Energy-aware MPSoC with Space-sharing for Real-time Applications

Stefan Aust, Harald Richter
Institute of Computer Science
Clausthal University of Technology
Clausthal-Zellerfeld, Germany
stefan.aust|harald.richter@tu-clausthal.de

Abstract—**Energy-awareness is an important design criterion for many mobile real-time applications such as phones, handhelds or cars. Normally, high-computing power excludes low electrical power consumption. However, with the two methods of space-sharing and adaptive clocking that are proposed in this paper, both can be reconciled. Space-sharing is an alternative design methodology for embedded systems that outperforms time-sharing with respect to simplicity and power dissipation. Adaptive clocking is an efficient means to further reduce power. The results were achieved by a set of measurements made with a single-chip multiprocessor (MPSoC) that implements space-sharing in a FPGA and by a model MPSoC that implements additionally adaptive clocking.**

Keywords-low power; multiprocessor; FPGA; MPSoC

I. INTRODUCTION

A MPSoC is a multiprocessor on a single silicon chip that may contain up to ten or even hundreds of processor-memory modules (PMMs). In case of up to ten PMMs, we speak also of a multi-core processor if the memory modules are lumped together into a single first-level cache that is shared between all cores. The term many core architecture is used for hundreds of cores and more on the same chip that are connected together by appropriate means. Fully custom-designed chips have traditionally implemented MPSoCs, multi- and many-core architectures. During the last few years the capabilities of Field Programmable Gate Arrays (FPGAs) have increased significantly and thus FPGAs allow the implementation of these architectures as an alternative, although with less powerful PMMs today.

Furthermore, there are many real-time applications that demand more and more computing power but have limited energy resources as in mobile devices or controller units in cars, for example. Inefficient usage of the available electrical energy reduces the usability of mobile devices and reduces the operational range of cars and has therefore to be avoided. Energy waste also causes thermal dissipation of heat, and as a consequence requires additional energy for cooling sensitive components. Finally, inefficient usage of energy limits the life expectancy of electronic devices because of aging processes in the semiconductor materials. The pn-junctions in the transistors deteriorate with increasing heat exposure. Because of these facts, energy-awareness in computers systems is an important design criterion. In addition to that, high computing power and low energy consumption normally exclude one another, which is bad for energy-aware embedded systems that require high performance CPUs.

In this paper, we suggest for energy-critical real-time applications the usage of MPSoCs on a single FPGA. This is possible by applying two methods that are presented by us: space-sharing and individual core clocking on the MPSoC. Space-sharing instead of time-sharing eliminates the problem of finding and guaranteeing a proper schedule of tasks that fulfills a given time constraint. As we will see, it allows also for a better energy management in embedded systems. This means that besides easier handling of complex real-time applications, space-sharing is able to save energy. In addition to that we propose an MPSoC with space-sharing and individual clocking of cores in this paper that optimizes the consumed energy by reducing the dynamic fraction of the chip's power dissipation to a minimum.

The paper is organized as follows: In Section 2, the architecture of the MPSoC is presented, together with the methods of space-sharing and individual core clocking. In Section 3, measurements of the energy consumption of MPSoCs on FPGAs are presented and discussed that do not use space-sharing or individual core clocking. In Section 4, a model MPSoC is investigated that uses both methods. Section 5 draws a conclusion of the performed work and gives an outlook to future work.

II. STATE-OF-THE-ART

Real-time applications demand the execution of a list of tasks within a given time limit or at a given time point. A common method to evaluate the soft or hard real-time capabilities of the underlying embedded system that executes that task list, is the analysis of how long each task list will need to be executed in the worst case (=worst-case execution-time analysis, WCET [1]).

However, the WCET analysis grows exponentially in CPU time with the number of tasks because it is a NP-complete problem. Many task scheduling strategies have been created to solve this by heuristic approaches, such as priority scheduling, earliest deadline first or round robin [2]. Also, many analysis tools as well have been developed and commercialized to simplify the WCET analysis for the designer of embedded systems [3,4].

With space-sharing, there is no need for scheduling tasks and therefore also no need for WCET tools as well. Space-sharing was introduced first by the authors in [5].

 ISBN: 978-1-61208-172-4

A. Space-Sharing

In time-sharing, a single processor is divided in its computing time among multiple tasks. Each task gets one time slice of the processor in a round-robin manner, and after a specific interval all tasks have got some of the computing power of the processor. After that, the whole procedure is repeated periodically. The more the number of tasks and the harder the required time constraints are, the more difficult it is for the processor to get all the tasks done before their deadlines come up. Thus, time-sharing requires that the cumulative CPU needs of all tasks do not exceed the processor's computing power and that a schedule can be found such that all time constraints are met.

In space-sharing, every task is allocated statically to an own processor. No scheduling is needed because tasks are never competing for the same computing resource. This means that the number of processors exactly matches the number of executable tasks (Figure 1).

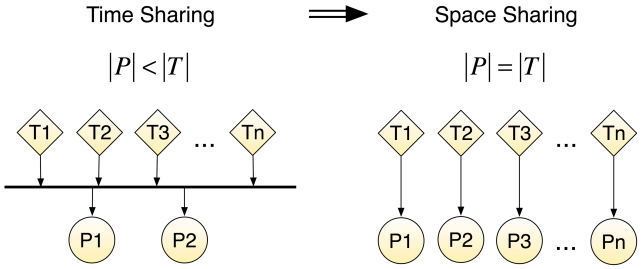

Figure 1. Time-sharing and space-sharing.

Space-sharing requires of course that so many processors are available as tasks are defined by the application programmer and that practical methods exist for allocating tasks to processors and for inter-task communication as well. On the other hand, complex real-time applications can comprise hundreds of tasks and thousands of inter-task communications. However, technological advancements are such that this can be handled by state-of-the art FPGAs. A Virtex-7 FPGA from Xilinx for instance is able to host hundreds of PMMs, provided that they have a low clock frequency compared to full-custom processors and only small on-chip memory. However, for many real-time applications this is sufficient because they have no need for GHz and GBytes in their feed forward and feed back control algorithms. This means that MPSoCs on a FPGA are a convenient way to implement space-sharing in real-time applications.

B. The MPSoC Architecture

Figure 2 shows the MPSoC we have implemented on various FPGAs from Xilinx. The MPSoC consists of n PMMs; where n is configurable, an interconnection network for inter-task communication, an on-chip shared memory and optionally of three hardware ports called bridges to peripheral I/O devices, network interfaces and external memory.

1) Soft Core Processors: The processors used in our MPSoC are so-called soft-core CPUs. This means that each processor exists only as a logic in a hardware description

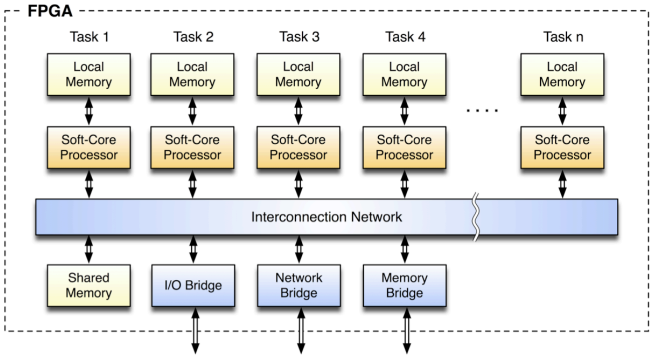

Figure 2. MPSoC architecture on a FPGA.

language, such as VHDL or Verilog that emulates a real processor. We have used a processor description from Xilinx of type MicroBlaze [6]. MicroBlaze emulates an in-order, non-superscalar 32 bit RISC CPU with a clock rate of up to 100 MHz. Because MicroBlaze is a simple RISC architecture it allows one to predict the needed CPU cycles for a given real-time task more easily than in the case of a fully featured CPU. Software development for MicroBlaze can be made in C or C++ because of compilers in the Xilinx EDK. Please note that a virtual processor that is emulated by a small area of the FPGA chip executes every compiled application code.

2) Local Memory: The local memories in our MPSoC are split into instruction and data stores according to Harvard architecture. The maximum memory sizes and increments depend on the FPGA type. For the Xilinx XC4VFX100 FPGA for example, local memories can be incremented in steps from 4 KB to 256 KB up to a maximum of 6768 KB for the whole chip.

3) Non-Blocking Real-Time Interconnection Network: An on-chip interconnection network establishes communication links between processors, i.e., between task pairs. The links carry messages in a point-to-point manner but it is also possible to implement multicast and broadcast in the 2x2 switches out of which the network is constructed. The network operates asynchronously to the PMM clocks and in circuit switching mode. Asynchronous communication requires a message FIFO at each network input to decouple message creating in a PMM from message transfer in the network. Circuit switching means that a path through the network is established for every point-to-point connection as long as the communication persists.

The most important feature of the interconnection network in our MPSoC is however that it is non-blocking by construction because it consists of multiple stages of 2x2 switches in Benes topology [7]. This non-blocking feature is mandatory for real-time. If the network would block, no upper limit could be guaranteed for the latency of message delivery from one task to another.

4) Software Development for Space-Sharing: The software development process for space-sharing is identical to that of time-sharing. The application is implemented in both cases by the code, i.e., the same set of tasks and the same inter-task communication. Only the way of mapping

 ISBN: 978-1-61208-172-4

the tasks and the way how inter-task communication is accomplished is different.

In time-sharing, all tasks are mapped onto time slots and communicate via system calls that a real-time operating system may provide. In space-sharing, all tasks are mapped onto physical areas on a silicon chip, and inter-task communication is implemented by an interconnection network. This is fully transparent to the application because the programmer can use the same API for inter-task communication. This allows for direct portability of code between conventional time-sharing and new space-sharing.

Once the application has been partitioned into tasks, the MPSoC can be built either manually or automatically by configuring in the VHDL or Verilog description of the MPSoC with so many PMMs as needed, plus an interconnection network to couple them. Furthermore, space-sharing allows the adapting of every local memory in its size exactly to the needs of the task. It is also possible to add a coprocessor to a PMM that executes parts of the task as a hardware accelerator. The processor description has to be modified and extended therefore. Finally, space-sharing allows for task isolation and memory protection by construction because each local memory in the MPSoC can be accessed only by its own processor. In space-sharing, inter-task isolation is a feature that requires a memory management unit and at least a basic operation system.

5) The number of PMMs on the FPGA: The number of achievable PMMs on the chip varies with the used FPGA type and the amount of external memory that is available on the FPGA board. In order to explore the achievable amount of PMMs per FPGA, we tested various types of FPGAs as listed in table I. On the Virtex-4 FPGA, which provides less than 5 percent of the capability of the recent Virtex-7 XC7V2000T FPGA, we could accomodate a maximum of 34 PMMs including 16 KB memory each.

TABLE I. LIST OF TESTED FPGAS

FPGA	**FPGA**	
	Chip	***Board***
Spartan-3	Xilinx XC3S1000	Digilent Starter-Kit
Virtex-4	Xilinx XC4VFX100	PLDA XpressFX
Virtex-5	Xilinx XC5VLX50T	Xilinx ML505

Independent from the used FPGA type, it holds that the summed computing and memory requirements cannot exceed the chip resources. Although only one FPGA and some external memory are involved, the cumulative processing power of all PMMs is nevertheless considerably high if the on-chip parallel processing is effectively organized. In addition to that, task-specific algorithms can be outsourced to a PMM coprocessor where they are executed in hardware, i.e., with maximum speed.

6) Energy awareness: The execution of every application needs consumption of electrical power. With space-sharing, only that chip area on a FPGA is occupied that is really needed by the application. It holds that chip areas that are not involved consume much less power than those whose transistors are switching between low and high states. Furthermore, the sizes of the local memories in space-sharing matches the sizes of the programmed code and data structures because no surplus is generated by VHDL or Verilog chip synthesis tools. As a consequence, power dissipation is as low as the underlying FPGA chip technoloy allows and as low as the application demands. This is in contrast to time-sharing on a classic embedded system that has to use standard microcontrollers and standard memory chips with given transistor count and therefore power dissipation.

Please note that the energy consumption of our MPSoC is low because of space-sharing but it does not yet reach the absolute minimum. Only if individual PMM clocking is added as a new feature to space-sharing then the key is found for combining real-time and high CPU power with extremely low energy consumption.

III. INDIVIDUAL CLOCKING OF PMMS

Individual clocking of PMMs means that every PMM gets that clock speed that it needs to fulfill the time constraints of the task it executes. Individual clocking saves energy because the faster the processor operates the more dynamic power it consumes.

Space-sharing allows the clocking of every PMM at its best rate and even to vary it dynamically over time, if needed. Individual clocking was implemented by us by means of a clock rate controller in every PMM and by extensions to the application code that can adaptively adjust the clock rate to the various phases a task can have.

A. Clock Rate Controller

Figure 3 shows the set-up of the clock rate controller. It consists of a master oscillator and a clock generator for all PMMs and an individual clock divider for each PMM. The clock divider is set by the processor program and controls thereby the processor clock rate.

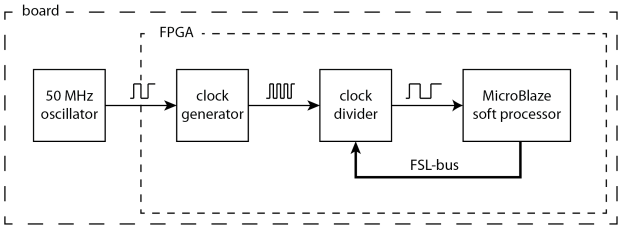

Figure 3. Set-up of the clock rate controller.

To benefit from the clock rate controller, the application programmer of the MPSoC must explicitly set the clock frequency for every task. If the task has several phases with different time constraints then he may set the rate in a fine-grain manner. A first step to accomplish this is to divide each task in clock segments according to Figure 4. After that, he can set the rate for every segment by two methods: either he uses a system call which becomes valid during runtime, or he uses a compiler directive which is evaluated during compile time. Both methods need the introduction of time constraints

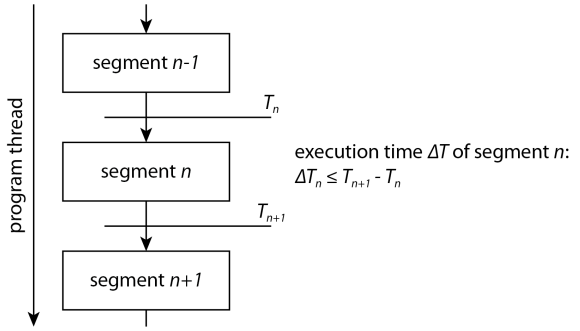

Figure 4. Clock rate adaptation by task segmenting.

that extend the real-time code. A paragon of real-time code extension by time constraints is given in [8].

B. System call for clock rate adaptation

Figure 5 shows an example code where the predefined routine *set_clock_rate* is responsible for the clock rate. The routine *check_new_data* periodically tests a sensor on the arrival of new data. This is accomplished at a slow sampling rate of 2 MHz. If new data is present, then the routine *calc_value* is executed at a rate of 40 MHz, which quickly processes the input. By this method, the processor clock varies dynamically during the program run and thus saves energy. The disadvantage of this method is that the programmer must define clock rates in the program. These clocks depend furthermore on the used processor type because more powerful soft processors can do more jobs at a lower clock rate. This means that the application code has to take into account the concrete MPSoC on which a task is executed and is thus less portable.

```
set_clock_rate(2);                          => clock rate = 2 MHz
while(new_data = 0){
   new_data = check_new_data(&data);
};
set_clock_rate(40);                         => clock rate = 40 MHz
value = calc_value(&data)
set_clock_rate(2);                          => clock rate = 2 MHz
```

Figure 5. System call for setting processor clock rate.

C. Compiler directive for clock rate adaptation

The second and better method for clock rate adaptation is to add deadlines to the code segments according to Figure 4. Each deadline Ti defines, until which each segment, i.e., task phase, must be completed. Compiler directives accomplish the definition of deadlines in the code. The directives are formatted as a comment to avoid language extensions or system calls but they are not treated as comments. Instead, a compiler preprocessor reads all meaningful comments and evaluates them. In addition, the compiler knows for what concrete MPSoC it compiles the code, and its preprocessor can therefore count the number of clock cycles needed to execute a task segment in that code. With that information, the preprocessor can set the clock rate MPSoC-dependent, and the programmer does not have to care about that.

```
/*! initiate */
while(new_data = 0){
   new_data = check_new_data(&data);
};
/*! event ≤ initiate + 20 */              => 40 clock cycles / 20µs = 2 MHz
value = calc_value(&data)
/*! terminate ≤ event + 30 */             => 1200 instructions / 30µs = 40 MHz
```

Figure 6. Compiler directive for setting processor clock rate.

Figure 6 shows the same example application of Figure 5, but with additional deadlines of 20 and 30 µs for the segments *event* and *terminate* that have to be met after the *initiate* segment. The resulting preprocessor evaluation is 2 MHz as a clock rate for the *event* and 40 MHz as clock rate for the *terminate* segment, which is the same as in method 1, but that result was achieved more conveniently.

IV. Measurements of Power Consumption

We conducted several series of measurements on MPSoCs with and without individual clocking to obtain concrete Figures of the power dissipation of every MPSoC component. All measurements were made on the boards of Section II.

According to [9], total power consumption P of a chip consists of static power P_{stat} and dynamic power P_{dyn}:

$$P = P_{stat} + P_{dyn} \quad (1)$$

Leakage currents inside the semiconductor material cause static power consumption. There are two ways for leakage currents: the sub-threshold leakage, which is an inversion current across the device, and the gate-oxide leakage, which is a tunneling current through oxide insulation of a transistor gate. Both are technology-dependent and cannot be altered by the chip user. Dynamic power in contrast arises from switch activities of transistors. Equation (2) determines the dynamic power consumption [10] as a function of supply voltage V, clock frequency f and chip capacitance C.

$$P_{dynamic} = \sum CV^2 f \quad (2)$$

In the following, measurements are presented for the case of no individual clocking. In Section 5, that technique is added.

A. Static and Dynamic Dissipation

In Figure 7, the static and dynamic dissipation is shown that we have measured for the Xilinx Virtex-4 FPGA. The diagram shows that dynamic power grows nearly linearly with the number of processors. Deviations from a straight line are caused by the place and route function of the used chip synthesis tool, which was XST from Xilinx. All processors have been clocked with 100 MHz first, while the total power consumption of the FPGA was measured. After this, the clock was disconnected to measure static power consumption only. Dynamic power was obtained as the difference between both.

 ISBN: 978-1-61208-172-4

Figure 7 shows that in MPSoCs of 10 PMMs and more the dynamic power dominates in the case of Virtex-4 on the PLDA board. Not shown in Figure 7 but measured by us is that in numbers Spartan-3 needs 268 mW of dynamic dissipation, Virtex-4 needs 120 mW, and Virtex-5 requires 53 mW for every added processor. The low power of Virtex-5 accounts mainly from the 65 nm process, while Spartan-3 and Virtex-4 share the older 90 nm process technology.

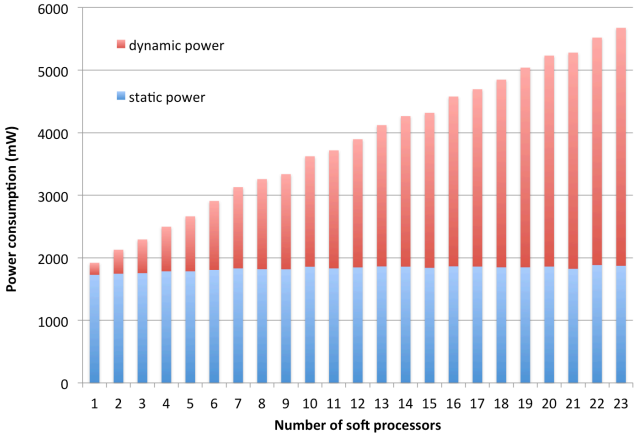

Figure 7. Static and dynamic power consumption versus the number of processors for Virtex-4.

B. Influence of Processor Clock Rates

A second measurement series was conducted to get the dynamic dissipation versus the processor clock rates. The number of processors was parameterized. In Figure 8, the results of several MPSoCs with 1 to 6 processors in a Spartan-3 FPGA are shown. The results were already predicted in theory by equation 2, but concrete values were not known.

C. Influence of Local Memory

Figure 9 shows the influence of local memory for a Virtex-4 FPGA at a clock rate of 100 MHz. The number of processors was again parameterized. As one can see in addition in Figure 9: if a PMM has no memory, about 100 mW remains that is needed by processor logic only.

D. Influence of Interconnection Network

As all PMMs, also the interconnection network has its own clock. In Figure 10, the influence of this clock rate on the dynamic power consumption of the FPGA is illustrated. The diagram shows that dynamic power dissipation increases linearly with a slope of about 0.85 mW per MHz for the tested Virtex-4 FPGA. Furthermore, we found out that the dynamic power consumption of the network mainly depends on the FIFOs at its inputs, not on the switching matrix itself. The FIFO in turn depends on the number of messages that have to be stored, their size and how big the differences between processor clock rates and network clock rates can become. Big differences need deep FIFOs to balance-out message sending and transporting, at least for a while.

E. Power Consumption Parameters

From the conducted measurements, we got the following numbers for parameters of an MPSoC that is implemented on a Virtex-4 for dynamic power consumption:

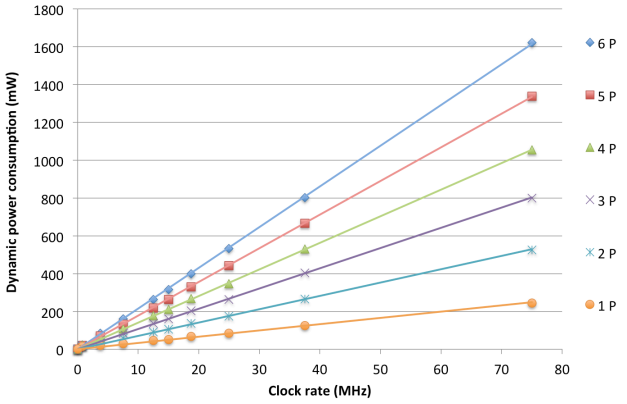

Figure 8. Dynamic power consumption versus processor clock rates for Spartan-3. The number of processors is parameterized.

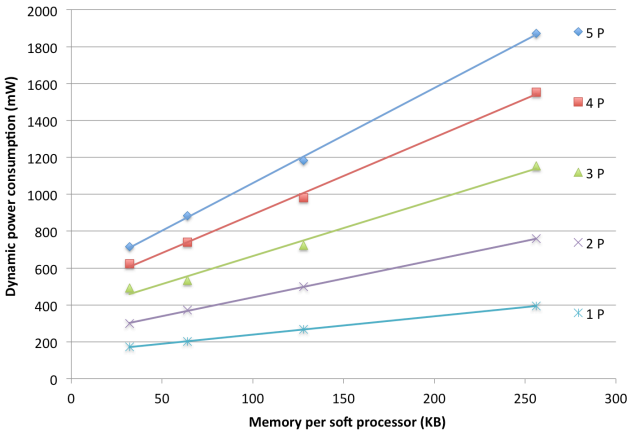

Figure 9. Dynamic power consumption versus size of local memory for Virtex-4. The number of processors is parameterized.

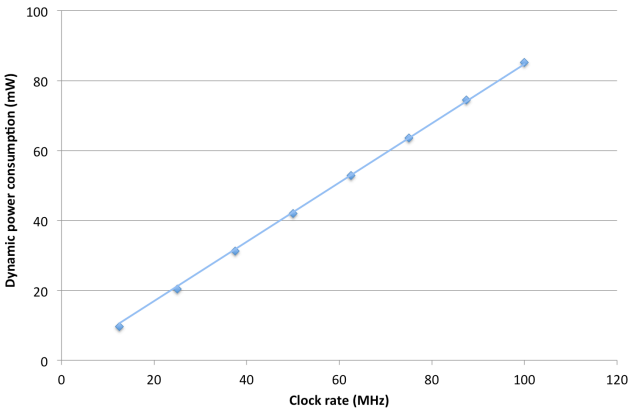

Figure 10. Dynamic power consumption of the interconnection-network versus its clock-rate for Virtex-4.

$$P_{processor} \approx 1 \frac{mW}{MHz} \quad (3)$$

$$P_{memory} \approx 0.01 \frac{mW}{KB \cdot MHz} \quad (4)$$

$$P_{network} \approx 0.85 \frac{mW}{MHz} \quad (5)$$

We used those parameters to define a model MPSoC that comprises 8 PMMs (Figure 11) in order to test the effectiveness of individual clocking.

V. Effectiveness of Individual Clocking

The defined model MPSoC has individual clock rates and local memory sizes that reflect the requirements of an example application. The accumulative dynamic power consumption of this configuration is 529 mW without phase-adaptive clocking. However, we did not employ that for the model MPSoC in order to have a better comparison to a MPSoC of same PMM number and architecture but with phase-adaptive clocking. It turned out that such a MPSoC would consume 1.9 W with a chip-wide clock rate of 100 MHz and with fixed memory sizes of 64 KB for all PMMs. Out of this, a factor of 3.6 less dissipation is computed. According to that it can be stated that individual clocking of PMMs is an efficient method for energy saving, already without phase-adaptivity. With phase-adaptivity, one can obtain the least possible power dissipation.

A. Set-Up of Model MPSoC

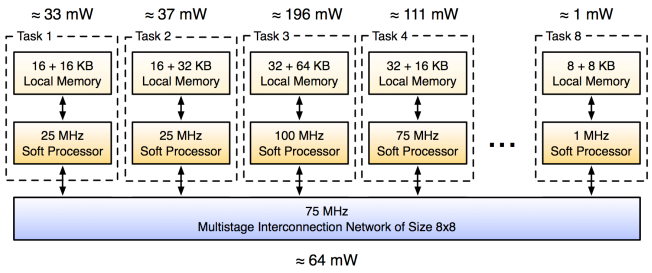

Figure 11. Set-Up of Model MPSoC.

VI. Conclusion and Future Work

In this paper, the concepts of space-sharing and adaptive clocking have been explained. Space-sharing eliminates the need to find a proper schedule for a list of tasks that have to be executed within a given time interval or at a given time point. It also eliminates the analysis of worst-case execution time of tasks in embedded systems, and the tools created here for. Additionally, space-sharing allows for a better electrical power management of real-time systems, for simpler task isolation and memory protection and for minimal power dissipation if combined with adaptive clocking. A convenient implementation of space-sharing is accomplished by a single-chip multiprocessor (MPSoC) in which the processor-memory modules are clocked individually due to the task phases and where the sizes of the local memories and the processor performance exactly match the application requirements. Furthermore, we conducted series of measurements to obtain concrete Figures for the power dissipation of the proposed MPSoC on Spartan-3, Virtex-4 and Virtex-5 FPGAs from Xilinx.

Future work will create a tool for the automatic FPGA configuration out of a XML-based description of the task set and inter-task communication of any given real-time application, such that the application can be ported into space-sharing and thus executed by our MPSoC without code modification to obtain minimal chip area consumption and thus very low power dissipation.

References

[1] P. Marwedel, “Embedded System Design,” 2nd edition, Dordrecht; Heidelberg, Springer, 2011.

[2] G. C. Buttazzo, “Hard Real-Time Computing Systems. Predictable Scheduling, Algorithms and Applications,” Boston; Dordrecht; London, Kluwer Academic Publishers, 1997.

[3] Rapita Systems Ltd., www.rapitasystems.com (last checked: 11-06-20).

[4] Symtavision GmbH, www.symtavision.com (last checked: 11-06-20).

[5] S. Aust and H. Richter, “Space Division of Processing Power For Feed Forward and Feed Back Control in Complex Production and Packaging Machinery,” Proc. World Automation Congress (WAC 2010), Kobe, Japan, Sept. 2010, pp. 1-6.

[6] Xilinx, “MicroBlaze Processor Reference Guide,” October 2009.

[7] S. Aust and H. Richter, "Real-time Processor Interconnection Network for FPGA-based Multiprocessor System-on-Chip (MPSoC)," The 4th International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP 2010), Florence, Italy, Oct. 2010, pp. 47-52.

[8] A. Leung, K. V. Palem, and A. Pnueli, “TimeC: A Time Constraint Language for ILP Processor Compilation,” Constraints, vol. 7, no. 2, 2002, pp. 75-115, doi: 10.1023/a:1015131814255.

[9] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, K. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, “Leakage Current: Moore‘s Law Meets Static Power,” IEEE Computer, vol. 36, issue 12, Dec 2003, pp. 68-75.

[10] L. Shang, A. S. Kaviani, and K. Bathala, “Dynamic Power Consumption in Virtex™-II FPGA Family,” Proc. of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays (FPGA '02), Monterey, CA, Feb. 2002, pp. 157-164.

 ISBN: 978-1-61208-172-4