Adaptive Idleness Distribution for Non-Uniform Aging Tolerance in MultiProcessor Systems-on-Chip

Francesco Paterna and Luca Benini
DEIS - University of Bologna
V.le Risorgimento 2, Bologna, Italy
Email: [francesco.paterna@unibo.it|luca.benini]@unibo.it

Andrea Acquaviva
DAUIN - Politecnico di Torino
Corso Duca Degli Abruzzi 24, 10129 Torino, Italy
Email: andrea.acquaviva@polito.it

Francesco Papariello and Giuseppe Desoli
ST Microelectronics
Agrate, Italy
Email: [francesco.papariello|giuseppe.desoli]@st.com

Mauro Olivieri
DIE - Universit "La Sapienza" di Roma
Roma, Italy
Email: olivieri@die.uniroma1.it

***Abstract*—In deep submicron designs of MultiProcessor Systems-on-Chip (MPSoC) architectures, uncompensated within-die process variations and aging effects will lead to an increasing uncertainty and unbalancing of expected core lifetimes. In this paper we present an adaptive workload allocation strategy for run-time compensation of variations- and aging-induced unbalanced core lifetimes by means of core activity duty cycling. The proposed techniques regulates the percentage of idle time on short-expected-life cores to meet the platform lifetime target with minimum performance degradation.**

Experiments have been conducted on a multiprocessor simulator of a next-generation industrial MPSoC platform for multimedia applications made of a general purpose processor and programmable accelerators.

I. Introduction

Technology process variations in next generation Multiprocessor Systems-on-Chip cause performance uncertainty and unbalancing. To cope with this effect, countermeasures at various levels have been developed, ranging from transistor level, architectural and system software level. Software approaches can be very effective because they can adapt to wear-out and temperature dependency. There are several hardware techniques that can be used to make software aware of chip degradation, namely sampling based detection [3], [1], periodic testing, error correction and detection circuitry [5]. Once this information is made available at the software level, a common purpose of various approaches recently proposed is to provide wanted performance and match real-time constraints through statistical scheduling [6] or learning algorithms [7].

The main challenge of these techniques in a multiprocessor systems is to cope with the non-uniform distribution of critical path delay variations. To handle this heterogeneous delay distribution, each core can be clocked with a different frequency, thus increasing the need of synchronization for intra-core communication. A more conservative approach is to run all the cores at the same clock frequency dictated by the slowest core [4]. In both cases the aging effect will deviate the system from the starting condition, affecting the expected lifetime and its distribution between the cores. In this scenario, some cores will have a lower lifetime expectation than others, thus decreasing reliability and predictability of the system.

The objective of the work presented in this paper is to mitigate the impact on lifetime uncertainty and unbalancing among the cores. To this purpose, we developed an idleness distribution policy that increases core expected lifetimes by duty cycling their activity. The idleness is distributed to equalize the expected lifetime of each core to a target value, imposed by the system designer or by the user. Since the actual impact on performance depends on the task model running on the target multicore system, in this work we consider three representative task models, namely batch execution, playout and streaming, for which we evaluate the impact of the policy on the performance level. The proposed approach is based on variability information that can be provided at run time by variability monitors, that are likely to be embedded in next generation MPSoC designs.

The contributions of this paper can be summarized as follows. First, we propose an on-line adaptive strategy for increasing MPSoC tolerance to non-uniform wear-out due to variations. The methodology is innovative as it is focused on aging tolerance to improve system lifetime rather than on recovery of performance lost because of wear-out. Moreover, it is not based on static task characterization, but on on-line execution time and wear-out monitoring. Second, we propose an efficient implementation based on a look-up table that directly correlates target lifetime with idleness distribution. Third, we studied the impact of the aging tolerance policy on performance for various representative task models, demonstrating its negligible overhead and adaptation to different workload characteristics.

The rest of the paper is organized as follows. Section 2 discusses the variability model considered in this work. Section 3 presents the hardware and software infrastructure. Section 4 describes the proposed policy and Section 5 presents experimental results.

II. Variability Model and Idleness Constraints

In this work we consider intra-die variations in multiprocessor systems on chip that are not compensated at fabrication time. These variations together with wear-out effects that are likely to be distributed in a non homogeneous way, lead to unbalanced performance levels. In particular we consider a non-uniform distribution of critical path delays. A conventional design approach is to assume that system clock frequency has been set to some conservative value, dictated by the slowest core [2]. Not only this impact performance, but also each core has a different expected lifetime depending on how close is the critical path delay to the clock time.

Research activity supported by the European project REALITY FP7-2008-IST-1-216537

978-3-9810801-5-5/DATE09 © 2009 EDAA

The typical behavior of core lifetime is qualitatively expressed by the well-known bathtub curve, which provides the probability of failure over time. We assume to operate in the middle region of the curve. The relationship between the variability of the critical path delay and actual lifetime for each core depends on two factors. First, an aging function which expresses the delay critical path degradation as a function of time. We assume that the aging function is modulated by core activity. This means that the delay critical path does not degrades when the core is idle. This means that we can increase the expected system core lifetime by putting it in some standby state when idle, which is a realistic assumption for state of the art SoCs. The second factor is the effectiveness of the error correction circuitry that is possibly embedded in the architecture. The wear-out effect causes more and more severe timing violations and an increasing number of paths violating them as the time elapses, thus increasing the percentage of corrected errors.

The error correction circuitry is able to correct up to a certain error rate. If this rate is reached, the core cannot be recovered and thus it fails. For this reason, the expected lifetime can be computed as the time to reach this maximum error rate. An error correction systems can be exploited as monitor of the aging process. Using an aging model, it is possible to determine the expected lifetime based on the amount of corrected errors. In this way, our policy can directly use the lifetime information to know how much idleness is needed to match a given target lifetime requirement. This opportunity is depicted in Figure 1. Starting from an initial expected lifetime (t_{max}) which is achieved with 100% core activity, by playing with idleness it is possible to increase the lifetime up to a target value t_{lf}. The dashed line represents the activity duty cycling performed by inserting idle periods between task executions. We assume that the system is required to match a lifetime requirement for the whole system and we play on idleness distribution of each core in order to increase the expected lifetime to match the target one.

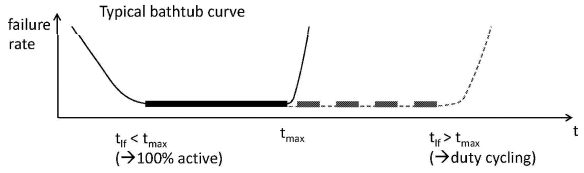


Fig. 1. Relationship between idleness and core lifetime.

III. Hardware and Software Platforms

A. MPSoC Simulation Platform

The target platform used to validate the policy presented in this work is a Multi-Processors System-on-Chip simulator provided by ST Microelectronics. The simulation model is at TLM level added with timing support. Its accuracy allows to model low level architectural details such as memory contention. The simulation platform has a configurable number of VLIW accelerators and a master processor of the st231 family. Memories can be configurable to be either local or shared. In the considered configuration, each processors has also a local memory for data and program. A shared memory between master and accelerators has been also instantiated. The interconnect is an ST-NoC.

B. Software Infrastructure and Task Models

The software organization of our system is composed by support functions for task loading, data communication and synchronization, statistic collection. All the cores load the same program, following a SPMD (Single Program Multiple Data) approach, where each core executes a different portion of the program depending on its identifier. The accelerator code contains all the possible tasks to be executed. Currently, dynamic loading of tasks is not supported. As such, to execute a certain task, cores have to jump to the related code portion, which is identified by a pointer. To control the execution on the accelerators, the master core changes the pointer depending on which task the accelerator has to run. Shared memory is used to exchange data among cores.

Batch execution model. In the batch execution model, the master core spawns a number of N independent tasks on the accelerators exploiting a non-blocking round-robin algorithm. The performance metric associated with this task model is the execution time, that in this case is defined as the time between the allocation of the first task and the completion of the last allocated task. Input and output data are stored in local memories of accelerators.

Output rate-constrained execution model (playout). This model is representative of playout activity performed by audio or video decoders. Also in this case the master allocates tasks on the accelerators. Accelerators read input data from their local memories. Output data items are stored in a common output queue allocated in shared memory with access regulated by semaphores. A consumer task runs in one dedicated core which periodically pick one data item from the output queue. The associated performance metric is the output throughput. When the output queue becomes empty, the consumer will experience a deadline miss. As such, the performance constraint is represented by the output rate.

Input-output rate-constrained execution model (streaming). While in the playout model input data for accelerators are available on local memories, in streaming task model data are provided to the accelerators by the master core. This is a typical model for a videoconferencing application where the input data are provided by a video camera and accelerators performs video encoding. Another example is a video decoder application receiving compressed frame from the network. An interprocessor communication queue is used as buffer between master and accelerators. As in the playout model, an output queue is used to synchronize data communication with the consumer core. The associated performance metric in this case is not only the output throughput, but also input throughput. If the input queue becomes full, this means that accelerators are not able to handle the input data rate. The constraint on the output still applies also in this task model.

IV. Adaptive Idleness Distribution Policy

The master core is responsible of allocating tasks on the accelerators. For this reason, it is the most suitable place where to implement the idleness distribution algorithm. Since the distribution algorithm depends on the reading of variability monitors of each core. Our target platform is equipped with a register accessible from the master and all the cores where the percentage of corrected errors (also called error rate) can be read for each core.

Our policy computes a required amount of idleness for each core. In order to make the policy implementation independent from the type of runtime information available, the policy

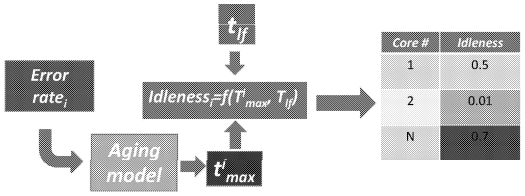


Fig. 2. Implementation scheme of the adaptive idleness distribution policy.

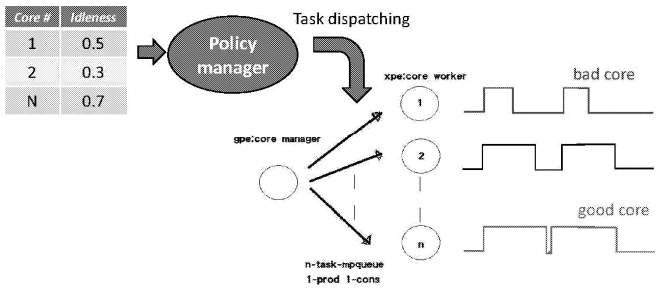


Fig. 3. Adaptive idleness distribution policy description.

takes as input a required idleness for each core. A conversion module fills up a table with the idleness values computed starting from error rate statistics for each core. An aging model as described in Section II is used to compute the time required to reach the max_error_rate value assuming zero idleness, that we call t^i_{max}, where i indicates the $i-th$ core. For each core, the target amount of idleness for a generic $i-th$ core is defined as:

$$idleness = \begin{cases} 1 - \frac{t^i_{max}}{t_{lf}}, t_{lf} > t^i_{max}; \\ 0, t_{lf} < t^i_{max} \end{cases}$$

where t_{lf} is the system lifetime requirement and idleness is expressed as a number between 0 and 1, where 0 indicates full activity and 1 indicating no activity. Once the wanted average $idleness$ has been computed it is stored in a table as shown in Figure 2. Then, the master processor must perform the task allocation policy accordingly, as depicted in Figure 3. To achieve the wanted average idleness, our policy allocates idle periods between task executions for each accelerator. This implies that the wanted idleness is achieved on a time scale on the order of task execution times. This is reasonable as long as the expected lifetime is typically several orders of magnitude larger than task durations. Indeed, the implementation on a smaller timescale would imply pre-emption of tasks on the accelerators, introducing an unnecessary overhead. It must be noted that the proposed policy does not assume a specific aging model. The unique assumption is that additional idleness increases core lifetime.

As a result, the master core exploit hardware timers to update a data structure where task start and completion times are stored. After each task completes, its activity interval is computed. The idle period to be allocated is obtained by multiplication of the last activity period by the wanted idleness. After the idle period expires for a core, a new task is allocated to it.

It must be taken into account that cores must also perform task management (i.e. loading and completion notification) and synchronization operations (i.e. waiting on semaphores), as needed to implement a given task model. When computing

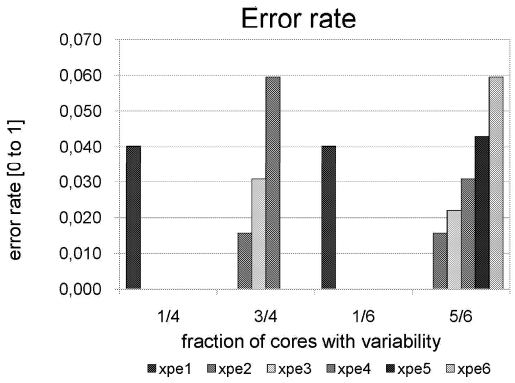


Fig. 4. Variability scenarios. Error rates are mean values of a gaussian distribution.

the idle period to be allocated to each core, this additional activity is taken into account by our policy. This is possible because the master core has full visibility and monitoring capability of accelerator's activity. The idleness for each core is conservatively updated by the master core at each task completion, depending on monitor readings. However, frequency of updates can be configured. Experimental results show that the implementation overhead of this policy is negligible and that the wanted idleness is obtained with a very high accuracy.

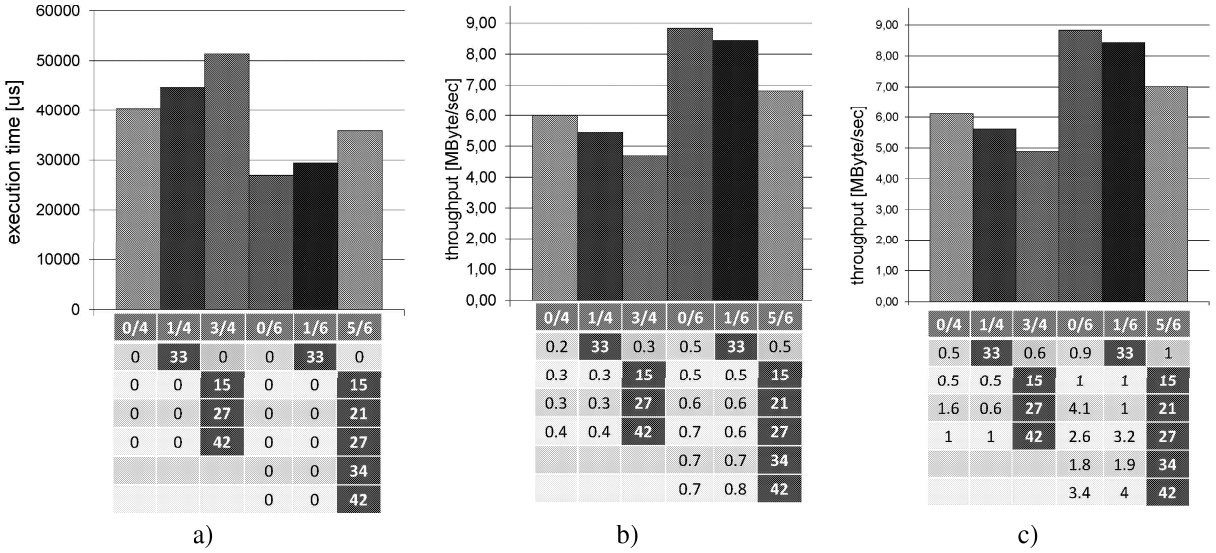
V. Experimental Results

The policy described in Section IV requires software support mechanism for task activity monitoring and idleness computation, that could impact the accuracy of idleness distribution. For our experiments we considered two platform configurations, namely four and six accelerators. For each configuration, we considered three variability scenarios. Each variability scenario defines the number of cores affected by variability issues and the mapping of error rates on the cores. In our simulation platform, error rates are extracted from a gaussian distribution. In our experiments we considered a static condition where monitor readings (i.e. variability conditions) are constant over time. However, we consider a worst case scenario where the master core reads the variability information at each task completion. The platform configurations and variability scenarios considered for our experiments are described in Figure 4.

It must be noted that minimum and maximum values of error rates are the same for the four and six core configurations. Benchmarks used for experiments are matrix multiplication kernels. To the purpose of characterization of idleness computation accuracy we measured the actual idleness and we compared it with the target one. The results we obtained about idleness accuracy, that are not shown here for space limitations, highlight that the maximum error in idleness assignment is within 0.1%, demonstrating the effectiveness of the proposed software infrastructure.

Batch Execution Results. The matrix multiplication benchmark $A \cdot B = C$ is composed by two phases. During the first one the matrix B is copied from shared memory to local memory, where A resides. In the second phase the actual matrix multiplication takes place. Results are stored in the local matrix C.

Increasing the lifetime may have an impact on performance depending on the task model. For batch execution, performance hit lead to an increase of the overall execution



a) b) c)

Fig. 6. Impact of variability on the output throughput for a) batch task model; b) playout model; c) streaming model.

	0/4	1/4	3/4	0/6	1/6	5/6
execution time [us]	40258	44715	51427	26965	29404	35875
relative impact [%]		11	28		9	33
throughput [Mbyte/s]	6,00	5,45	4,70	8,83	8,44	6,81
relative throughput [%]		9,14	21,67		4,42	22,86
throughput IN [Mbyte/s]	6,11	5,61	4,89	8,83	8,44	7,01
relative throughput IN [%]		8,24	20,00		4,42	20,59
throughput OUT [Mbyte/s]	5,96	5,45	4,72	8,67	8,29	6,86
relative throughput OUT [%]		8,57	20,79		4,35	20,86

Fig. 5. Relative impact of variability on performance for all the scenarios and configurations

time of N tasks, where N has been fixed to 60. Results are shown in Figure 6.a, where associated idleness values for each core are also reported for clarity. In Figure 5 the relative impact on execution time is shown. For each platform configuration (i.e. four vs six cores), this has been computed using the scenario without variations as reference. By comparing the two platform configurations, it can be noted that the impact on execution time is proportional to the fraction of variability-affected cores. For instance, the execution time of the 5/6 configuration has a larger increase than for the 3/4 one. However, with the given error rate distribution this is not enough to make any of the four cores configuration more performing.

Output Rate-Constrained Processing Results. In this case the metric to be considered is the output throughput. In order to consider a worst case condition, we set the consumer frequency corresponding to the maximum throughput that can be delivered by the six core configuration, which is about 9MBytes/sec. As such, introducing idleness has an immediate impact on throughput, as it can be observed in Figure 6.b. Differently from the execution time for the previous task model, throughput degradation here is less sensitive to the fraction of variability affected cores. Indeed, in Figure 5 the 5/6 scenario has a throughput drop of 23% while the 3/4 scenario has a throughput drop of 22%. However, by comparing 1/4 and 1/6 scenarios, the relative throughput drop is 9.1% compared to 4.4%.

Input-Output Rate-Constrained Streaming Results. Both input and output throughput are critical in this case. Figure 6.c shows variability effects on the input throughput. The same results have been obtained for the output throughput (not shown). Interestingly, also for the input throughput the relative performance drop for high throughput values is similar for 3/4 and 5/6, being around 20% in both cases (see Figure 5).

VI. Conclusion

In this work we presented an adaptive idleness distribution policy aimed at reducing the impact of variations and aging on the lifetime of MPSoCs. The policy exploits variability monitors and online task execution statistics to determine the duration of idle intervals to be distributed to the cores to match a given lifetime requirement. The proposed strategy has been implemented on a industrial simulator of a next generation nanoscale multiprocessor platform.

References

[1] Jason Blome, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. Online timing analysis for wearout detection. In . *Second Workshop on Architectural Reliability (WAR)*, December 2006.

[2] Keith A. Bowman, Alaa R. Alameldeen, Srikanth T. Srinivasan, and Chris B. Wilkerson. Impact of die-to-die and within-die parameter variations on the throughput distribution of multi-core processors. In *ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design*, pages 50–55, New York, NY, USA, 2007. ACM.

[3] S. Das, Sanjay Pant, D. Roberts, Seokwoo Lee, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. A self-tuning dvs processor using delay-error detection and correction. *VLSI Circuits, 2005. Digest of Technical Papers. 2005 Symposium on*, pages 258–261, June 2005.

[4] David Roberts, Ronald G. Dreslinski, Eric Karl, Trevor Mudge, Dennis Sylvester, and David Blaauw. When homogeneous becomes heterogeneous. In *Third workshop on Operating Systems for Heterogeneous Multiprocessor Architectures (OSHMA)*, 2007.

[5] Jared Smolens, Brian Gold, James Hoe, Babak Falsafi, and Ken Mai. Detecting emerging wearout faults. In *Third Workshop on Silicon Errors in Logic*, April 2007.

[6] Honggang Wang, Wei Wang, Dongming Peng, and Hamid Sharif. A route-oriented sleep approach in wireless sensor networks. In *CS*, 2007.

[7] J.A. Winter and D.H. Albonesi. Scheduling algorithms for unpredictably heterogeneous cmp architectures. In *38th International Conference on Dependable Systems and Networks*, pages –, 2008.