A Self-adaptable Distributed DFS Scheme for NoC-based MPSoCs

Thiago da Rosa, Guilherme Guindani, Douglas Cardoso, Ney Calazans, Fernando G. Moraes
PUCRS – FACIN – Av. Ipiranga 6681 – Porto Alegre – 90619-900 – Brazil
{thiago.raupp, douglas.cardoso}@acad.pucrs.br, {guilherme.guindani, ney.calazans, fernando.moraes}@pucrs.br

ABSTRACT

As processor count in MPSoCs increases, the use of NoCs becomes relevant, if not mandatory. However, power and energy restrictions, especially in mobile applications, may render the design of NoC-based MPSoCs over-constrained. The use of traditional dynamic voltage and frequency scaling (DVFS) techniques proved useful in several scenarios to save energy/power, but it presents scaling problems and slow response times. This work proposes a self-adaptable distributed dynamic frequency scaling (DFS) scheme for NoC-based MPSoCs. It takes into account the communication load and the utilization level of each processor to dynamically change its operating frequency. Frequency change decisions and clock generation are executed locally to each processor. Clock generation is simple, based on clock gating of a single global clock. The overhead of the scheme is minimum, the range of generated clocks is wide, and the response time is negligible. Experimental results with synthetic applications shows that the proposed scheme has an average execution time overhead below 7%, and may lead to considerable power and energy savings, since it allows an average reduction of 27% on the total number of executed instructions. Evaluating the proposed method with a real application, the execution time overhead reached 13%, while the total number of executed instructions was reduced by 64%.

Categories and Subject Descriptors

B.7.1 **[Integrated Circuits]**: Types and Design Styles – *advanced technologies, VLSI*; C.1.2 **[Processor Architectures]:** Multiple Data Stream Architectures (Multiprocessors) – *Interconnection architectures*.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

DFS, MPSoC, NoC, power management technique.

1. INTRODUCTION

NoC-based MPSoCs provide massive computing power on a single chip. Such devices can support the convergence of several appliances (e.g. HDTV, multiple wireless communication standards, media players, gaming, etc.). However, performance and energy consumption are conflicting targets, and designers have to select the best trade-off between performance and energy for the set of applications the system will execute.

Energy consumption in CMOS circuits can be reduced by controlling two main variables: the supplied voltage and the operating frequency. The later has a linear impact on energy consumption, and a given frequency can only be supported with some minimum voltage supply. Voltage has a quadratic impact on energy consumption. Accordingly, this is the most used factor to reduce energy consumption. Controlling these two variables at runtime is the basis of Dynamic Voltage and Frequency Scaling (DVFS) techniques. These techniques can be controlled by hardware components alone, by software algorithms, or by some hybrid hardware-software configuration. On NoC-based MPSoCs, the DVFS infrastructure can be introduced into the processing element (PE), on the NoC router or in both.

This paper proposes and evaluates a new technique for Dynamic Frequency Scaling (DFS) with fixed system voltage. The main goal of this technique is to enable fast frequency switching according to each processor's workload and communication load. The proposed DFS scheme is evaluated in a synthesizable NoC-based MPSoC.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the clock generation module. Section 4 describes the target MPSoC architecture. The proposed DFS controller and the required modifications to enable the DFS scheme are presented in Section 5. Section 6 presents the experimental setup and results. Finally, conclusions and future works are drawn in Section 7.

2. RELATED WORK

This section review, in Table 1, the related work in DFVS applied to processors and MPSoCs, comparing it to the proposed technique. The target architecture of works [1] and [2] is a single CPU, using a parameter monitored in hardware to take the DVFS decisions. The first uses the supply current driven by the CPU to adjust the frequency/voltage pair of the chip. The second one uses the CPU temperature as monitoring parameter, with the DVFS algorithm implemented in the Linux kernel. In [3] the architecture is also a single CPU, but the monitoring parameter is the application statistics, which are collected by a performance monitoring unit and temperature sensors at runtime. Works [4], [5], [6], and [7] target more than one CPU using a bus-based interconnection infrastructure. Alimonda et. al. in [4] use the load of communication queues between two processors to control the DVFS scheme. In [6] and [7] the parameter which controls the DVFS scheme comes from the tasks slacks and applications profile, respectively.

Table 1 - DVFS state-of-the art comparison.

Author	Architecture	Monitoring Parameter	Implementation
Pourshaghaghi [1] 2009	Single CPU	CPU Supply Current	Fuzzy Logic Controller in Hardware
Shu [2] 2010	Single CPU	CPU Temperature	Temperature Sensors and Software Algorithm
Salehi [3] 2010	Single CPU	Application History	Software Tracking Application Workload
Alimonda [4] [5] 2006/2009	Bus-Based MPSoC	Queues Load	Central Controller Hardware
Liu [6] 2009	2 CPUs, Bus-based Interconnect	Tasks Slacks	Task Graph Unrolling Software
Kong [7] 2008	Bus-Based MPSoC	Application Profile	Software computes Suitable DVFS Level and Informs Controller Hardware
Chabloz [8] 2010	Synchronous Islands	Tasks Deadlines	GRLS scheme, Local clock generation
Yin [9] 2009	NoC	Queues Load	Voltage Selection via Transistors
Herbert [10] 2009	NoC-Based MPSoC	Process Variation	Off-line Calibration (Design Variability), Algorithms in Software
Puschini [11] [12] 2008/2009	NoC-Based MPSoC	Temperature and Task Synchronization/Latency	Parameter Modeling, Game Theory Algorithm
Goossens [13] 2010	NoC-Based MPSoC	Tasks Slacks	Voltage and Frequency Scaling Hardware, Software to adjust the Controller
Beigné [14] 2008	NoC-Based MPSoC	-	Application chooses the IP Voltage Level
Proposed Work	NoC-Based MPSoC	Communication and CPU Load	Local Clock Generation, Controller sets Correct Frequency Level. Software updates Controller with Current CPU state.

In [8] the Authors use synchronous islands, adopting an approach named GRLS (Globally Ratiochronous-Local Synchronous). The parameter to control the DVFS is the tasks' deadlines.

A DVFS controlling scheme is proposed for NoCs in [9]. Similarly to [4] and [5], the monitoring parameter is the communication load. However, in [9], the communication queues are placed between each 2 neighbor routers and the voltage scaling is done via power supply networks. Works [10], [11], [12], [13] and [14] target NoC-based MPSoCs architectures. In [10], the process variation is the parameter used to control DVFS decisions. In [11] the monitoring parameters are temperature and task synchronization, while in [12] the parameter monitored is the communication latency. In both works the monitoring parameters were modeled, and used to build an objective function, used by a game theory algorithm to define the voltage and frequency levels. In [13] the Authors use a metric to compute the suitable DVFS level according to the application profile. Authors present DVFS architecture for IPs integration within a GALS NoC in [14], where the user is able to program the desirable voltage level of the IP in the application.

As shown, DVFS schemes may use hardware or software controlling parameters. Hardware parameters for controlling DVFS include temperature, process variation, current and load in communication buffers. Software parameters include application profile and scheduling tasks. In terms of implementation, most proposals employ software algorithms, releasing to the hardware the monitoring process (when a hardware parameter is monitored).

The present works proposes a DFS scheme, with fixed system voltage, through hardware and software mechanisms. The hardware mechanism obtains data from the Network Interface (NI) and from the processor to parameterize the clock generation module, setting the correct frequency for the NoC and PEs. Software mechanisms are responsible for monitoring a set of parameters, making them available to the hardware mechanism.

3. CLOCK GENERATION

The clock generation module uses as input a reference clock, which consists in the highest frequency usable in the system as a local clock. The principle of the clock generation process is to achieve clock division by simply omitting selected cycles of the reference clock, as Figure 1 illustrates: initially, inputs *num* and *den* are natural numbers 2 and 5, respectively. This corresponds to set the frequency of the clock generator to two-fifths (40%) of the reference clock. In other words, for each *den* reference clock cycles, *num* cycles are propagated to the output clock. According to the DFS literature review, this is the first work to apply this method to generate new frequencies in NoC-based MPSoCs.

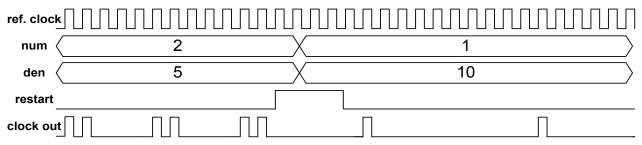


Figure 1 - Example of the proposed clock generation process.

Any frequency obtained by changing the *num* and *den* values can be generated with the obvious exceptions (*den*=0 is not an acceptable value, *num*=0 corresponds to a clock gating action and the constraint *num* ≤ *den* must be respected). Before changing the *num* and *den* values, the *restart* signal must be asserted to momentarily stop the output clock and reinitialize internal registers. After releasing *restart*, the new frequency, defined by the modified values of *num* and *den* appears at the output.

This clock generation scheme translates into a simple logic that can multiply the reference clock frequency by selected values distributed evenly inside the closed interval [0,1]. The amount of distinct values is dictated by the range of values assignable to *num* and *den*, which amounts to define the size of internal registers used to store these inside the clock generator.

A system using only this clock generation scheme is clearly classifiable as *ratiochronous* [8], since any relation between two frequencies in the system is a rational number. In the scheme, if all clock edges of all clocks can be kept always in phase with the corresponding edge of the reference clock, the system can simply dispense the use of synchronizers. However, due to the use of one clock generation module to each processor, keeping all clock signals in phase may have a strong impact on clock distribution control, and is ignored here to keep the clock generators simple.

In this way, synchronizers must be employed later to guarantee reliable communication between two modules controlled by distinct clock generation modules.

Figure 2 presents the block diagram of the clock generation module. The module has two outputs: (*i*) *clock_plasma*, the divided clock, used by PEs; (*ii*) *clock_router*, the half frequency with a duty cycle of 50%, used by the NoC. The inputs *num* and *den* are used to configure the clock generator. As the example discussed, *num* represents the numerator of the fraction that multiplies the reference clock, while *den* is the denominator of the fraction.

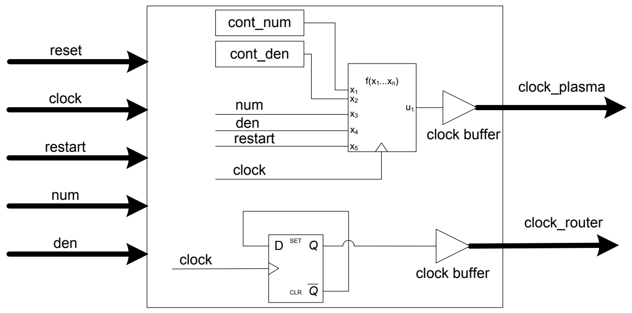


Figure 2 - Clock generation module.

The main advantages of this clock generation module are the low area overhead and a large set of generated frequencies. For example, for *num* and *den* being 4-bit values the module takes only 12 flip-flops, 31 LUTs and two clock buffers (BUFG) of a Virtex-5 Xilinx FPGA. In this same example, 120 different fractions can be formed. Although several of these correspond to a same frequency (e.g. 1/1, 2/2 etc) still a large number of distinct frequencies can be produced with small resolution for *num* and *den*. In addition, the clock output is always stable, contrary to what happens in standard DFS methods, where the time required to stabilize a new frequency can be significant. In [15] it is also presented a controller that can provide fast frequency switching. However, the Authors use two extra PLLs in the proposed scheme, which induces large area and power overhead. The proposed module is also glitch free by construction. Such features make the use of the proposed clock generator module appropriate for distributed DFS in MPSoCs, where each PE may have its own frequency according to its load.

Yet, the most recent technologies are restricting the supply voltage scaling margins, which is the key component behind power savings through DVFS [16]. Therefore, designing the system to work at a fixed supply voltage, coupled to the DFS method herein proposed, is an option to efficiently manage the energy consumption in nanoscale technologies.

4. SYSTEM ARCHITECTURE

The reference MPSoC [17] is a homogeneous multiprocessing NoC-Based MPSoC. Figure 3 shows an instance of this MPSoC. The 2-D mesh NoC used in the reference MPSoC has the following features: wormhole packet switching, flit width equal to 16 bits, XY routing algorithm, round-robin arbitration, input buffers with 8-flits depth. Each PE includes the following modules: (*i*) a 32-bit Plasma processor (a MIPS-like architecture); (*ii*) a local memory (RAM); (*iii*) a DMA module, responsible for transferring the task object code to the memory and messages to/from the NoC to the local memory; (*iv*) a network interface (NI). Two types of PEs are used: slave and master. Slave-PEs are responsible for executing application tasks, while the Master-PE is responsible for managing task mapping and system debug. The task repository is an external memory, responsible to store all object codes of applications that will eventually be executed.

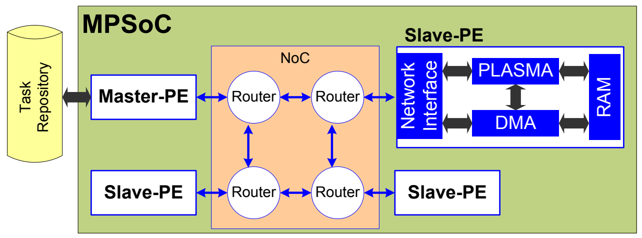


Figure 3 - Block diagram of the MPSoC architecture.

Each slave processor runs a multitask *microkernel* that enables the communication between tasks through *send* and *receive* primitives, respectively called *WritePipe()* and *ReadPipe()*. Each *microkernel* contains a vector, named *pipe*, which contains messages to be exchanged between tasks. When a given task executes a *WritePipe()*, the message is stored in the processor communication *pipe*, and computation continues. On the other side, when a given task executes a *ReadPipe()*, a system function is executed. If the target task is located in the same processor, the task executes a read in the communication *pipe*. If the task is located in another processor, the *microkernel* sends a request message through the NoC and the task enters in wait state. When the message arrives from the network, the *microkernel* stops the executing task and reschedules the waiting task. Thus, the communication scheme employs non-blocking writes and blocking reads.

The *microkernel* was modified in the present work to monitor the CPU utilization and communication *pipe* occupancy, storing them in new memory-mapped registers. By monitoring the *microkernel* scheduler it is possible to evaluate the CPU utilization and by monitoring the communication *pipe* occupation it is possible to evaluate the communication load.

5. THE DFS CONTROLLER

The DFS controller computes the communication load and CPU utilization level according to values provided by the *microkernel*. Such values are used by the controller to define the PE frequency. The controller always operates at the *reference frequency* (the highest frequency in the system, used as input to the clock generation module). As shown in Figure 4, the slave-PE feeds the DFS controller with values stored in memory-mapped registers:

- *pipe_ocup* and *req_msg*: related to the communication load, correspond to the number of messages stored in the communication *pipe* (an integer value) and if there is request for a message not yet produced by the processor (a Boolean value).
- *not_scheduled* (Boolean values): when true, only the *microkernel* is running, meaning that no task is being executed; when false, at least one task is being executed. The DFS controller may define the CPU utilization counting how many clock cycles this signal is asserted, in a sampling period.

Due to the delay induced by the clock generation circuitry, the clock phase at the outputs of the DFS controller is not the same of the reference frequency. Therefore, synchronizers [18] are added to capture the control signals generated by the Slave-PE module (module *synchronizers* in Figure 4).

To cope with different clock phases and frequencies, the original router-PE interface was also modified, adopting the GALS paradigm. This is achieved by adapting the existing buffers in the NoC and network interface to work as bisynchronous FIFOs [19] (a minimum area overhead is introduced, corresponding to the FIFO control signals).

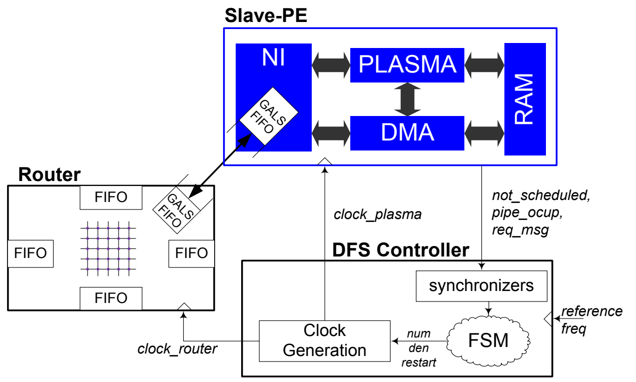


Figure 4 - Router-PE GALS interface and the DFS controller responsible for generating the router and PE frequencies.

The FSM represented in Figure 4 corresponds to the behavior detailed in Table 2. The controller uses the clock generation module, detailed in Section 0, to provide the two output clocks. The role of the FSM is to choose the correct PE frequency, by evaluating the following parameters:

- *Pending message requests from other tasks*. This situation takes place when the processor is not producing data to the consumer task (*req_msg* = 0).
- Occupancy of the pipe. If the communication pipe has a high occupancy, the processor is producing messages at a higher rate than the consumer tasks can consume, while the inverse scenario means a lack of produced messages. Upper and lower parameterizable thresholds define the high and low occupancy states, respectively. Occupancy between these values defines an operational state.
- *CPU utilization*. When the utilization is low the CPU is not executing any task or tasks are blocked, e.g., waiting message(s) from other tasks. When the utilization is high, tasks are using the processor at the maximum rate. Two parameterizable thresholds define high, low and operational CPU utilization states.

Table 2 - DFS Controller behavior (↓/↑ mean decrease/ increase one frequency step, ↑↑ means increase two frequency steps, = means keep frequency unchanged and - denotes don't care conditions).

Action in frequency	*Pending Message*	*Current Pipe Occupancy*	*Previous Pipe Occupancy*	*CPU Utilization*
1 - ↓	0	high	-	-
2 - ↓	0	operational	low	-
3 - ↓	0	low	-	low
4 - =	0	operational	operational	-
5 - =	0	low	-	operational
6 - =	1	-	-	low
7 - ↑↑	1	-	-	operat./high
8 - ↑	0	low	-	high
9 - ↑	0	operational	high	-

Frequency decreases in three situations: (*i*) the communication pipe is almost full (action 1 of the Table 2); (*ii*) the communication pipe occupation is increasing, i.e. in the previous evaluation its state was low and the present state is operational (action 2); (*iii*) the communication pipe occupation is almost empty and the CPU usage is low, meaning that even at a lower frequency the data in the communication pipe is being consumed (action 3).

Frequency increases in three situations: (*i*) existence of pending messages with operational or high CPU utilization (action 7) – the clock generator increases in two steps the frequency; (*ii*) the communication pipe is almost empty and the CPU has high utilization (action 8); (*iii*) the communication pipe occupation is dropping, meaning soon the processor can present lack of messages, i.e. in the previous evaluation its state was high and the present state it is operational (action 9).

When a given PE receives a message request, and it has data to transmit, this PE goes to the reference frequency during the message transmission. This action avoids stalling consumer PEs operating at higher frequencies than the producer PEs.

The period between consecutive evaluations is also parameterizable. In this work, the evaluation period corresponds to four time slices. When an evaluation is triggered, the controller stores the values generated by the *microkernel*, computing the current communication pipe occupation and CPU utilization.

6. EXPERIMENTAL RESULTS

This section employs an instance of the reference MPSoC with 6 processors (1 Master-PE and 5 Slave-PEs) and a 3x2 NoC to demonstrate the characteristics and advantages of the proposed DFS scheme. NoC and Plasma peripherals (NI and DMA) are described in VHDL, while an ISS model is used to describe Plasma CPU and RAM. The simulations were performed in ModelSim. Three applications, written in the C language, were used to evaluate the proposed method:

a. *Pipeline* – Data-flow application with 3 tasks: *producer*, *worker*, and *consumer*.
b. *Communication* – Application modeled with four tasks, with a communication graph that has two tasks working in parallel. Two initial tasks run in parallel, as producers, providing data for the worker task, which sends data to the consumer task.
c. *Partial MPEG filter* – Real application used to evaluate the performance of the proposed DFS controller. The partial MPEG filter is composed by five tasks, modeled as a pipeline.

Applications (a) and (b) are synthetic, with execution time emulated by a loop, with 100 messages being sent from the producer task(s) to the other(s) task(s). The DFS controller was parameterized to generate 9 different frequencies: 5, 10, 25, 40, 50, 60, 75, 90 and 100% of the reference frequency. In the graphs presented in this Section, these frequencies are plotted in the y-axis, with values ranging from 0 to 8.

6.1 Synthetic applications

Table 3 details the three simulated test cases for the Pipeline application, and results are presented in Figure 5. The number of executed instructions is reduced in average 32%, and the execution time overhead ranges from 1.8 to 3.2%.

Obviously, the number of instructions to execute the applications does not change when reducing the frequency. This reduction is obtained from reducing the instructions executed by the microkernel, e.g, execution of the scheduler when there is no task being executed.

Table 3 - Pipeline Evaluation Scenarios.

Test case	*Data rate generation*			*Number of executed machine instructions (in thousands)*	
	Producer	*Worker*	*Consumer*	***Without** DFS*	***With** DFS*
1	+	++	++++	24,482	16,049
2	++++	++	+	19,965	14,565
3	+++	+	++++	23,227	15,022

Table 4 - Communication Evaluation Scenarios.

Test case	*Data rate generation*			*Number of executed machine instructions (in thousands)*	
	Producer	*Worker*	*Consumer*	***Without** DFS*	***With** DFS*
1	++	+	++++	14,515	10,960
2	++	++++	+	23,011	13,933
3	+	++++	++	16,326	13,916

In test case 1 the slowest task (in this work, the term slowest or faster refers in fact to the data generation or consumption rate, not to the task execution time) is the producer (continuous line in Figure 5). As expected, the slowest task goes to the reference frequency, and the fastest task (consumer - dashed line) goes to a frequency proportional to the data generation rate (¼ of the reference frequency). The worker task frequency stays between the other two frequencies, varying two frequency steps when it tries to read messages from the producer task, and the message is not yet available.

In test case 2 the *consumer* is the slowest task. In this test case, the consumer task reacts more slowly than test case 1, taking more time to achieve the reference frequency. The consequence is that the *producer* and the *worker* fill the respective communication buffers, reducing their operating frequencies. Once the consumer starts to consume data at the reference frequency, the system stabilizes (between 50 to 80 ms after start).

In test case 3, the *worker* is the slowest task. The *worker* task quickly reaches the reference frequency, due to the pending requests coming from the *consumer* task. The *producer* task keeps its initial frequency, since the consumption rate of the worker task maintains the communication buffer in the operational state. The *consumer* frequency is decreased due to the *worker* data generation rate.

Table 4 details the three simulated test cases for the Communication application, and results are presented in Figure 6. The number of executed instruction is reduced in average 26%, and the execution time overhead ranges from 2.8 to 6.9%.

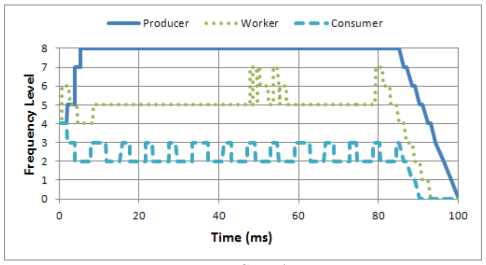
In test case 1 the *worker* task, which receives data from two *producer* tasks, reaches the reference frequency, since it is the slowest task. Note that the relationship between the *worker* and *consumer* frequency is around two (*worker* frequency level equal to 8 and *consumer* frequency level equal to 5), even if the generation rate between them is four (Table 4 - test case 1). The reason is that the *worker* receives data from 2 *producers,* transmitting these data to the consumer.

In test case 2 the *consumer* is the slowest task, going to the reference frequency. Also, both *producers* had their frequencies increased to the reference level, due to pending messages requested by the *worker* (the fastest task in this case). However, as the *consumer* consumes data too slowly, the other three tasks reach the minimal frequency due to the high communication pipe occupancy (10 - 20 ms). The system achieves a steady state between 60 to 80 ms.

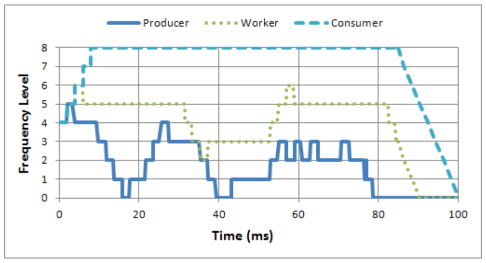
The third test case quickly stabilizes, with the *producer* and *consumer* working at the reference frequency, and the *worker* operating at ¼ of the reference frequency. The data generation rate relation between the three tasks explains this behavior.

6.2 MPEG

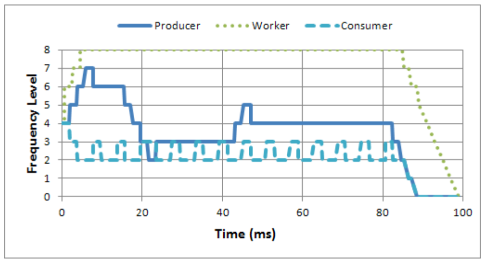
The result for the partial MPEG decoder is shown in Figure 7. In this application *iVLC* is a CPU-intensive task. Tasks *iQuant* and *IDCT* are simpler than *iVLC*. Tasks *Start* and *Print* are used to initialize the system and to print the results, respectively. In this test case, 200 frames were transmitted. The graphic in Figure 7 shows that only the task executing a high amount of computation had its frequency increased to the reference frequency, while *Print* and *Start* tasks had their frequency decreased to the lowest frequency level. The execution time overhead, compared to the



Test Case 1

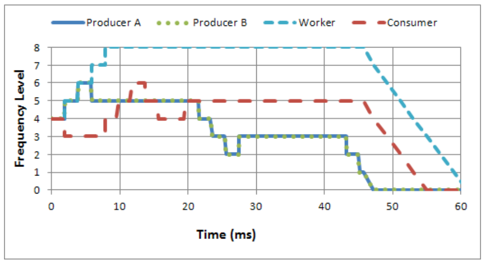


Test Case 2

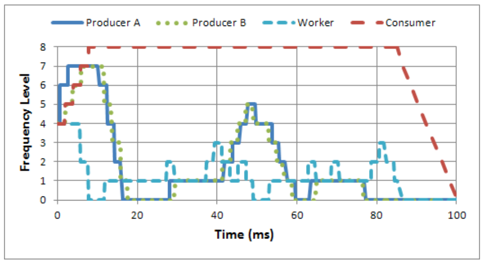


Test Case 3

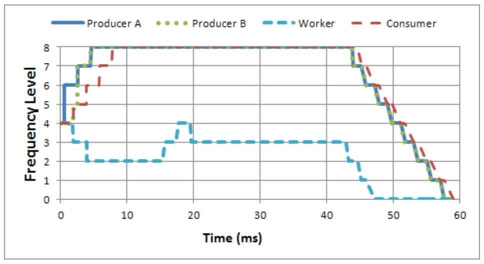
Figure 5 - Pipeline application. (1) Consumer as fastest task and producer as slowest. (2) Consumer as slowest task and producer as fastest. (3) Worker as slowest task and consumer as fastest.



Test Case 1



Test Case 2



Test Case 3

Figure 6 - Communication. (a) Worker as slowest task and consumer as fastest task. (b) Worker as fastest task and consumer as slowest task. (c) Producers as slowest tasks and worker as fastest task.

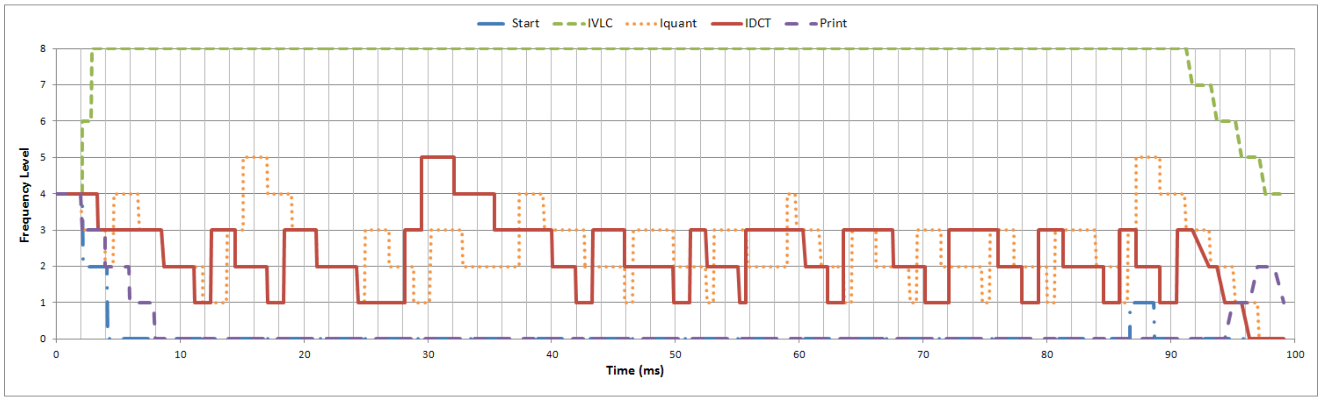


Figure 7 - Partial MPEG filter execution for 200 frames.

execution with the whole system at reference frequency was 13%. The number of executed instructions is reduced in 64%.

When the whole system executes with no DFS scheme, the six processors and the NoC operate at the reference frequency. On the other hand, using the proposed DFS scheme, only one processor operates at the reference frequency, while three other processors and the NoC operate, in average, at half of the reference frequency (including the Master-PE) and two processors operate at the lowest frequency level.

7. CONCLUSION

This work proposes a new DFS scheme and evaluates it in a real MPSoC platform. The frequency scaling scheme is based in the communication load and CPU utilization of each MPSoC PE. A clock generation module was designed to enable frequency changing. This module presents a low-area overhead and requires no stabilization time at each frequency switching.

Results show that the DFS scheme adjusts the processor frequency according to the load injected into the network. As shown in the MPEG benchmark, the CPU-intensive task has its frequency increased to generate more data to the other tasks. The limiting factor is the reference frequency. Once the tasks with lower injection rate reach the reference frequency, the system stabilizes, reducing the frequency of other tasks. Also, processors with no scheduled tasks have their frequency reduced. The proposed DFS method has a small impact in the total execution time. Therefore, an important energy reduction is expected, since few processors of the MPSoC operate at the reference frequency, drastically reducing the number of executed instructions.

Future works include: (*i*) enhancements in the DFS controller to dynamically adjust the evaluation time; (*iii*) evaluate the method when more than one task is executed in the same processor; (*iii*) evaluate the energy reduction obtained applying the method; (*iv*) apply the same method to the NoC.

8. ACKNOWLEDGMENTS

The Authors acknowledge the support of CNPq and FAPERGS, projects 301599/2009-2 and 10/0814-9, respectively.

9. REFERENCES

[1] Pourshaghaghi, H.R.; de Gyvez, J.P. "Dynamic voltage scaling based on supply current tracking using fuzzy Logic controller". In: ICECS, pp.779-782, 2009.

[2] Shu, L.; Li, X. "Temperature-aware energy minimization technique through dynamic voltage frequency scaling for embedded systems". In: ICETC, pp. 515-519, 2010.

[3] Salehi, M. E.; Samadi, M.; Najibi, M.; Afzali-Kusha, A.; Pedram, M.; Fakhraie, S. M. "Dynamic Voltage and Frequency Scheduling for Embedded Processors Considering Power/Performance Tradeoffs." IEEE Transactions on VLSI Systems, *in press*, 2010.

[4] Alimonda, A.; Carta, S.; Acquaviva, A.; Pisano, A. "Non-Linear Feedback Control for Energy Efficient On-Chip Streaming Computation". In: IES, pp.1-8, 2006.

[5] Alimonda, A.; Carta, S.; Acquaviva, A.; Pisano, A.; Benini, L. "A Feedback-Based Approach to DVFS in Data-Flow Applcations". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 28, no. 11, pp. 1691-1704, 2009.

[6] Liu, S.; Qiu, M. "A Discrete Dynamic Voltage and Frequency Scaling Algorithm Based on Task Graph Unrolling for Multiprocessor System". In: Scalcom-Embeddedcom, pp.3-8, 2009.

[7] Kong, J.; Choi, J.; Choi, L.; Chung, S. W. "Low-Cost Application-Aware DVFS for Multi-core Architecture". In: ICCIT, pp.106-111. 2008.

[8] Chabloz, J. M.; Hemani, A. "Distributed DVFS using rationally-related frequencies and discrete voltage levels". In: ISLPED, pp.247-252, 2010.

[9] Yin, A. W.; Guang, L.; Nigussie, E.; Liljeberg, P.; Isoaho, J.; Tenhunen, H. "Architectural Exploration of Per-Core DVFS for Energy-Constrained On-Chip Networks". In: DSD, pp.141-146, 2009.

[10] Herbert, S.; Marculescu, D. "Variation-aware dynamic voltage/ frequency scaling". In: HPCA, pp. 301-312, 2009.

[11] Puschini, D.; Clermidy, F.; Benoit, P.; Sassatelli, G.; Torres, L. "Temperature-Aware Distributed Run-Time Optimization on MP-SoC Using Game Theory". In: ISVLSI, pp. 375-380, 2008.

[12] Puschini, D.; Clermidy, F.; Benoit, P.; Sassatelli, G.; Torres, L. "Adaptive energy-aware latency-constrained DVFS policy for MPSoC". In: SOCC, pp. 89-92, 2009.

[13] Goossens, K.; She, D.; Milutinovic, A.; Molnos, A.; "Composable Dynamic Voltage and Frequency Scaling and Power Management for Dataflow Applications". In: DSD, pp. 107-114. 2010.

[14] Beigné, E.; Clermidy, S.; Miermont, P. “Dynamic Voltage and Frequency Scaling Architecture for Units Integration within a GALS NoC”. In NOCS, pp. 129-138, 2008.

[15] Tschanz, J. et. al. "Adaptative Frequency and Biasing Techniques for Tolerance to Dynamic Temperature-Voltage Variations and Aging". In: ISSCC, pp 292-293, 2007.

[16] Chakraborty, K.; Roy,S. "Topologically Homogeneous Power-Performance Heterogeneous Multicore Systems". In: DATE, pp. 125-130, 2011.

[17] Carara, E., Oliveira, R., Calazans, N., Moraes, F. HeMPS - a Framework for NoC-based MPSoC Generation. In: ISCAS, 2009, pp.1345-1348.

[18] Sparso, J.; Furber, S. “Principles of Asynchronous Circuit Design – A Systems Perspective”. Kluwer Academic Publishers, 2001, 337p.

[19] Panades, I.; Greiner, A. "Bi-Synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures". In: NOCS, pp.83-94, 2007.