Scheduling of Distributed Algorithms for Low Power Embedded Systems

Stanislaw Deniziak
Department of Information Systems
Kielce University of Technology
Kielce, Poland

Albert Dzitkowski
Department of Information Systems
Kielce University of Technology
Kielce, Poland

***Abstract*—Recently, the advent of embedded multicore processors has created interesting technologies for power management. Systems consisting of low-power and high-efficient cores create new possibilities for the optimization of power consumption. However, new design methods, dedicated to these technologies should be developed. In this paper we present a method of static task scheduling for low-power real-time embedded systems. We assume that the system is specified as a distributed algorithm, then it is implemented using multi-core embedded processor with low-power processing capabilities. We propose a new scheduling method to create the optimal or suboptimal schedule. The goal of optimization is to minimize the power consumption while all time constraints will be satisfied or the quality of service will be as high as possible. We present experimental results, obtained for sample systems, showing advantages of our method.**

Keywords*—Embedded system; distributed algorithm; task scheduling; big.LITTLE; low power system

I. INTRODUCTION

Embedded systems are dedicated computer-based systems that are highly optimized for a given application. Besides the cost and performance, power consumption is one of the most important issue considered in the optimization of embedded systems. Design of energy-efficient embedded systems is important especially for battery-operated devices. Although the minimization of power consumption is always important, because it reduces the cost of running and cooling the system. It was observed that power demands are increasing rapidly, yet battery capacity cannot keep up [1].

Embedded systems are usually real-time systems, i.e. for some tasks time constraints are defined. Therefore, power optimization should take intconsideration that all time requirements should be met. In general, higher performance requires more power, hence, the optimization of embedded system should consider the trade-off between power, performance and cost. Performance of the system may be increased by applying a distributed architecture. The function of the system is specified as a set of tasks, then during the co-design process, the optimal architecture is searched [2]. Distributed architecture may consist of different processors, dedicated hardware modules, memories, buses and other components. Recently, the advent of embedded multicore processors has created an interesting alternative to dedicated architectures. First, the co-design process may be reduced to task scheduling for multiprocessors systems. Second, advanced technologies for power management, like DVFS (Dynamic Voltage and Frequency Scaling) or ARM big.LITTLE [3], create new possibilities for designing of low-power embedded systems.

Although there are a lot of synthesis methods for low-power embedded systems [4], the problem of optimal mapping of a distributed specification onto the multicore processor is rather a variant of the resource constrained project scheduling (RCPSP) [5] one, than the co-synthesis. Since the RCPSP is NP-complete, only heuristic approach may be applied to real-life systems. According to the best of our knowledge there is no synthesis methods taking into consideration ARM big.LITTLE architecture as a target platform for real-time embedded systems. Only, some work considering run-time scheduling were done [6].

The most of RCPSP approaches are dedicated to the task graph specification of a system. But in many cases, especially in case of embedded software, more general distributed models [7] would be more convenient. It was occurred that the function of a real-time distributed system may be efficiently specified as a distributed echo algorithm [8]. Moreover, such specification may also be statically scheduled [9].

In this paper we present the novel method for synthesis of the power-aware scheduler for real-time embedded systems. We assume that the function of the system is specified as a distributed echo algorithm [10] that should be executed by the multicore processor supporting the ARM big.LITTLE technology. The goal of the static scheduling is the reduction of power consumption by moving some tasks to low-power cores (LPCs), while critical tasks are assigned to high-performance cores (HPCs), to satisfy all time constraints. The proposed method is dedicated to high performance embedded computing systems.

II. RELATED WORK

The problem of design of low-power embedded systems has attracted researchers for many years. One direction of these research is finding the low-power architecture by optimizing the allocation of resources and task assignment according to the power consumption (e.g. COSYN-LP [11], SLOPES [12], LOPOCOS [13]). The overview of some power aware codesign methods is presented in [4]. But all above methods create the dedicated hardware/software architecture and cannot

be applied to multicore processors.

Another direction of research concerning the design for low-power is to develop methodologies that takes into consideration dynamic reduction of the power consumption during runtime. AVR (Average Rate heuristic) [14] is a task scheduling method for variable speed processor. Dynamic Power Management [15] tries to assign optimal power saving states. Other methods reduces power consumption by efficiently using voltage scale processors [16]. All above methods are based on power-aware scheduling called YDS. Above methods schedules dynamically a set of tasks by selecting the proper speed for each task. ARM big.LITTLE uses only 2 predefined speeds, thus it is rather not possible to adopt above methods to this technology.

There are a lot of scheduling methods for real-time embedded systems. Earliest Deadline First (EDF) [17] or Least Laxity First (LLF) [18] is ones of the most efficient dynamic scheduling methods. But above methods are dedicated to homogeneous architectures (SMP). Discussion concerning the problems of task scheduling in real-time systems is presented in [19]. Most of them optimize schedule length.

Embedded software consists of the given set of tasks. Usually it is possible to estimate the task parameters like execution time, power consumption, memory requirements. Most static scheduling methods are based on specification represented as task graph [20]. But in many cases it is difficult to specify function as a task graph, some other models e.g. distributed algorithms [7] are more suitable. It seems that the echo algorithm [10] would be attractive for this purpose.

According to our best knowledge there is no scheduling method for real-time systems specified as distributed echo algorithm, as well as the static scheduling method optimizing energy consumption in embedded systems based on the big.LITTLE platform.

III. PRELIMINARIES

We assume that the target embedded system is based on multi-core processor with LPCs and HPCs. LPC requires less power to execute tasks but execution times are longer. HPC executes tasks faster but consumes more energy. We consider soft real-time systems, i.e. all tasks should be executed before the specified deadline. But it is acceptable to slightly exceed the deadline. In this case the quality of service (QoS) decreases with increasing delay. The goal of optimization is to find schedule for which the power consumption is minimal while time constraints are satisfied or QoS is maximal. Since we consider shared memory architecture, transmissions between tasks may be neglected

A. Echo algorithms

Echo algorithms [18] are a class of wave algorithms [7] used for describing distributed computations. The system is specified as a set of tasks communicating by message passing. One task is an initiator, which starts all computations. After finishing its execution the initiator sends explorer messages to all neighbours. After receiving the first explorer message the task stores source node as an activator and after execution sends explorer message to all neighbour nodes except the activator. After finishing execution of all tasks, all tasks which were not activators execute again to compute echo message which is sent to their activators only. Each task, after receiving echo messages from all activated task, executes again and sends echo message to its activator. Finally, the initiator should receive all echo messages and then it computes the final result.

Fig. 1 presents sample echo algorithm consisting of 10 processes. Assume that task 0 is the initiator. Therefore this task will be executed first. Then, tasks 1, 5 and 4 should be executed. It should be noted that the order of activation of tasks depends on times of execution of the following tasks, e.g. task 6 may be activated by task 5 but also it may be activated by task 7, in case when tasks 1, 2, 3, and 7 will finish their execution before finishing task 5. Thus, the scheduling on heterogeneous processors is complex even when the execution times of all tasks are known e.g. are estimated or when the worst case execution times are assumed.

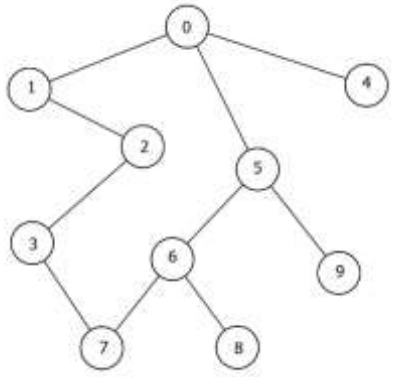

Fig. 1. Sample distributed algorithm

B. Functional Specification of Distributed Systems

We assume that the system is specified as a collection of sequential processes coordinating their activities by sending messages. Specification is represented by a graph $G = \{V, E\}$, where V is a set of nodes corresponding to the processes and E is a set of edges. Edges exist only between nodes corresponding to communicating processes. Tasks are activated when required set of events will appear. As a result, the task may generate other events. External input events will be called requests (Q), external output events are responses (O) and internal events correspond to messages (M). The function of the system is specified as finite sequences of activation of processes. There is a finite set of all possible events $\Lambda = Q \cup O \cup M = \{\lambda_i : i = 1,\ldots,r\}$. System activity is defined as the following function:

$$\Phi: V \times C \rightarrow \Omega \times \Pi \times 2^{\Lambda} \quad (1)$$

where C is an event expression (logical expression consisting of logical operators and Boolean variables representing events), $\Omega=[\omega_L, \omega_H]$ are workloads of the activated process defined for LPC and HPC respectively, and $\Pi=[\pi_L, \pi_H]$ defines power consumption.

Using function Φ it is possible to specify various classes of distributed algorithms. The algorithm from Fig.1 may be

described using 20 actions. Assume that estimation of task workloads and energy consumption are given (Tab. I). Thus, actions will be the following:

A_0: $\Phi(v_0,\{q_0\}) \rightarrow ([3,2], [9,20], \{m1_1,m2_5,m3_4\})$

A_1: $\Phi(v_1,\{m1_1\}) \rightarrow ([13,8],[34,81],\{x_1,m4_2\})$ |
$\Phi(v_1,\{m10_1\}) \rightarrow ([13,8],[34,81],\{x_2,m5_0\})$

A_2: $\Phi(v_5,\{m2_5\}) \rightarrow ([6,4],[16,40],\{x_3,m6_6,m7_9\})$ |
$\Phi(v_5,\{m15_5\}) \rightarrow ([6,4],[16,40],\{x_4,m8_0,m7_9\}$

A_3: $\Phi(v_2,\{m4_2\}) \rightarrow ([10,5],[26,52],\{x_5,m9_3\})$|
$\Phi(v_2,\{m12_2\}) \rightarrow ([10,5],[26,52],\{x_6,m10_1\})$

A_4: $\Phi(v_3,\{m9_3\}) \rightarrow ([7,4],[18,40],\{x_7,m11_7\})$ |
$\Phi(v_3,\{m17_3\}) \rightarrow ([7,4],[18,40],\{x_8,m12_2\})$

A_5: $\Phi(v_6,\{m6_6\}) \rightarrow ([5,3],[13,29],\{x_9,m13_7,m14_8\})$ |
$\Phi(v_6,\{m16_6\}) \rightarrow ([5,3],[13,29],\{x_{10},m15_5,m14_8\}$

A_6:$\Phi(v_7,\{m11_7\}) \rightarrow ([9,5],[23,50],\{x_{11},m16_6\})$ |
$\Phi(v_7,\{m13_7\}) \rightarrow ([9,5],[23,50],\{x_{12},m17_3\}$

A_7: $\Phi(v_4,\{m3_4\}) \rightarrow ([11,6],[28,58],\{x_{13}\})$

A_8: $\Phi(v_9,\{m7_9\}) \rightarrow ([12,7],[30,67],\{x_{14}\})$

A_9: $\Phi(v_8,\{m14_8\}) \rightarrow ([4,2],[11,22],\{x_{15}\})$

A_{10}:$\Phi(v_0,\{(m5_0|m18_0)\&(m8_0|m19_0)\&m20_0)\}) \rightarrow ([9,5],[23,52], \{r_1\})$

A_{11}:$\Phi(v_1,\{x_1\&m21_1\}) \rightarrow ([17,9],[42,93],\{m18_0\})$ |
$\Phi(v_1,\{x_2\&m1_1\}) \rightarrow ([17,9],[42,93],\{m22_2\})$

A_{12}:$\Phi(v_2,\{x_5\&m23_2\}) \rightarrow ([7,4],[17,40],\{m21_1\})$ |
$\Phi(v_2,\{x_6\&m22_2\}) \rightarrow ([7,4],[17,40],\{m24_3\})$

A_{13}:$\Phi(v_3,\{x_7\&m25_3\}) \rightarrow ([2,1],[4,10],\{m23_2\})$ |
$\Phi(v_3,\{x_8\&m24_3\}) \rightarrow ([2,1],[4,10],\{m26_7\})$

A_{14}:$\Phi(v_5,\{x_3\&m27_5\&m28_5\}) \rightarrow ([11,6],[27,59],\{m19_0\})$ |
$\Phi(v_5,\{x_4\&m2_5\&m28_5\}) \rightarrow ([11,6],[27,59],\{m32_6\})$

A_{15}:$\Phi(v_6,\{x_9\&m29_6\&m30_6\}) \rightarrow ([6,3],[15,31],\{m27_5\})$ |
$\Phi(v_6,\{x_{10}\&m32_6\&m30_6\}) \rightarrow ([6,3],[15,31],\{m31_7\})$

A_{16}:$\Phi(v_7,\{x_{11}\&m31_7\}) \rightarrow ([3,2],[8,20],\{m25_3\})$ |
$\Phi(v_7,\{x_{12}\&m26_7\}) \rightarrow ([3,2],[8,20],\{m29_6\})$

A_{17}:$\Phi(v_4,\{x_{13}\}) \rightarrow ([3,2],[8,21],\{m20_0\})$

A_{18}:$\Phi(v_8,\{x_{15}\}) \rightarrow ([2,1],[4,9],\{m30_6\})$

A_{19}:$\Phi(v_9,\{x_{14}\}) \rightarrow ([5,3],[12,30],\{m28_5\})$

Each action is activated only once, when the corresponding condition will be equal to true. Actions $A_1 \div A_6$, and $A_{11} \div A_{16}$ contain alternative sub-actions. Only the first action, for which the condition will be satisfied, will be activated. According to the echo algorithm specification, process v_0 is the initiator, messages $m1_1$, ..., $m17_3$ are explorer messages, while $m18_0$, ..., $m31_7$ are echo messages (indices are added only for readability, mx_i means that message mx is sent to v_i. Events x_1, ...,x_{15} are internal events, used for storing the state of processes between successive executions.

TABLE I. TASK CHARACTERISTICS

Task no.	Task execution time [ms]				Energy consumption [mJ]			
	Exploration mode		Echo mode		Exploration mode		Echo mode	
	HPC	LPC	HPC	LPC	HPC	LPC	HPC	LPC
0	2	3	5	9	20	9	52	23
1	8	13	9	17	81	34	93	42
2	5	10	4	7	52	26	40	17
3	4	7	1	2	40	18	10	4
4	6	11	2	3	58	28	21	8
5	4	6	6	11	40	16	59	27
6	3	5	3	6	29	13	31	15
7	5	9	2	3	50	23	20	8
8	2	4	1	2	22	11	9	4
9	7	12	3	5	67	30	30	12

Since different requests may be processed by distinct algorithms, the function of a system may be specified using a set of functions Φ sharing the same processes. Each function has only one initiator (process activated by the request). Processes may be activated many times, but the algorithm should consists of the finite number of actions and infinite loops are not allowed.

C. ARM big.LITTLE technology

ARM big.LITTLE technology is an architecture where high-performance CPU cores are combined with the most efficient ones. In this way the peak-performance capacity, higher sustained performance, and increased parallel processing performance, at significantly lower average power, are achieved. It was shown that using this technology it is possible to save up to 75% CPU energy in low to moderate performance systems and it is possible to increase the performance by 40% in highly threaded workloads.

Three different methods of applying big.LITTLE technology for minimizing the power consumption were proposed [3]:

1) In the cluster switching, LPCs are grouped into “little cluster", while HPCs are arranged into “big cluster". The system uses only one cluster at a time. If at least one HPC core is required then the system switches to the “big cluster", otherwise the “little cluster" is used. Unused cluster is powered off.

2) In CPU migration approach, LPCs and HPCs are paired. At a time only one core is used while the other is switched off. At any time it is possible to switch paired cores.

3) The most powerful model is a Global Task Scheduling (GTS). In this model all cores are available at the same time i.e. tasks may be scheduled on all HPC as well as LPC cores.

Different configurations of LPC/HPC core are available.

For example Samsung Exynos 5 Octa consists of 4 LPCs (Cortex-A7) and 4 HPCs (Cortex-A15), Exynos 5 Hexa uses 2LPC/4HPC configuration, while the MediaTek MT8173 contains only 2 LPCs (Cortex-A53) and 2 HPCS (Cortex-A72). Big.LITTLE technology is applied also in Qualcom Snapdragon, NVidia Tegra X1, Apple A10 Fusion and HiSilicon processors.

Our approach is dedicated to the global task scheduling model. GTS is the most flexible and the most efficient method of applying big.LITTLE architecture. Moving tasks between HPCs and LPCs is fast, it requires less time than a DVFS state transition or SMP load balancing action.

IV. Power-Aware Scheduling

The draft of our algorithm of power-aware scheduling is given in Fig.2. First, a list of schedulable tasks (S_{list}) consists of the initiator, only. Then time marker (T) is initialized to 0. The main loop schedules the successive tasks, ordered according to their priorities. Priority of each task is based on the laxity (*L*), defined as a difference between task start times, obtained using ALAP (As Late As Possible) and ASAP (As Soon As Possible) methods, assuming the deadline (*TL*). These methods are applied assuming non limited number of cores. The *Sort()* method orders all schedulable tasks according to increasing laxity.

Tasks with the lowest laxity are scheduled first. If the laxity is higher than the difference between task execution times for LPC and HPC, then the task is scheduled on the LPC (if any LPC is available). If the laxity is lower than above difference, then the task is scheduled on the HPC (if any HPC is available). If no HPC is available, the task is scheduled on the LPC (if any LPC is available). If the difference between the time limit (deadline) and the time maker is higher or equal the system execution time obtained from ASAP method (in version for LPC), then the task is scheduled on the LPC. If none of the above conditions is fulfilled, the task stay in S_{list} and will be attempted to schedule in the next time frame.

Finally, all scheduled tasks are removed from the S_{list}. Before starting the next iteration of the main loop, the next tasks are added to the S_{list} using *NextReadyTasks()* method. The tasks are chosen according to rules of distributed echo algorithm. When all cores are busy or S_{list} is empty, the time marker is moved to the next time frame (function *NextAvailableTimeFrame()*), i.e. the nearest time when any core will finish executing task.

The presented algorithm is a greedy approach. First, it tries to reduce the power consumption by assigning tasks to LPC whenever it is possible. Although it is heuristic, we observed that in most cases it is able to find the solution for which all time constraints are satisfied.

```
S_list = all source nodes;
T=0;
while S_list≠ϕ do
{
  ASAP_h scheduling of all unscheduled tasks (HPC);
  ALAP_h scheduling of all unscheduled tasks (HPC);
  ASAP_l scheduling of all unscheduled tasks (LPC);
  for each t_i
  {
      L_i = ALAP_h (t_i)- ASAP_h (t_i);
  }
  Sort(S_list);
  for each  t_i∈ S_list{
      if  L_i > lt_i-ht_i and available(T,LPC) then
      {
          Assign t_i  to LPC;
          Mark t_i as scheduled;
      }
      else
      if systemExecutionTime(ASAP_l) ≤ TL – T
            and available(T,LPC)    then
      {
          Assign t_i  to LPC;
          Mark t_i as scheduled;
      }
      else
      if  L_i ≤ ht_i-lt_i then
            if available(T,HPC) then
            {
                Assign t_i  to HPC;
                Mark t_i as scheduled;
            }
            else if available(T,LPC) then
            {
                Assign t_i  to LPC;
                Mark t_i as scheduled;
            }
      if scheduled(t_i) then
          remove t_i from S_list;
  }
  add NextReadyTasks() to S_list;
  T=NextAvailableTimeFrame();
}
```

Fig. 2. Power-aware scheduling algorithm

V. EXAMPLE

Assume that the target embedded system is based on multi-core processor with 2 LPCs and 4 HPCs. The sample system specification (Fig.1) consists of 10 tasks that are executed twice, first time in the exploration phase and the second time during the echo phase. The initiator is defined as task 0. It starts the computations in the exploration mode and it returns the final result after execution in the echo mode. Assume that the soft deadline is equal 37 ms.

The algorithm starts with S_{list}={0} and T=0. During the first pass all tasks are initially scheduled using the ASAP and ALAP methods. Results are given in Tab. II.

TABLE II. INITIAL TASK SCHEDULING

Exploration phase										
Task	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
$ASAP_h$	0	2	10	14	2	2	6	9	9	6
$ALAP_h$	1	6	14	15	24	3	7	10	20	16
$ASAP_l$	0	3	16	23	3	3	9	14	14	9
L	1	4	4	1	22	1	1	1	11	10
Echo phase										
Task	***0e***	***1e***	***2e***	***3e***	***4e***	***5e***	***6e***	***7e***	***8e***	***9e***
$ASAP_h$	30	19	15	18	8	24	21	19	11	13
$ALAP_h$	32	23	19	19	30	26	23	20	22	23
$ASAP_l$	52	33	26	30	14	41	35	32	18	21
L	2	4	4	1	22	2	2	1	11	10

During the exploration phase tasks are identified by the task number, for the echo mode tasks are identified by adding suffix "e" to the task number. It may be observed that according to the $ASAP_L$ method scheduling on LPCs, the minimal system execution time is equal 61 ms and it requires 5 cores. The energy consumption equals 368 mJ. Initial $ASAP_L$ scheduling gives an information about minimal execution time using LPCs only. It gives also the solution with minimal energy consumption. The initial $ASAP_H$ scheduling returns the following results: execution time=35 ms, energy consumption=824 mJ, and requires 5 HPCs. Above results specifies the fastest solution, which consumes the maximal power.

The algorithm proceeds as follows:

1. T=0: S_{list}={0}
   Task 0: L_0=1, lt_0-ht_0=1, (65<37-T) =*false*, 0→ HPC1
2. T=3: S_{list}={5,1,4}
   Task 5: L_5=1, lt_5-ht_5=2, (64<37-T)=*false*, 5→ HPC1
   Task 1: L_1=4, lt_1-ht_1=5, (64<37-T)=*false*, 1→ HPC2
   Task 4: L_4=22, lt_4-ht_4=5, 4 → LPC1
3. T=6: S_{list}={6,9}
   Task 6: L_6=1, lt_5-ht_5=2, (62<37-T)=*false*, 6→ HPC1
   Task 9: L_9=10, lt_9-ht_9=5, 9→ LPC2
4. T=9: S_{list}={7,8}
   Task 7: L_7=1, lt_7-ht_7=4, (60<37-T)=*false*, 7→ HPC1
   Task 8: L_8=11, lt_8-ht_8=2, *LPC not available*
5. T=10: S_{list}={2,8}
   Task 2: L_2=4, lt_2-ht_2=5, (60>37-T)=*false*, 2→ HPC2
   Task 8: L_8=10, lt_8-ht_8=2, *LPC not available*
6. T=13: S_{list}={8,4e}
   Task 8: L_8=8, lt_8-ht_8=2, 8 → LPC1
   Task 4e: L_{4e}=22, lt_{4e}-ht_{4e}=1, *LPC not available*
7. T=14: S_{list}={3,4e}
   Task 3: L_3=1, lt_3-ht_3=3, (56<37-T)=*false*, 3→ HPC1
   Task 4e: L_{4e}=21, lt_{4e}-ht_{4e}=1, *LPC not available*
8. T=15: S_{list}={4e}
   Task 4e: L_{4e}=20, lt_{4e}-ht_{4e}=1, *LPC not available*
9. T=17: S_{list}={8e,4e}
   Task 8e: L_{8e}=11, lt_{8e}-ht_{8e}=1, 8e → LPC1
   Task 4e: L_{4e}=18, lt_{4e}-ht_{4e}=1, *LPC not available*
10. T=18: S_{list}={3e,2e,9e,4e}
    Task 3e: L_{3e}=1, lt_{3e}-ht_{3e}=1, (53<37-T)=*false*, 3e→ HPC1
    Task 2e: L_{2e}=1, lt_{2e}-ht_{2e}=1, (53<37-T)=*false*, 2e→ HPC2
    Task 9e: L_{9e}=10, lt_{9e}-ht_{9e}=2, 9e→ LPC2
    Task 4e: L_{4e}=17, lt_{4e}-ht_{4e}=1, *LPC not available*
11. T=19: S_{list}={7e,4e}
    Task 7e: L_{7e}=1, lt_{7e}-ht_{7e}=1, 7e→ LPC1
    Task 4e: L_{4e}=16, lt_{4e}-ht_{4e}=1, *LPC not available*
12. T=22: S_{list}={1e,6e,4e}
    Task 1e: L_{1e}=1, lt_{1e}-ht_{1e}=8, (52<37-T)=*false*, 1e→ HPC1
    Task 6e: L_{6e}=1, lt_{6e}-ht_{6e}=3, (52<37-T)=*false*, 6e→ HPC2
    Task 4e: L_{4e}=13, lt_{4e}-ht_{4e}=1, 4e→ LPC1
13. T=25: S_{list}={5e}
    Task 5e: L_{5e}=1, lt_{5e}-ht_{5e}=5, (49<37-T)=*false*, 5e→ HPC2
14. T=31: S_{list}={0e}
    Task 0e: L_{0e}=1, lt_{0e}-ht_{0e}=4, (6<37-T)=*false*, 0e→ HPC1

The final schedule is presented in Fig. 3. After executing ASAP and ALAP initial scheduling, task 0 has laxity equal 1 and difference between execution time for LPC and HPC equals to 1. Since the first two conditions specified in *if*

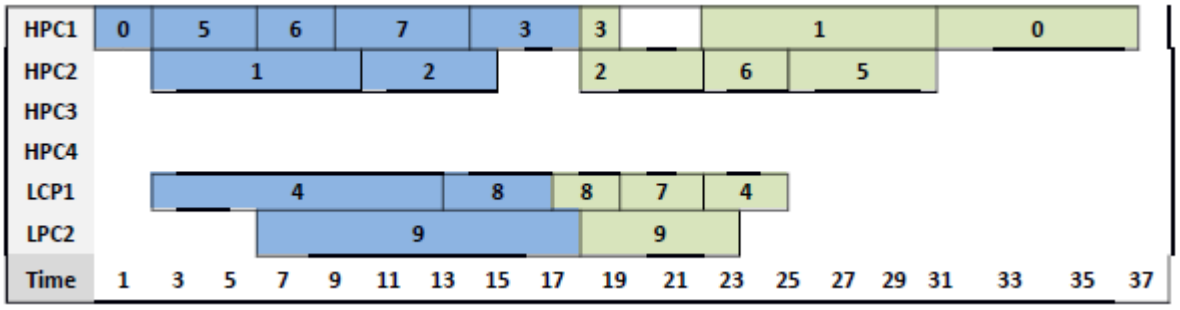

Fig. 3. Sample schedule for algorithm from Fig.1

statements evaluate to *false*, task 0 is scheduled on the HPC1.Next, tasks 1, 4 and 5 are added to the list to be scheduled in the next time frame. Tasks 1 and 5 have lower laxity than the difference between LPC and HPC, therefore they are assigned to HPC. Otherwise, the laxity of task 4 is significantly greater than the above difference, thereby the task is scheduled on the LPC. Similar cases take place for tasks 9, 8, 8e, 7e, 9e and 4e. All other tasks are scheduled on HPCs in order to fulfil given time constraint. The energy used by the processor is equal to 698 mJ. It gives 15% power saving, in comparison with the fastest solution. We observed that for lower time constraints our method can give even 55% of energy savings.

VI. Experimental Results

The efficiency of our method was evaluated using the example from Fig.1 as well as using other examples consisting of 25 and 45 tasks. Unfortunately there is no standard benchmark sets for echo algorithms. There is also no similar approaches of scheduling that may be compared with our approach. Therefore for comparison the classical list scheduling and ASAP methods were chosen.

Tables III, IV and V present results obtained for all sample algorithms using our method (EchoLPS) and list scheduling. Two different big.LITTLE architectures were examined, the first consists of 4 LPCs and 2 HPCs, the second one consists of 2 LPCs and 4 HPCs. For each architecture 4 different deadlines were examined. The mildest constraint was chosen in such a way that all tasks may be scheduled on LPCs. Such systems are found for reference, as the most power savings systems. Experimental results show how the tightening of time constraints affects the energy consumption. It should be noted that, nevertheless that our method is heuristic, in all cases solutions satisfying the deadline were found. But of course EchoLPS does not guarantee the fulfilment of hard real-time constraints.

For comparison the results obtained using classical list scheduling was given. List scheduling, first assigns tasks to HPCs i.e. it tries to find the fastest solution. Lists of tasks are ordered according to priority that is based on ALAP-ASAP values. We may observe that for comparable results (as far as the execution time is concerned) the solution found using the List Scheduling consumes significantly more energy than solutions obtained using our method.

For reference we also performed scheduling of all sample systems using List Scheduling, ASAP and ALAP methods. Table VI presents the results. For each solution the minimal number of LPC or HPC cores was found. Using List Scheduling it was possible to find the lowest energy consuming solutions. In some cases solutions are faster than obtained our method, but more LPC cores are required. Solutions found using ASAP/ALAP methods usually found the fastest solutions, but these methods do not minimize the number of cores required to execute task.

VII. Conclusions

In this paper a power-aware static scheduling method for embedded systems was presented. The method schedules real-time tasks on multi-core processor with power management capabilities. We applied our method to processors supporting ARM big.LITTLE technology, but the method may be adopted also to DVFS. The method gives better results than classical scheduling methods adopted to low-power embedded systems.

The method assumes the specification of the system in the form of a distributed echo algorithm. Such specification is more general than task graphs used in the most of existing static scheduling methods for real-time embedded systems. According to our best knowledge this is the first static scheduling method for real-time embedded software specified as the echo algorithm.

Our future work will concentrate on extending our method to systems specified using other classes of distributed algorithms, systems using other power management technologies as well as adaptive systems [21], considering the dynamic power optimization. Other direction of our work is to perform scheduling of the set of applications on the same system. Another interesting result may be obtained by developing quasi-static or quasi-dynamic scheduling method for distributed specifications. Such methods may be applicable to systems where the time of execution for tasks is not known or it is difficult to estimate.

The presented method uses simple heuristic to find the best tradeoff between the power consumption and efficiency of the system. Although the method gives quite good results, we will consider to apply more sophisticated optimization methods like constraint logic programming, mathematical programming [22] and developmental genetic programming [23].

TABLE III. RESULTS FOR 10 TASKS

System size	10					10				
Architecture	2 HPC and 4 LPC					4 HPC and 2 LPC				
Algorithm	EchoLPS				List Scheduling	EchoLPS				List Scheduling
Time constraint	65	56	46	37	NO	79	65	51	37	NO
Execution time	65	56	45	36	37	79	65	51	36	37
Energy consumption	368	499	637	698	716	368	458	570	698	813
Power increase	100%	136%	173%	190%	195%	100%	124%	155%	190%	221%
Time decrease	100%	86%	69%	57%	57%	100%	82%	65%	47%	47%

TABLE IV. RESULTS FOR 25 TASKS

System size	25					25				
Architecture	2 HPC and 4 LPC					4 HPC and 2 LPC				
Algorithm	EchoLPS				List Scheduling	EchoLPS				List Scheduling
Time constraint	201	168	143	117	NO	357	280	204	127	NO
Execution time	194	167	142	117	125	357	279	202	127	99
Energy consumption	1672	2033	2280	2604	2864	1672	2165	2631	3177	3431
Power increase	100%	122%	136%	156%	171%	100%	129%	157%	190%	205%
Time decrease	100%	86%	73%	60%	64%	100%	78%	57%	36%	28%

TABLE V. RESULTS FOR 45 TASKS

System size	45					45				
Architecture	2 HPC and 4 LPC					4 HPC and 2 LPC				
Algorithm	EchoLPS				List Scheduling	EchoLPS				List Scheduling
Time constraint	246	215	185	154	NO	466	368	271	171	NO
Execution time	246	215	185	154	133	466	363	271	171	130
Energy consumption	2169	2559	2900	3313	3630	2169	2821	3437	4169	4537
Power increase	100%	118%	134%	153%	167%	100%	130%	158%	192%	209%
Time decrease	100%	87%	75%	63%	54%	100%	78%	58%	37%	28%

TABLE VI. RESULTS FOR LIST SCHEDULING, ASAP AND ALAP

System size	10	25	45	10	25	45	10	25	45
Architecture	6 LPC			5 HPC	9 HPC	15 HPC	4 HPC	8 HPC	13 HPC
Execution time	65	193	230	37	109	130	37	109	130
Energy consumption	368	1672	2169	824	3864	5202	824	3864	5202
Algorithm	List Scheduling			ASAP			ALAP		

REFERENCES

[1] M. Ditzel, R.H.Otten, W.A.Serdijn, Power-Aware Architecting for data-dominated applications, Springer, 2007.

[2] S. Deniziak, Cost-efficient synthesis of multiprocessor heterogeneous systems. Control and Cybernetics, vol.33, 2004, pp.341-355.

[3] big.LITTLE Processing with ARMCortexTM - A15 & Cortex-A7, ARM Holdings, http://www.arm.com/files/downloads/ big.LITTLE_Final.pdf, September 2013,

[4] L.Benini, A.Bogliolo, G. De Michelli, A Survey of Design Techniques for System-Level Dynamic Power Management, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.8, no.3, June 2000, pp.299-316.

[5] S. Hartmann, D. Briskorn, A survey of variants and extensions of the resource-constrained project scheduling problem, European journal of operational research : EJOR. - Amsterdam : Elsevier, Vol. 207., 1 (16.11.), 2010, pp. 1-15.

[6] L. Costero, F. D. Igual, K. Olcoz, S. Catalán, R. Rodríguez-Sánchez and E. S. Quintana-Ortí, "Refactoring Conventional Task Schedulers to Exploit Asymmetric ARM big.LITTLE Architectures in Dense Linear Algebra," *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Chicago, IL, 2016, pp. 692-701.

[7] G. Tel, "Introduction to Distributed Algorithms" Cambridge University Press, 2nd edition, 2001.

[8] S. Bąk, R. Czarnecki, S. Deniziak "Synthesis of real-time cloud applications for Internet of things" *Turkish Journal of Electrical Engineering and Computer Sciences*, vol.23, no.3, 2015, pp. 913- 929.

[9] S. Bąk, S. Deniziak "Synthesis of real-time distributed applications for cloud computing", *IEEE Federated Conference on Computer Science and Information Systems*, IEEE, 2014.

[10] Ernest J.H. Chang: Echo Algorithms: Depth Parallel Operations on General Graphs. IEEE Transactions on Software Engineering, Vol. 8, No. 4, July 1982.

[11] B. P. Dave, G. Lakshminarayana and N. K. Jha, "COSYN: Hardware-software co-synthesis of heterogeneous distributed embedded systems," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 7, no. 1, March 1999, pp. 92-104.

[12] L.Shang. R. P. Dick. N.Jha. SLOPES: Hardware/Software Cosynthesis of Low-Power Real-Time Distributed Embedded Systems With Dynamically Reconfigurable FPGAs, IEEE Trans on Computer-Aided Design of Integrated Circuits and Systems, vol.26, no.3, 2007, pp.508-526.

[13] M. T. Schmitz, B. M. Al-Hashimi, P. Eles, System-Level Design Techniques for Energy-Efficient Embedded Systems, Kluwer Academic Publishers, 2004.

[14] F.F. Yao, A.J. Demers and S. Shenker. A scheduling model for reduced CPU energy. Proc. 36th IEEE Symposium on Foundations of Computer Science, 1995, pp.374–382.

[15] J.Luo, N.K. Jha, Low Power Distributed Embedded Systems: Dynamic Voltage Scaling and Synthesis, Proc. 9th Int. Conference High Performance Computing - HiPC 2002, Lecture Notes in Computer Science, vol. 2552, 2002, pp. 679-693.

[16] C.Steger, C.Bachmann, A. Genser, R. Weiss, J, Haid, Power-aware hardware/software codesign of mobile devices, e & i Elektrotechnik und Informationstechnik, vol.127, no. 11, 2010.

[17] J. Anderson, V. Bud, and U. C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In IEEE ECRTS, July 2005, pp. 199-208.

[18] Han, Sangchul and Park, Minkyu, Predictability of Least Laxity First Scheduling Algorithm on Multiprocessor Real-Time Systems, Proc. of EUC Workshops, Lecture Notes in Computer Science, vol.4097, 2006, pp.755-764.

[19] Li Jie, Guo Ruifeng and Shao Zhixiang, "The research of scheduling algorithms in real-time system," 2010 International Conference on Computer and Communication Technologies in Agriculture Engineering, Chengdu, 2010, pp. 333-336.

[20] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, Christopher Gill, "Parallel Real-Time Scheduling of DAGs," IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 12, Dec., 2014, pp. 3242-3252.

[21] S.Deniziak, L.Ciopinski, "Synthesis of power aware adaptive schedulers for embedded systems using developmental genetic programming", Proc. *Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2015, pp.449-459.

[22] P. Sitek, J. Wikarek, "A Hybrid Programming Framework for Modeling and Solving Constraint Satisfaction and Optimization Problems", Scientific Programming, vol. 2016, Article ID 5102616, 2016.

[23] S. Deniziak, L. Ciopinski, G. Pawinski, K. Wieczorek, and S. Bak, "Cost optimization of real-time cloud applications using developmental genetic programming", in IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC 2014), December 2014, pp.774-779.