Predicting Bus Contention Effects on Energy and Performance in Multi-Processor SoCs

Sandro Penolazzi, Ingo Sander and Ahmed Hemani
Dept. of Electronic Systems, School of ICT, KTH, Stockholm, Sweden
{sandrop, ingo, hemani}@kth.se

***Abstract*—We present a high-level method for rapidly and accurately predicting bus contention effects on energy and performance in multi-processor SoCs. Unlike most other approaches, which rely on Transaction-Level Modeling (TLM), we infer the information we need directly from executing the algorithmic specification, without needing to build any high-level architectural model. This results in higher estimation speed and allows us to maintain our prediction results within $\sim$2% of gate-level estimation accuracy.**

I. Introduction

Multi-Processor SoCs (MPSoCs) have emerged as one possible solution to cope with the demand of increasing performance and computational power. However, since all processors are typically connected to the same bus, contention can occur, which results in a degradation of performance and in an increase of energy consumption.

Being able to estimate these two quantities early in the design cycle is essential to avoid costly and time-consuming design reiterations. Making it possible is the goal of this paper.

Raising abstraction is a fundamental step towards achieving early design estimation [3] and Transaction-Level Modeling (TLM) is a good example of how the abstraction level can be raised above RTL. However, TLM still implies a considerable engineering effort, since it relies on simulating both the software and the underlying hardware. This can make even TLM slow when running large and complex use-case scenarios.

Our approach differs from TLM in that it does not simulate a high-level architecture, but it infers the architectural implications by combining native code execution and back-annotation with target-specific information. The abstraction level of our approach is higher than TLM, thus resulting in faster estimation. We rely on the following two steps:

1) Characterizing bus contention (Section III): this step consists in identifying and analytically expressing the main factors responsible for bus contention. Although time consuming, this step is a one-time activity that should be carried out by the IP provider.

2) Predicting bus contention (Section IV): this step aims at predicting the effects of the aforementioned factors, in terms of performance degradation and energy consumption increase, for an MPSoC with N processors running N applications. Unlike the step above, this step is very fast, as the prediction algorithm is applied on a code profiled during native execution on the host and back-annotated with target-specific information. This step is responsibility of the system designer.

978-3-9810801-7-9/DATE11/©2011 EDAA

II. Related work

Many high-level modeling approaches rely on TLM. In [6] the authors choose an approximately-timed transaction-level implementation to evaluate the timing effects due to bus contention when mapping applications to MPSoCs. The model is written in SystemC. In [5], the authors also use TLM to increase communication speed in MPSoC design. In [4], a methodology is proposed for software estimation in heterogeneous multiprocessor systems. Applications are represented as a set of tasks forming a fork-join task graph, while the architecture is specified using the High-level Machine Description (HMDES) language [2].

Although more abstract and faster than RTL, these approaches rely on the actual simulation of the target architecture and, as such, they may still be too slow for modeling complex scenarios. As opposed to them, our approach runs the application code natively on the host and back-annotates it with target-specific information. After that, a stochastic/analytical model for contention detection is applied.

III. Characterizing bus contention

In this section, we identify the factors involved in bus contention and create an analytical model for them. Although time-consuming since done at gate level for gaining higher accuracy, this is a one-time activity done by the IP provider.

We define e and E as the execution time and energy consumption of an application, assuming that this runs on a single-processor SoC and has all the resources available for itself. In this ideal case, no bus contention occurs. Executing N times the same application on this single-processor SoC would result in an execution time $e_N = N \cdot e$ and energy consumption $E_N = N \cdot E$. Assuming now to have an MPSoC with N processors and to run the same application in N identical copies, each one mapped onto one processor, we still expect an energy consumption $E_N = N \cdot E$, but we now expect an execution time $e_N = e$. However, because of bus contention, we get $E_N = N \cdot (E + \Delta E)$ and $e_N = e + \Delta e$. Identifying and analytically expressing the factors responsible for Δe and ΔE is the focus of this section.

Our reference architecture is a SoC composed of one SPARC compliant Leon3 processor [1], one shared SRAM

memory and one AHB controller. This configuration is then extended to a homogeneous MPSoC hosting up to 7 processors. A Round Robin scheduling policy is chosen for bus arbitration.

We use an FFT, a Fibonacci and a Viterbi as a characterization benchmark. Such applications, which are of common-use and show regular algorithms, run at gate level, which allows to get accurate numbers for execution time, energy consumption and a detailed instructions trace. We vary the number N of processors between 1 and 7. Besides, the same application runs in N identical copies, each one mapped onto one processor. Due to the very long time necessary to run gate-level simulations and energy extractions, we have limited to three the number of applications used for calibration and we have only considered Round Robin for bus arbitration.

We identify two major factors affecting bus contention: the number of processors (Subsections III-A) and the number of w/r accesses to memory (Subsection III-B).

A. *Factor 1: number of processors*

In Figure 1 we show the execution time of each of the three applications running on the different SoC configurations.

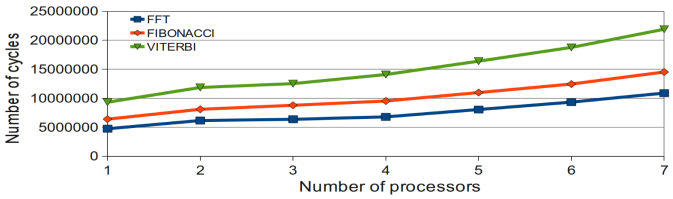

Fig. 1. Execution time e (in number of cycles) for FFT, Fibonacci and Viterbi

The first factor involved in bus contention is the number of processors, which has a nearly linear impact and shows a similar progression in all the three benchmarks considered. We associate the increase of execution cycles to stall conditions for the processors.

We define the percentage of cycles difference $C^{\%}_{x,y}$ between the N-processor and the 1-processor case, where $x = N$ and $y = 1$, for the benchmark applications chosen. The percentage is calculated having the 1-processor case as a reference. The results suggest that this value is quite uniform for the same comparison interval across the different applications.

If we indicate with N the total number of processors and consider the i-th application running on the i-th processor, where $i \leq N$, we can refer to the execution time e and energy consumption E of each i-th application as $e_i(N) = e_i(1) + \Delta e_i(N)$ and $E_i(N) = E_i(1) + \Delta E_i(N)$. Note that $e_i(1)$ and $E_i(1)$ correspond to the ideal case where an application runs on a single-processor SoC, with no bus contention. We also introduce the quantity $C_i(N)$ as the number of execution cycles for the i-th application in the N-processor SoC case. Note that the value of $C_i(1)$ is available through the relation $C_i(1) = e_i(1)/T$, being T the clock period. We can thus express $\Delta e_i(N)$ with $N > 1$ using the simple Equation 1:

$$\Delta e_i(N) = T \cdot C^{\%}_{N,1} \cdot C_i(1) \qquad (1)$$

We now consider the percentage of cycles difference $C^{\%}_{x,y}$ between the N-processor case and the $(N-1)$-processor case, where $x = N$ and $y = N-1$, for the benchmark applications chosen. The $(N-1)$-processor case is taken as a reference. As an interesting result, we found that $C^{\%}_{7,6} = 16.67\%$ for all the three applications.

In order to understand whether this quantity would maintain itself constant also for configurations with a higher number of processors, the FFT benchmark was run on a SoC with up to 15 Leon3 processors. The results are shown in Figure 2.

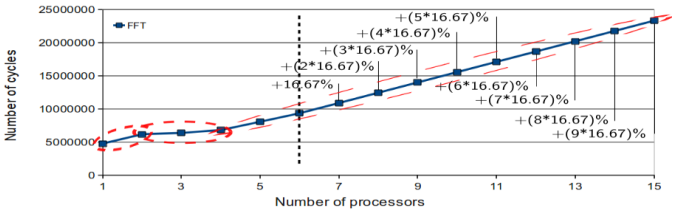

Fig. 2. Execution time e (in cycles) for FFT, $1 \leq N \leq 15$

It is evident that, taken the 6-processor configuration as a reference, it results that $C^{\%}_{N,6} = (N-6) \cdot 16.67$, for $N > 6$. This indicates a strong regularity in the increment of cycles for configurations with a number of processors $N > 6$. Note that these results are representative of the case where Round Robin is used for bus arbitration.

We identify 3 regions in Figure 2, highlighted by dashed circles. The circle in the middle shows an almost constant number of cycles for the SoC configurations with 2, 3 and 4 processors. In this case the bus controller allocates to each processor requesting the bus a time slot with the width of a bus transfer. In case of a 4-processor SoC, each processor will thus be granted a bus access every fourth bus transfer. It appears that this time frame is short enough to allow a processor to continue its execution without stalling until it regains control over the bus. That is why the performance penalty introduced is insignificant.

However, for configurations with more than 4 processors, this time frame gets too long and the processor has to stall for some time before it can access the bus again and continue its execution. This leads to the behavior highlighted by the circle on the right and to the 16.67% time penalty for any extra processor.

Finally, the circle on the left highlights an execution time difference between the configurations with 1 and 2 processors, which apparently contradicts the interpretation we have given above for the configurations with 2 to 4 processors. The explanation is found in the way the bus controller operates: if there is only one master requesting the bus, no arbitration is done and thus no arbitration penalty is introduced.

We have also measured the energy difference for the same benchmark applications on the 7 SoC configurations, and we have obtained a linear energy increase proportional to the processors number. The increase in energy consumption is a direct consequence of the increased execution time, which we have previously attributed to stall conditions for a

processor. Indirectly, we can thus say that the energy difference $E_i(N) - E_i(1)$, with $N > 1$, is also due to stall conditions. In particular, if we indicate as $\frac{E_{stall}}{cycle}$ the stall energy overhead per clock cycle, we can write the following Equation 2:

$$\frac{E_{stall}}{cycle} = \frac{E_i(N) - E_i(1)}{C_i(N) - C_i(1)} \quad (2)$$

We expect $\frac{E_{stall}}{cycle}$ to be a constant value. Therefore, we calculate the value of $\frac{E_{stall}}{cycle}$ by applying Equation 2 to every SoC configuration with $2 \leq N \leq 7$ and running our three reference benchmarks. The consistency of the results confirms the predictability of the stall energy consumption per clock cycle, which has an average value $\frac{\hat{E}_{stall}}{cycle} = 24.85\ pJ$ and a very low standard deviation $\sigma = 0.53$.

With this in mind, we want to analytically express the energy overhead $\Delta E_i(N)$ for the i-th application as a function of the number of processors N. This quantity was previously defined as the extra energy consumed by the i-th application because of bus contention, which is as a function of the number of processors N. The relation is shown in Equation 3.

$$\Delta E_i(N) = \frac{\hat{E}_{stall}}{cycle} \cdot \frac{\Delta e_i(N)}{T} \quad (3)$$

B. Factor 2: number of w/r accesses to memory

Besides the number of processors, we also investigate the impact that w/r memory accesses have on bus contention compared to instructions fetching. We remind the reader that our reference SoC is composed of processors without cache.

Using the FFT as a reference, we measure the instruction-level difference $C^{\%}_{7,1}$ in the execution cycles between the 7-processor and the single-processor SoC. We are comparing these two limit cases in order to make the differences more visible.

Clearly, for all instructions the value of $C^{\%}_{7,1}$ is positive. This makes sense, since we are using processors without cache, which implies that all instructions need to be fetched directly from memory. Contention can therefore happen for any instruction.

Being the average value for $C^{\%}_{7,1} = 129.32\%$ when measured on an application-level basis, all the load/store-related instructions show a value for $C^{\%}_{7,1}$ above average. Infact, in this case, not only fetching represents a possibility for contention, but also writing/reading data to/from memory. Nonetheless, there are also some non-store/load-related instructions exhibiting execution time above average. Some of them operate on data though – such as *add*, *sll* and *sub* – and a delay can be introduced if the data is not available when the instruction is issued.

Explicitly considering the load/store-related instructions as a second factor in modeling bus contention can thus improve the estimation accuracy for Δe. This can be done by separately considering the contribution of the load/store-related instructions and the non-store/load-related ones. If we denote the former and latter group of instructions by the symbol wr and $\overline{wr}$ respectively, we can reformulate Equation 1 as Equation 4:

$$\Delta e_i(N, wr) = T \cdot [C^{\%}_{wr,N,1} \cdot C_{i,wr}(1) + C^{\%}_{\overline{wr},N,1} \cdot C_{i,\overline{wr}}(1)] \quad (4)$$

IV. Predicting bus contention

In the present section, we use a case study to elaborate on how we can use and extend the results from Section III to predict the bus contention effects on an N-processor SoC running N arbitrary applications. We remind the reader that we evaluate such effects in terms of variation in the execution time $\Delta e_i(N, wr)$ and energy consumption $\Delta E_i(N, wr)$ for each i-th application.

As opposed to the bus contention characterization process, which is a one-time and time-consuming activity carried out by the IP provider, the bus contention prediction is very fast and is made by the SoC architect during design-space exploration. The speed comes in this case from the fact that the prediction relies on well-defined equations, such as Eq. 4 and 3, which are applied on a code profiled during native execution on the host and back-annotated with target-specific information.

A. Case study

In this case study we want to generalize Equation 4, so that it can still be applied to the case of an MPSoC with N processors running N different applications that can start and finish at any time.

We define an MPSoC execution time e_{mpsoc} as the range of time beginning when the first processor starts its execution at time t_s and ending when the last processor finishes its execution at time t_f. It is therefore true that $e_{mpsoc} = max(t_f) - min(t_s)$. In addition, the number of active processors n_a can vary in time, until it becomes 0 when the last application has completed its execution. Since bus contention can only be caused by an active processor trying to access the bus, we can rewrite $e_i(N, wr)$ and $E_i(N, wr)$ more precisely as $e_i(n_a, wr)$ and $E_i(n_a, wr)$. Keeping track of the variation of n_a over time is critical. Note also that, when $0 < n_a \leq N$, there will be $N - n_a$ processors in the idle state. This scenario is shown in Figure 3, where we consider 5 different applications running on 5 processors $[P_1, ..., P_5]$, and where each i-th application starts and finishes at different times $t_{s,i}$ and $t_{f,i}$. The processors in idle state are represented by dotted lines.

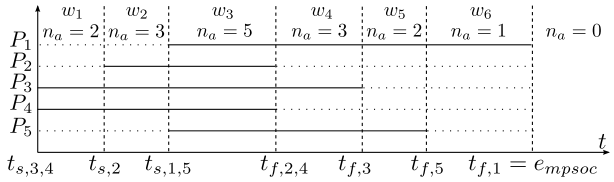

Fig. 3. MPSoC execution time for N processors and N *different* applications

Using Equation 4, we want to predict the value $e_i(n_a, wr)$ for any possible application. Being $C^{\%}_{\overline{wr},N,1}$ and $C^{\%}_{wr,N,1}$ the only unknown elements in the equation, we can now use the average $\hat{C}^{\%}_{\overline{wr},N,1}$ and $\hat{C}^{\%}_{wr,N,1}$ calculated on our characterization benchmarks, i.e. FFT, Fibonacci and Viterbi, and apply them to other applications as well. In Section V, we prove

the generality of our model and that the error introduced is minimal. Equation 4 generalized to any i-th application becomes therefore:

$$\Delta e_i(n_a, wr) = T \cdot [\hat{C}^{\%}_{wr,n_a,1} \cdot C_{i,wr}(1) + \hat{C}^{\%}_{\overline{wr},n_a,1} \cdot C_{i,\overline{wr}}(1)] \quad (5)$$

The energy consumption caused by the idle state can instead be expressed as follows:

$$E_{i,idle} = P_{idle} \cdot (e_{mpsoc} - e_i) \quad (6)$$

where P_{idle} is the power consumption of the processor when in idle state. The complete equations for execution time and energy consumption can thus be rewritten as $e_i(n_a, wr) = e_i(1, wr) + \Delta e_i(n_a, wr)$ and $E_i(n_a, wr) = E_i(1, wr) + \Delta E_i(n_a, wr) + E_{i,idle}$ respectively.

Note that the starting time $t_{s,i}$ for each i-th application is a user-defined parameter. The value of $t_{f,i}$ is instead calculated by the prediction algorithm, as explained below. The times t_s and t_f are the only points where the n_a value is updated. The reason is that there is no other time when this value can change. The result is a static decomposition of the whole applications set into multiple time windows w_k. In Figure 3, six time windows $[w_1, w_6]$ are identified.

An algorithm can be used for predicting bus contention, where the equations introduced before are used iteratively to predict both execution time and energy consumption for each i-th application. Note that, since the values are updated only in correspondence of the beginning/end of a time window, prediction is very fast and its speed is proportional to the number of applications, not to their length. In detail, the prediction algorithm consists of the following 4 steps, performed whenever a new window w_k starts:

1) The value of n_a is updated.

2) The ending time of a window is the minimum time between all the $t_{f,i}$ of the applications that are still running and the next closest starting time $t_{s,i}$. The value of $t_{f,i}$ is calculated as a function of the n_a value for the present window.

3) For each i-th application, the values of $\Delta e_i(n_a, wr)$ and $\Delta E_i(n_a, wr)$ are updated using Equations 5 and 3.

4) The process iterates until all applications have completed and $n_a = 0$. The value of $E_{i,idle}$ can then also be updated using Equation 6, since e_{mpsoc} is known.

V. Validating bus contention prediction accuracy

Since very accurate, gate level has been chosen to validate the accuracy in predicting bus contention for both execution time and energy consumption. The reference SoC architecture is the same used during the bus contention characterization phase. The 1-processor configuration is taken as a reference and measured values $e_i(1)$ and $E_i(1)$ are used.

We consider N different applications running on N processors. Each i-th application is mapped onto only one processor. We consider the case where N=5. The five applications are a Viterbi on P_1, a Dhrystone on P_2, the N-queen problem on P_3, a Fibonacci on P_4 and an FFT on P_5. Since we do not have any OS running, all applications start at the same time, but can finish at different times.

In Table I we show the results for measured versus predicted execution time and energy. The table shows the number of active processors n_a and the current state of each processor, i.e. active or idle. The predicted values for execution time and energy are calculated by using the algorithm described in Subsection IV-A. The prediction error is below 2%. In the energy prediction case, the idle energy is also considered in the error calculation.

TABLE I
Measured vs. predicted execution time $e_i[ms]$ and $E_i[\mu J]$.

n_a			P_1 Viterbi	P_2 Dhrystone	P_3 Queens	P_4 Fibonacci	P_5 FFT
5	measured	e_i	active	active	114.06	active	active
		E_i			156.87		
	predicted	e_i			115.25		
		E_i			158.50		
4	measured	e_i			idle		182.95
		E_i					257.11
	predicted	e_i					183.64
		E_i					258.01
3	measured	e_i				232.11	idle
		E_i				326.56	
	predicted	e_i				231.92	
		E_i				326.19	
2	measured	e_i		281.64		idle	
		E_i		407.42			
	predicted	e_i		281.61			
		E_i		407.26			
1	measured	e_i	327.02	idle			
		E_i	485.73				
	predicted	e_i	326.99				
		E_i	485.56				
Error[%]		e_i	-0.01	-0.01	+1.04	-0.08	+0.38
		E_i	-0.03	-0.07	+1.07	-0.18	+0.29

VI. Conclusions and future work

We have presented a high-level method for rapidly and accurately predicting bus contention effects on energy and performance in MPSoCs. Our approach consists of two steps: first, a one-time activity done at gate level, aimed at identifying and analytically expressing the main factors involved in bus contention; second, the development of a prediction algorithm to be used for design-space exploration and applied at the level of native code execution. Our prediction results are within 2% of gate-level estimation accuracy.

References

[1] AEROFLEX GAISLER. http://www.gaisler.com.

[2] L. N. Chakrapani, J. Gyllenhaal, W. mei W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran: An infrastructure for research in instruction-level parallelism. In *LNCS*. Springer, 2004.

[3] F. Ghenassia. *Transaction-Level Modeling with SystemC*. 2005.

[4] A. Sahu, M. Balakrishnan, and P. Panda. A generic platform for estimation of multi-threaded program performance on heterogeneous multiprocessors. In *DATE 2009*, Nice, France.

[5] G. Schirner, A. Gerstlauer, and R. Dömer. Fast and accurate processor models for efficient mpsoc design. *ACM Trans. Des. Autom. Electron. Syst.*, 15(2):1–26, 2010.

[6] M. Streubuhr, J. Gladigau, C. Haubelt, and J. Teich. Efficient approximately-timed performance modeling for architectural exploration of mpsocs. In *Forum on Specification Design Languages (FDL), 2009*, Sophia Antipolis, France, 2009.