International Journal of Computer Applications (0975 8887)
Volume 97 - No. 10, July 2014

An Effective Resource Partitioning Heuristic for Embedded Applications on an MPSoC

Hassan Salamy
Texas State University
601 University Drive
San Marcos, TX, USA, 78666

Olalekan Sopeju
Texas State University
601 University Drive
San Marcos, TX, USA, 78666

ABSTRACT

As the utilization of multiprocessors system-on-chip (MPSoC) is becoming ubiquitous, demands for effective allocation and scheduling techniques are needed more than ever to harness the power of MPSoCs. An MPSoC is a system consisting of multiple heterogeneous processing cores, memory hierarchies, and communication infrastructure to effectively overcome the power and clock constraints from single core architectures. MPSoCs provide the performance demanded by embedded applications especially real-time multimedia applications. This article presents effective techniques to partitioning the processing cores and memory budget in an MPSoC among multiple embedded applications possibly entering the system at different times. The proposed framework will study the structure of each application and predict the possible reduction in schedule time if more processors and/or memory budget are assigned to this application. The objective is to fairly divide the resources such that the schedule times for the applications are minimized. Results on different embedded applications workloads and under different system resources show the effectiveness of our techniques that were able to reduce the cycle count by 10.2 % on average compared to an effective technique in the literature.

Keywords:

MPSoC, Allocation, Scheduling, Scratchpad

1. INTRODUCTION

Multi-core designs are now considered the trend to overcome the limiting performance return from single core designs that are facing serious clock, power, and physical constraints. This trend found its way in general purpose architectures as well as embedded systems. The utilization of multiple cores improves the system performance through possible task parallelization. This opened the door to achieve higher performance levels to solve low-end and high-end computing challenges. Following this trend, multi-processor System-on-chip (MPSoC) architecture designs are now ubiquitous. This kind of system usually includes multiple processing cores that are often heterogeneous, complex interconnected architecture for input and output components as well as memory hierarchy that usually spreads between fast on-chip levels of memory to slower large external memory components. MPSoCs are often viewed as flexible high performance systems with optimized power consumption. With the heavy utilization of MPSoCs, the trend of memory performance is lagging that of the processors. Hence, in embedded systems especially those mainly used for real-time computing, memory types and access speed are often two main research items that should be addressed to be able to harness the power of MPSoCs. Memory access latency is considered to be a main obstacle to improve the speed of embedded applications scheduled on such systems. This problem is even more serious in MPSoC due to the heavy contention the communication network encounters and due to the trend of using shared memories in many cases. Execution time predictability is another critical aspect of memory in systems utilizing real-time embedded applications. Caches usually fall short to these real-time requirements as they are hardware-controlled and hence modeling their exact behaviors for execution time prediction is often not attainable. Hence, many MPSoCs use software-controlled *scratchpad memories* (SPMs). Scratchpad memories are software controlled and therefore they are suitable to accurately predict the run times of real-time embedded applications. But due to their limited size in embedded systems, many multi-processors systems-on-chip use some kind of a memory hierarchy with small capacity but fast on-chip memories and large capacity slower off-chip memories. This difference in access latency implies that the proper allocation of variables to the fast on-chip memory is essential in reducing the run times of embedded applications utilizing the MPSoC as often the latency of the off-chip memory is in the range of 100 times slower than that of the on-chip memory.

Many complex embedded applications consist of multiple concurrent real-time tasks [1]. The execution time of a task depends on the processors it is allocated to as well as the available SPM budget. Often multiple applications are simultaneously utilizing the MPSoC and hence they compete for the available cores and memory resources. Proper allocation of the system resources among competing embedded applications and effective scheduling techniques play an essential role in minimizing the execution times of the applications. This article assumes an MPSoC system with multiple processing cores, a fast on-chip SPM memory budget and a large of-chip memory. The system is being utilized by multiple applications with start times possibly not known a priori. Based on such system, effective heuristics are presented to fairly divide the system resources among the embedded applications simultaneously utilizing the system. Based on the applications currently on the system, our framework studies the structure of each application provided through profiling and allocates the resources accordingly. The prob-

lem of resource partitioning on MPSoCs is an NP-complete problem [2].
The rest of this article is organized as follows. Section 2 presents related work. Section 3 presents the architectural model and our approach. Section 4 presents our effective proposed approach. Section 5 is the results and Section 6 presents the conclusion.

2. RELATED WORK

The problem of allocation and scheduling of embedded applications on multiple processors has been studied by many research groups. Benini et al. [3] used integer linear programming and constraint programming to solve the problem. Different scheduling algorithms were compared on a set of diverse benchmarks [2]. Also, an integer linear programming approach was used to solve the hardware/software codesign partitioning problem [4]. A branch and bound algorithm to solve the hardware/software partitioning problem with pipelined scheduling was introduced in [5].
Panda et al. [6, 7] published a comprehensive technique to SPMs allocation on a single processor to reduce the run time through maximally utilizing the available fast SPM memory. Integer linear programming approaches to optimally solve the memory allocation problem for SPMs were presented in [8, 9]. An ILP formulation for the scratchpad memory allocation was also used in [10] to reduce the code size. Kuang et al. [11] proposed an integer linear programming solution to partitioning and pipelined scheduling. Angiolini et al. [12] utilized dynamic programming to effectively solve the problem of mapping memory locations to SPM locations.
The problem of memory allocation on multi-processor system-on-chips was studied by many research groups. Data parallelism to improve performance in a system of homogeneous multiprocessor systems is mainly the main focus of many of such research. In order to obtain optimal distributed shared memory architecture to reduce the memory and data access costs, Meftali et al. [13] used an optimal integer linear programming formulation. Kandemir et al. [14] used a compiler based approach to optimize energy and memory access latency on MPSoCs. In [15], hard real-time utilization was improved using a memory-centric scheduling technique. The scheduling of memory intensive periodic tasks onto real-time multi-core systems was introduced in [16]. Blagodurov et al. [17] presented a contention-aware scheduling algorithm on multicore systems. Vaidya et al. [18] proposed a dynamic scheduling algorithm based on hosting the scheduler on all cores of a multi-core processor and accesses a shared Task Data Structure (TDS) to pick up ready-to-execute tasks. Power and energy efficient scheduling on multicore systems has been studied in [19] and [20].
Suhendra et al. [21] and Salamy [22] studied the problem of integrating task scheduling and memory partitioning among a heterogeneous multiprocessor system on chip with scratch pad memory. Other works [23, 24, 25, 26, 27, 28, 29, 30] have studied issues related to task scheduling/allocation and memory partitioning on multiprocessor systems. Xue et al. [31] proposed a dynamic resource partitioner for embedded applications in an MPSoC. Their proposed approach partitions the resources in a manner proportional to the requirements of each application. The system allocates and deallocates resources as new applications get or leave the system. This is the closest approach to what we propose and hence we compare our results against it.

3. THE ARCHITECTURAL MODEL AND OUR APPROACH

This article assumes an MPSoC consisting of a set of processing cores, a limited size on-chip SPM budget, and large off-chip memory. Applications utilizing the system will compete for the available resources. Processor cores and SPM budgets will be allocated among the applications simultaneously using the system. Our approach will examine the structure of each application in the system to decide the number of cores as well as the SPM memory budget to allocate to that application. A simple view of our architectural model is presented in Figure 1. The example model consists of three applications with the corresponding processor cores and memory budget divided among the applications.

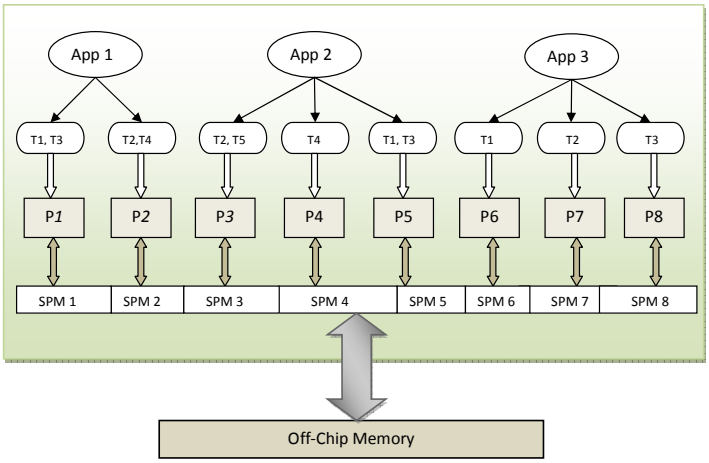

Fig. 1. An example MPSoC with three applications, eight processors, an SPM budget, off-chip memory, and interconnection bus.

Problem Definition: *Given (i) an MPSoC architectural model with multiple processor cores, on-chip SPM memory, and large off-chip memory and (ii) a set of applications to be executed at this system with possibly unknown start times, fairly divide the processor cores and the SPM budget among all concurrently executing applications in the system to minimize the execution times of the applications.*

The main question to be addressed in this article is how to fairly divide the resources among the available embedded applications. Our approach will examine the nature of each application to allocate resources. Generally speaking, more cores will be allocated to applications more parallel in nature whereas memory intensive applications will enjoy a larger memory budget. An application is of parallel nature if its task dependence graph has the potential of increased parallelism. Such applications benefit from more cores as tasks can be run in parallel. On the other side, an application is memory-intensive in nature if accessing memory is what constitutes the larger percentage of the run time. Clearly, this type of applications benefits more from a larger SPM memory budget as it is assumed that accessing the on-chip memory is many times faster than that of the external memory.
Our proposed system will receive applications of possibly unknown start times and then a set of information will be extracted by the

profiler that reflects the nature of each application. The extracted set of information by the profiler will be used by the proposed resource partitioner to decide on the amount of resources to allocate to each application so that the execution times of the applications in the system are minimized. The resources will be allocated based on the structure of the applications concurrently using the system. Since the resources in the system are assumed to be limited, the proper allocation plays a major role in minimizing the schedule time of the applications. Once the resources are allocated, a schedule for each application's tasks will be produced based on the resources mapped to this application. This article only studies the allocation problem and it uses our previously published scheduling heuristic [22] to construct the schedule. Notice that as an application enters or leaves our system, the resources will be redistributed to reflect the current applications in the system and this adds to the dynamic essence of our proposed solution.

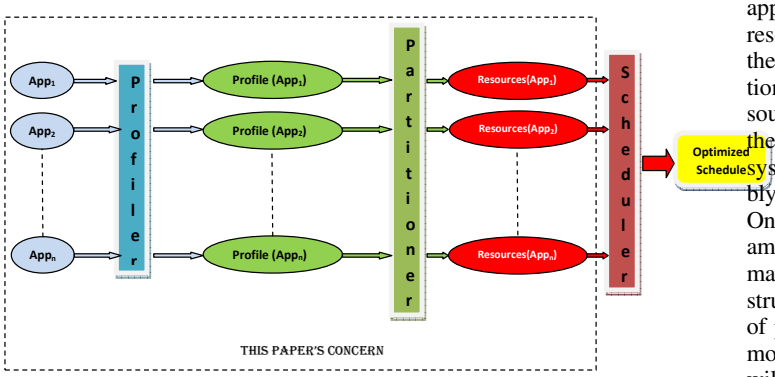

Fig. 2. Our Proposed Framework.

4. OUR EFFECTIVE PROPOSED APPROACH

Our proposed framework is presented in Figure 2. The system receives applications possibly at different times. Then the profiler extracts important information about the application and forwards them to the resource partitioner that partitions the resources among the applications. The resource budgets are then sent to the scheduler to generate an effective minimum time schedule. This paper is only concerned with the profiler and the resource partitioner. Our resource allocation techniques depend on the structure of the applications utilizing the MPSoC system at the same time. The profiler will examine the applications and provide the necessary information about each application. This information will be used by the resource partitioner. The profiler part of our proposed approach is detailed next.

4.1 The Profiler

Once the system receives a new application, the profiler will study its structure and extract important information that will be sent to the resource partitioner. One important piece of information is the task dependence graph (*TDG*). A task dependence graph is a directed acyclic graph with weighted edges where each task in the embedded application is represented by a vertex. The profiler will indentify the main computation blocks. Computation blocks will be used as the vertices in the construction of the task dependence graph (TDG). Dependencies between the computation blocks will be represented as weighted edges between tasks in the TDG with the weights representing the communication costs. Communication costs will be estimated from the information about control and data flow.

The profiler will also extract a set of important information about the tasks of each application, namely Max_{ij}, Avg_{ij}, and Min_{ij}. Min_{ij}, Avg_{ij}, and Max_{ij} of a task of an embedded application in a system of p processors represent the computation time for task T_i on processor P_j if all the SPM budget is assigned to this P_j , $1/p$ of the SPM budget is assigned to P_j, and no SPM budget is assigned to P_j, respectively.

Note that not all these information are necessarily needed by the resource partitioner to be discussed in the next section but they are an essential information needed by our scheduler technique in [22].

4.2 The Resource Partitioner

This section provides effective techniques to partition the available system processor cores and SPM memory among the embedded applications currently using the system. As mentioned earlier, the resource partitioner will receive the necessary information about the applications extracted by the profiler. Based on this information, the resource partitioner is supposed to divide the system resources among the applications so that the schedule times for all the applications are minimized. Since we are assuming a typical system with limited resources, embedded applications will probably receive fewer resources than that is optimally needed.

Once the resource partitioner receives a new application, it will examine the information about its structure and compute an approximate value that represents its level of parallelism mainly from the structure of its corresponding task dependence graph. This level of parallelism will reflect how much this application benefit from more processor cores. On the other hand, the resource partitioner will also examine the application to determine how much it can benefit from a higher SPM budget. This will mainly be reflected through a computed value called *elasticity*. The proposed resource partitioner is made up of two main parts, the SPM partitioner (see Figure 3), and the processors partitioner (see Figure 5) detailed next.

4.3 The SPM Partitioner

The SPM partitioner is responsible about partitioning the SPM budget in the system. The limited SPM budget in the MPSoC system will be partitioned among the embedded applications concurrently using the system. Due to the limited SPM resources, usually not all applications variables can fit in the SPM. Therefore proper allocation of the SPM is essential to reduce the computation time as accessing the SPM is 100 times faster than accessing the external memory. Applications will receive an SPM budget based on their structure. Applications that benefit more from a larger SPM budget will get more SPM compared to applications where more SPM budget is less beneficial. This added benefit will be reflected in the *elasticity* value.

The *elasticity* value represents the extent an application can benefit from more SPM budget. Given an application with multiple tasks and a set of possible processors that might execute each task, the *elasticity* of task T_i on processor P_j is a number between 0 and 1 where a higher value implies that the computation time of this task is amendable to more reduction if more SPM budget is allocated to processor P_j assigned to run this task. The *elasticity* value defined in Equation 1 of task T_i on processor P_j depends on the Cur_{ij} and the Min_{ij} values. As defined earlier Min_{ij} is the run time of task T_i on processor P_j assuming that all the available SPM budget is assigned to processor P_j. On the other hand, Cur_{ij} represents the run time of task T_i on processor P_j under the current SPM budget

assigned to processor P_j. In our case, Cur_{ij} is calculated based on partitioning the remaining SPM budget equally over the applications who have not already received an SPM budget. Based on the two values Cur_{ij} and Min_{ij}, *elasticity* reflects the room for improvement from more SPM budget. The $Curr_{ij}$ value is a dynamic value as it depends on the SPM budget distribution among the applications and hence it lends this dynamic essence to *elasticity*.

$$elasticity(T_{ij}) = \frac{Cur_{ij} - Min_{ij}}{Cur_{ij}} \quad (1)$$

elasticity is then used to define the predicted reduction fraction PRF of an application that reflects the degree an added SPM can reduce the computation time of the whole application rather than individual tasks. More precisely, the PRF of an application is defined in Equation 2 as the average of the elasticity values of all its t tasks among all the p processors.

$$PRF(APP_i) = 1/p \cdot \sum_{j \in p} \sum_{T_i \in APP_i} \frac{elasticity(T_{ij})}{t} \quad (2)$$

Our proposed heuristic in Figure 3 to partition the available SPM consists of two main steps. First, it determines the memory requirement of each application from its nature mostly established by the profiler. Then, it allocates the SPM memory to the applications based on the nature of each application. The heuristic takes as input the SPM size (m) and the n applications concurrently using the system. It starts by creating a list of applications in decreasing order of their computed PRF values. Based on the structure of the each application which translates to its data requirement obtained through the function *SPM_requested()*, our proposed SPM heuristic finds the total SPM budget needed (SPM) to satisfy the requested budget by all the applications. If the available SPM is larger than the requested SPM budget then each application will simply receive the memory it requested.

On the other hand, if the available SPM in our system is less than the requested memory, then the heuristic effectively divide the memory among the competing embedded applications. This is mainly done in Lines 12–21 in the heuristic in Figure 3 in which an application will receive an SPM budget proportional to what it requested such that an application with higher predicated reduction factor (PRF) will receive an SPM budget closer to the requested budget compared to an application of smaller PRF value. This is done since PRF of an application reflects the added benefit from more SPM budget based on the nature of its tasks through the *elasticity* value defined earlier. The proposed heuristic then updates the list L, the Min and $Curr$ values of the remaining applications based on the remaining SPM budget in the system. It also updates the PRF values.

4.4 The Processing Cores Partitioner

The processing core partitioner will receive information from the profiler pertaining to the structure of the embedded applications. The received information will be used to approximate the degree of parallelism of an application. More parallel application will benefit more from more processing cores as in this case more tasks can run in parallel. We define an approximate value to capture the degree of parallelism (DP) as in Equation 3. A large value of DP implies that the application holds a higher degree of parallelism compared to a smaller DP value. The DP value reflects the structure of the application mainly through its task dependence graph (TPG). DP reflects the potential parallelism between different tasks of the embedded application. Two tasks can run in parallel if they are independent

SPM_Partitioner(n, m)

```
1.  L = List applications in the system in decreasing order of PRF
2.  SPM =0 and Total_PRF = 0
3.  For i = 1 to n do:
4.    SPM = SPM + SPM_requested(i)
5.    Total_PRF = Total_PRF + PRF(i)
6.  End For
7.  If (SPM≤ m)
8.    For i = 1 to n
9.      SPM_received(i) = SPM_requested(i)
10.   End For
11.  Else
12.   While L is not empty
13.     i = First application in list L.
14.     Temp_Value = UpperBound((PRF(i)/Total_PRF)* m)
15.     SPM_ received(i) = MIN(SPM_requested(i),Temp_Value )
16.     m = m - SPM_ received(i)
17.     Remove i from L.
18.     Recompute the Min and Curr values of the remaining
          applications based on the new m value.
19.     Find the new PRF and the Total_PRF values.
20.     Rebuild L.
21.   End While
```

Fig. 3. The proposed SPM partitioning heuristic.

on each other whereas dependent tasks in the TDG are to be run sequentially. Two tasks (Ti, Tj) in the TDG are said to be independent if there is no path that goes through these two tasks. A higher number of independent tasks in an application loosely imply a higher potential for parallelism.

To find the DP value for a given embedded application, first add two dummy nodes to the TDG, a dummy start node S and a dummy end node E. From node S add an outgoing edge to all the nodes (tasks) in the TDG with zero incoming edges. Whereas to node E add an edge from each node (task) in the TDG with no outgoing edges. Then the heuristic will find all the paths between S and E. Two paths are called distinct if they differ in at least one task. Now for any two distinct paths, say p_i and p_j, find the pairs ($T_i \in p_i$, $T_j \in p_j$) such that T_i and T_j are independent and hence can run in parallel.

The degree of parallelism (DP) defined in Equation 3 consists of two main parts. The first part, $paths_i$, is the number of distinct paths in the TDG corresponding to an embedded application as explained earlier. Even though a higher number of distinct paths might reflect a bigger potential for parallelism, it is not a sufficient metric to reflect the parallelism in a TDG and hence the second component in Equation 3

$$DP(APP_i) = paths_i + \frac{pairs_i}{paths_i} \quad (3)$$

The second part of the DP definition captures the fact that two more balanced paths will benefit more from two processors compared to two unbalanced paths. To clarify this point, consider the simple TDG examples in Figure 4. These two example TDGs represent two applications with the same number of tasks and the same number of distinct paths. Now assume that two processors are allocated to each TDG. For the unbalanced TDG in Figure 4-(a), if processor 2 is mapped to Task T_6 then this processor is no more needed after the execution of T_6 as all other tasks in the TDG are dependent.

Table 1. List of pairs of independent tasks.

Unbalanced TDG	Balanced TDG
(1, 6)	(1, 4)
(2, 6)	(1, 5)
(3, 6)	(1, 6)
(4, 6)	(2, 4)
(5, 6)	(2, 5)
NA	(2, 6)
NA	(3, 4)
NA	(3, 5)
NA	(3, 6)

However, in the case of the balanced TDG in Figure 4-(b), the two processors can be maximally utilized to run the tasks in parallel with minimum idle times. To reflect that the embedded application corresponding to Figure 4-(b) has higher potential for parallelism compared to that in Figure 4-(a), the second part, $\frac{pairs}{paths}$, of our DP definition is next introduced and explained.

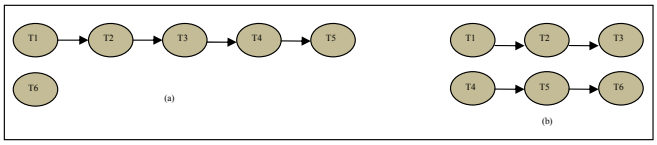

Fig. 4. (a) An unbalanced TDG. (b) A balanced TDG

The $pairs$ value in Equation 3 is defined as the number of task pairs that can be executed in parallel. The pairs for the example TDGs in Figure 4 are listed in Table 1. The first column in the table list such pairs corresponding to the unbalanced TDG in Figure 4-(a) whereas the pairs corresponding to the balanced TDG in Figure 4-(b) are listed in Column 2. Based on the values in Table 1, the DP value for the unbalanced TDG computes as 2 + 5/2 = 4.5 whereas the DP value for the balanced TDG is 2 + 9/2 = 6.5. Now assume that the two embedded applications corresponding to the two TDGs in Figure 4 are to be executed on a system with 3 available processor cores. Based on the computed DP values, our processor resource partitioner heuristic detailed later on, will assign two processors to the balanced TDG and 1 processor to the unbalanced TDG which is an efficient allocation under the stated scenario.

Please note that even though the *pairs* term represents the number of independent pairs of tasks, it is in no way means that all of those tasks can run in parallel. To explain this point, assume a simple *TDG* with the following pairs of tasks between two paths p_i and p_j: ($T_i \in p_i, T_j \in p_j$), (T_1, T_3), (T_1, T_4), and (T_1, T_5). What this tells us is that task T_1 that belongs to path p_i can run in parallel with either of the tasks T_3, T_4 or T_5. Although the *pairs* value will be equal to the number of such pairs, only one such pair can run in parallel since T_3, T_4, and T_5 belong to the same path p_j. This simple example shows that the *pairs* value does not represent the number of pairs that run in parallel but rather the potential for such parallelism that is reflected by the degree the paths in the *TDG* are balanced.

Our effective heuristic to processor partitioner presented in Figure 5 tries to divide the available processing cores in our system fairly among the available competing embedded applications in the system to reduce the schedule times. This will be achieved by allocating to each application a number of processors that is proportional to the DP value and the number of distinct paths. The heuristic takes as input the number of processing cores (p) in the system as well

Processor_Partitioner(n, p)

```
1.   Path = 0 and Path_DP = 0
2.   For i = 1 to n
3.     DP(i) = Compute_DP(i)
4.   End For
5.   L = List the applications in decreasing order of (1 + α DP(i)) * path(i)
6.   For i = 1 to n
7.     Path = Path + path(i)
8.     Path_DP = Path_DP + (1 + α DP(i)) * path(i)
9.   End For
10.  If (Path ≤ p)
11.    For i = 1 to n
12.      Processor_received(i) = path(i)
13.    End For
14.  Else
15.    While L not empty
16.      i = First application in L.
17.      Temp_Value = UpperBound((1 + α DP(i)) * path(i)/Path_DP * p)
18.      Processor_received(i)= MIN(path(i),Temp_Value )
19.      Update: Path_DP = Path_DP - (1 + α DP(i)) * path(i)
20.      p = p - Processor_ received(i)
21.      Remove i from L.
22.    End While
```

Fig. 5. The proposed processor partitioning heuristic.

as the number of concurrently running applications (n) with their profile data extracted by the profiler. First, the heuristic will sort the applications in Line (5) in the decreasing order of their parallelism potential. The expression in Line (5) produces better results than simply using the DP value to sort the applications in the system. This is mainly due to the fact that the DP value is an exaggeration of the tasks that can actually run in parallel and hence the refinement in Line (5). Our proposed heuristic will then store the total number of requested processors by all the applications in the system in the term *Path*. Since it is not easy to come up with the exact value for the optimal number of cores for an application, our heuristic uses the number of distinct paths as an approximation for the number of cores. If the available number of cores is larger than the requested cores by the applications, then each application will simply receive the number of cores it requested.

On the other hand, if the number of available processing cores in our system is less than the requested cores, then it will effectively divide the available cores among the competing embedded applications. This is mainly done in lines (15–22) where competing applications will receive cores in a way such that applications with higher DP values will be allocated number of processing cores closer to what they requested. A good value for α in the processing heuristic in Figure 5 is 0.1 which is found through fine tuning. Please note that from the way DP and paths are defined and the way the proposed heuristic is set up, two applications with the same number of paths might receive different number of cores. This is mainly due to the fact the DP is defined not only to depend on the number of paths but the number of independent pairs of tasks that loosely reflect how balanced the TDG is. As mentioned earlier, this is an effective utilization of the processing cores in the system as a more balanced TDG implies that the cores can be better utilized compared to less balanced TDGs where in such case cores might exhibit more idle times.

Run_Time_Partitioner(n, p)

1. ∀ applications already in the system:
2. Reconstruct their TDG including only unscheduled tasks.
3. Processor Partitioner(n, p);
4. SPM Partitioner(n, m);

Fig. 6. Our run time partitioning heuristic.

Our processor allocation heuristic works also for a heterogeneous set of computing cores in an SoC. Based on the structure of each application and its requirements, the partitioner will decide on which kind of processing cores to allocate to each application.

4.5 Run Time Resource Allocation/Deallocation

The proposed heuristics can be used under different scenarios. The first scenario is a system with multiple embedded applications that get to the system at the same time. The second scenario is a system with multiple embedded applications with known start times but not necessarily equal start times. The third scenario is to use our techniques at run time where applications can get to the system at different times with no prior knowledge about the start times. The proposed heuristics are fast enough that online allocation during run time is possible. Under this scenario, the resources will be allocated or deallocated based on what applications are utilizing the system. When an application gets to the system, the heuristics will be called to perform a new resource allocation based on the new set of embedded applications. The same happens when an application leaves the system as in this situation the resources previously allocated to this application are free and they can be allocated to other applications if needed. Our run time heuristic in Figure 6 is called any time a new application gets or leaves the system. Although run time allocation and scheduling is always a harder problem, our proposed heuristics can be effectively used in such situation if needed. This is mainly because our proposed allocation and deallocation can be performed very fast by our heuristics and thus they are suitable for this situation.
However, the scheduler should be designed carefully as how to behave when some resources of an application need to get deallocated. We will not go into details as how this might be handled as the scheduler part is beyond the scope of this article. However, we will summarize the proposed approach to this scenario. Due to this possible deallocation, some memory budget allocated to an application may need to be allocated to another application. First, if a new application gets into the system, the resource partitioner will be invoked based on the new set of applications currently using the system. If the memory budget m derived from our memory partitioner is greater than the currently available memory budget that is not used by any other application, some of the memory budget will be freed from other applications to meet the new requirements. To prevent any data loss, task preemption is not allowed. That is, a task of an application that is required to give part of its memory will run into completion before this deallocation occurs. In order to determine which parts of the SPM memory budget are best candidates to deallocate, our techniques keeps track of the frequency memory locations are used during execution and least recently used (LRU) ones will top the candidates list for deallocation.

Table 2. Characteristics of our benchmarks.

Benchmark	# of variables	# of tasks	Total Var size (Kbytes)
Lame	128	4	294.83
Osdemo	46	7	78.64
Enhance	44	6	7192.35
Cjpeg	20	5	690.31

5. EXPERIMENTS

5.1 Benchmarks and Set up

In this section, the resource partitioning techniques are tested to show the effectiveness of the proposed work. Real life embedded applications are used from different benchmark suites including [25], *Mediabench* and *MiBench,* [30, 32]. The benchmarks used are $enhance$, $lame$, $osdemo$, and $cjpeg$ with their characteristics presented in Table 2
Our profiler will examine each embedded application and extract the necessary information to be used by the resource partitioner as detailed earlier. The profiler will first identify the main computation blocks (tasks) which will translate to the nodes in the task dependence graph (TDG). The profiler will also study the dependencies between different blocks based on the control/data flow information and add the appropriate edges to the TDG. This control/data flow information to estimate the communication costs will be represented by the weights of the edges in the TDG. An instrumented version of the architectural simulation tool *Simplescalar* was used to get some of the profile information. *Simplescalar* is utilized to find the computation time of an application on a certain processor under a specific memory budget. In addition to the TDG and the computation time of tasks on different processors, the profiler will extract important information like the Min and Cur values and the size of the variables with their frequency of appearance in a task.

5.2 Results

There are not many techniques in the literature that can compare directly to our dynamic technique to resource partitioning among multiple applications in the system. The closest to what we are doing is the technique presented in [31]. Hence, we implemented the following two techniques and performed our experiments and compared the results:

—[31]: This is the approach presented in [31]. In this approach, the authors divide the available resources carefully among the applications to reduce the run time.

—Ours: Our approach detailed in this paper to fairly divide the available resources among competing applications in the system.

Our resource allocation techniques for memory and processing cores are implemented and tested under different scenarios to show their effectiveness. As mentioned earlier, we compare our approach to the effective technique in [31]. Testing our techniques is not an easy task as this article assumes that applications can get into the system at random times in contrast to predefined times. Our techniques will adapt to the number of applications in the system and their structure. Once an application enters or leaves our system, the resource partitioner will be called to allocate/deallocate resources based on the new set of applications in the system as explained earlier.
To get actual run times of the applications, our effective scheduler published in [22] is utilized. The scheduler is based on memory-aware scheduling where the step of partitioning the memory budget

Table 3. System's assumed resources for different workloads.

Workload Combination	(# Processors, SPM Budget)
$(Lame, Osdemo)$	(3, 24KB), (4, 12KB) & (6, 64KB)
$(Lame, Cjpeg)$	(3, 64KB), (4, 32KB) & (6, 256KB)
$(Lame, Osdemo, Cjpeg)$	(4, 256kB), (6, 128KB) & (10, 512KB)
$(Lame, Enhance, Cjpeg, Osdemo)$	(4, 2MB), (8, 2MB) & (10, 4MB)

assigned to an application among the processor cores allocated to that application is integrated with the scheduling step. It was shown in [22] that this integrated approach improves over the traditional decoupled approaches that treat memory partitioning and scheduling as two independent tasks. This is mainly due to the fact that the appropriate configuration of a processor's scratch pad memory depends on the tasks scheduled on that processor.

To test the proposed resource partitioning heuristics, MPSoC systems under different workloads from the pool of the following embedded applications are utilized: $Lame$, $Cjpeg$, $Osdemo$, and $Enhance$. The systems were tested under the following workloads: $(Lame, Osdemo)$, $(Lame, Cjpeg)$, $(Lame, Osdemo, Cjpeg)$ and $(Lame, Enhance, Cjpeg, Osdemo)$. Each workload in a system was tested under different scenarios of arrival times to imitate a real life system where application can get to the system at any time and the resource partitioner is required to provide the necessary allocation/deallocation based on the application concurrently utilizing the system. Based on the applications in the workloads, our approach is tested in systems with different processing and memory budgets. The choice of system resources for a set of embedded applications is essential to test our proposed approach as too little or too many resources may not reflect the effectiveness of our techniques. Based on the nature of each embedded application, we came up with the system resources scenarios in Table 3 to effectively test the techniques for different workloads. The off-chip memory size is assumed to be unlimited, that is, it can hold all the data variables needed by the embedded application.

The results from our techniques and those in [31] for workloads: $(Lame, Osdemo)$, $(Lame, Cjpeg)$,$(Lame, Osdemo, Cjpeg)$ and $(Lame, Enhance, Cjpeg, Osdemo)$ are presented in Figure 7, 8, 9, and 10, respectively. The results represent the average cycle count of multiple runs for each workload with different start times under each set of system resources. Our techniques were able to reduce the cycle count in all cases compared to that in in [31] with a reduction range of 7 % to 16 % with an average cycle count reduction of 10.2 %. These results show the effectiveness of our techniques that effectively allocated the resources for different combinations of embedded applications under different system resources under different scenarios of arrival times and order of arrivals. Note in Figure 8 the importance of the memory requirement of an application on the results. The cycle count went up from 3 to 4 processors since the memory budget was reduced by 32KB that resulted in adverse effect on the run time of the $Lame$ application as the run time of $Lame$ is heavily reduced with memory budget close to 32 KB compared to that of 8 KB as shown in our scheduler results in [22]. This does not show in Figure 7 since the cycle count is dominated by that of $Osdemo$.

One of the main reasons for our improvements over those in [31] is that our techniques allocate the resources based on a deep analysis of the structure of each application reflected partially in the $elasticity$, PRF, and DP values.

6. CONCLUSION

This article presents effective techniques to resource partitioning among multiple applications on a multiprocessor system-on-chip.

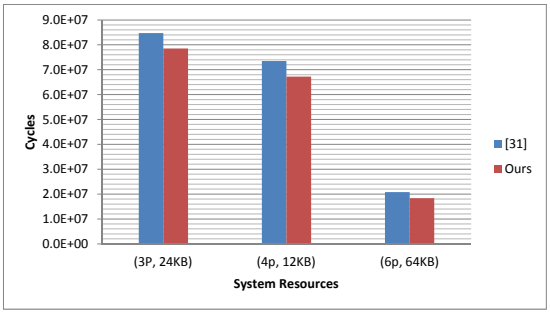

Fig. 7. Lame-Osdemo benchmarks

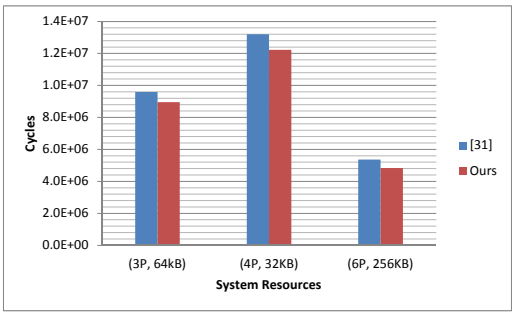

Fig. 8. Lame-Cjpeg benchmarks

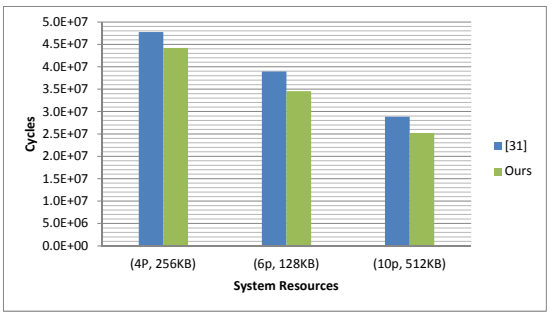

Fig. 9. Lame-Osdemo-Cjpeg benchmarks

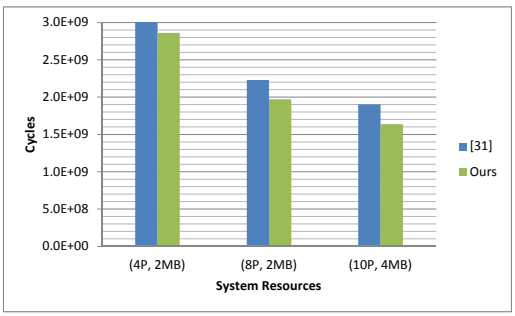

Fig. 10. Lame-Enhance-Osdemo-Cjpeg

Our techniques effectively divide the processor cores and the memory budget among competing applications based on the structure of each application. Applications are assumed to enter or leave the system at different times. Results on real life benchmarks show the effectiveness of our framework.

7. REFERENCES

[1] Z. Ma, C. Wong, S. Himpe, E. Delfosse, F. Catthoor, J. Vounckx, , and G. Deconinck, "Task concurrency analysis and exploration of visual texture decoder on a heterogeneous platform," in *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, 2003.

[2] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, 1999.

[3] L. Benini, D. Bertozzi, A. Guerri, and M. Milano, "Allocation and scheduling for mpsoc via decomposition and no-good generation," in *Proc. International Joint conferences on Artificial Intelligence (IJCAI)*, 2005.

[4] R. Neimann and P. Marwedel, "Hardware/software partitioning using integer programming," in *Proc. Design Automation and Test in Europe (DATE)*, 1996.

[5] K. S. Chatha and R. Vemuri, "Hardware-software partitioning and piplined scheduling of transformative applications," *IEEE Transactions on VLSI*, vol. 10, no. 3, 2002.

[6] P. Panda, N. Dutt, and A. Nicolau, "Memory issues in embedded systems-on-chip: optimization and exploration," *Kluwer Academics Publisher*, 1999.

[7] P. Panda, N. D. Dutt, and A. Nicolau, "On chip vs off chip memory: the data partitioning problem in embedded processor-based systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 5, no. 3, 2000.

[8] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 1, no. 1, 2002.

[9] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *Journal of Embedded Computing*, 2005.

[10] J. Sjodin and C. V. Platen, "Storage allocation for embedded processors," in *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2001.

[11] S.-R. Kuang, C.-Y. Chen, and R.-Z. Liao, "Partitioning and pipelined scheduling of embedded systems using integer linear programming," in *Proc. International Conference on Parallel and Distributed Systems (ICPADS)*, 2005.

[12] F. Angiolini, L. Benini, and A. Caprara, "Polynomial-time algorithm for on-chip scratchpad memory partitioning," in *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2003.

[13] S. Meftali, F. Gharsalli, F. Rousseau, and A. Jerraya, "An optimal memory allocation for application-specific multiprocessor system-on-chip," in *Proc. International Symposium on Systems Synthesis (ISSS)*, 2001.

[14] M. Kandemir, J. Ramanujam, and A. Choudhury, "Exploiting shared scratch pad memory space in embedded multiprocessor systems," in *Proc. Design Automation Conference (DAC)*, 2002.

[15] G. Yao, R. Pellizzoni, S. Bak, E. Betti, , and M. Caccamo, "Memory centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, pp. 1–35, 2012.

[16] S. Bakz, G. Yaoz, R. Pellizzoni, and M. Caccamoz, "Memory-aware scheduling of multicore task sets for real-time systems," in *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 300–309, 2012.

[17] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 4, 2010.

[18] V. G. Vaidya, P. Ranadive, and S. Sah, "Dynamic scheduler for multicore systems," in *2nd International Conference on Software Technology and Engineering(ICSTE)*, 2010.

[19] J. L. March, J. Sahuquillo, S. Petit, H. Hassan, and J. Duato, "Power-aware scheduling with effective task migration for real-time multicore embedded systems," *Concurrency and Computing: practice and experience*, 2012.

[20] J. Cong and B. Yuan, "Energy-efficient scheduling on heterogeneous multi-core architectures," in *The International Symposium on Low Power Electronics and Design (ISLPED)*, 2012.

[21] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for mpsoc architecture," in *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2006.

[22] H. Salamy and J. Ramanujam, "An effective solution to task scheduling and memory partitioning for multi-processor system-on-chip," *IEEE Transactions on Computer Aided Design*, vol. 28, no. 4, 2012.

[23] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proc. International Conference on Hardware-Software Codesign (CODES)*, 2002.

[24] M. Kandemir and N. Dutt, "Memory systems and compiler support for mpsoc architectures," *Multiprocessor Systems-on-Chips*, 2005.

[25] F. Sun, N. Jha, S. Ravi, and A. Raghnunathan, "Synthesis of appication-specific heterogeneous multiprocessor architectures using extensible processors," in *Proc. International Conference on VLSI Design*, 2005.

[26] J. Chen, C. Yang, T. Kuo, and C. Shih, "Energy-efficient real-time task scheduling in multiprocessor dvs systems," in *Proc. Asia and South Pacific Design Automation Conference*, 2007.

[27] A. K. Coskun, T. T. Rosing, K. A. Whisnant, and K. Gross, "Static and dynamic temperature-aware scheduling for multiprocessor socs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16(9), no. 1127 – 1140, 2008.

[28] R. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *Proc. 30th IEEE Real-Time Systems Symposium*, 2009.

[29] C. Chou and R. Marculescu, "Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29(1), no. 78 – 91, 2010.

[30] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. of IEEE International Symposium on Microarchitecture*, pp. 330–335, 1997.

[31] L. Xue, O. Ozturk, F. Li, M. Kandemir, and I. Kolcu, "Dynamic partitioning of processing and memory resources in embedded mpsoc architectures," in *Design, automation and test in Europe (DATE)*, 2006.

[32] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown., "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE 4th Annual Workshop on Workload Characterization*, 2001.