encodes the symbols on line 18. Compressed image bitstream is written into the file on the line 21.

4 EXPERIMENTAL RESULTS

In order to validate our proposal we measured the execution time of our modified CBPC encoder with streaming kernel for predictive and error modeling steps on three different sets of test images:

- **Jpeg:** A set of natural grayscale images which are widely used for benchmarking the compression efficiency in the literature and in the JPEG standard, as shown in Table 2.
- **Jpeg (RGB):** The same set of images as previous set in RGB color format. We chose this set of images to demonstrate the scalability of proposed scheme. Our algorithm independently processes each color channel equivalent to processing a luminance channel in grayscale images.
- **Satellite:** A set of satellite images in high resolution. Images were obtained in various spectral bands. Because of high resolution and visual properties, satellite images present a challenge to efficient lossless compression which diminishes the benefits of potential heuristics incorporated into adaptive predictors.
- **Medical:** A set of grayscale medical images obtained with various methods which range from high resolution MR images to computerized tomography images.

Our final SCBPC compression method consists of a host code written in C++ and a streaming kernel written in the StreamIt programming language. Streaming kernel consists of PM (Predictive Modeling) and CM (Contextual Modeling) steps, while the host performs all other steps as described in the Fig. 4. We performed our benchmarks on two different platforms which resemble the increase of processing cores in upcoming computational trends:

- **2 Cores:** A system containing a 2.66GHz Intel Core 2 Duo CPU with 4GB of RAM, running a Ubuntu 10.04 operating system with Linux 2.6.32 kernel. The system had *gcc* version 4.4.3 installed.
- **4 Cores:** A system containing a 2.66GHz Intel Core 2 Quad CPU with 8GB of RAM, running Fedora 15 operating system with Linux 2.6.41 kernel. For C++ source compilation we used *gcc* version 4.6.1.

Streaming kernel `pmcm.str` was first processed by streaming compiler as shown in Fig. 4. We did not use any stream–specific optimizations in this step, and the only option we used was the target number of threads for the kernel. Streaming compiler produces a C++ code which was compiled and packaged in the library and exposed through the STREAMGATE.

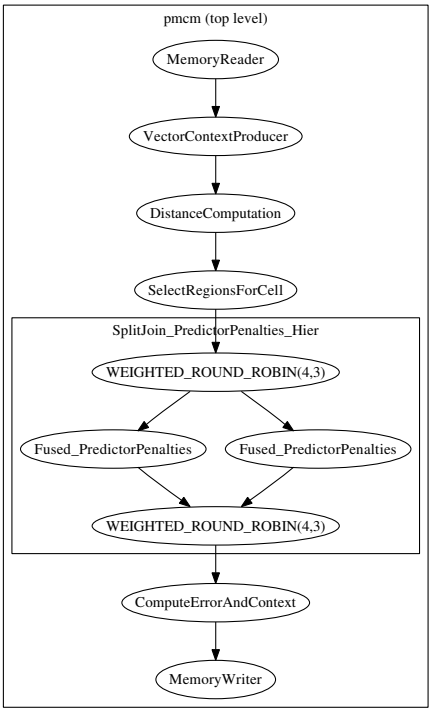


Fig. 7. Resulting stream graph for 2 cores

For the host part of the SCBPC code and for the C++ code generated by StreamIt compiler we used *gcc* with `-O3` optimization flag. We also used this setting for compilation of original, serial CBPC C++ code so that we can provide a fair comparison between parallelized (SCBPC) and sequential (CBPC) implementation of the compression method.

Fig. 7 shows the resulting, partitioned stream graph of our streaming kernel. The original stream graph shown in Fig. 5 was finally partitioned to 9 threads. For RGB images from Jpeg (RGB) set the streaming compiler was instructed to target 17 threads which practically duplicate the pipelines from Fig. 7 for each color channel.

We measured the speedup of SCBPC obtained on our test sets normalized to two realizations:

1. Optimized sequential realization of original CBPC algorithm from our previous publications [1, 16]. This realization doesn't incorporate any stream–specific code and is written in C++. As noted, the executable is compiled with respective *gcc* compiler with `-O3` optimization switch. It represents a highly optimized serial implementation of the algorithm.

2. Sequential realization of SCBPC with the streaming kernel compiled with the target number of threads equal to one. This gives a realization of our algorithm with

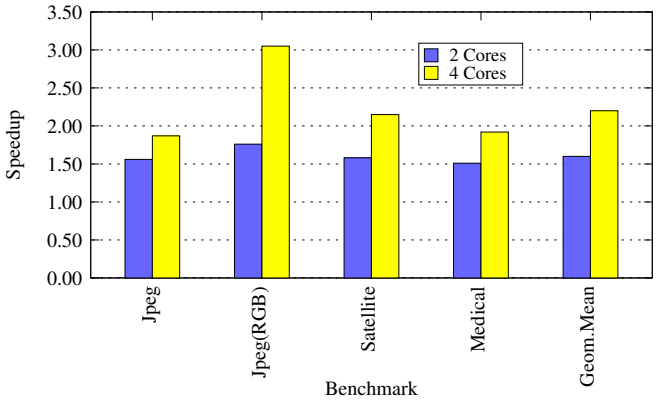


Fig. 8. Speedup normalized to optimized serial implementation

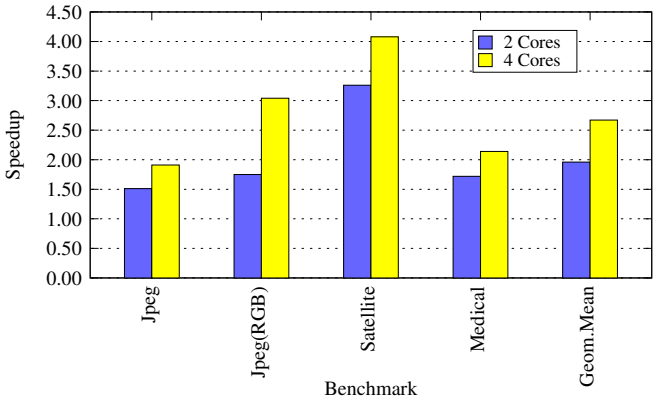


Fig. 9. Speedup normalized to serial implementation of pmcm kernel

stream–specific implementation of PM and CM steps, but compiled for sequential execution. Speedup of parallelized SCBPC (to specific number of threads for streaming kernel) relative to this serial streaming realization gives a fair comparison of benefits obtained through the incorporation of high–level streaming abstraction into the design flow.

Fig. 8 shows the average speedup obtained on our sets of images normalized to the first realization of the algorithm. The last pair of bars shows the geometric mean. For each set we obtained comparative results which increase with the number of cores. For RGB images the speedup of 3.05 is especially noticeable for a four core machine. This is because of increased amount of data–parallelism available through independent color channels. This specific case clearly demonstrates the portability and scalability of our approach. However, in order to have this effect in other applications, we would have to rely on exploiting not only data parallelism, but other types of parallelism as well. We also see here an opportunity to exploit a high–level pipeline parallelism between the host code and streaming kernel. On overall, we obtained mean speedup of 1.6 for two cores and 2.2 for four cores.

Fig. 9 shows the speedup normalized to the second, serial streaming realization of the algorithm. This figure gives a fair comparison of scalability over the increasing number of cores (and parallelizing the streaming kernel) when we concentrate solely on the effects of scalable implementation of streaming kernel. As illustrated, increasing the number of processing cores improves the execution time without any modifications of the source code, either host or kernel code. Interestingly, we obtained superlinear speedup for both test configurations (3.26 for two cores, 4.08 for four cores) on satellite images. We attribute this to cache effects with the current implementation of tapes in generated streaming kernel code, which are implemented as FIFO buffers [13]. Placing large amount of data on the tapes between filters can potentially cause cache misbehavior for coarse grained stream graphs, such as the graph which the streaming compiler produces when instructed to compile the kernel code for serial execution (or insufficient number of threads). On average, we obtained linear speedup when the number of cores is increased.

5 CONCLUSION

Current and upcoming processor architectures are characterized with the increased number of processing cores. Another trend in computing landscape is the emergence of data–intensive applications that require high processing power. In that sense, those data–intensive applications are natural targets for performance optimizations.

In this paper we demonstrated the benefits of using streaming programming model for high–level implementation of computational kernels for data–intensive applications such as predictive lossless image coding. We proposed an interface and tool which can provide portable and scalable performance with the increase of processing cores. This is obtained through a high–level implementation of performance hungry computations as kernels in streaming model and their integration into existing code as reusable modules.

Our approach is demonstrated in our adaptive prediction–based lossless image coding algorithm which clearly benefited from streaming implementation of its computational kernels. These kernels are run in parallel using available processing resources. Our approach can be extended to heterogeneous platforms where compute–intensive and data–parallel parts can be mapped into graphical processing units without tedious programming at the level of graphical libraries.

REFERENCES

[1] J. Knezović, M. Kovač, and H. Mlinarić, "Classification and blending prediction for lossless image compression," in *Electronic Proceedings 13th IEEE Mediterranean Electrotechnical Conference*, (Malaga, Spain), 2006.

[2] N. Memon, X. Wu, "Recent Developments in Context-Based Predictive Techniques for Lossless Image Compression," *The Computer Journal*, vol. 40, no. 2/3, pp. 127–136, 1997.

[3] X. Wu, "Lossless Compression of Continuous-tone Images via Context Selection, Quantization, and Modeling," *IEEE Transactions On Image Processing*, vol. 6(5), pp. 656–664, 1997.

[4] G. Motta, J.A. Storer, B. Carpentieri, "Lossless Image Coding via Adaptive Linear Prediction and Classification," *Proceedings of The IEEE*, vol. 88, pp. 1790–1796, November 2000.

[5] X. Li, M.T. Orchard, "Edge–Directed Prediction for Lossless Compression of Natural Images," *IEEE Transactions on Image Processing*, vol. 10(6), pp. 813–817, 2001.

[6] M.J. Weinberger, G. Seroussi, G. Sapiro, "The LOCO–I Lossless Image Compression Algorithm: Principles and Standardization into JPEG–LS," *HP Laboratories Technical Report*, vol. HPL-98-198, 1998.

[7] V. P. Baligar, L. M. Patnaik, and G. R. Nagabhushana, "High compression and low order linear predictor for lossless coding of grayscale images," *Image Vision Comput.*, vol. 21, no. 6, pp. 543–550, 2003.

[8] S. Marusic and G. Deng, "Adaptive prediction for lossless image compression," *Image Commun.*, vol. 17, no. 5, pp. 363–372, 2002.

[9] B. Meyer, P. Tischer, "TMW – a New Method for Lossless Image Compression," in *Proc. of the 1997 International Picture Coding Symposium*, (Berlin, Germany), VDE–Verlag, 1997.

[10] Y. Yu and C. Chang, "A new edge detection approach based on image context analysis," vol. 24, pp. 1090–1102, October 2006.

[11] T. Fang, "On performance of lossless compression for HDR image quantized in color space," *Image Commun.*, vol. 24, no. 5, pp. 397–404, 2009.

[12] A. Kingston and F. Autrusseau, "Lossless image compression via predictive coding of discrete radon projections," *Image Commun.*, vol. 23, no. 4, pp. 313–324, 2008.

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

[14] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe, "A Stream Compiler for Communication-Exposed Architectures," in *Proc. ASPLOS 02 International Conference on Architectural Support for Programming Languages and Operating Systems*.

[15] 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, no. 4, pp. 387–396, 2010.

[16] J. Knezović, M. Kovač, I. Klapan, H. Mlinarić, v. Vranješ, J. Lukinović, and M. Rakić, "Application of novel lossless compression of medical images using prediction and contextual error modeling," *Collegium antropologicum*, vol. 31, no. 4, pp. 1143–1150, 2007.

[17] T. Seemann, P. Tischer, "Genralized Locally Adaptive DPCM," in *Proc. 1997 Data Compression Conference*, (Snowbird, UT), pp. 473–488, IEEE, March 25–27 1997.

[18] M.J. Slyz, D.L. Neuhoff, "A Nonlinear VQ–based Lossless Image Coder," in *Proc. 1994 Data Compression Conference*, (Snowbird, UT), pp. 491–500, IEEE, March 1994.

[19] P.G. Howard, J.S. Vitter, "Arithmetic Coding for Data Compression," in *Proceedings of the IEEE*, vol. 82(6), pp. 857–865, June 1994.

[20] D. Santa–Cruz, T. Ebrahimi, J. Askelof, M. Larsson, C. Christopoulos, "JPEG 2000 still image coding versus other standards," in *Proc. of the SPIE's* 45^{th} *annual meeting, Applications of Digital Image Processing XXIII*, vol. 4115, (San Diego, California), pp. 446–454, August 2000.

[21] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. 101 Morris Street, Sebastopol, CA 95472: O'Reilly, 1998.

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 as a research assistant 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 the member of IEEE and ACM.

Hrvoje Mlinarić 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 1996, 2002 and 2006, respectively. Since 1997 he has been with Faculty of Electrical Engineering and Computing currently holding the assistant professorship position. He is also the vice-head of the Department of Control and Computer Engineering. His research interests include data compression, programmable logic and advanced hardware and software design. He is the member of Croatian Academy of Engineering.

Martin Žagar received his Ph.D.C.S. in 2009, and B.S.C.S. in 2004 at the Faculty of Electrical Engineering, University of Zagreb, Croatia. Currently he is a teaching assistant at the Department of Control and Computer Engineering, University of Zagreb. His main areas of interest are data compression and multimedia architectures. He published a numerous papers in journals and proceedings.

AUTHORS' ADDRESSES

Josip Knezović, Ph.D.
Asst. Prof. Hrvoje Mlinarić, Ph.D.
Martin Žagar, Ph.D.
Departement of Control and Computer Engineering
Faculty of Electrical Engineering and Computing
University of Zagreb
Unska 3, HR-10000, Zagreb, Croatia
email: {josip.knezovic, hrvoje.mlinaric, martin.zagar}@fer.hr

Received: 2012-04-26
Accepted: 2012-05-11