A Low Overhead Self-adaptation Technique for KPN Applications on NoC-based MPSoCs

Onur Derin[1], Prasanth Kuncheerath Ramankutty[1], Paolo Meloni[2] and Giuseppe Tuveri[2]
[1]*ALaRI - Faculty of Informatics, University of Lugano, Via G. Buffi 13, 6904 Lugano, Switzerland*
[2]*DIEE - Faculty of Engineering, University of Cagliari, 09123 Cagliari, Italy*

Keywords: Kahn Process Networks, Networks-on-Chip, Self-adaptation, Event-based Control.

Abstract: Self-adaptive systems are able to adapt themselves to mutating internal/external conditions so as to meet their goals. One of the challenges to be tackled when designing such systems is the overhead introduced in making the system monitorable and adaptable. A large overhead can easily compensate the benefits of adaptation. In this work, we are addressing this challenge within the context of KPN applications on NoC-based MPSoCs. In particular, parametric adaptations at the application level are considered. We present a low overhead technique for the implementation of the monitor-controller-adapter loop, which is present in self-adaptive systems. The technique is fundamentally based on an extended network interface which provides the ability to interrupt remote tiles on a NoC-based multiprocessor platform. Results from the MJPEG case study show that the proposed interrupt-based approach incurs an overhead as low as 0.4% without compromising the quality of the adaptation control. Our new technique provides an improvement of approximately 6.25% compared to another state-of-the-art technique that interacts with the application using KPN semantics (i.e., blocking channels). Moreover, the sensitivity of the overhead to the complexity of the adaptation controller is much lower in case of our interrupt-based technique as compared to the blocking channel based scheme.

1 INTRODUCTION

There has been a paradigm shift in the design of processors in order to achieve scalable performances by moderately increasing the power consumption. Processors are being integrated on the same chip with ever-increasing numbers alongside a network-on-chip (NoC) (Micheli and Benini, 2006) as the communication backbone. Memory organization and programming models are also affected by these changes. No remote memory access (NORMA) and message passing models allow scalability on such platforms.

As systems become more complex, modeling their behavior and analyzing their performances become more challenging. Offline analysis may prove to be more difficult or impossible. Even if possible, changing internal or external conditions may turn the optimal configuration chosen at design time into a non-optimal one. Self-adaptivity has emerged as a solution to these issues. Self-adaptivity is the ability of a system to adapt itself dynamically to changing internal and external conditions. It allows systems to meet their non-functional goals such as high performance, high dependability and low power consumption.

An important challenge in designing self-adaptive systems is the implementation of monitoring and adaptation capabilities with low overheads. There are two types of overhead. The *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 *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.

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) (Kahn, 1974) model due to its suitability for NORMA-based NoC multiprocessor systems-on-chip (MPSoCs). KPN is a stream-oriented computation model, where an application is organized as streams and computational blocks; streams represent the flow of data, while com-

262 Derin O., Kuncheerath Ramankutty P., Meloni P. and Tuveri G..
A Low Overhead Self-adaptation Technique for KPN Applications on NoC-based MPSoCs.
DOI: 10.5220/0004389802620269
In *Proceedings of the 3rd International Conference on Pervasive Embedded Computing and Communication Systems* (SANES-2013), pages 262-269
ISBN: 978-989-8565-43-3
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

putational 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.

The particular focus of this paper is the self-adaptivity overhead 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 a low overhead technique for the implementation of the monitor-controller-adapter (MCA) loop based on an extension in the Network Interface (NI) which allows interrupting remote tiles on the NoC platform. In doing so, the realization of self-adaptation is investigated with regard to two interaction styles between the MCA and the application. Experimental results are provided comparing these approaches.

The rest of the paper is organized as follows. An overview of the related work is provided in Section 2. Section 3 describes the NoC-based platform and inter-processor interrupt support. Section 4 presents the proposed interrupt-based self-adaptation technique. Section 5 explains shortly the MJPEG case study, followed by the experimental results in section 6. Finally, section 7 concludes the paper.

2 RELATED WORK

Run-time management on MPSoC platforms have gained an increasing attention in the recent years. It can be used for various purposes such as management of quality of service, power, temperature, variability as well as load balancing and fault tolerance. (Nollet et al., 2010) surveys several works from the academia and the industry that address different aspects of run-time adaptation. In terms of their classification criteria, our work can be classified as quality management since we address application-level performance goals. It can be incorporated into the generic self-adaptive run-time environments proposed in (Nollet et al., 2010; Derin et al., 2009) and reside alongside other self-adaptation services such as fault-tolerance (Meloni et al., 2012).

Some emerging NoC-based multi-core architectures provide inter-processor interrupt (IPI) support. For example, Tilera's Tile64 can deliver interrupts to notify user-space processes of message arrival (Wentzlaff et al., 2007). This allows it to support both polling and interrupt-based message delivery. Intel's SCC also provides a hardware message passing mechanism that triggers an interrupt on the receiving core, before returning from a *write()* call (Mattson et al., 2010). The OpenScale platform (Busseuil et al., 2011), which is quite similar to the presented NoC-based MPSoC, can handle packet reception by both interrupt and polling methods; the interrupt occurs when the number of elements inside the incoming FIFO reaches a given threshold. Anyway, to the best of our knowledge there are no works evaluating the impact of the use of interrupts on the overhead related to the self-adaptivity of the system.

The self-adaptation framework presented in (Derin et al., 2012) (hereafter referred to as MCA-EB, short for *event-based MCA using blocking channels*) uses blocking FIFO channels for the interaction between MCA and the application tasks. Fig. 1(a) and Fig. 1(b) shows a simple KPN application and its self-adaptive version based on this framework respectively. MCA-EB 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 (APC) alongside the original task graph. MCA-EB deploys event-driven control (Sandee, 2006), where the monitor and control actions are triggered upon the generation of a specific application event. Such an event causes the monitoring task to measure the required application parameter and sent it to the controller task using the monitor channel (MC). The controller task reads this data from the blocking channel, runs the control algorithm to generate the control command and sends it to the adaptive task using the control channel (CC). Adaptive task reads these commands from the blocking channel (at a predefined place in the code) and performs the adaptation.

Self-adaptive applications built using MCA-EB approach suffer from reduced throughput due to pipeline stalls caused by the blocking-channel interaction between the application tasks and the controller. That is, for every event that triggers the adaptation control, the adaptive tasks in the application pipeline have to wait for the control algorithm to finish, thus preventing the processing of subsequent input data by them. This causes the pipeline to be emptied and filled often, thus reducing the throughput.

In P^3N (Zhai et al., 2011), a parameterized polyhedral process network model is defined to support run-time parametric adaptations. P^3N allows analyzing where and how parameter values can be changed dynamically and consistently according to dependence relations between parameters. It also enables extracting the dependence relation between dependent parameters at design time. As explained later, in our approach, the identification of the adaptation execution points and adaptation propagation channels is to be done manually by the application programmer.

3 INTER-PROCESSOR INTERRUPT SUPPORT IN NoCs

The platform proposed in (Meloni et al., 2012) is adopted as the hardware base for this work. In the proposed approach the system architecture can be seen as a network of tiles, interconnected by means of a NoC communication infrastructure.

The communication network is built by using an extended version of the the ×pipes-lite library of synthesizable components (Bertozzi and Benini, 2004). Network Interfaces (NI) are in charge of constructing the packets on the basis of the communication transactions requested by the cores. NIs have been extended with support for message-passing communication model. A programmable message manager with DMA capabilities is integrated with the NI inside a module called Network Adapter (NA). Communication and synchronization mechanisms are managed accessing memory-mapped registers at the network interfaces.

3.1 Programming Model

Reference primitives implementing message-passing communication are built, according to the general definition of such model, upon two base functions: *send()* and *receive()*. These two primitives are implemented in C, and interact with memory mapped registers inside the NA. According to the usual message-passing signatures, to send a message with a *send()*, the programmer has to specify the address (*SendAddress* hereafter) inside the private memory that contains the information to be sent (message data), a tag assigned to the message (*SendTag*), the size of the transfer (*SendDim*), and the ID of the destination processor (*SendID*). The *receive()* parameters are the tag of the expected message (*ReceiveTag*), the sender ID (*ReceiveID*) and the address where the received message data has to be stored (*ReceiveAddress*). Two implementations of the *receive()* are provided, with blocking and non-blocking behavior.

3.2 Message Passing Support

The *Network Adapter* architecture is connected to dual port data and instruction private memories, in order to allow the processor to continue with the execution while the other ports are used to load/store data from/to the memory in case of message send/receive. The *NA* integrates a module called *DMA message-passing handler* (MPH), embedding the memory-mapped registers to be programmed by the processor, when controlling send and receive operations, to set the needed primitive parameters.

It also includes an address generator in charge of generating the addresses when the private memories must be accessed from the port reserved for message passing.

When the processor wants to call a *send()*, the microcode that implements the primitive stores the values of primitive parameters into a set of send-related memory-mapped registers. As soon as the registers are programmed, the *address generator* starts to load *SendDim* words from the memory, starting from address *SendAddr*, and propagates them to the NA of the processor that has *SendID* as network address. The *SendTag* is an identifying tag assigned to the specific message and is sent, together with the size and the destination address inside the packet header, to be decoded by the receiving network interface. The storing inside the *SendTag* register triggers the starting of the NoC transaction.

At the other end of the communication, the processor needs to execute a *receive()* to complete the transaction. The *receive()* microcode, as a first step, stores the primitive parameters inside three memory-mapped registers. Once such registers are programmed, the processor keeps polling the DMA handler, where a dedicated circuitry is in charge of comparing the incoming messages with the contents of the three registers. In case of matching, the message data is stored in memory, at the location identified by *ReceiveAddress*. It is useful to point out that, when the receive is not already executed at the time that the message reaches the destination NA, the address generator is capable of temporarily store the data in a reserved region of the memory used as buffer until it is needed.

3.3 Interrupt Generation Support

As a further extension, a tag decoder has been instantiated inside the Network Adapter. It is in charge of detecting a set of predetermined tag configurations, that are reserved for the purpose of remote interrupt generation. In case of matching, the tag decoder triggers an interrupt signal that is connected to the processor interrupt controller. This feature can be used to allow a processor in the system to generate an asynchronous event on another processor, such as, for example, the monitoring and control related signaling that is exploited in this paper. The number and the range of reserved tag configurations are configured at design time. By default, the tag is 16 bits wide, and 16 different configurations generate a different interrupt signal to the processor. This means that 16 compara-

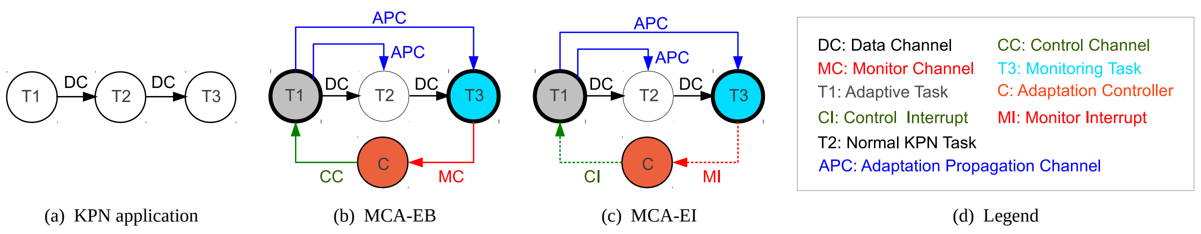


Figure 1: Self-adaptation approaches for KPN applications on NoC.

tors and 16 registers are instantiated inside the NA netlist after synthesis. If the interrupt signal is generated directly by the comparing logic, some spurious fluctuations can be generated at the receiving of the message, during the transient of the circuitry switching. Being the interrupt controller set to be sensitive on rising edge of the interrupt signal, in order to avoid such fluctuations to generate an interrupt in the processor, the interrupt signal is buffered in a register before being forwarded to the interrupt controller. This hardware overhead can be customized reducing the number of reserved tags according to the application features.

Besides implementing the monitoring and control interrupts that will be described more in detail in the following, the interrupt generation is used to implement a request based flow control between KPN tasks mapped on the NoC platform. Such flow control relies on two software FIFOs created for every communication channel in the task graph, more in detail respectively in the producer and consumer private memory. The producer keeps writing the produced data inside the software FIFO and blocks only when it is full. The consumer, at the other end of the channel, keeps reading input data from its software FIFO. When this FIFO is empty, a *request* interrupt message is sent from the consumer to the producer, triggering it to perform a *send* of all the data inside its software FIFO.

4 INTERRUPT-BASED SELF-ADAPTATION FOR KPN APPLICATIONS ON NoC

In this section, we present our new approach, *event-based MCA using inter-processor interrupts* (MCA-EI), for implementing self-adaptive KPN applications on NoC-based MPSoCs. Similar to MCA-EB, MCA-EI introduces an MCA feedback loop into the application pipeline. The monitor (equivalent to sensors) measures various parameters to check whether the application meets its goals. The controller takes decisions so as to steer the system towards the goal, whereas adapters (similar to actuators) are in charge of actually performing adaptations. Since processes are generally mapped to different resources on MPSoCs, it is quite possible that the parameter to be monitored is present on 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. Both MCA-EB and MCA-EI incorporate a generic fuzzy logic based adaptation controller and implement similar monitoring and adaptation techniques. Similarly both uses event-based control; which means the adaptation control is triggered upon the occurrence of specific events in the system. However they differ based on how the MCA mechanism interacts with the application. MCA-EB uses *blocking-channels* to this end, whereas MCA-EI is based on *inter-processor interrupts*.

Similar to MCA-EB scheme, MCA-EI also represents a self-adaptive application in terms of the following entities: monitoring tasks calling monitoring functions, adaptive tasks implementing adapter functions, adaptation controller(s) and adaptation propagation channels alongside the original task graph. A simple self-adaptive application pipeline built using MCA-EI framework is shown in Fig. 1(c). The design and usage of monitoring functions, adapter functions and fuzzy adaptation control algorithm are as detailed in (Derin et al., 2012). Hence only the main differences in comparison with the MCA-EB is presented here.

Fig. 2 depicts the pseudo-code representing a monitoring task in MCA-EI scheme. The modification done on a normal KPN task to convert it to a monitoring task (i.e.; by adding calls to monitoring functions) is colored in blue. In this example the task is equipped with throughput (in terms of bit-rate) monitoring capabilities. The difference to be noted compared to MCA-EB is that, instead of sending the monitored parameter value over blocking channel it is sent as an interrupting message to the NoC tile where the controller is run.

As compared to MCA-EB (where the controller is a separate task), the control algorithm is run as part of interrupt handler in MCA-EI. As shown in the pseudo-code of the controller given in Fig. 3,

```
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);
     adaptIntr.Value = calculateBitrate();
     adaptIntr.Type = MONITOR_INTR;
     sendInterrupt(BR_CTRL_TILE, adaptIntr);

  }
}
```

Figure 2: A monitoring task in MCA-EI scheme.

```
adaptIntrHandler(adaptIntr)
{
  switch(adaptIntr.Type) {

    case MONITOR_INTR:
      adaptIntr.Value = fuzzyCtrl(adaptIntr.Value);
      adaptIntr.Type = CONTROL_INTR
      sendInterrupt(ADAPT_TASK_TILE, adaptIntr);
    break;

    case CONTROL_INTR:
      cacheCtrlParam(adaptIntr.Value);
      CtrlIntrPending = TRUE;
    break;
  }
}
```

Figure 3: Adaptation interrupt handler in MCA-EI scheme.

the interrupt handler handles two kinds of interrupts; a) monitor interrupts (MI) - from monitoring task's tile to adaptation controller's tile, b) control interrupts (CI) - from adaptation controller's tile to adaptive task's tile. Upon receiving an MI, the interrupt handler runs the fuzzy control algorithm with the received monitored parameter value as the argument to generate the control command. Subsequently it interrupts the adaptive task's tile (using CI) to send the control command. On the other hand, CI is handled as follows; first the received control command will be cached so that it can be processed by the adaptive task later, second a flag is set to inform the adaptive task that a control interrupt had occurred.

An adaptive task in MCA-EI is shown in Fig. 4. It checks for any previous interrupts from the controller tile at a fixed location in the task body. In case of any previous interrupts the adaptive function will be invoked (with the cached value of control command) to perform the required actions. Further it also sends the modified values of the adapted parameter to other

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

     If(ctrlIntrPending) {
       getCachedCtrlCmd(&ctrlCmd);
       adaptParam(ctrlCmd);
       write(ADAPT_PROP_CH, newParam);
     }

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

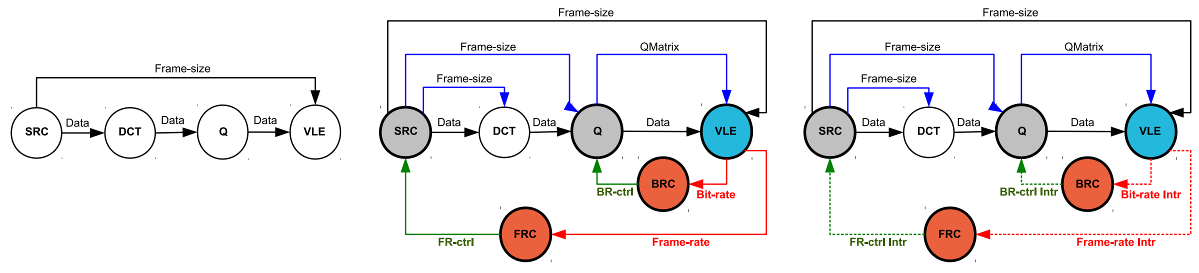
Figure 4: An adaptive task in MCA-EI scheme.

tasks (using APC) which need these updated parameters. In KPN model, processes can receive external input only via blocking FIFO channels. Checking the pending controller interrupt flag is a non-blocking operation, thus it does not affect the liveness of the application.

The functioning of MCA-EI can be summarized as follows. When the event which triggers the adaptation control is generated, the monitoring task performs monitoring and interrupts the adaptation-controller's tile. The controller tile receives this interrupt and runs the control algorithm as part of the interrupt handler. Subsequently, the controller interrupts the tile of the adaptive task to send the control command. The interrupt handler of the adaptive task tile caches the interrupts received, to be processed by the adaptive task later. When the adaptive task reaches the predefined point of execution, it checks for any cached interrupts and performs the required adaptations if necessary.

The MCA-EI approach increases the application throughput by using an interrupt mechanism instead of blocking FIFO channels in the MCA feedback loop. MCA-EI scheme has the advantage of not stalling the application pipeline since the adaptive tasks never wait for blocking control commands from the controller. This helps the system to attain higher throughput as compared to the MCA-EB scheme.

In our work, the correctness of the reconfiguration relies on the KPN processes reaching an execution point at which their state does not relate to any computation done using the parameter to be updated. An adaptation command via a control channel or an adaptation propagation channel can only be served at such points. For a general KPN application, the identification of the adaptation execution points and adaptation propagation channels is to be done manually by the application programmer. It should be done carefully to guarantee a correct parameter reconfiguration.



(a) MJPEG Encoder Pipeline (b) MCA-EB (c) MCA-EI

Figure 5: Self-adaptive MJPEG encoder using different approaches.

Otherwise, the functional correctness may be compromised and also deadlocks may be introduced. In the specific case of polyhedral process networks (PPN) (Verdoolaege, 2010), the state of processes reduce to the iterator values at the top of the for-loops of each process. Therefore such locations, as shown in Fig. 4, are ideal candidates for adapting PPN processes.

5 CASE STUDY: MOTION JPEG (MJPEG)

This section presents MJPEG (Lieverse et al., 2001), as a case study to demonstrate our technique. A typical MJPEG encoder pipeline is shown in Fig. 5(a), where all the components are modeled as KPN tasks. The *SRC* task captures the input video frame-by-frame and feeds it to the *DCT* task one block (8×8 pixels) at a time, to perform the discrete cosine transform. The *Q* task quantizes each video block using an 8×8 QMatrix, whereas VLE task does entropy coding on the video blocks before generating the final MJPEG stream by inserting headers/markers to indicate the start/end of each frame.

5.1 Self-adaptive MJPEG

We implemented two self-adaptive MJPEG encoders on our 2×2 NoC-based FPGA platform using the MCA-EB and MCA-EI approaches respectively as shown in Fig. 5. Our implementations support autonomous control of bit-rate (BR) and frame-rate (FR) at run-time. 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. The implementation details of the sliding-window based monitor, fuzzy-logic controller and the bit-rate/frame-rate adapters are provided in (Derin et al., 2012).

Table 1: Comparison of steady-state overheads.

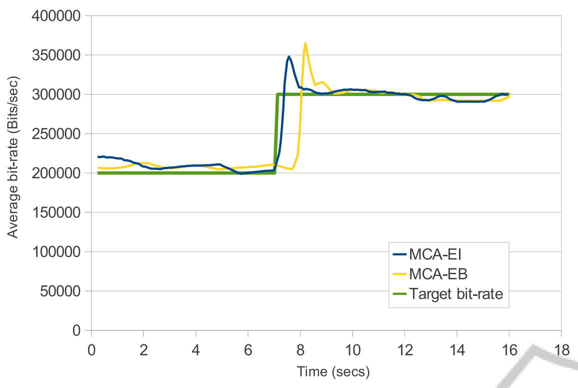
	FR-Overhead (%)	BR-Overhead (%)
MCA-EI	0.385	0.386
MCA-EB	6.663	6.667

6 RESULTS

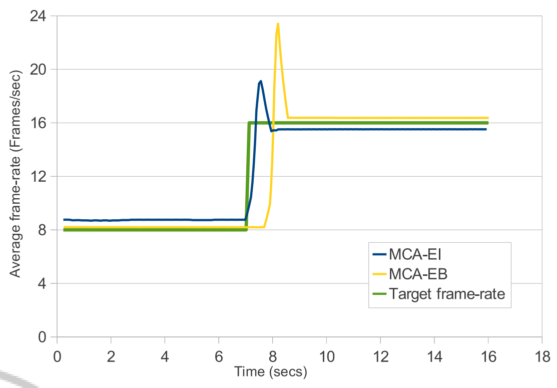
In this section, the results from the adaptive MJPEG case study using a 128×128 test video are presented. The MCA-EI approach is compared against MCA-EB in terms of the adaptation overhead and the control quality. Adaptation overhead is measured as the reduction in frame-rate and bit-rate, whereas control quality is quantified using rise-time/fall-time (RT/FT) and mean-absolute-error (MAR). 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 time taken for the parameter to reach within a tolerance band ($\pm5\%$) about its target value is the rise/fall time. The absolute error value for the parameter is calculated for all measurements starting from where it reached the tolerance band till the last frame. The mean of these absolute error values gives the MAR.

6.1 Adaptation Overhead

To measure the overhead due to the introduction of the MCA 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 values of FR and BR without the framework. The experiment is repeated after introducing the MCA loop and the additional channels to obtain the reduced values. In this case both BR and FR control is turned off inside the controller, since only the overhead due to the framework needs to be measured. Table 1 com-



(a) Bit-rate adaptation



(b) Frame-rate adaptation

Figure 6: Results for initial BR = 200000 bps, initial FR = 8 fps and final BR = 300000 bps, final FR = 16 fps.

pares the steady-state overhead (in terms of bit-rate and frame-rate reduction) for MCA-EI and MCA-EB schemes. It can be seen that the overhead in case of the MCA-EB is much more than MCA-EI because the application pipeline is stalled at the end of every frame. Firstly, the tasks from adaptive tasks till the monitoring task are stalled consecutively until the final frame block is processed by the monitoring task. Only then, the monitoring can be performed and the controllers can provide the control command required to unblock the adaptive tasks. In case of MCA-EI the pipeline is never stalled, yielding higher throughput. Also in MCA-EI, the adapter tiles will not be interrupted by the controller during steady-state; whereas in MCA-EB, the adaptive tasks have to wait for the control command for every frame even in the steady-state. The throughput reduction in the MCA-EI scheme (about 0.4%) is due to the execution of the controller inside the interrupt service routine (in our experiments, the controller is mapped on the processor with the heaviest workload).

6.2 Control Quality

Fig. 6 shows how the framework adapts (using MCA-EB and MCA-EI) when the encoder is run with initial BR = 200000 bits/s (bps), initial FR = 8 frames/s (fps), final BR = 300000 bps and final FR = 16 fps. Here the goals are changed from initial to final at the 60^{th} frame and the framework is configured such that the control algorithm is run for every third frame.

Table. 2 compares the two schemes in terms of rise-times (RT) and mean-absolute-errors (MAR) for this experiment. It can be seen that the rise-times for the MCA-EB scheme is considerably higher than MCA-EI due to the stalling of the pipeline at the end of each frame. However, the MAR is within tolerable limits (±5% of final values) for both.

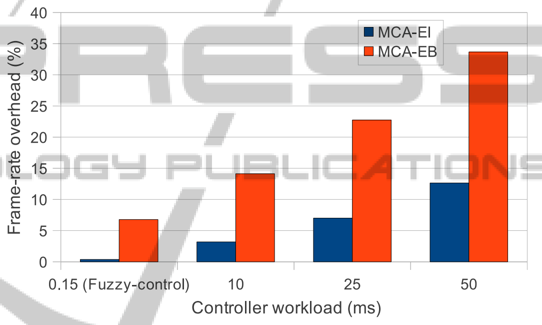


Figure 7: Effect of controller workload on adaptation overhead.

6.3 Adaptation Overhead vs. Controller Workload

Workload of the controller - represented as the calculation time of the control decision, is simulated by introducing delay in the controller code. To obtain the steady-state overhead, the MJPEG encoder is run with the MCA feedback loop in place but the bit-rate and frame-rate control being turned off. Fig. 7 shows the variation in the steady-state overhead (as percentage frame-rate reduction) with respect to the controller workload for the MCA-EB and MCA-EI schemes. As obvious, the overhead increases with increasing controller workload. But this increase is far less in interrupt-based scheme as compared to the blocking-channel case. In case of the MCA-EB scheme the entire encoder pipeline is stalled while the control algorithm is run, irrespective of on which tile the controller is located. The throughput reduction in the MCA-EI scheme is due to the controller stealing several cycles from the application task that runs on its tile. These experiments are carried out with the controller running on the processor with the heaviest workload (i.e., the tile of DCT) and shows that the in-

Table 2: Comparison of control-quality.

	RT-BR (secs)	RT-FR (secs)	MAR-BR (bits)	MAR-FR (frames)
MCA-EI	0.364	0.364	6551 (2.2%)	0.57 (3.6%)
MCA-EB	0.579	0.579	7958 (2.7%)	0.63 (4.0%)

terrupt based approach is much superior for complex controllers that consume more time. Furthermore, the overhead can be almost eliminated in the MCA-EI scheme by running the controller on a tile where no application task is present. However, such an improvement is not possible for the MCA-EB scheme.

7 CONCLUSIONS

In this paper, we presented the MCA-EI approach aimed towards developing low-overhead self-adaptive KPN applications on NoC-based MPSoCs. Compared to the MCA-EB scheme (Derin et al., 2012), it makes use of *inter-processor interrupts* to increase the application throughput. Results from the MJPEG case study show that the MCA-EI scheme outperforms MCA-EB in terms of overhead (about 6.25% reduction) while offering similar or better quality of control. The sensitivity of adaptation overhead to controller workload is also much less in case of MCA-EI. However, MCA-EI requires platform support to send data to remote tiles using interrupting messages over the NoC. This support is implemented by extending the network interface with a tag decoder.

ACKNOWLEDGEMENTS

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

Bertozzi, D. and Benini, L. (2004). Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *Circuits and Systems Magazine, IEEE*, 4(2):18 – 31.

Busseuil, R., Barthe, L., Almeida, G. M., Ost, L., Bruguier, F., Sassatelli, G., Benoit, P., Robert, M., and Torres, L. (2011). Open-Scale: A scalable, open-source NOC-based MPSoC for design space exploration. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig'11)*, pages 357–362, Los Alamitos, CA, USA.

Derin, O., Ferrante, A., and Taddeo, A. V. (2009). Coordinated management of hardware and software self-adaptivity. *J. Syst. Archit.*, 55(3):170–179.

Derin, O., Ramankutty, P. K., Meloni, P., and Cannella, E. (2012). Towards self-adaptive KPN applications on NoC-based MPSoCs. *Advances in Software Engineering*, 2012(Article ID 172674):13 pages.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Information Processing '74: Proceedings of the IFIP Congress*. North-Holland.

Lieverse, P., Stefanov, T., van der Wolf, P., and Deprettere, E. (2001). System level design with SPADE: an M-JPEG case study. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'01)*.

Mattson, T. G., Riepen, M., Lehnig, T., Brett, P., Haas, W., Kennedy, P., Howard, J., Vangal, S., Borkar, N., Ruhl, G., and Dighe, S. (2010). The 48-core SCC processor: the programmer's view. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10.

Meloni, P., Tuveri, G., , Raffo, L., Cannella, E., Stefanov, T., Derin, O., Fiorin, L., and Sami, M. (2012). System adaptivity and fault-tolerance in NoC-based MPSoCs: the MADNESS Project approach. In *Proceedings of 15th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD'12)*.

Micheli, G. D. and Benini, L. (2006). *Networks on Chips: Technology and Tools*. Morgan Kaufmann, San Fransisco, 1st edition.

Nollet, V., Verkest, D., and Corporaal, H. (2010). A safari through the mpsoc run-time management jungle. *J. Signal Process. Syst.*, 60(2):251–268.

Sandee, J. (2006). *Event-driven control in theory and practice - trade-offs in software and control performance*. PhD thesis, Eindhoven University of Technology.

Verdoolaege, S. (2010). Polyhedral process networks. *Handbook of Signal Processing Systems*, pages 931–965.

Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., Brown III, J. F., and Agarwal, A. (2007). On-chip interconnection architecture of the Tile processor. *IEEE Micro*, 27(5):15–31.

Zhai, J., Nikolov, H., and Stefanov, T. (2011). Modeling adaptive streaming applications with parameterized polyhedral process networks. In *Proceedings of the 48th Design Automation Conference*, pages 116–121. ACM.