Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 172674, 16 pages
doi:10.1155/2012/172674

Research Article

Towards Self-Adaptive KPN Applications on NoC-Based MPSoCs

Onur Derin,[1] Prasanth Kuncheerath Ramankutty,[1] Paolo Meloni,[2] and Emanuele Cannella[3]

[1] *ALaRI, Faculty of Informatics, University of Lugano, 6904 Lugano, Switzerland*
[2] *Department of Electrical and Electronic Engineering, Faculty of Engineering, University of Cagliari, 09123 Cagliari, Italy*
[3] *LIACS, Leiden University, 2333 CA Leiden, The Netherlands*

Correspondence should be addressed to Onur Derin, derino@alari.ch

Received 29 June 2012; Revised 23 September 2012; Accepted 25 September 2012

Academic Editor: Phillip A. Laplante

Self-adaptivity is the ability of a system to adapt itself dynamically to internal and external changes. Such a capability helps systems to meet the performance and quality goals, while judiciously using available resources. In this paper, we propose a framework to implement application level self-adaptation capabilities in KPN applications running on NoC-based MPSoCs. The monitor-controller-adapter mechanism is used at the application level. The monitor measures various parameters to check whether the system meets the assigned goals. The controller takes decisions to steer the system towards the goal, which are applied by the adapters. The proposed framework requires minimal modifications to the application code and offers ease of integration. It incorporates a generic adaptation controller based on fuzzy logic. We present the MJPEG encoder as a case study to demonstrate the effectiveness of the approach. Our results show that even if the parameters of the fuzzy controller are not tuned optimally, the adaptation convergence is achieved within reasonable time and error limits. Moreover, the incurred steady-state overhead due to the framework is 4% for average frame-rate, 3.5% for average bit-rate, and 0.5% for additional control data introduced in the network.

1. Introduction

Recent trends in microprocessor design have witnessed a paradigm shift from single core architectures towards Multi-processor-System-on-Chips (MPSoCs) due to unsustainable increases in power dissipation while running a single processor at very high frequencies. As the number of cores in the system increases, on-chip communication becomes a bottleneck. To address the scalability issues of MPSoCs, Networks-on-chips (NoCs) [1] have emerged as a new communication paradigm. However, even in the case of NoC-based systems, if shared memory access is employed, memory coherence protocols induce an overhead in the communication network rendering the gain from additional cores useless. No Remote Memory Access (NORMA) [2] model addresses this problem by assigning a private local memory for each NoC tile. This solution is especially suited for programming models based on message passing.

Embedded systems are often subject to stringent non-functional goals such as high computational performance and dependability, low power consumption, memory usage, and chip area. Satisfying such requirements imposed by the application designer on systems with increasing complexity of the underlying architectures is a fundamental challenge. To deal with this problem, designers often resort to self-adaptation-based techniques [3]. Self-adaptive systems are able to react when the actual operating conditions of the system such as the workload, the internal/external conditions, and the quality-of-service goals differ from the design-time assumptions.

The techniques to be developed for implementing self-adaptive applications depend heavily on the adopted model of computation. In this paper, we adopt Kahn Process Networks (KPN) [4] model due to its suitability for NORMA-based NoC multiprocessors. KPN represents a stream-oriented computation model, where an application is organized as streams and computational blocks; streams represent the flow of data, while computational blocks represent operations on a stream of data, making it a suitable computation model to represent most of the signal processing and multimedia applications of the embedded systems world.

There are certain challenges to be tackled when designing self-adaptive systems. A general concern is the overhead introduced in making the system monitorable and adaptable. A large overhead can easily compensate the benefits of adaptation. There are two types of overhead. The first type, which can be called *steady state overhead*, is the overhead experienced simply due to the additional hardware or software for enabling monitoring and adaptation capabilities. It is present even when there are no ongoing adaptations. This overhead should be minimized because it has to be afforded at all times. The second type, which can be called *transient overhead*, is the overhead experienced while an adaptation is taking place. The major sources of this overhead are the adaptation control logic and the realization of an adaptation. If the system is expected to have frequent adaptations, then care must be taken to minimize this type of overhead.

Separation of concerns is a key feature for self-adaptive systems. However, the realization of this principle is quite challenging for several reasons. It emphasizes that the application programmer should be involved as minimally as possible in making the system self-adaptive. Although it may be possible to realize this for adaptations at the run-time environment and hardware levels because of the clear interface between the application and the execution platform, it is a more difficult task for application level adaptations. Intrinsic application knowledge by the application programmer is required in order to expose the feasible adaptations in the application. Automatic inference of such adaptations would be very difficult, if possible at all. Depending on the adaptation goal, another difficulty is in the inference of what to monitor and how to monitor it without involving the programmer. There is a semantic gap to be bridged between the given goal and the application. Monitoring involves choosing the correct program variables and operating on them in order to calculate the actual metric that corresponds to the goal. Another issue with the separation of concerns principle is that it is likely to conflict with the low overhead goal mentioned previously. The less the programmer has to do would lead to the more the self-adaptation logic has to do, thus yielding to more overhead due its complexity. Last but not least, the behavior of the adaptation controller is application-dependent. Machine learning algorithms can be used to obtain the application knowledge, particularly the relation between the goal and the adaptations, but it would result in a complex control logic with a bigger overhead which may not be acceptable for embedded systems. Alternatively, the required application knowledge can be provided to the controller by the application programmer.

Another fundamental challenge for system-wide self-adaptivity is presented by the management of the adaptations. The systems are usually faced with multiple goals to satisfy at run-time such as a desired throughput, low power consumption, and high dependability. Satisfying all the goals by controlling various possible adaptation options is a difficult task. Changing the set of goal types would require a complete or partial redesign of the controller. Possible solutions to this problem are automated controller synthesis or designing generic controllers.

In this paper, we are addressing the application level self-adaptation challenges mentioned above in the context of streaming applications based on the KPN model and running on NORMA-based NoC multiprocessors. The main contribution of this work is the investigation of fuzzy control as a generic adaptation management mechanism for self-adaptive systems. In doing so, methods for adding adaptation and monitoring capabilities to KPN applications are proposed. Results are presented showing the generality and quality of control. The overhead of the proposed self-adaptive framework is also reported with a case study.

The remaining part of the paper is organized as follows. An overview of the related work is provided in Section 2. Section 3 introduces our framework with implementation details of monitoring, controlling, and adaptive tasks. Section 4 discusses adaptive-MJPEG as a case study, which is built using the proposed M-C-A framework. The results of the case study are provided in Section 5. Section 6 discusses the main design principles and generality of the proposed framework, while Section 7 concludes our work.

2. Related Work

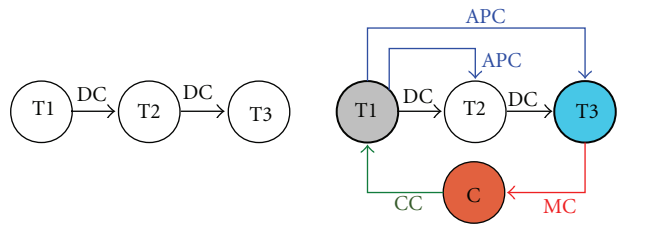
In [5], authors presented a monitor-controller-adapter-based framework to enable self-adaptivity for streaming applications. The paper introduces a framework based on the KPN computation model. However, implementation details of monitors and adaptation controllers are not provided. Even though a case study is provided, no results are available to evaluate the framework in terms of its effectiveness and performance (timing, convergence, etc.).

A standardized way to manage self-adaptivity at application level is provided in [6], which proposes *separation of concerns* between adaptation management and system functionalities. Self-adaptivity is obtained by applying a set of *adaptation policies* on software components, while these policies are triggered by certain configurable system events. Possible adaptations for component behavior and application parameters are also discussed. Unfortunately, they do not discuss if and how a general goal is achieved.

In [7], the authors present the results of project MADAM that has delivered a comprehensive solution for the development and operation of context-aware, self-adaptive applications. The main contributions of this work are (a) a sophisticated middleware that supports the dynamic adaptation of component-based applications, and (b) an innovative model-driven development methodology based on abstract adaptation models and corresponding model-to-code transformations.

A generic model of a self-adaptive system is presented in [8], in which a proposal to manage self-adaptivity at hardware and software levels by means of a decentralized control algorithm is provided. A goal management methodology, a goal specification interface, along with a decentralized and coordinated control mechanism is proposed as part of this work.

In [9–11], several techniques for fine-grained QoS control of multimedia applications are presented. The proposed methods generate a controlled application that meets given



DC: data channel
MC: monitor channel
T1: adaptive task
C: adaptation controller
CC: control channel
APC: adaptation propagation channel
T3: monitoring task

(a) KPN application (b) Self-adaptive KPN application

Figure 1: A self-adaptive KPN application based on M-C-A framework.

QoS requirements from an input application software. The controller monitors the progress of the computation in a cycle and chooses the next action to run and its quality level, guided by safety and optimality constraints for the system. Our work differs from these methods in two aspects: (1) we target applications running on MPSoCs in a distributed manner, whereas these works consider single threaded applications for which it is possible to estimate by using timing analysis and profiling techniques, worst-case execution times and average execution times, for different levels of quality. (2) Our controller is generic and requires minimal knowledge of application characteristics as compared to these methods which require deep knowledge of the data-flow structure of the application.

A middleware-based approach to enable run-time migration of processes among tiles of a NoC is presented in [12]. Such a technique helps in achieving application-independent adaptivity support such as fault tolerance. On the other hand, our work deals with application-dependent parameter adaptations using a M-C-A based approach.

3. Self-Adaptive KPN Applications Using Distributed M-C-A Framework

This section presents our framework to build self-adaptive component-based applications by incorporating a distributed monitor-controller-adapter (M-C-A) mechanism in the KPN application pipeline (as proposed in [5]). Monitoring involves measurements of various parameters to check whether the system meets the assigned goals. The controller is capable of driving adaptations when goals are not met, whereas adapters are in charge of performing adaptations. In case of KPN applications running on MPSoCs, various tasks of the application will be mapped onto different tiles of the platform. Hence it is quite possible that the parameter to be monitored is present in one tile, whereas the task to be adapted may exist on a different tile. This forces the monitor, controller, and adapters to be implemented on different tiles in a distributed manner. For example, in case of a video encoder application, bit-rate monitoring should be done on the tile where sink task is present whereas the frame-size adapter logic has to be present on the tile where source task is located. Our framework represents a self-adaptive application in terms of the following entities: adaptive tasks implementing adapter functions, monitoring tasks calling monitoring functions, adaptation controller(s), and adaptation propagation channels alongside the original task graph. Figure 1 depicts a simple KPN application and its self-adaptive version based on our framework.

3.1. Adaptive Task. In order to implement application specific adaptations, each task should expose its adaptation space (set of adaptable parameters) to the external world. Adaptive tasks will have *control channels* and multiple optional *adaptation propagation channels* in addition to nominal input/output data channels. Control channels carry the control commands from the controller to adaptive tasks whereas adaptation propagation channels carry new parameter values from adaptive tasks to other tasks which require these updated values. For example, in case of an adaptive source task (which supports frame-size adaptation) in a video encoding application, control channel will carry the frame-size control command from the controller, whereas the adaptation propagation channels will carry the new frame-size to any other relevant tasks. The frequency at which these channels will be read/written by the task depends on the application as well as the granularity required for the control. In order to perform the adaptation, the task should read the control command from the control channel and call the adapter functions, with control command as the argument. It should also send the modified values of the adapted parameter to other tasks which need these updated parameters. Figure 2 shows the modifications required (shown in blue) to transform a KPN task into an adaptive KPN task.

```
adaptiveTask()
{
  for(i=0; i<M; i++) {

    read(CTRL_CH, &ctrlSignal);
    adaptParam(ctrlSignal);
    write(ADAPT_PROP_CH, newParam);

    for(j=0; j<N ; j++) {
      read(DATA_IN_CH, &inData)
      outData = process(inData);
      write(DATA_OUT_CH, outData);
    }
  }
}
```

Figure 2: An adaptive task.

```
monitoringTask()
{
  for(i=0; i<M; i++) {

    int dataCounter = 0;
    for(j=0; j<N ; j++) {
      read(DATA_IN_CH, &inData);
      dataCounter += sizeof(inData);
      outData = process(inData);
      write(DATA_OUT_CH, outData);
    }

    timeStamp t = getCurrentTime();
    alignSlidingWindow(dataCounter, t);
    br = calculateBitrate();
    tr = calculateTokenrate();
    write(MONITOR_CH_BR, br);
    write(MONITOR_CH_TR, tr);
  }
}
```

Figure 3: A monitoring task.

3.1.1. Adapter Functions. Adapter functions perform the actions needed to perform the adaptations. The implementation of adapter functions is parameter dependent. An adapter function for adapting "param1" has the following signature: void adaptParam1(CtrlCommand c); where control command argument can take one of the following values: (a) −2: modify the adaptable parameter so as to aggressively reduce the monitored parameter, (b) −1: modify the adaptable parameter so as to mildly reduce the monitored parameter, (c) 0: maintain same value for the parameter, (d) +1: modify the adaptable parameter so as to mildly increase the monitored parameter, (e) +2: modify the adaptable parameter so as to aggressively increase the monitored parameter. The adapter functions need to be implemented by the application programmer with appropriate interpretation of the mild/aggressive changes to the parameter.

3.2. Monitoring Task. Monitoring refers to the measurement of a parameter in the system, that is, of interest. The accuracy and timing of these measurements are critical, since it impacts the overall quality of adaptation. A normal KPN task is converted to a monitoring task by calling monitoring functions provided by the framework. Figure 3 shows a simple monitoring task obtained by modifying a typical KPN task by adding calls to the monitoring functions (shown in blue). Our framework supports two types of throughput monitoring: bit-rate and token-rate. The granularity of monitoring is application-dependent and it

is the application programmers responsibility to insert calls to the monitoring functions at an appropriate place in the code. Furthermore, the framework assumes support from the platform to measure the current time. Monitoring task should also send the monitored parameter values to the adaptation controller using monitor channels.

3.2.1. Monitoring Functions. The following are the monitoring functions provided.

AlignSlidingWindow. We propose sliding-window monitoring, which is triggered by a call to the alignSlidingWindow function. Sliding window method is deployed to find the average of last few instantaneous values of a monitored parameter. It is realized using two circular arrays of size equal to *monitor-width*, which is configurable in the implementation. These arrays are used to hold the parameter values and the timestamp of their measurements. When alignSlidingWindow is called (with the newly captured parameter value and its timestamp as arguments), the windows are adjusted so that the arrays contain the most recent parameter values.

CalculateTokenRate. Token rate is the number of tokens received per unit time by the monitoring task. It is calculated using the number of entries in the sliding window and the difference in timestamps between the latest and oldest entries.

CalculateBitRate. This function calculates the throughput of the generated data. Throughput (bit-rate) is calculated by dividing the sum of all entries in the monitoring window by the difference in timestamps between the latest and oldest entries.

The width of the sliding window can be specified by the application programmer using the monitor-width parameter. This parameter decides the sensitivity of the control mechanism (i.e., how fast the variations in the monitored variables are perceived). If the monitor-width is too large, the sensitivity will be low, that is, the effect of a particular adaptation strategy will be reflected in the average value only after many values got generated under that strategy. On the other hand, a very small monitor window helps in detecting changes in the parameter very fast. However, this may cause large ripples in the output since any adaptation strategy needs some settling time before its effects are visible. Hence it is very important to keep monitor-width at an optimum value to obtain a good quality of adaptation.

3.3. Controller. The most important entity of any adaptation scheme is the controller, because it takes decisions to steer the monitored parameters towards their target values. The correctness and speed of the decisions taken by the controller influence the effectiveness of the adaptation mechanism. Hence controller is the most critical entity in the design of self-adaptive systems. In order to free the application developer from self-adaptivity concerns, our framework provides a generic *fuzzy logic* [13] based adaptation controller that should work for any application being run on the platform.

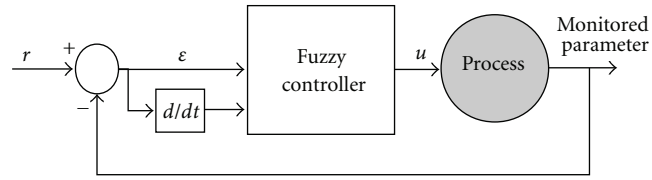


Figure 4: A simple fuzzy control-based system.

Fuzzy logic is a form of multivalued logic that deals with reasoning in an approximate way rather than precise. It is derived from the fuzzy set theory which is based on the understanding that every fact is present or not up to a certain degree. Fuzzy control represents formal methodology for presentation, manipulation, and implementation of human heuristic knowledge about how certain processes should be controlled by using a simple, rule-based *"if X and Y then Z"* approach rather than attempting to model a system mathematically. For example, instead of dealing with temperature control in precise terms, fuzzy controller uses linguistic terms such as *"if (process is too cool) and (process is getting colder) then (heat the process)"* or *"if (process is too hot) and (process is heating) then (cool the process quickly)."* These terms are imprecise yet very descriptive of what must actually happen. We chose to use fuzzy logic control since the mathematical models of most application processes are unknown and would be very difficult to build, yet it can easily be described linguistically such as if process is very hot and the temperature is increasing, it is clear that the process has to be cooled quickly.

Figure 4 depicts a simple fuzzy logic controlled system. Here some parameter of interest within the system is monitored. The error signal (ϵ) is the difference between the reference value (r) of the parameter and its monitored value. The fuzzy control logic takes this error signal and its rate of change as inputs and generates the control signal (u) as the output, which will be fed to the adapter logic.

We propose to implement a separate fuzzy controller for each specified goal of the system, thus offering scalability in terms of adding new goals. This also allows controller tasks to be placed in tiles which are at optimum distances from the corresponding adaptive and monitoring tasks, hence reducing the latency and the amount of network traffic introduced by control data. The frequency at which the controller should be run is application-dependent. For example, in case of a frame-rate control in a video encoder, the algorithm can be run for every third video frame.

Our design of the fuzzy controller is based on the following parameters.

Error (ϵ). The difference between the monitored value of a parameter and its target value.

Delta Error ($\Delta\epsilon$). The difference between current error and previous error.

Control Settling Width. The duration for which the controller should wait for a control decision to take its effect on

Table 1: Error ranges for the fuzzy controller.

Error range	Range name
(Error threshold high) $< \epsilon$	Positive huge
(Error threshold low) $< \epsilon \leq$ (error threshold high)	Positive large
$0 \leq \epsilon \leq$ (error threshold low)	Positive small
$-$(Error threshold low) $\leq \epsilon < 0$	Negative small
$-$(Error threshold high) $\leq \epsilon < -$ (error threshold low)	Negative large
$\epsilon < -$(error threshold high)	Negative huge

Table 2: Delta-error ranges for the fuzzy controller.

Delta-error range	Range name
(Delta error threshold) $< \Delta\epsilon$	Positive large
$0 \leq \Delta\epsilon \leq$ (delta error threshold)	Positive small
$-$(delta error threshold) $\leq \Delta\epsilon < 0$	Negative small
$\Delta\epsilon < -$(delta error threshold)	Negative large

Table 3: Control levels and their meanings.

Control levels	Meaning
-2	Aggressively reduce the monitored parameter
-1	Mildly reduce the monitored parameter
0	Maintain same value for the monitored parameter
+1	Mildly increase the monitored parameter
+2	Aggressively increase the monitored parameter

the monitored parameter before taking the next decision. In other words, settling width represents the duration between two consecutive control decisions. For example, in the case of frame-rate control, the settling width can be represented in terms of the number of frames between two consecutive control decisions.

Error Threshold Low and Error Threshold High. Threshold values divide the error axis into distinct intervals (i.e., error ranges). The decision taken by the controller depends on which interval in the error axis the current error value belongs to.

Delta Error Threshold. Similar to the error thresholds, delta-error threshold divides the delta-error axis into sub intervals (i.e., delta-error ranges). The interval in which delta-error falls also influences the decision of the controller.

Depending on which interval the value of error/delta-error falls, they are assigned a range value. Tables 1 and 2 give all the possible range values and the corresponding range names for errors and delta-errors, respectively.

Our controller implements five discrete levels of control as detailed in Table 3. For example, to reduce the monitored parameter aggressively, controller generates -2 as the control command. Similarly +1 at the controller output seeks for mild increase in the parameter. The interpretation of these discrete outputs is parameter dependent and has to be done by the adapter functions.

The decision making algorithm of the controller is summarized as follows.

(i) If error range is *positive huge* then control command is -2 (i.e., if the current value of the parameter is very much greater than the target value then seek to decrease it aggressively).

(ii) If error range is *positive large* and delta-error range is *negative large* then control command is 0 (i.e., if the current value of the parameter is greater than the target value and the error is decreasing at a very fast pace then seek to maintain previous situation. This means that the decision taken at the previous step was correct, so do not change anything).

(iii) If error range is *positive large* and delta-error range is not *negative large* then control command is -1 (i.e., if the current value of the parameter is greater than the target value and the error is not decreasing at a very fast pace then reduce the parameter mildly. This means that the decision taken at the previous step was not enough and further reduction of parameter value is needed).

(iv) If error range is *positive small* and delta-error range is *positive large* then control command is -1 (i.e., if the current value of the parameter is slightly greater than the target value and the error is increasing at a very fast pace then reduce the parameter mildly. This means that even though error is within the tolerance band it is deviating in the positive direction very fast, so try reducing the parameter value mildly).

(v) If error range is *positive small* and delta-error range is not *positive large* then control command is 0 (i.e., if the current value of the parameter is slightly greater than the target value and the error is not increasing at a very fast pace then seek to maintain previous situation. This means that error is smoothly maintaining its value within the tolerance limits, so no action needed).

(vi) If error range is *negative small* and delta-error range is *negative large* then control command is +1 (i.e., if the current value of the parameter is slightly lesser than the target value and the error is decreasing at an abrupt pace then increase the parameter mildly. This means that even though error is within the tolerance band it is deviating in the negative direction very fast, so try increasing the parameter value mildly).

(vii) If error range is *negative small* and delta-error range is not *negative large* then control command is 0 (i.e., if the current value of the parameter is slightly lesser than the target value and the error is not decreasing fast then nothing needs to be changed. This means that error is smoothly maintaining its value within the tolerance limits, so no action needed).

(viii) If error range is *negative large* and delta-error range is *positive large* then control command is 0 (i.e., if the current value of the parameter is much smaller than the target value and the error is increasing at a very

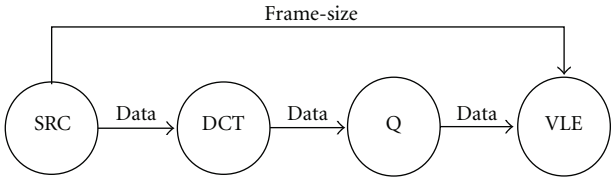


Figure 5: MJPEG encoder pipeline.

fast pace then seek to maintain previous situation. This means that the decision taken at the previous step was correct, so no action required).

(ix) If error range is *negative large* and delta-error range is not *positive large* then control command is +1 (i.e., if the current value of the parameter is much smaller than the target value and the error is not increasing at a very fast pace then seek to increase the parameter mildly. This means that the decision taken at the previous step was not enough and further increase of parameter is needed).

(x) If error range is *negative huge* then control command is +2 (i.e., if the current value of the parameter is very much smaller than the target value then seek to increase it aggressively).

Table 4 captures the behavior of the algorithm for all possible situations.

The functioning of the controller can be summarized as below. For every new received value of the monitored parameter, the controller decides whether to take a new control decision depending on the settling-width. If this input has to be ignored for a parameter then the corresponding adaptive task will be asked to maintain its previous situation (by sending 0 as the control command). On the other hand, if this input has to be considered for a parameter then following actions are performed. Error and delta-error for that parameter are calculated first. Then control algorithm will be run using these values to decide the control command. The generated command will be communicated to the respective adaptive task through the control channel.

4. Case Study: Motion JPEG (MJPEG)

This section presents MJPEG [14], a popular video compression standard, as a case study to demonstrate our framework. This algorithm is selected because its processes are coarse grained with high computation/communication ratio, a characteristic of an application suited for NoC-based MPSoCs. A typical MJPEG encoder pipeline is shown in Figure 5, where all the components can be modeled as KPN tasks.

Video Source (SRC). This component captures the input video frame-by-frame and feeds it to the succeeding components in the pipeline one block (8 × 8 pixels) at a time.

Discrete Cosine Transform (DCT). This component performs discrete cosine transform on each video block received from the SRC component and sends it to the Quantizer for further processing. DCT is widely used for multimedia compression algorithms such as MP3, JPEG, and MPEG, where high frequency components of less amplitude can be discarded without compromising quality.

Quantization (Q). Quantization refers to reducing the amplitude of a signal to achieve compression. In MJPEG, an 8 × 8 matrix of coefficients (QMatrix) is used for this purpose and the resultant data is rounded off to the nearest integer. The Quantizer also performs a 2D to 1D conversion of the quantized blocks by doing a zig-zag scan.

Variable Length Encoding (VLE). VLE is the last stage of MJPEG pipeline, where entropy (Huffman) encoding is done on the received video blocks. VLE also acts as the sink component, generating the final MJPEG stream by inserting headers/markers to indicate the start/end of each frame.

4.1. Self-Adaptive MJPEG. In this section, we present the implementation of the self-adaptive MJPEG encoder on our NoC-based platform using our M-C-A framework. This implementation supports autonomous control of bit-rate (BR) and frame-rate (FR) to match the target values set by the user. Bit-rate adaptation is achieved by controlling the quality of encoding (by scaling the QMatrix accordingly), whereas frame-size scaling is used to control the frame-rate.

Table 4: Adaptation control algorithm.

Control command		Delta-error			
		POS_ large	POS_ small	NEG_ small	NEG_ large
Error	POS_huge	−2	−2	−2	−2
	POS_large	−1	−1	−1	0
	POS_small	−1	0	0	0
	NEG_small	0	0	0	1
	NEG_large	0	1	1	1
	NEG_huge	2	2	2	2

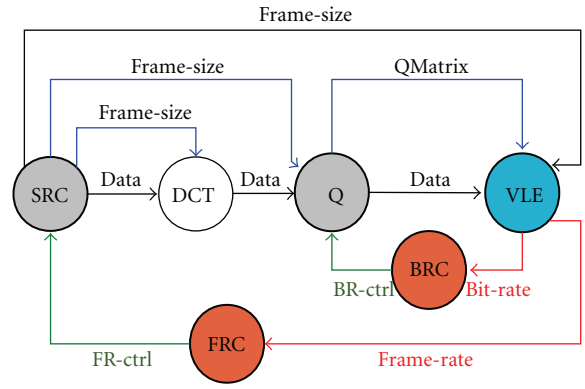


Figure 6: Adaptive-MJPEG encoder pipeline (see the color legend of Figure 1).

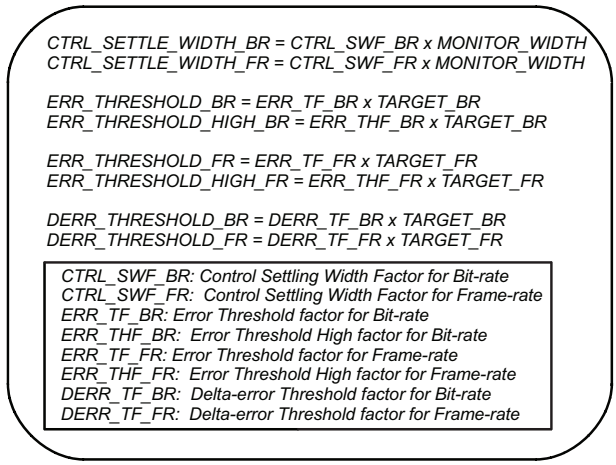


Figure 7: Settling widths and error thresholds for controller.

The modifications done on the MJPEG pipeline to make it self-adaptive are shown in Figure 6 and are as follows.

4.1.1. Monitoring VLE. The VLE task is equipped with monitoring capabilities (for bit-rate and frame-rate) by adding calls to the monitoring functions. Monitoring is done at the frame-level, hence these function calls are made after the task has accumulated all the blocks corresponding to one frame. Timestamp of a frame is measured by reading the hardware timer register of the NoC platform. Every time a new frame is generated, *alignMonitorWindow()* function is called with the frame-size and timestamp as arguments. Average values of the bit-rate and frame-rate are obtained by calling *calculateBitRate()* and *calculateTokenRate()* functions, respectively.

4.1.2. Controllers. We decided to implement two independent controllers, one for bit-rate and the other for frame-rate. The design principles are as detailed in the previous section. In our implementation, the settling-widths are specified as a fraction of the monitor-width whereas the threshold values of error and delta-error signals are taken as a percentage of the target parameter values. These fraction parameters are exposed such that they can be fine tuned by the user. Calculation of settling-widths and threshold values used in our implementation are shown in Figure 7. For every newly generated frame, the controllers receive the monitored values of bit-rate and frame-rate. Depending on the settling-width, they decide whether to take a new control decision. If a new control decision is needed, the fuzzy control algorithm will be run using the error and delta-error values for the input. The generated control-command for bit-rate is sent to the adaptive Quantizer task, whereas the frame-rate control-command is sent to the adaptive Source task.

4.1.3. Adaptive Quantizer. The quantization of the data has a direct impact on the generated bit-rate of the encoder. The output bit-rate can be adapted to the required level by scaling the QMatrix. For example, when the quantization coefficients are small, the output of the quantizer has more nonzero values and hence the VLE component will produce more bits per frame. On the other hand, when the input data is quantized using large quantization coefficients, fewer bits will be generated per frame. Figure 8 shows the output bit-rates for various scaling factors of QMatrix in case of a slow, 128×128 pixel present the results of running our self-adaptive MJPEG encoder onel video.

To make the quantizer adaptive, adaptBitrate() function is implemented, which takes the control command from the bit-rate controller as input. The implemented bit-rate adapter logic supports two levels of scaling for the QMatrix-aggressive and mild. The algorithm maintains three parameters (configurable by the user), namely QuantScaleCoeff, AggrQScaleFactor, and MildQScaleFactor to perform the adaptations.

QuantScaleCoeff (Quantization Scaling Coefficient). The coefficient by which all Q-Matrix coefficients will be multiplied to produce its scaled version.

AggrQScaleFactor (Aggressive Quantization Scaling Factor). The constant by which previous value of QuantScaleCoeff will be multiplied/divided to obtain its current value in case of aggressive scaling.

MildQScaleFactor (Mild Quantization Scaling Factor). The constant by which the previous value of QuantScaleCoeff will be multiplied/divided to obtain its current value in case of mild scaling.

The bit-rate adapter works as follows. Before reading the data for a new frame, the quantizer task reads the bit-rate control command from the controller and calls adaptBitrate() function with this value. If the decision by the controller is to aggressively decrease the bit-rate, the current value of the *QuantScaleCoeff* will be multiplied by AggrQScaleFactor to obtain its new value. On the other hand, if the adapter is asked to mildly increase the bit-rate, previous value of *QuantScaleCoeff* will be divided by *MildQScaleFactor* to get its new value. The value of *QuantScaleCoeff* will be left unchanged to keep the bit-rate at

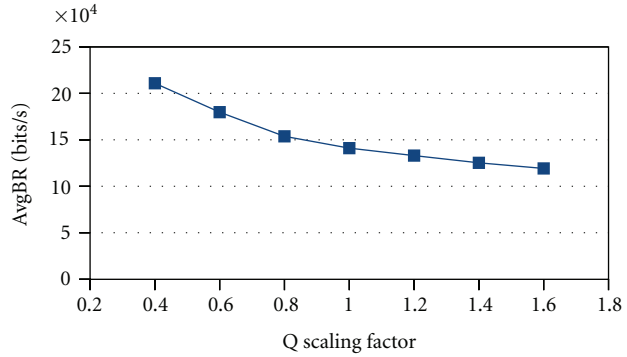


Figure 8: Bit-rate variation with respect to QMatrix scaling.

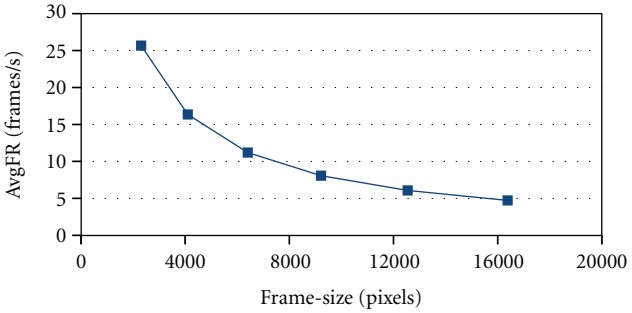


Figure 9: Frame-rate variation with respect to frame-size scaling.

the current level. Once the new value for the *QuantScaleCoeff* is decided, the scaled version of the QMatrix is calculated by multiplying all its elements by this new *QuantScaleCoeff*. For all the blocks of the frame, this scaled version of the QMatrix will be used. The quantizer task also sends the newly generated QMatrix to the VLE through a dedicated adaptation propagation channel so that it can be inserted in the frame header of the generated frame.

4.1.4. Adaptive Source. The output frame-rate is decided by how fast the encoder can complete the processing of one frame. Since the processing time for a frame is proportional to the amount of data contained in it, frame-rate can be controlled by scaling the dimensions of the input video. Even though this will produce smaller images at the output, target frame-rate can be easily achieved by using this method. Figure 9 shows the impact of the frame-size parameter on the output frame-rate.

The source task is made adaptive by providing the adaptFramerate() function, which takes care of scaling the input frame size. The implementation of frame-size scaling logic is based on the following configurable parameters.

CurFrameNumVBlocks. The number of vertical blocks in the current frame.

CurFrameNumHBlocks. The number of horizontal blocks in the current frame.

AggrFsScaleFactor (Aggressive Frame-Size Scaling Factor). The constant by which current value of frame-size (number of vertical and horizontal blocks) will be multiplied/divided to obtain its new value in case of aggressive scaling.

MildFsScaleFactor (Mild Frame-Size Scaling Factor). The constant by which current value of frame-size (number of vertical and horizontal blocks) will be multiplied/divided to obtain its new value in case of mild scaling.

The algorithm functions as follows. Similar to the bit-rate adaptation, the Source task reads the frame-rate control command from the controller and passes this value to the adaptFrameSize() function. If the decision by the controller is to aggressively decrease the frame-rate, the previous values of the *curFrameNumVBlocks* and *curFrameNumHBlocks* will be multiplied by *AggrFsScaleFactor* to obtain their new values. Similarly, if the adapter is asked to mildly increase the frame-rate, these parameters will be divided by *MildFsScaleFactor* to calculate their new values. To keep frame-rate at the current level, the number of blocks in the frame will be left unchanged. The frame-rate adapter also sends the new value of the frame-size to DCT, Q, and VLE tasks using separate adaptation propagation channels so that they know exactly how many blocks to be processed for the next frame.

4.1.5. Adaptation Propagation Channels. Some additional channels need to be added to the pipeline to communicate the changes done by the adaptive tasks to other tasks. A channel to send the scaled version of the QMatrix from Quantizer to VLE is added. This is necessary because the QMatrix used for a particular frame needs to be inserted in its header so that the decoder can use the correct value while decoding the frame. Channels to propagate new frame-size values are also added between Source-DCT and Source-Q tasks. Quantizer and DCT should know the frame-size to calculate the number of blocks to be processed for each frame. To send the frame-size values from Source to VLE, we use the existing channel in the original task graph.

5. Results

In this section, we present the results of running our self-adaptive MJPEG encoder on a 2 × 2 NoC-based FPGA platform. The platform is generated by the SHMPI builder tool [15] and it is a mesh-based 2 × 2 NoC consisting of Microblaze processors emulated on a Xilinx Virtex6 FPGA. The software stack enabling the execution of KPN applications on this platform is based on the request-driven middleware explained in [12].

5.1. Design Space Exploration for Adaptation Control. As evident from the design of M-C-A framework, the quality of the adaptation control is influenced by various parameters used inside the monitors, controllers, and adapters. In order to achieve smooth and fast adaptation, a careful selection of these parameters is needed. To find such a combination, a design space exploration (DSE) is performed. The first step in DSE is to determine the design space for the configurable parameters. The design space tends to be enormous due to the large number of parameters and the different values each can assume. To carry out the DSE within a reasonable time,

Table 5: Design space for the M-C-A framework.

Parameter	Values
Monitor width	6, 12, 20
Settling width factor (BR)	0.1, 0.2
Settling width factor (FR)	0.1, 0.2
Error threshold factor (BR)	0.05, 0.1
Error threshold factor high (BR)	0.15, 0.2
Delta-error threshold factor (BR)	0.03
Error threshold factor (FR)	0.1, 0.2
Error threshold factor high (FR)	0.2, 0.3
Delta-error threshold factor (FR)	0.05
Mild Q scaling factor	1.1, 1.2
Aggressive Q scaling factor	1.4, 1.6
Mild frame-size scaling factor	1.1, 1.2
Aggressive frame-size scaling factor	1.25, 1.4

we chose only a few values for each parameter. The design space used is captured in Table 5.

The evaluation of a design point is based on the following two metrics speed of adaptation (quantified by rise/fall time) and convergence of adaptation (quantified by mean absolute error).

Rise/Fall Time. It is the time (in number of frames) taken by the system to move from an initial state to the target state.

Mean Absolute Error. It is the mean of absolute error values for a monitored parameter over several consecutive frames.

To calculate these two metrics for a monitored parameter, the encoder is run for a fixed number of frames of a test video with an initial value of the parameter. Then its value is changed to the target value and the system is allowed to adapt. The number of frames taken for the parameter to reach within a tolerance band (±5%) about its target value is the rise/fall time. The absolute error value for the parameter is calculated for all frames starting from where it reached the tolerance band till the last frame. The mean of these absolute error values gives the mean absolute error. We have used the following sets of goals for the DSE experiments: initial BR = 200000 bits/sec, initial FR = 8 frames/sec, final BR = 300000 bits/sec, and final FR = 16 frames/sec.

Since both bit-rate and frame-rate control are considered in the case study, the DSE is a four-dimensional minimization problem consisting of rise-times and mean-absolute-errors for BR and FR control as objectives. In order to simplify the procedure, the following facts are taken into account. Bit-rate adaptation has no impact on the frame-rate, since it only scales the QMatrix (i.e., data to be processed per frame does not change). On the other hand, frame-rate adaptation affects also the bit-rate, since it changes the amount of data generated per frame. So, initially the DSE is done by varying only those parameters that affect the frame-rate. The bit-rate controller is turned off during this step to obtain the optimum frame-rate control parameters. This step is a two-dimensional optimization problem over a smaller design space. The results of the first DSE stage are presented in Figure 10. The Pareto points are represented with rectangular markers, whereas the selected Pareto point is colored in red. In the second step, only bit-rate control parameters are varied while using the values of the selected Pareto point from the first step for frame-rate control parameters. Both bit-rate and frame-rate control are enabled in this step. Pareto points are obtained with respect to the four optimization objectives. The selected parameter values after each DSE step is shown in Table 6.

Table 6: Two step DSE for adaptation control.

Parameter	After step 1	After step 2
Monitor width	—	12
Settling width factor (BR)	—	0.2
Settling width factor (FR)	0.2	0.2
Error threshold factor (BR)	—	0.05
Error threshold factor high (BR)	—	0.2
Delta-error threshold factor (BR)	—	0.03
Error threshold factor (FR)	0.2	0.2
Error threshold factor high (FR)	0.3	0.3
Delta-error threshold factor (FR)	0.05	0.05
Mild Q scaling factor	—	1.1
Aggressive Q scaling factor	—	1.6
Mild frame-size scaling factor	1.1	1.1
Aggressive frame-size scaling factor	1.25	1.25

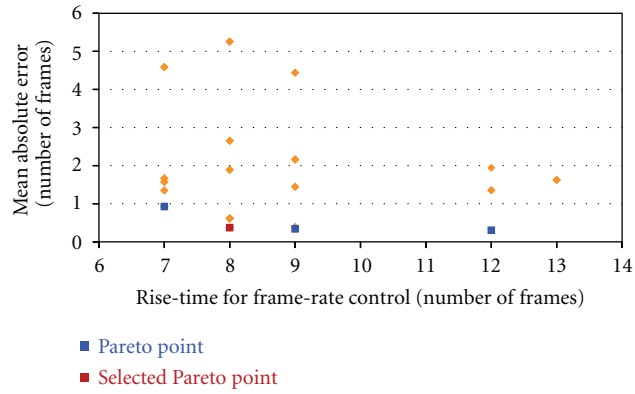


Figure 10: DSE for frame-rate control (step 1).

In order to assess the sensitivity of the control quality to the design parameters, we calculated the distribution of all the points in the design space. The cumulative distributions of error and rise-time for bit-rate are shown in Figure 11. It can be seen that 95% of the design points have less than 5% bit-rate error, whereas the rise-time of 90% of them are below 8 frames. Similar plots for frame-rate are shown in Figure 12. Here 80% of the design points have less than 12% frame-rate error, whereas the rise-time of 84% of them are below 9 frames. This shows the generality of the proposed solution, because even for nonoptimal parameter configurations, the system is able to adapt fast while keeping the error within tolerable limits.

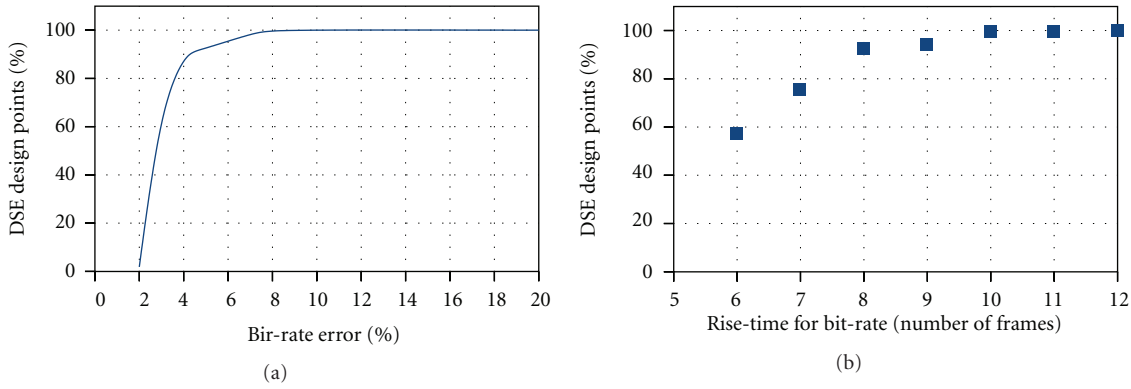


Figure 11: Cumulative distribution for bit-rate control.

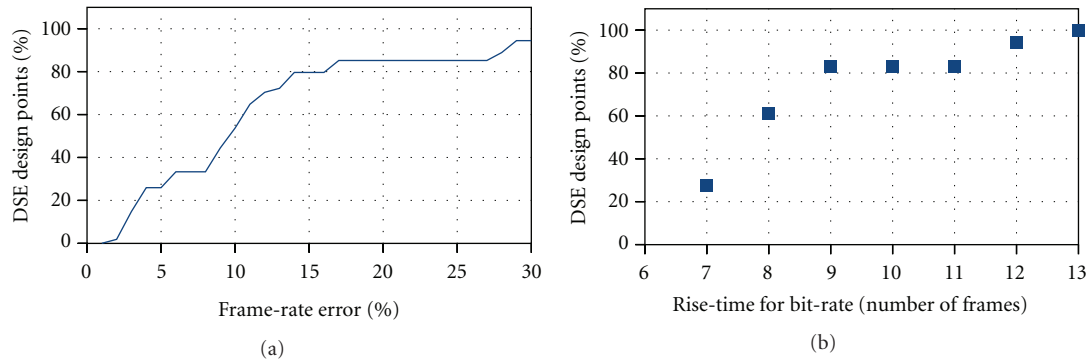


Figure 12: Cumulative distribution for frame-rate control.

5.2. Bit-Rate and Frame-Rate Adaptation Tests. To demonstrate the effectiveness of our adaptation scheme, we conducted various experiments by setting different goals for bit-rate and frame-rate. All these tests are carried out using 128 × 128 video (for 200 frames) stored in the memory. The selected Pareto point (shown in Table 6) obtained from the DSE is used to configure the controller. Figure 13 shows the results when the encoder is run with initial BR = 200000 bits/sec, initial FR = 8 frames/sec, final BR = 300000 bits/sec and final FR = 16 frames/sec. The adaptation in terms of quantization scaling coefficient and frame-size is also shown. It can be seen that the scaling coefficient is reduced from its initial value of 1 to a value of 0.5 to meet the initial bit-rate. But after frame 60, its value is further reduced to 0.35 to increase the bit-rate to its final value. Similarly, the frame-size is reduced from its initial value of 16000 pixels to 9000 pixels in order to achieve the initial frame-rate of 8 fps. But after frame 60, it is further reduced to about 4000 pixels to increase the frame-rate to 16 fps. The rise time and mean absolute errors for this scenario are: rise time (BR) = 6 frames, mean absolute error (BR) = 7684 bits (2.56%), rise time (FR) = 9 frames, mean absolute error (FR) = 0.33 frames (2.06%).

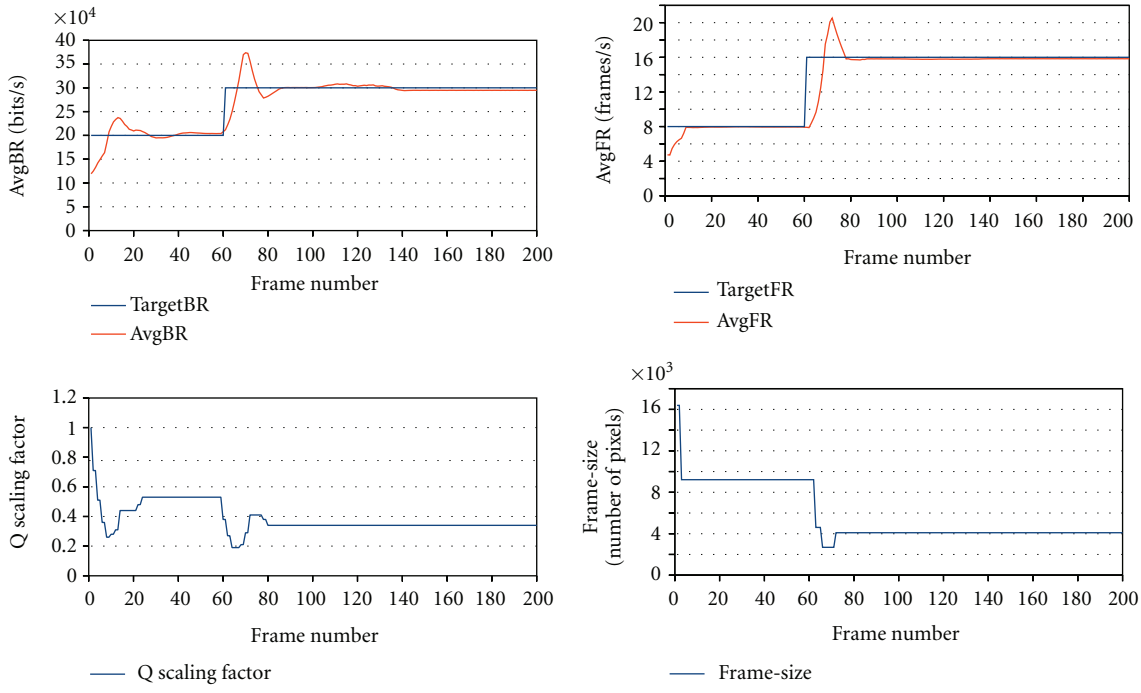
5.3. Fast Video versus Slow Video. Figure 14 shows the results of evaluating the framework using slow and fast video inputs. Figure 14(a) characterizes the two videos in terms of the number of bytes generated by the encoder per frame (for 128 × 128 video), when there is no bit-rate/frame-rate control. From Figures 14(b) and 14(c), it can be seen that for both videos, the targets are achieved with the following metrics.

Slow Video. Rise time (BR) = 6 frames, mean absolute error (BR) = 7790 bits (2.59%), rise time (FR) = 9 frames, mean absolute error (FR) = 0.36 frames (2.25%).

Fast Video. Rise time (BR) = 5 frames, mean absolute error (BR) = 10440 bits (3.48%), rise time (FR) = 9 frames, mean absolute error (FR) = 0.24 frames (1.5%).

The results reveal that for slow video the bit-rate control converges fast whereas for fast video, a lot of ripples are observed at the output, resulting in a higher mean absolute error. In case of frame-rate control the fastness or slowness of the input does not have much impact and the frame-size converges to the same value in both cases without ripples.

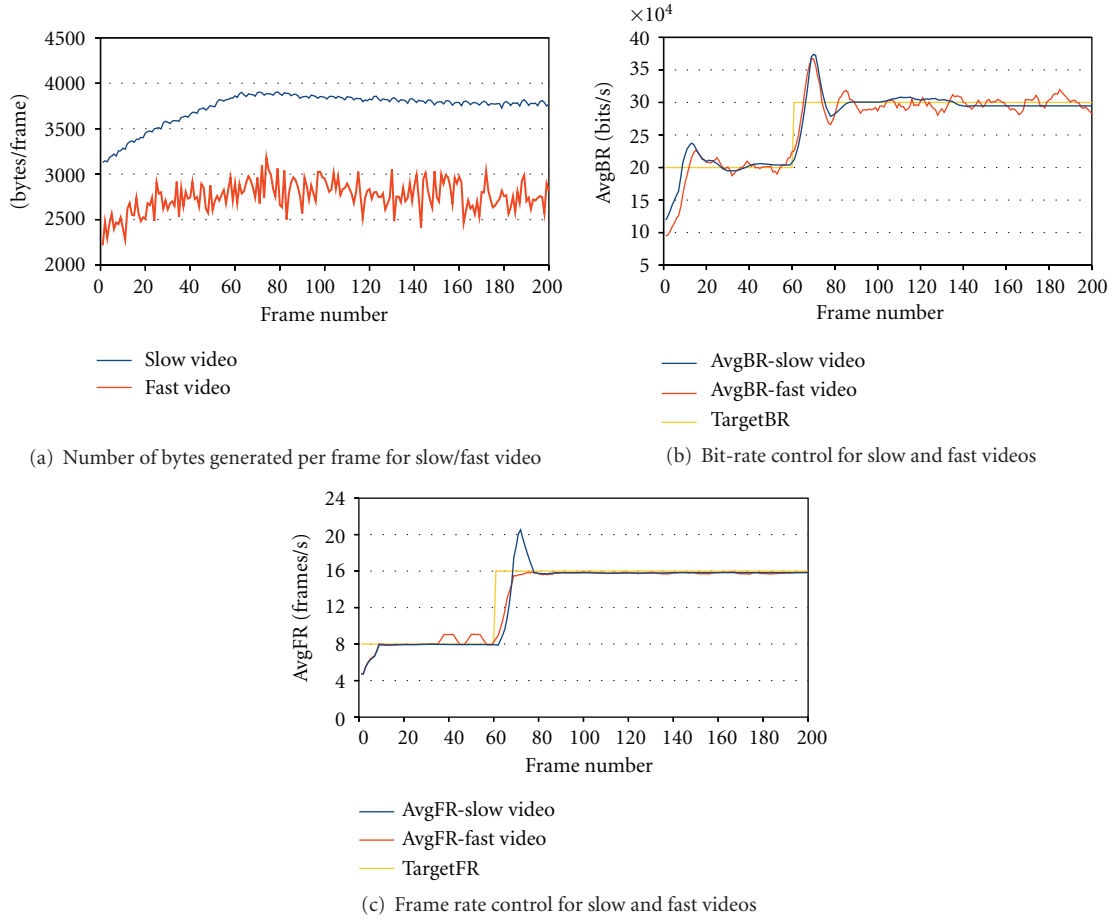
5.4. Cost of Adaptation. To measure the steady-state overhead due to the introduction of the M-C-A feedback loop in the application pipeline, the following procedure is used. First, the encoder is run without the feedback loop as well as the adaptation propagation channels to obtain the average value of frame-rate without the framework. The experiment is repeated after introducing the M-C-A loop



(a) AvgBR and Q scaling factor variations

(b) AvgFR and Frame-size variations

Figure 13: Results for initial BR = 200000 bps, initial FR = 8 fps, and final BR = 300000 bps, final FR = 16 fps.



(a) Number of bytes generated per frame for slow/fast video

(b) Bit-rate control for slow and fast videos

(c) Frame rate control for slow and fast videos

Figure 14: Results for bit-rate/frame-rate control for slow and fast videos.

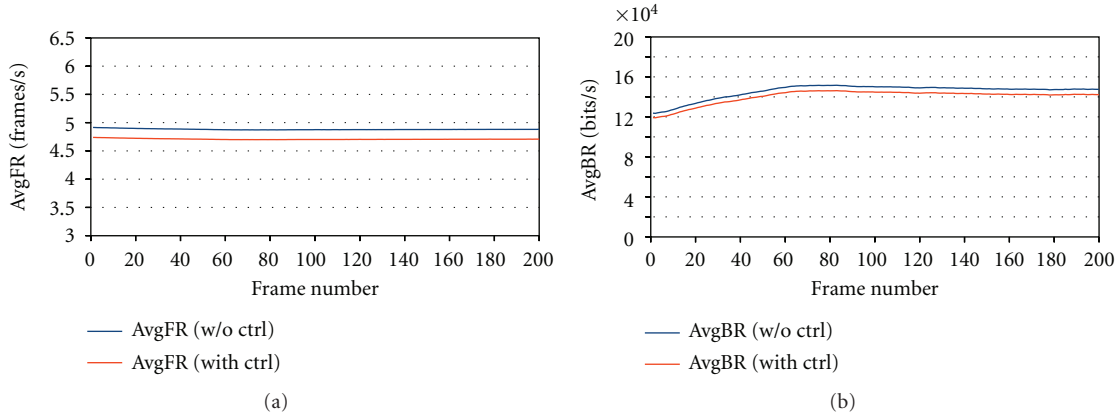


Figure 15: Cost of adaptation in terms of reduction in BR/FR.

and the additional channels to obtain the reduced frame-rate. In this case, both bit-rate and frame-rate control are turned off inside the controller, since only the overhead due to the framework needs to be measured. Figure 15 depicts the outcome of this test for a 128 × 128 test video. It is observed that the introduction of the framework results in a frame-rate reduction of only 4%. Similarly, the reduction in bit-rate is about 3.5%. The reduction in the bit-rate and frame-rate is due to the increase in the inter-arrival time between frames.

The overhead in terms of the additional control data introduced by our M-C-A mechanism is minimal. For every video frame, it sends a total of 72 additional tokens over the network. This includes one token from monitoring task to bit-rate controller, one token from monitoring task to frame-rate controller, one token from bit-rate controller to Quantizer task, one token from frame-rate controller to Source task, 64 tokens from Quantizer to VLE (to send the QMatrix), two tokens from Source to DCT (to send the height and width of the frame), and two tokens from Source to Quantizer (to send the height and width of the frame). This is equivalent to 288 bytes of data since a token is represented as integer type by the middleware. For a 128 × 128 frame the total video data to be sent over the NoC is 49152 bytes. This includes the pixel data sent from Source to DCT, DCT to Quantizer, and Quantizer to VLE. So the framework introduces approximately 0.5% of additional control data.

5.5. Effect of Parameter Variations. This section presents the experimental results regarding the effect of varying various design parameters on the quality of adaptation control.

5.5.1. Monitor-Width. Figure 16 shows the impact of varying the monitor-width on the four quality metrics of the controller. Monitor-width plays an important role in deciding the sensitivity of the control mechanism. If the monitor-width is too large the sensitivity will be low, because the effect of a particular adaptation decision will be reflected in the average value only after many frames are generated with that decision, resulting in an increase in rise/fall time. On the other hand, very small monitor windows will help in detecting changes in the monitored parameter at a very early stage. However, this may cause large ripples in the output since any adaptation strategy needs some time for its effect to be visible at the output, resulting in large errors in the monitored parameter. Similarly, when the monitor-width is large, the error will increase due to slow response of the controller. Hence it is very important to keep the monitor-width at an optimum value.

5.5.2. Q Scaling Factors. Figure 17 shows the effect of variations in Q scaling factors on the quality of control. Adapters deploy aggressive scaling when the monitored value of the parameter deviates too much from the target, whereas mild scaling is used otherwise. A high value of aggressive scaling factor will help to reduce the rise/fall time, but it may cause large overshoots in the output and hence may increase the average error. Similarly, a small value for mild scale factor will help in reducing the ripples after the output converges. From the results, it can be deduced that large values for Q scaling factors will cause larger average errors in the output, in spite of the reduction in rise/fall time.

5.5.3. Error Thresholds. Figure 18 shows the effect of variations in error thresholds on the bit-rate control. The *x*-axis represents bit-rate error thresholds in the format (error-threshold-low, error-threshold-high, delta-error-threshold). From the results, it can be seen that when *error-threshold-low* is increased keeping *error-threshold-high* as constant, the mean-error increases. This is due to the possibility of the monitored parameter settling at a value which is far from its target, thus increasing the error. On the other hand, when *error-threshold-high* is increased keeping *error-threshold-low* as constant, the rise-time increases. This can be explained as follows: when *error-threshold-high* is large, aggressive scaling is used less often, causing an increase in the rise-time.

5.6. Reusing the Adaptation Controller. The results presented in Section 5.1 use the MJPEG encoder case study and are performed with the aim of minimizing both rise-time and mean-error. The framework requires fine tuning in order to be used for a different application. The effect of parameter

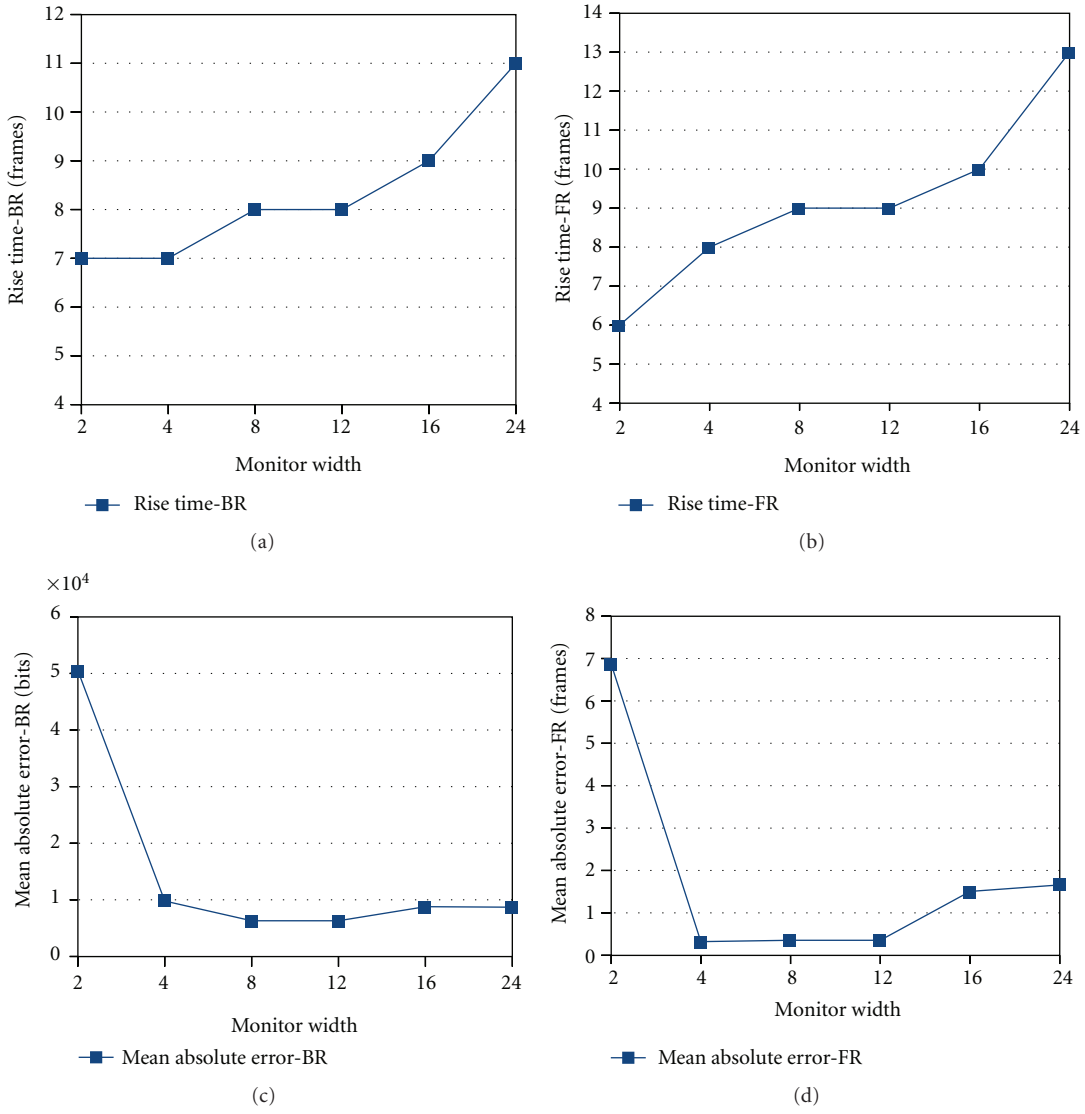


Figure 16: Effect of monitor-width on the quality of control.

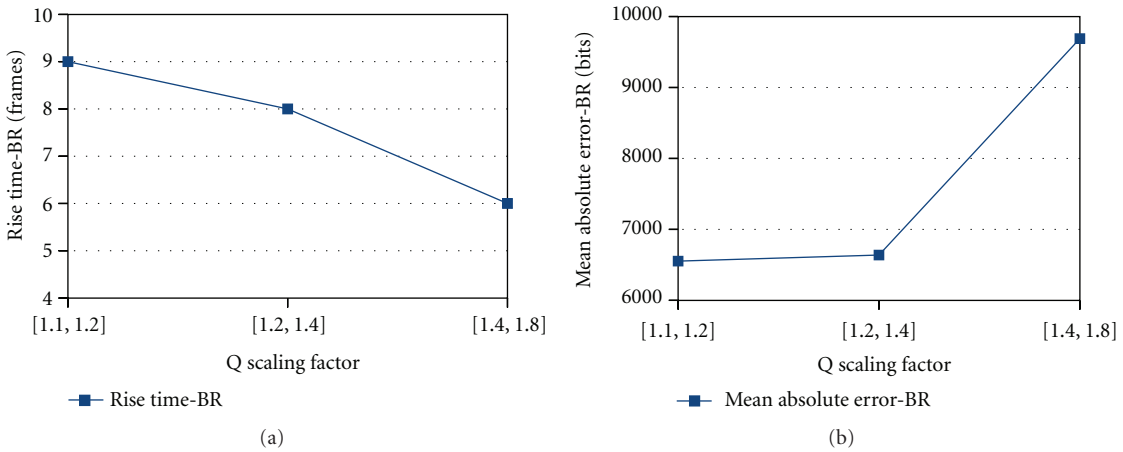


Figure 17: Effect of Q scaling factors on quality of control.

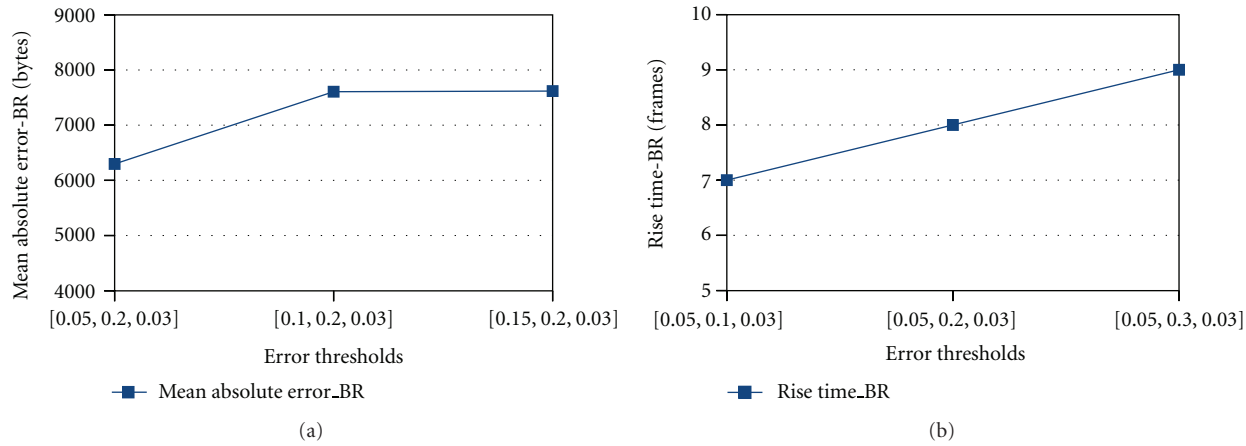


Figure 18: Effect of error thresholds on quality of control.

variations studied in Section 5.5 can be used as a guideline while configuring the adaptation controller for a new set of application requirements. For example, to increase the responsiveness of the control, either the *aggressive scaling factor* can be increased or the *error-threshold-high* can be reduced. Similarly, if the application demands a closer convergence of a parameter to the target value while tolerating slow response, the user can tighten the *error-threshold-low* and lessen the *aggressive scaling factor*.

6. Discussion on the Framework

The self-adaptivity mechanism proposed in this work relies on monitoring, controlling and adaptation capabilities. For the monitoring and adaptation support, despite the fact that some general mechanisms such as monitoring and adaptive functions are used, the methods are based on some advantages that come with the KPN computation model. In this work, we are particularly interested in throughput monitoring and parametric adaptations. The KPN model facilitates implementation of such monitoring and adaptation capabilities. For the former, since KPN is composed of computational blocks and their explicit communication with tokens over channels, monitoring the throughput (e.g., the rate at which tokens are produced as well as the bit-rate on a channel) can be achieved in an application independent manner. For the latter, the reconfiguration of the application to work with a new value of an application parameter requires that the relevant parts of the application to be updated consistently. Consistency implies that a token is processed by tasks throughout the application pipeline with the right parameter value. KPN helps in achieving this by synchronizing the updating of tasks via blocking channels. These properties of the KPN model address the aforementioned *separation of concerns* challenge.

On the other hand, the fuzzy control approach is not specific to KPN and can be used for controlling any self-adaptive system. The control is event-based rather than time-based. Unlike the widely practiced periodic monitoring and control, this approach involves monitoring in an event-based manner (e.g., at the end of processing of a frame). Such an approach is suited better for networked systems as it is less sensitive to possible delays in the network and incurs less overhead on the amount of data transferred on the network. Comparison of these approaches is left as a future work.

7. Conclusion

In this paper, we proposed an approach to implement application level self-adaptation capabilities for KPN applications running on networks-on-chip based MPSoCs. The proposed framework is based on introducing a monitor-controller-adapter mechanism in the application pipeline. Techniques to add monitoring and adaptation capabilities to normal KPN tasks are discussed along with the design of a generic fuzzy logic-based adaptation controller. Finally, we presented an adaptive MJPEG case study on a FPGA based 2×2 NoC platform. Our results show that even if the parameters of the fuzzy control are not tuned optimally, the adaptation convergence is achieved within reasonable time and error limits for most of the designed controllers. Moreover, the steady-state overhead introduced due to the framework is low (4%) in terms of frame-rate reduction. Since the controller is a generic one, this framework can be easily integrated to other applications also, requiring minimal modifications to the code.

Acknowledgments

This work was funded by the European Commission under the project MADNESS (no. FP7-ICT-2009-4-248424). The paper reflects only the authors' view; the European Commission is not liable for any use that may be made of the information contained herein.

References

[1] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools*, Morgan Kaufmann, 2006.

[2] E. Carara, A. Mello, and F. Moraes, "Communication models in networks-on-chip," in *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP '07)*, pp. 57–60, IEEE Computer Society Press, May 2007.

[3] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Proceedings of the Future of Software Engineering (FoSE '07)*, pp. 259–268, Washington, DC, USA, May 2007.

[4] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress on Information Processing*, J. L. Rosenfeld, Ed., pp. 471–475, North-Holland, New York, NY, 1974.

[5] O. Derin and A. Ferrante, "Enabling self-adaptivity in component based streaming applications," in *SIGBED Review, Special Issue on the 2nd International Workshop on Adaptive and Reconfigurable Embedded Systems (APRES '09)*, vol. 6, 2009.

[6] P. Charles David and T. Ledoux, "Towards a framework for self-adaptive component-based applications," in *Proceedings of Distributed Applications and Interoperable Systems*, vol. 2893 of *Lecture Notes in Computer Science*, pp. 1–14, Springer, 2003.

[7] K. Geihs, P. Barone, F. Eliassen et al., "A comprehensive solution for application-level adaptation," *Software - Practice and Experience*, vol. 39, no. 4, pp. 385–422, 2009.

[8] O. Derin, A. Ferrante, and A. V. Taddeo, "Coordinated management of hardware and software self-adaptivity," *Journal of Systems Architecture*, vol. 55, no. 3, pp. 170–179, 2009.

[9] J. Combaz, J. C. Fernandez, T. Lepley, and J. Sifakis, "Fine grain QoS control for multimedia application software," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, vol. 2, pp. 1038–1043, March 2005.

[10] J. Combaz, J. C. Fernandez, J. Sifakis, and L. Strus, "Using speed diagrams for symbolic quality management," in *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS '07)*, pp. 1–8, March 2007.

[11] M. Jaber, J. Combaz, L. Stras, and J. C. Fernandez, "Using neural networks for quality management," in *Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA '08)*, pp. 1441–1448, September 2008.

[12] E. Cannella, O. Derin, P. Meloni, G. Tuveri, and T. Stefanov, "Adaptivity support for mpsocs based on process migration in polyhedral process networks," *VLSI Design*, vol. 2012, Article ID 987209, 17 pages, 2012.

[13] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965.

[14] P. Lieverse, T. Stefanov, P. Van der Wolf, and E. Deprettere, "System level design with spade: an M-JPEG case study," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD '01)*, pp. 31–38, November 2001.

[15] P. Meloni, S. Secchi, and L. Raffo, "An FPGA-based framework for technology-aware prototyping of multicore embedded architectures," *IEEE Embedded Systems Letters*, vol. 2, no. 1, pp. 5–9, 2010.