Synchronization-Driven Dynamic Speed Scaling for MPSoCs

Mirko Loghi
Politecnico di Torino
Torino, Italy
mirko.loghi@polito.it

Massimo Poncino
Politecnico di Torino
Torino, Italy
massimo.poncino@polito.it

Luca Benini
Università di Bologna
Bologna, Italy
lbenini@deis.unibo.it

ABSTRACT

Equalizing the ratios between workloads and speeds of processing elements provides the optimal speed allocation. Based on that principle, this work describes a dynamic speed setting policy for multi-processor systems-on-chip (MPSoCs) that relies on the estimation of processor idle times specifically due to the synchronization work. The policy provides two advantages: first, it does not rely on any assumption about the communication pattern of the application executed by the system. Second, it is purely architectural; it automatically detects changes in the system workload and sets processors speeds accordingly by means of a custom hardware block.
Results on a parallel MPEG video decoding application show an EDP saving above 55%, averaged over several datasets, corresponding to an energy saving above 50%, and a corresponding penalty in performance below 8%.

Categories and Subject Descriptors: C.1.2 [Computer Systems Organization] : Multiprocessors; J.6 [Computer Applications]: COMPUTER-AIDED ENGINEERING

General Terms: Algorithm, Design, Experimentation.

Keywords: Power Optimization, MPSoC, Dynamic Voltage/Frequency Scaling.

1. INTRODUCTION

The power crisis in current digital integrated systems is fueling an architectural paradigm shift toward multi-core architectures: Single-chip multi-core engines, that have first become widespread in embedded computing, are now making deep inroads in general purpose computing [1].
Energy-efficient multi-core design is getting an increasing level of technology support. The concept of "voltage islands" [2] has gathered momentum, and many recently announced SoC architectures feature tens of power domains [3]. Multi-core architectures with independently controllable supply voltage and clock frequency enable an unprecedented level of control on the performance vs. power/energy tradeoff. Voltage and speed setting allows to virtually eliminate all wasteful mismatches created by non-uniform workloads allocated to cooperating cores.

ISLPED'06, October 4–6, 2006, Tegernsee, Germany.

A tough challenge in this area is the accurate and timely detection of workload mismatches at execution time. In many realistic use cases, applications cannot be accurately pre-characterized, and efficient run-time mechanisms are required to detect when one or more cores are over-clocked with respect to their workload. To address this challenge, several techniques have been proposed that require modification in the applications ([6]–[8]), or impose a particular inter-processor communication style that helps making speed mismatches more easily detectable [9, 10, 11, 12].
This paper moves towards the development of minimally-invasive detection of workload mismatches. Instead of requiring additional efforts from the programmer's side, we augment the hardware architecture with self-monitoring and autonomous control capabilities. We *monitor synchronization* to detect non-constructive idleness, which is a direct manifestation of non-optimal speed setting. We infer the idleness by observing the memory accesses performed by processors, and use such an information to drive a dynamic speed setting policy.
Results on a MPEG decoder show that our approach can save more than 50% of the energy spent with respect to a system running at the maximum speed, also improving by 25% a static assignment of speeds based on the off-line monitoring of the system workload.

2. PREVIOUS WORK

The problem of voltage/frequency selection is a quite mature research topic: many techniques have been proposed in the literature, requiring various levels of support by either hardware or software (a survey of DVS/DFS techniques can be found in [4, 5]). All these schemes, however, target single-processors systems.
In the multi-processor system domain, conversely, the spectrum of solution is much more limited. Most approaches have focused on a task-based model of the system, in which system behavior consists of a set of tasks with well-defined execution times and deadlines. The availability of an RTOS is usually assumed, and the issue of speed selection is embedded into the task scheduling problem with real-times constraints ([6]–[8]).
The approaches followed by [11, 12] introduce a control-theoretic perspective of the speed assignment problem, which is solved by modeling the system as a queue network. Characterizing embedded applications in terms of service and arrival rates tends to be difficult, however; these approaches thus tend to be more suitable for an analytical exploration of workload allocation policies than for detailed frequency assignments.
In [9], authors concentrate on the balance between computation and communication, concurrently determining an optimal speed assignment for both communication and computation tasks. This work is explicitly targeted for network processors, for which communication power is significant.

Our scheme shares with [9] the assumption that tasks are statically mapped to processing elements: in this scenario, scheduling of the tasks is immaterial, and only speed assignment is relevant. Our solution differs from this approach in two main aspects: First, we do not make *any relevant assumptions about the application running on the system*, including its data-flow or communication patterns; second, our method is *purely architectural*, in the sense that it automatically detects, through a custom hardware block, changes in the system workload, and sets processors speeds accordingly.

3. SYNCHRONIZATION-DRIVEN SPEED SCALING

The proposed speed scaling scheme is based on the idea of *relating the speed assignment of a given processor to the amount of time spent waiting for synchronization*.

In this work we consider applications that rely on a shared-memory paradigm, and execute on a multi-core platform explicitly based on shared-memory. We therefore assume that synchronization is based on shared variables. Our "low-level" synchronization primitive is therefore based on the polling of some shared variables.

We then assume that the system consists of n cores numbered $1, 2, \ldots, n$. Each core i can be set to a speed S_i, which can take values from a set of m possible speeds $S_k, k = 1, ..., m$. The i-th core executes a *workload* W_i, measured in terms of the amount of work (e.g., instructions) to be performed; its actual unit is irrelevant since our formulation relies on the ratio between workload and speed W_i/S_i, which yields a measure of time.

Finally, we denote by T_i and T_{S_i}, respectively, the time spent by processor i executing its workload and busy waiting for some synchronization variable to become available.

3.1 Energy, Workload and Synchronization Time

The time spent for synchronization by one processor represents a form of non-constructive idleness; unlike idleness due to other factors (e.g., I/O waiting), this idleness is caused by the *interference* of other processors, making thus its nature (and thus the issue of speed setting) quite different from the single-processor case.

Intuitively, time spent during synchronization wastes energy, since a core busy-waits without performing useful computation. In order to support this intuition, we ran an experiment to establish a relation between the time spent for synchronization and energy, consisting of an exhaustive exploration on a synthetic application.

The latter implements a prime number search on a shared vector, implementing a parallel version of the Eratosthenes' algorithm. By allocating slices of the vector with different sizes to cores, it is possible to split the total amount of computational work as desired, thus allowing explorations with different workload allocations.

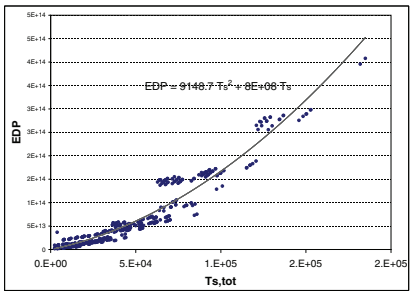

Figure 1: Relation Between EDP vs. Total Synchronization Time.

Figure 1 shows the energy-delay product versus the total synchronization time $T_{S,tot} = \sum_{i=1}^{n} T_{S_i}$ (values of T_{S_i} have been measured through simulation on the platform described in Section 4). The plot clearly supports the intuition and shows how execution time and energy consumption are strongly impacted by the amount of time spent in synchronization. Using these observations, we derive a speed setting policy driven by the idle synchronization time.

Le us consider an arbitrary time window TW; over this window, the time spent by the application running on the system will consists, in general, of three distinct components: the time for executing "useful" instructions, the time spent for synchronizing with other processors, and the time used to wait for the bus to access its private memory. In formula: $TW = T_i + T_{S_i} + T_{BW_i}$. Without a significant loss in accuracy, we can neglect T_{BW_i}, since caches will filter out most of the memory accesses.

Reformulating, and by exposing processor speed, we get:

$$TW = \frac{W_i}{S_i} + T_{S_i} \tag{1}$$

Notice that TW is an invariant, that is, it is the same for all cores. Based on the results of Figure 1, the objective is to reduce T_{S_i} to 0; this can be achieved by either slowing down some processors, or by speeding up some others. Consider for instance the situation depicted in Figure 2, where processor P_2 waits, for a time T_{S_2}, processor P_1 to produce an event (e.g., unlock of a semaphore). This

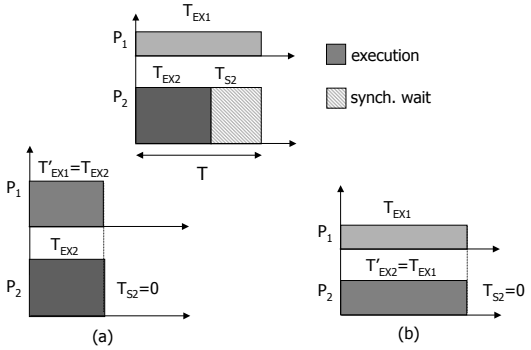

Figure 2: Alternative Speed Assignments: Speeding Up P_1 (a) or Slowing Down P_2 (b).

time can be reduced to 0 by speeding up P_1 of a quantity T_{S_2}/T (a), or by slowing down P_2 of the same quantity. The two alternatives differ in the execution time, which is correlated to the speed of the two processors. While from the point of view of EDP they are both identical (because $T_{S_2} = 0$), we should tend to privilege the scheme where the overall execution time is smaller.

The example suggests a possible qualitative criterion for speed setting: *If a processor P_i waits, then slow it down if it is running at the maximum speed, otherwise speed up the other cores.*

3.2 Speed Scaling Algorithm

We consider time to be split into a set of disjoint time windows of the same size TW; the speeds of the cores are determined upon expiration of each time window, based on the quantities measured in the previous window. This discretization relies on the assumption that the *workloads are slowly changing with respect to the observation window*, which is quite reasonable if TW is much smaller than the total execution time.

Computation of the optimal speeds requires the estimation of the incoming workloads; from Equation 1, a workload can be espressed as $W_i = (TW - T_{S_i}) \cdot S_i$. At the end of a time window, all the three quantities on the right hand side of this equation are known (TW

and S_i), or can be measured (T_{S_i}). This allows to compute W_i for the just expired time slot; under the assumption that workload will not change abruptly in the next slot, we use W_i as an estimate of the incoming workload in the next slot.
In order to achieve the optimal speed assignment ($T_{S_i} \equiv 0, \forall i$), we must enforce that all workloads are executed in the same time, that is, all the ratios W_i/S_i are identical:

$$\frac{W_1}{S_1} = \frac{W_2}{S_2} = \ldots = \frac{W_n}{S_n} \quad (2)$$

This condition implies that all processors will execute their workloads without synchronization idle time (e.g., as in Figure 2).
Formula 2 is a system of $n-1$ equations with n unknowns. In fact, the time to which each W_i/S_i evaluates to depends on the speed assignments and it is not defined a priori.
Since we aim at obtaining the same performance of a full speed system, still with a reduced energy consumption, we solve Equation 2 by *forcing the processor with the maximum workload to run at maximum speed.* All the other speeds will thus be automatically derived based on Equation 2.
More precisely, let j be the index corresponding to the processor with maximum workload during the k-th time window:

$$j = \text{index}(\max_{1,\ldots,n} W_i)$$

The speed setting for the $k+1$-th time window is thus defined as

$$\begin{cases} S_{j,opt} = S_{max} \\ S_{i,opt} = \frac{W_i}{W_j} \cdot S_{max} \end{cases} \quad (3)$$

where S_{max} is the maximum available speed in the system.
Notice how this simple speed setting criterion is able to deal naturally with the most critical difficulty for a DVS scheme, that is, the problem of workloads that are variable over time.

4. EXPERIMENTAL RESULTS

To verify the effectiveness of the proposed policy, we first run a set of parametric benchmarks that expose different workload allocations and different kinds of parallelism. We have then applied it to a real-life embedded application, (an MPEG video decoder), for which some parametric exploration has also been carried out.
In both experiments, we compared our policy against (i) a full-speed system (all cores running at the maximum speed), and (ii) one static speed assignment. Since the only way to determine the optimal static assignment would be by exhaustive exploration of all possible frequency assignments, we determined a "reasonably good" assignment, obtained by extracting some execution information from the application code, and by a further exploration of other points around this possible optimal value. This procedure is supposed to emulate the choice of a designer that has to select a static assignment for a target application.

4.1 The Multiprocessor Platform

We implemented our policy by augmenting a virtual multiprocessor platform ([13]), consisting of a configurable number of processing elements, with their private memories, a shared memory, some hardware device for basic interprocessor synchronization (a hardware interrupt module, and a hardware device that provides test-and-set features), a shared-bus interconnect, and a clock divider that feeds the cores.
We augmented the platform by adding a speed setting device based on the above scheme. The device estimates the T_{S_i} and updates the processors' frequencies accordingly. Synchronization is evaluated by monitoring accesses performed by each core. We assume that a processor is performing a busy wait when it repeatedly accesses the same address with no accesses to other locations between them. Accesses must be detected by inspecting the interface between each core and its cache controller, since many accesses are filtered by the data cache.

4.2 Application Kernels

- **Eratost**: A prime number search on a shared vector, implementing a parallel version of the Eratosthenes' algorithm. Each processor scans a disjoint portion of the vector, and cores synchronize using barriers.
- **Mergesort**: A parallel version of the mergesort algorithm; it sorts a pool of randomly generated vectors stored in shared memory. The application is parallelized in a barrier-based fashion for the sorting phase, and in a pipeline style for the final merging phase. Therefore, when the merging processor is working on the i-th vector, the other ones can sort the $(i+1)$-th. The workload allocation among cores is not uniform and it varies during the execution.
- **DES**: A DES algorithm parallelized in a master-workers-slave fashion. The master core sends the stream to two workers that encode it and send the encoded data to the slave core. The resulting workload is not uniform among cores (workers bear the heavier computational load) but it is static, since the amount of computation does not depend on data streams.
- **FFT**: a variant from the *SPLASH-2* [15] suite, modified to perform two concurrent FFTs on two distinct vectors of different sizes. Half of processors are in charge to perform the computation of the first vector, while the other cores handle the second vector. The computation is repeated several times and the role of processing elements is changed after each iteration, in order to obtain a non-uniform and non-static workload allocation.

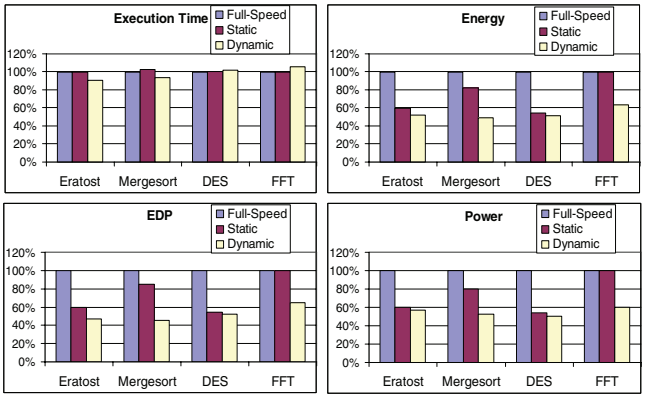

Figure 3: Results for Application Kernels.

Results of those experiments are shown in Figure 3. For each benchmark, we compare execution time, energy consumption, energy-delay product, and power, all normalized to the full-speed case. We can observe that our policy outperforms the static policy whenever the workload is variable in time (as usually happens in real complex applications). Conversely, when the workload is static (as in *DES*), the dynamic policy provides results comparable to the "optimal" static policy.
For the *FFT* application, since the best static allocation is the one with all cores at full speed, no savings can be obtained. Conversely,

our dynamic policy, following the workload variations, can achieve appreciable benefits even in such a condition.
Notice also that, in some cases, a good speed setting policy can provide, as a side effect, a small improvement in terms of execution time. This happens because slowing down some processor reduces the traffic on the shared medium, hence allowing faster accesses to cores that execute at higher speeds.

4.3 Case Study

The application used in our case study decodes an MPEG video stream. One core is in charge of decoding the frame headers, and performs the entropic decompression of the data stream (Huffman and RLE decoding). The other 2 cores perform the de-quantization and the inverse-DCT (for I-frames), or they apply the motion compensation (for P- and B-frames). The last processor collects decoded data and sends them to the output buffer. The workload is not static, because different kinds of frames in a MPEG stream require different amounts of computation to be decoded.
Figure 4 shows how our policy outperforms the static policy in decoding a video stream where 20% of frames are I-frames. We reported data for execution time, energy spent, energy-delay product, and average power, all normalized with respect to the full-speed case (as a quantitative measure, the full-speed execution requires 472 million cycles and 293 mJ).

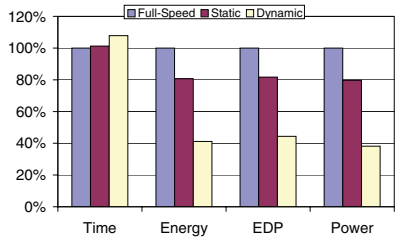

Figure 4: Comparison of Different Policies.

Since the workload behavior depends on the ratio between the various frame types, we used streams with different features to test our policy, varying the $(\mathrm{Bframes} + \mathrm{Pframes})/\mathrm{Iframes}$ ratio. Figure 5 shows results of this exploration: we can observe how savings and penalties are just marginally affected by the input characteristics (time penalties do not exceed 8%, while the energy saving ranges between 56% and 64% with respect to full speed execution).

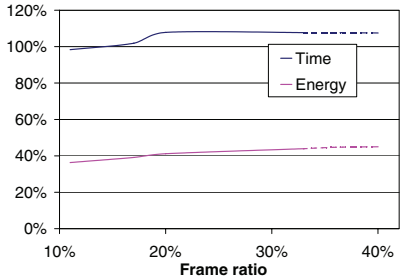

Figure 5: Dependency on the input features.

Last, we explored the dependency of our policy on TW (the size of the sampling window). Figure 6 shows how execution time and energy vary for window sizes between 2 and 64 KCycles: it is evident from the plot how the behavior of our policy is substantially independent of the window size.

5. CONCLUSIONS

We presented a purely architectural solution for the reduction of dynamic energy in MPSoCs, based on the principle of the equalization

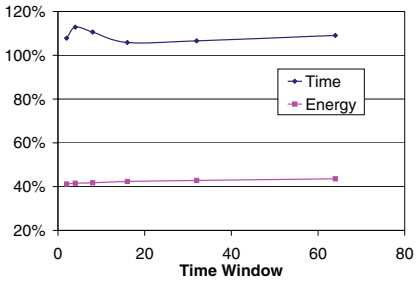

Figure 6: Dependency on the size of the sampling window.

of the normalized workload of the cores. Balancing is driven by the idleness due to synchronization, and provides relevant benefits in terms of energy consumption, with marginal impact on performance. The dynamic speed allocation policy does not require any software support, since it can infer the time spent in synchronization by examining the behavior of the processors on-line.
Results on a real life application (a parallel MPEG decoder) show that our policy outperforms other static policies, and it does not suffer from relevant performance penalties. We reduce energy by more than 40% with respect to a static assignment, and about 60% with respect to the full-speed system. Performance penalties remain below 8% with respect to the system with all cores running at the maximum speed, thus leading to savings in energy-delay product larger than 50% with respect to a full-speed execution.

6. REFERENCES

[1] David Geer, "Industry Trends: Chip Makers Turn to Multicore Processors," *IEEE Computer*, Vol. 38, No. 5, pp. 11-13, May, 2005.

[2] D. Lackey, et al., "Managing Power and Performance for SOC Designs using Voltage Islands," *ICCAD 2002*, pp. 195–202, Nov. 2002.

[3] "A Power Management Scheme Controlling 20 Power Domains for a Single-Chip Mobile Processor," Y. Kanno, et al., *ISSCC'06*,pp 540–541, Feb. 2006.

[4] W. Kim, D. Shin, H. S. Yun, J. Kim, S. L. Min, "Performance comparison of dynamic voltage scaling algorithms for hard real-time systems", *RTAS'02:IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 219–228, 2002.

[5] B. Zhai, D. Blaauw, D. Sylvester, K. Flautner, "Theoretical and Practical Limits of Dynamic Voltage Scaling", *DAC'04:41st Design Automation Conference*, pp. 868–873, 2004.

[6] P. Yang, et al., "Energy-Aware Runtime Scheduling for Embedded Multiprocessor SOCs," *IEEE Design and Test of Computers*, pp. 46–58, Vol. 18, No. 5, 2001.

[7] D. Zhu, R. Melhem, B. Childers. "Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 7, pp. 686–700, July 2003.

[8] W. Kwon and T. Kim. "Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors". *IEEE Transactions on VLSI*, pp. 125–130, June 2003.

[9] J. Liu, P. Chou, N. Bagherzadeh, "Communication Speed Selection for Embedded Systems with Networked Voltage-Scalable Processors", *CODES'02*, pp. 169–174, May 2002.

[10] Z. Lu, J. Lach, M. Stan, "Reducing Multimedia Decode Power using Feedback Control," *ICCD'03:*, pp. 489-494.

[11] Q. Wu, P. Juang, M. Martonosi, L.-S. Peh and D.W. Clark. "Formal Control Techniques for Power-Performance Management". *IEEE Micro*, Vol. 25, No. 5, Sep./Oct. 2005, pp. 52–62.

[12] A. Alimonda, S. Carta, A. Pisano, A. Acquaviva, "A Control Theoretic Approach to Run-Time Energy Optimization of Pipelined Elaboration in MPSoC," *DATE'06*, pp. 876–877.

[13] MPARM Home Page, *www-micrel.deis.unibo.it/sitonew/research/mparm.html*

[14] A. Bona, V. Zaccaria, R. Zafalon, "System-Level Power Modeling and Simulation of High-End Industrial NoC", *DATE'04*, pp. 318–323, Mar. 04.

[15] J. P. Singh, W.-D. Weber, A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory", *Computer Architecture News*, Vol. 20, No. 1, pages 5-44, March 1992.