A Framework to Resource Partitioning and Task Scheduling of Multiple Applications on an MPSoC

Hassan Salamy*1, Olalekan Sopeju2

1, 2 Electrical Engineering-Computer Science, Texas State University
601 University Drive, San Marcos, TX, USA, 78666
*1hsalamy@txstate.edu; 2 olalekan@txstate.edu

Abstract- In the era of multicore dominance, multi-processor system-on-chips (MPSoCs) are now increasingly utilized as the appropriate platform in complex embedded systems. An MPSoC consists of multiple heterogeneous processing elements, a memory hierarchy, and input/output components linked together by an on-chip interconnect structure. Many embedded systems employ software-managed memories known as scratch-pad memories (SPM). Scratchpad memories, unlike caches, are software-controlled and hence the execution time of applications on such systems can be accurately predicted and controlled. Often multiple embedded applications are utilizing the system at the same time. The proper allocation of system resources to active applications and scheduling of applications tasks are two critical issues in reducing the computation times as well as the communication overhead. In this article, we present effective resource allocation techniques and a memory-aware task scheduling on an MPSoC with multiple embedded applications possibly entering and leaving the system at different times. The objective is to reduce the execution time of all the applications in the system. Results on real life benchmarks show the effectiveness of our techniques.

Keywords- MPSoC; Scheduling; Allocation; Scratchpad

I. INTRODUCTION

As the ability to extract more performance out of a single core is ever diminishing due to clock and power constraints, architectures with multiple cores became an attractive solution to achieve higher levels of performance to solve a broad range of problems from both high-end and low-end computing. Thanks to recent advances in architecture, VLSI and electronic design, multiprocessors system-on-chip (MPSoCs) that include a large number of different processing cores are now common for a variety of reasons. Generally speaking, an MPSoC consists of multiple heterogeneous processing elements (PEs), memory hierarchies, and I/O components interconnected by complex communication architectures. Such architectures provide the flexibility of simple design, high performance, and optimized energy consumption.

While embedded systems become increasingly complex, the processor-memory speed gap has continued to increase. Over the last several years, increase in memory access speed has failed to keep up with the increase in processor speed. This makes the memory access latency a major aspect in scheduling embedded applications on embedded systems. The increase in processor-memory speed gap is more of a problem in the case of MPSoCs due to the heavier contention on the network and the use of shared memories in some cases.

An MPSoC provides an attractive solution to the problems brought forth by increasing complexity and size of embedded applications. Execution time predictability is a critical issue for real-time embedded applications; this makes the use of data caches not suitable as a cache is hardware-controlled and hence it is hard to model the exact behaviour and to predict the execution time of programs. To alleviate such problems, many modern MPSoC systems use software-controlled memories known as scratchpad memories (SPMs).

An SPM is fully software-controlled and hence the execution time of an application on such memories can be predicted with accuracy. Unfortunately, scratchpad memories are expensive and hence they are usually of limited size and as a result not all the application data variables can be stored in the on-chip scratchpads. Many multi-processor system-on-chip models use a memory hierarchy with slow off-chip memory (DRAM) and fast on-chip scratchpad memories. Such hierarchy infers that the proper allocation of variables to the on-chip memory is an essential part in reducing the off-chip accesses. The computation time of a task on a processor depends on how much SPM is allocated to that processor as accessing an element from the off-chip memory is usually in the order of 100 times slower than accessing elements stored locally in the on-chip memory.

Often multiple embedded applications are utilizing the system at the same time. The proper allocation of system resources to active applications and scheduling of applications' tasks are two critical issues in reducing the overall computation time as well as the communication overhead. The problem of resource partitioning and task scheduling on MPSoCs is an NP-complete problem [9]. Traditionally at a single application level, the step of task scheduling is applied separately from the memory partitioning step. Such decoupled techniques may not result in a global minimum in terms of minimizing the computation time of the whole application and consequently that of the overall applications utilizing the system. The appropriate configuration of a processor's scratch pad memory depends on the tasks scheduled on that processor. Therefore, the integration of those two steps is critical to improve the performance.

This article presents an effective framework for resource partitioning and memory-aware task scheduling for multiprocessor system-on-chip with multiple concurrent applications. The applications utilizing the MPSoC can arrive and leave the system at different times. The developed solution is a two-level solution where in the first level the available resources are dynamically partitioned among the available applications, and in the second level, the tasks of a single application are then scheduled based on an integrated approach of task scheduling and memory partitioning.

II. RELATED WORK

Benini et al. [4] solved the scheduling problem using constraint programming and the memory partitioning problem using integer linear programming. The authors argued why these two choices fit the two problems the best. Kwok and Ahmed [9] presented a comparison among algorithms for scheduling task graphs onto a set of homogeneous processors on a diverse set of benchmarks to provide a fair evaluation of each heuristic based on a set of assumptions. De Micheli et al. [11] studied the mapping and scheduling problem onto a set of processing elements as a hardware/software codesign. A tool for hardware-software partitioning and pipelined scheduling based on a branch and bound algorithm was presented in [8]. Their objective was to minimize the initiation time, number of pipeline stages, and memory requirements. Their solution is based on integer linear programming formulation.

Panda et al. [12, 13] presented a comprehensive allocation technique for scratchpad memories on uniprocessor to maximally utilize the available SPM memories to decrease the programs execution times. Optimal ILP formulations for memory allocation for scratch-pad memories were presented in [3]. An ILP formulation to the SPM allocation problem to reduce the code size was presented in [14]. Steinke et al. [15] formulated the same problem with the objective to minimize the energy consumption. Angiolini et al. [1] optimally solved the problem of mapping memory locations to SPM locations using dynamic programming.

Blagodurov et al. [5] presented a contention-aware scheduling algorithm on multicore systems. Vaidya et al. [18] proposed a dynamic scheduling algorithm in which the scheduler resides on all cores of a multi-core processor and accesses a shared Task Data Structure (TDS) to pick up ready-to-execute tasks. Suhendra et al. [16] studied the problem of integrating task scheduling and memory partitioning on a heterogeneous multiprocessor system on chip with scratch pad memory. This is the only paper, to the best of our knowledge, which addressed this problem in an integrated approach for MPSoC. They formulated this problem as an integer linear problem (ILP) with the inclusion of pipelining. ILP solutions require long computation time for large applications. Unlike our approach, they assumed only one application utilizing the system at the same time. A technique to effectively divide system resources among competing applications is presented in [19].

III. ARCHITECTURAL MODEL AND OUR APPROACH

The architectural model assumed in this article consists of a number of processing cores and a scratchpad budget that is shared concurrently by all processors. A large off-chip memory of unlimited size is assumed. Each executing application in our model is mapped to a set of resources. The number of cores mapped to an application depends on the structure of the application and the degree of potential parallelism. A simple view of our model is presented in Figure 1, where a set of processors are mapped to each application and then the tasks of the application are scheduled on the available processors mapped to this application. As mentioned earlier, the schedule is SPM budget-aware, i.e., the schedule considers the dynamic execution time of a task based on the processor to which it is mapped as well as the SPM memory budget assigned to that processor.

**Problem Definition:** Given (i) an MPSoC architectural model of a set of processors, on-chip SPM budget, and large off-chip memory and (ii) a set of applications to be executed at this architecture with possibly unknown start times, divide the processor cores and the SPM budget among all concurrently executing applications then divide the resources mapped to each application among its tasks to minimize the execution times of the applications in the system.
The question is how to divide the available processors and memory budget among different applications at different times. The number of processors and the SPM memory budget allocated to an application depends on the nature of the application. More processors will usually be allocated to an application that has larger potential for parallelism compared to an application of a more sequential nature.

Our framework consists of three major components: (i) the profiler, (ii) the resource partitioner, and (iii) the scheduler. The profiler is responsible of profiling an application. Once the profiler receives a new application, it will extract a set of information that will be later used by the resource partitioner. This information will then be used by the resource partitioner to allocate processor cores and SPM budget to such application based on its nature and structure as well as the number of applications currently using the system. Then the scheduler will schedule the tasks of each application on the processor cores and under the SPM budget allocated by the partitioner to this application.

The number of resources assigned to a certain application is based on the structure of the application. For instance, it makes sense to assign more processors to a highly parallel application in nature so that more tasks can run in parallel. Moreover, typically a memory-intensive application should be assigned a bigger scratchpad memory budget compared to an application that requires less memory. Note that a memory-intensive application is one in which memory accesses are a significant percentage of the execution time; whereas, an application where most of its computation time is consumed in big loops rather than memory accesses is classified as non-memory-intensive.

The processor cores and the scratchpad memory budget are shared resources and therefore they need to be carefully partitioned among the competing applications in the system. Once this partitioning is over, the scheduler will schedule the tasks of a certain application on the processors mapped to that application under the memory budget allocated to it. Notice that as an application enters or leaves our system, the resources will be redistributed and the tasks will be rescheduled and thus the framework is dynamic based on the applications using the system at a certain point in time. The profiler, the resource partitioner, and the scheduler are further discussed in the next three sections.

IV. THE PROFILER

Once the profiler receives a new application to be scheduled on the MPSoC, it analyzes its structure and extracts important information that will be sent to the resource partitioner and the scheduler as annotations. For example, the task dependence graph is a very basic part that is needed by the resource partitioner and the scheduler. A task dependence graph (TDG) is a directed acyclic graph with weighted edges where each task in the embedded application is represented by a vertex. An edge between two tasks, say Ti and Tj in the TDG, represents some kind of a scheduling order due to data dependence. Thus a certain processor cannot start executing task Tj unless all the necessary data communication is performed. We profile to identify the key computation blocks which will represent the tasks in the task dependence graph. The communication between tasks is estimated based on the control and data flow information. Shared variables between different tasks will contribute to the communication cost. These variables will not be stored in the SPM in our case. Hence, a task graph is constructed with a vertex for each task and weighted edges to represent communication costs between tasks. We assume that tasks mapped to the same processor exhibit zero communication cost.

Another important piece of information extracted by the profiler is a set of \[ \{ \text{Max}_{ij}, \text{Avg}_{ij}, \text{Min}_{ij} \} \] values for each of the tasks. The \( \text{Min}_{ij} \) of a task represents the computation time for task \( Ti \) on processor \( Pj \) assuming all of the available SPM budget is assigned to \( Pj \). The \( \text{Avg}_{ij} \) represents the computation time for task \( Ti \) on processor \( Pj \) assuming \( 1/n \) of the available SPM budget is assigned to \( Pj \) where \( n \) is the number of processors. And the \( \text{Max}_{ij} \) represents the computation time for task \( Ti \) on processor \( Pj \) assuming no SPM budget is assigned to \( Pj \) which means that all the data variables will be accessed from the slow off-chip memory. In summary, the profiling by the profiler is intended to: (i) divide each application into computation blocks referred to as tasks; (ii) find the computation times for each task on each available processor in processor cycles; (iii) find the number of variables; (iv) the number of times each variable is used, \( \text{freq} \); and (v) the size in bytes for each variable in the current application.

V. THE RESOURCE PARTITIONER

The resource partitioner is responsible of dividing the available resources among the concurrently executing applications. Given a system of \( p \) processing cores and an SPM budget of size \( m \) and \( k \) executing applications, divide the available resources among the applications so that the schedule times of the applications are minimized. The partitioner will receive the profiling information from the profiler and then decide how much SPM budget and how many processor cores should be assigned to each application. The scheduler will most probably assign fewer resources to an application than the optimal resources required as we are assuming a system of limited resources and with more than one concurrently executing application competing for the available resources.

Once the partitioner receives a new application, it will read its structure and computes the level of parallelism based on the structure of the TDG to determine the degree of benefit from assigning more processors to such an application. Also, based on the elasticity value, the scheduler will be able to figure out how much this application benefits from more SPM budget.
define *elasticity* of a task in an application as the extent to which this task can benefit from a bigger SPM budget. Although it can be defined in different ways, we define *elasticity* as a value between 0 and 1 to represent the extent to which the computation cost of a task (and consequently an application) on $P_i$ may decrease as the SPM budget of $P_i$ is increased from the current budget to *size* where *size* is the maximum amount of SPM budget available in our model. Equation 1 defines *elasticity* of task $T_i$ on processor $P_i$ where $Cur$ is the entire computation time of the task under the current memory allocation among the applications. In the resource partitioner heuristics, the $Cur$ value is based on equally partitioning the remaining SPM budget among the applications in the system who have not been allocated a memory budget yet in the heuristic in Figure 2 to be discussed later on. The *elasticity* of a task $T_i$ is a measure of the room for computation time reduction of $T_i$ with more SPM budget.

$$elasticity(T_i) = \frac{Cur_i - Min_i}{Cur_i}$$  

A bigger value of *elasticity* means that the computation time of $T_i$ is more amenable for reduction with bigger SPM. Note that $elasticity(T_i)$ is a dynamic value since the current computation time of $T_i$, $Cur_i$, may change as the SPM budget distribution among the applications progresses.

The SPM budget in the system will be partitioned among the concurrently executing applications in the system so that the computation times of the applications in the system are minimized. If the system does not have enough SPM budget to cover all the variables in the system, then each application will receive an SPM budget proportional to its predicted reduction fraction *PRF* value. This concept will be further discussed in the Subsection A. The *PRF* of an application is the average elasticity of its $t$ tasks among all the $p$ processors and is defined in Equation 2.

$$PRF(APP) = 1/p \cdot \sum_{i} \sum_{j} \frac{elasticity(T_{ij})}{t}$$  

As mentioned earlier, the number of processor cores mapped to a certain application is directly related to the degree of parallelism (DP) of such an application. The degree of parallelism is defined in Equation 3. A larger DP value means that the application’s structure has higher degree of potential parallelism whereas a small DP value means that the application is more sequential in nature. The degree of parallelism of an application will be extracted from the structure of its task dependence graph. We define the DP in a way to reflect the degree of parallelism between the tasks in the TDG. Two independent tasks can be executed in parallel on two different processors (if possible) whereas two dependent tasks, say $A$ and $B$ ($B$ depends on $A$), must be executed sequentially as task $B$ needs to wait for some input information from task $A$.

Given a TDG with $t$ tasks. A pair of tasks ($T_i$, $T_j$) can be run in parallel if there is no path in the TDG between $T_i$ and $T_j$ and thus they are independent. The number of such pairs in an application loosely reflects its degree of parallelism. The DP value will be used by the resource partitioner as a guideline when dividing the processors in the system among the concurrently executing applications.

Given a TDG, find all the paths between a dummy start node and a dummy end node. The dummy start node $S$ is a node with an outgoing edge to each task (node) in the TDG with zero incoming edges and the dummy end node $E$ is such that there is an edge between this node and all the tasks in the TDG with no outgoing edges. Two paths are distinct if they have at least one task not in common. For any two paths $p_i$ and $p_j$, find all the pairs ($T_i \in p_i, T_j \in p_j$) such that $T_i$ and $T_j$ can be run in parallel. As shown in Equation 3, the degree of parallelism is made up of the sum of two components. The first component represents the number of distinct paths in the TDG. Intuitively speaking, an application with a TDG of large number of distinct paths is more parallel in nature and thus it can benefit more from additional processors. However, the number of distinct paths is not enough to represent the degree of parallelism in an application.

An application with two more balanced paths will benefit more from two processors compared to an application with two unbalanced paths as in the latter case one of the processors might be idle for longer periods of time.

$$DP(APP) = paths_i + \frac{pairs}{paths}$$  

To take this observation into consideration, we added the second part, $pairs$, to our DP definition in Equation 3. The value *pairs* is the number of pairs that can be executed in parallel. Although the *pairs* value represents the number of parallel paths, that number is mostly an exaggeration of the number of actual tasks that can run in parallel. To clarify this point, assume that we have the following pairs of tasks between two paths $p_i$ and $p_j$ ($T_1 \in p_i, T_j \in p_j$), ($T_1, T_3$), ($T_1, T_2$), and ($T_5, T_3$). What this means is that task $T_1$ that belongs to path $p_i$ can run in parallel with task $T_3, T_4$ or $T_5$. Since $T_3, T_4$, and $T_5$ belong to the same path $p_j$, they are dependent and should be run sequentially. Thus, $T_1$ can be run in parallel with only one of the tasks at a time. As a result, the *pairs* variable in our DP definition is in no way a reflection of how many pairs can run in parallel at the same time but rather a reflection of how much the paths are balanced in a certain TDG.
The processors will be divided among all the applications in the system so that the overall run times of the applications in the system are minimized. As will be discussed next, each application will receive a number of processors proportional to its number of distinct paths and its degree of parallelism DP.

A. Our Resource Partitioning Heuristic

The main task of our resource partitioning heuristic is to partition the resources among different applications in the system to minimize the overall schedule time. The resource partitioner heuristic is mainly made up of two major parts, the SPM partitioner (see Figure 2), and the processors partitioner (see Figure 3). The job of the partitioner is to partition the available resources among the concurrently executing applications in an effective way.

In the SPM partitioner case, the heuristic operates in two steps. In the first step, it determines the memory requirements of the applications from the annotations sent by the profiler and then in the second step, it allocates the available SPM memory space to the applications. The SPM partitioner takes the SPM size (m) and the number of applications concurrently running (n) as an input. The heuristic starts by creating a list of applications in decreasing order of their PRF values. Based on the data requirements of each application obtained by the function call SPM_requested(i), the heuristic calculates the total SPM budget, SPM, needed to meet the demands of all the available applications. If this SPM value is less than or equal to the available SPM budget (m), then each application takes all the SPM space it requested. In this case, we do not have a space partitioning problem.

However, if the total requested SPM is larger than the available SPM budget, the SPM partitioner tries to distribute the available SPM among all the concurrently executing applications in an effective way. It does that in Lines 12–21 in the heuristic by allocating to each application an SPM budget proportional to its requested size under the criterion that an application with higher PRF will receive an SPM budget closer to it requested budget compared to an application with smaller PRF value. Note that the overall objective is a short schedule of each application in the system.

The processor partitioner heuristic in Figure 3 works in a similar fashion to the SPM partitioner. The processor partitioner takes the number of processors in the system (p) and the number of applications concurrently running (n) as inputs. The heuristic starts by creating a list of applications in decreasing order of their potential level of parallelism in Line (5). We found that the expression in Line (5) produces better results than just using the DP values as the heuristic starts by creating a list of applications in decreasing order of their PRF values. Based on the data requirements of each application obtained by the function call SPM_requested(i), the heuristic calculates the total SPM budget, SPM, needed to meet the demands of all the available applications. If this SPM value is less than or equal to the available SPM budget (m), then each application takes all the SPM space it requested. In this case, we do not have a space partitioning problem.

However, if the total requested SPM is larger than the available SPM budget, the SPM partitioner tries to distribute the available SPM among all the concurrently executing applications in an effective way. It does that in Lines 12–21 in the heuristic by allocating to each application an SPM budget proportional to its requested size under the criterion that an application with higher PRF will receive an SPM budget closer to it requested budget compared to an application with smaller PRF value. Note that the overall objective is a short schedule of each application in the system.

The processor partitioner heuristic in Figure 3 works in a similar fashion to the SPM partitioner. The processor partitioner takes the number of processors in the system (p) and the number of applications concurrently running (n) as inputs. The heuristic starts by creating a list of applications in decreasing order of their potential level of parallelism in Line (5). We found that the expression in Line (5) produces better results than just using the DP values as the heuristic starts by creating a list of applications in decreasing order of their PRF values. Based on the data requirements of each application obtained by the function call SPM_requested(i), the heuristic calculates the total SPM budget, SPM, needed to meet the demands of all the available applications. If this SPM value is less than or equal to the available SPM budget (m), then each application takes all the SPM space it requested. In this case, we do not have a space partitioning problem.

However, if the total requested SPM is larger than the available SPM budget, the SPM partitioner tries to distribute the available SPM among all the concurrently executing applications in an effective way. It does that in Lines 12–21 in the heuristic by allocating to each application an SPM budget proportional to its requested size under the criterion that an application with higher PRF will receive an SPM budget closer to it requested budget compared to an application with smaller PRF value. Note that the overall objective is a short schedule of each application in the system.

A good value for the α expression in DP is an exaggeration of the number of actual tasks that can run in parallel. It then sums up the total number of requested processors by all the applications, namely, the value Paths. For simplicity, we assume that the optimal number of processors for an application is equal to the number of distinct paths in its corresponding TDG. If the total number of processors requested (Paths) is less than or equal the number of available processors, then each application will be granted a number of processors equal to its number of distinct paths.

```
SPM_Partitioner(n, m)
1. L = list of applications 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 not empty
13.   i = First application in L.
14.   Temp = UpperBound((i, PRF(i))* m)
15.   SPM_received(i) = MIN(SPM_request(i), Temp)
16.   m = m - SPM_received(i)
17.   Remove i from L.
18.   Recompute the Min values of remaining applications based on the updated m value.
19.   Recompute PRF and the Total_PRF values.
20.  End While
```

Fig. 2 Our SPM partitioning heuristic.
applications with the same number of paths may end up receiving different number of processors based on the DP values. The application with the higher DP value will probably receive more processors compared to the one with smaller DP value. This is fair as the DP value loosely reflects the degree of balance between all the paths of an application. Also a more balanced-path application will more effectively utilize the processors as more tasks can be executed in parallel.

\[
\text{Processor\_Partitioner}(n, p)
\]

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 of applications in decreasing order of \((1 + \alpha DP(i)) \times \text{path}(i)\)
6. For i = 1 to n
7. Path = Path + path(i)
8. Path\_DP = Path\_DP + \((1 + \alpha DP(i)) \times \text{path}(i)\)
9. End For
10. If (Path \leq 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 = \text{UpperBound}\((1 + \alpha DP(i)) \times \text{path}(i) / \text{Path\_DP} \times p\)
18. Processor\_received(i) = \text{MIN}(\text{path}(i), \text{Temp})
19. Path\_DP = Path\_DP - \((1 + \alpha DP(i)) \times \text{path}(i)\)
20. p = p - Processor\_received(i)
21. Remove i from L.
22. End While

Fig. 3 Our processor partitioning heuristic.

VI. OUR SCHEDULER

The scheduler receives all the applications with their structure, assigned number of cores, and SPM budget from the resource partitioner. For each application \(i\), the scheduler will receive its task dependence graph, TDG\(_i\), the cores mapped to this application \(P_i\), and its SPM budget \(S_i\) as \(\text{APP}_i(\text{TDG}_i, P_i, S_i)\). The scheduler then schedules the tasks of each application \(i\) on the resources assigned to it; i.e., at this stage, the tasks will be mapped to the available processors \(P_i\) and the SPM budget \(S_i\) will be partitioned among the processors so that the overall computation time is minimized.

Most works so far have treated task scheduling and memory partitioning as two decoupled steps that are performed independently. Given a set of tasks and an MPSoC model with a certain amount of available scratchpad memory budget, tasks are usually scheduled on the processors and then memory is partitioned among used processors. In this aspect, those two steps are performed independently. However, the configuration of a processor’s scratchpad memory is highly dependent on the tasks scheduled on this processor. As a result, task scheduling and memory partitioning are inter-dependent on each other and they should be integrated in one step in order to get high quality schedules.

Unlike current approaches that have studied the task scheduling and memory partitioning problems as two separate problems, we solve these two problems in an integrated fashion. This section presents an effective memory-aware scheduler of an embedded application on the resources allocated to it based on our integrated concept where the private on-chip SPM memory budget allocated to a processor is decided as tasks are mapped to this processor.

The computation time of a task depends on the processor to which it is mapped as well as on the SPM memory available for that task. Therefore, task scheduling should take into consideration the varying computation time of a task based on the processor and on the SPM budget. Considering static computation time, meaning that the computation time is fixed from the scheduler point of view, may limit the quality of the schedule.

A good heuristic for task scheduling and memory partitioning should take into consideration the varying execution time of a task throughout the process of building the schedule. This varying execution time is the result of the dynamic SPM budget assignment to processors throughout the course of our heuristic. Using profiling of the tasks in the embedded application, \(\text{Min}\), \(\text{Avg}\), and \(\text{Max}\) values (defined earlier) are calculated for each task on each of the available processors.

Our scheduler heuristic in Figure 4 starts with sorting the tasks in increasing order of the \textit{ASAP} values in a list \(L_i\), with the smaller \textit{ALAP} value as a tie-break for tasks with equal \textit{ASAP} values. For each task, following the \textit{ASAP} sort, we evaluate the best processor to assign this task so that the overall computation time is minimally increased.
The minimum start time of a task $T_i$, on processor $P_j$, $Start\_time(T_i, P_j)$, is equal to the maximum of the end time of processor $P_j$, $End\_time(P_j)$, and the maximum end time of all its parent tasks, $Max(End\_time_{T_j=PPEC(T_j)}(T_j))$, plus the corresponding communication time (see Equation 7). Two dependent tasks mapped to the same processor will have zero communication cost. In general, task $T_i$ will be scheduled on the processor $P_j$ corresponding to the minimum additional overhead time in the schedule.

However, $T_i$ may be scheduled on a processor $P_k$ of higher overhead time provided that the predicted end computation time ($PEC(P_k)$, defined by us in Equation 4) of this processor is at least $\delta$ % less than that of $P_j$ (Line 17 of our heuristic in Figure 4). $P_j$ in Line 17 of in Figure 4 represents the processor corresponding to the current $min$ value. The $min$ value in our heuristic is the minimum additional overhead time that will be added to the schedule based on a certain task scheduling decision. We choose $\delta$ of 10 in our experimental evaluations. This $PEC(P_j)$ value is a guide to the scheduler of how much this overhead time may decrease with additional SPM memory transfers in future steps if $T_i$ is mapped to $P_k$. $PEC$ is an estimate of how much the end time of processor $P_k$ will be if more SPM budget is assigned to it.

<table>
<thead>
<tr>
<th>Memory-aware Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Read the TDG from the Profiler based on dependence analysis and communication costs.</td>
</tr>
<tr>
<td>2. Read all the processors to this application from the Resource Partitioner.</td>
</tr>
<tr>
<td>3. Read the SPM budgets allocated to this application from the Resource Partitioner.</td>
</tr>
<tr>
<td>4. Divide the SPM memory equally between the processors.</td>
</tr>
<tr>
<td>5. For each task $T_i$ and processor $P_j$, read from the Profiler the following:</td>
</tr>
<tr>
<td>6. (i) Minimum computation time on $P_j$, $Min_i$.</td>
</tr>
<tr>
<td>7. (ii) Maximum computation time on $P_j$, $Max_i$.</td>
</tr>
<tr>
<td>8. (iii) Average computation time on $P_j$, $Avg_i$.</td>
</tr>
<tr>
<td>9. Find ASAP for all the tasks based on $Avg$ values.</td>
</tr>
<tr>
<td>10. $L_1$ = List of tasks in increasing order of ASAP.</td>
</tr>
<tr>
<td>11. While ($L_1$ not empty) do:</td>
</tr>
<tr>
<td>12. Get the first task $T_f$ from $L_1$.</td>
</tr>
<tr>
<td>13. For each processor $P_k$:</td>
</tr>
<tr>
<td>14. Calculate the elasticity and $PEC$ of $P_k$ if $T_f$ is mapped to $P_k$.</td>
</tr>
<tr>
<td>15. Find the minimum start time of $T_f$ on $P_k$.</td>
</tr>
<tr>
<td>16. Find $END_time(P_k)$ if $T_f$ is mapped to $P_k$.</td>
</tr>
<tr>
<td>17. if (($END_time(P_k) &lt; min &amp;&amp; PEC(P_k) \geq (1 - \delta) \times PEC(P_j)$)</td>
</tr>
<tr>
<td>18. $min = END_time(P_k)$.</td>
</tr>
<tr>
<td>19. else if ($END_time(P_k) = min$)</td>
</tr>
<tr>
<td>20. $min = END_time$ of processor with the higher elasticity.</td>
</tr>
<tr>
<td>21. End For</td>
</tr>
<tr>
<td>22. Assign $T_f$ to $P_k$ corresponding to $min$.</td>
</tr>
<tr>
<td>23. Delete $T_f$ from $L_1$.</td>
</tr>
<tr>
<td>24. Call Balance().</td>
</tr>
<tr>
<td>25. End While</td>
</tr>
<tr>
<td>26. For $i = 1$ to $j$ do:</td>
</tr>
<tr>
<td>27. Call Balance().</td>
</tr>
</tbody>
</table>

Fig. 4 Our memory-aware scheduler.

The $PEC$ of a processor is closely related to the $elasticity$ (defined in Section V) of the tasks scheduled on that processor. $Cur$ value in the $elasticity$ and the $PEC$ is the computation time of the task under the current memory budget. The $PEC$ value provides the flexible essence of our scheduler as at each step it looks beyond the current SPM budgets distribution in its task mapping decision to an estimate of future distributions in future steps. In our heuristic, if assigning task $T_i$ to either of two different processors will have the same increase value in the overall schedule time, we schedule $T_i$ on the processor with the higher $elasticity$ under the current SPM budget. The $elasticity$ of a processor is the average value of the $elasticity$ of the tasks scheduled on this processor.

$$PEC(T_i) = End\_time(T_i) - \sum_{T_j \in T_i} \left( Cur(T_j) - \frac{Cur(T_j)}{1 + elasticity(T_j)} \right)$$

After scheduling any task, we try to balance the schedule in a way to decrease the total computation time (see Figure 5). We do so by changing the SPM budget for each processor to reach a better balance. We start by trying to reduce the
computation time of tasks on processor $P_j$ with maximum end time so far. This can be done by transferring an $\alpha$% of the memory budget, $\text{Mem}_j$, corresponding to processor $P_j$ with the minimum (End_time * elasticity) and such that $\text{End\_time}(P_j) < \text{End\_time}(P_k)$ and assigning it to processor $P_k$. Doing so will probably decrease the end time of processor $P_k$ and in the same time increase the end time of processor $P_j$. Considering processor $P_j$ to be of low total elasticity, reducing its memory budget will result in minimal increase in its End_Time. We do memory transfer $\alpha$% at a time as long as $\text{End\_time}(P_j) < \text{End\_time}(P_k)$. Based on tuning, we choose an $\alpha$ equals to 10 in our experiments.

After any SPM memory budget redistribution among different processors, the Recompute() subroutine in Figure 6 will be invoked to recompute the start time, computation time, and end time of tasks $T_i$ referred to, respectively, as Start_time($T_i$), End_time($T_i$), and Time($T_i$). First a Gain value, $\text{Gain}(T_i, \text{Mem}_j)$, is computed for $T_i$ with the newly budget SPM memory, $\text{Mem}_j$, assigned to the processor to which $T_i$ is mapped. This Gain value in Equation 6, represents the execution cycles reduced due to allocating variables of $T_i$ to $\text{Mem}_j$, following the increasing order byte/ freq of the data variables where $\text{byte}_i$ is the size of the variable $V_i$ and $\text{Freq}_i$ is the number of times $V_i$ is accessed. In Equation 6, $\beta_1$ is the cost of accessing a variable from the off-chip memory and $\beta_2$ is the cost of the SPM access. This is a simple data allocation technique that we adopted in our heuristic. The new computation time of $T_i$, Time($T_i, \text{Mem}_j$), is the time taken to execute $T_i$ where $\text{Time}(T_i, \text{Mem}_j) = \text{End\_time}(T_i) + \text{Comm\_time}(T_i, \text{Mem}_j)$. The end time of each processor is thus the end time of the last task assigned to this processor (Equation 9).

$$\text{Time}(T_i, \text{Mem}_j) = \text{Time}(T_i, 0) - \text{Gain}(T_i, \text{Mem}_j)$$

$$\text{Gain}(T_i, \text{Mem}_j) = \sum_{i \in T_j, i \in \text{Mem}_j} ((\beta_1 - \beta_2) * \text{freq}_i).$$

$$\text{Start\_time}(T_i, P_j) = \text{Max}(\text{End\_time}(T_i, \text{Parent}(T_i)), \text{End\_time}(P_j))$$

$$+ \text{Comm\_time}(T_i, \text{Parent}(T_i), \text{End\_time}(P_j))$$

$$\text{End\_time}(T_i) = \text{Start\_time}(T_i, P_j) + \text{Time}(T_i, \text{Mem}_j)$$

$$\text{End\_time}(P_j) = \text{Max}(\text{End\_time}(P_j), \text{End\_time}(P))$$

After all the tasks are scheduled, we call the Balance() procedure in Figure 5, to try to further reduce the schedule time through reducing the end time of the processor with the largest end time by performing memory transfers. At this point, we tune the Balance() procedure so that it allows the last memory transfer between $P_i$ and $P_j$ that will result in $\text{End\_time}(P_j) > \text{End\_time}(P_i)$ We run Balance() $t$ times where $t$ is the number of tasks in the TDG. It is hard to find the best number of times to do so but our results showed the using the number of tasks improved the results in some cases. Notice that if a processor ends up with no scheduled tasks, then the SPM budget for such processor will be distributed among other processors using the Balance() procedure to reduce the schedule time the most.

### Balance()

1. $P_i$ = processor with maximum end time, End_time($P_i$).
2. $P_j$ = processor with minimum End_time($P_j$)*elasticity.
3. while (End_time($P_j$) < End_time($P_i$)) do:
   4. $\text{Mem}_k$ = $\text{Mem}_e + \alpha \% \text{Mem}_j$.
   5. $\text{Mem}_j$ = $\text{Mem}_j - \alpha \% \text{Mem}_j$.
   6. Recompute().
4. if (End_time($P_i$) \leq End_time($P_j$)).
   8. Perform the memory update.
5. End while
6. Recompute().

### Recompute()

1. Following the ASAP sort of scheduled tasks $T_i$ and the new SPM budget distribution:
2. Recompute $\text{time}(T_i, \text{Mem}_j)$.
3. Recompute $\text{Gain}(T_i, \text{Mem}_j)$.
4. Recompute $\text{Start\_time}(T_i, P_j)$ where $T_i$ is
In the case of multiple applications executing concurrently on an MPSoC, the important question is what happens when the system receives a new application or one application finishes executing and leaves the systems. As mentioned earlier, when a new application enters the system, the resource partitioner will repartition the resources among all the executing applications guided by the level of parallelism of an application and its PRF value. At this point, the scheduler will receive a new set of information about the applications in the system as well as the new application. The unscheduled tasks of the applications already in the system under the previous resource allocation need to be rescheduled according to the new resource allocation.

Some of the resources that will be assigned to the new application will come from the unused cores and SPM budget whereas others will come from resources already assigned to some other already executing applications. Applications may lose some processor cores and a part of the SPM budget assigned to them under the previous resource partitioning prior to the arrival of the new application. The heuristic, however, will wait for all the tasks currently executing to finish before deallocating any of their resources; i.e., preemption is not allowed. However, based on the outcome of the resource partitioning, we will assign to the new application available resources at this point and only postpone the assignment of other resources currently used by some tasks till they all finish executing. The rest of the schedule will be generated based on the new resources assignment and on the updated TDG for each application. The updated TDG of a certain application is the TDG including only non-scheduled tasks. An equivalent scenario will occur when an application finishes executing. The resources that were assigned to this application will be distributed, if needed, among other still executing applications. Our complete heuristic for the resource partitioning and memory-aware task scheduling is presented in Figure 7.

Showing the effectiveness of our techniques is not straightforward and hence we divided testing of our techniques into three different parts; (i) testing the scheduler in Section VI, (ii) testing the resource partitioner in Section V, and (iii) testing our whole allocation and scheduling heuristic.

The key computation blocks of each application are identified through profiling. Each computation block will be used as a task (vertex) in our TDG. Dependencies between different blocks will be represented as edges between tasks in the TDG. We divided the application into tasks or basic blocks. We then used the control/data flow information to find the dependencies among tasks and to come up with an estimate of the communication cost. We used SimpleScalar architectural simulation to profile the used benchmarks [2]. Through an instrumented version of the SimpleScalar profiler, we extract the profiling data such as variable sizes and access frequencies and execution time of each task. The MPSoC architecture used is similar to the one in Figure 1. As discussed in Section IV, the profiling is intended to (i) divide each application into computation blocks referred to as tasks, (ii) find the computation times (Min, Avg, Max) for each task on each available processor in processor cycles, (iii) find the number of variables, (iv) the number of times each variable is used, freq, and (v) the size in bytes for each variable in the current application. We used the following real life programs from [17], Mediabench and MiBench, [7, 10] enhance, lame, osdemo, and cjpeg as test benchmarks. The characteristics of our benchmarks are shown in Table I.

In the first set of experiments, we tested the scheduler part of our framework discussed in Section VI. Recall that the scheduler is concerned with a single application as it is assumed that it received the proper SPM and processor cores assigned to this application from the resource partitioner. We implemented four approaches to fairly show the effectiveness of our scheduler heuristic namely,

- decoupled task scheduling and memory partitioning assuming equally partitioned SPM among all available processors TSPM_EQUAL;
- decoupled task scheduling and memory partitioning with SPM partitioned among different processors with any ratio, TSPM_ANY; In this case, we partition the SPM in ratios based on the structure of the embedded systems. A more memory intensive application will receive more SPM memory.
our integrated task scheduling and memory partitioning scheduler heuristic described in Section VI, \textit{TSMP\_INTEG};

- the optimal solution based on the ILP formulation in [16], \textit{ILP} using the \textit{CPLEX ILP} solver [6]; and

- Our integrated approach with pipeling, \textit{TSMP\_PIPE}.

**TABLE I CHARACTERISTICS OF OUR BENCHMARKS.**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># variables</th>
<th>#tasks</th>
<th>Var size (Kbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lame</td>
<td>128</td>
<td>4</td>
<td>294.83</td>
</tr>
<tr>
<td>Osdemo</td>
<td>46</td>
<td>7</td>
<td>78.64</td>
</tr>
<tr>
<td>Enhance</td>
<td>44</td>
<td>6</td>
<td>7192.35</td>
</tr>
<tr>
<td>Cjpeg</td>
<td>20</td>
<td>5</td>
<td>690.31</td>
</tr>
</tbody>
</table>

We tested the following benchmarks, \textit{enhance}, \textit{lame}, \textit{osdemo}, and \textit{cjpeg} assuming a multiprocessor system-on-chip of two processors and a scratch pad memory of size that varies between 4KB and 4 MB. We assume a 100 cycle latency for off-chip memory access compared to 1 cycle latency for the SPM on-chip memory. We tested each of our benchmarks under three SPM budgets chosen based on the size of the benchmark. The choice of SPM sizes for each benchmark is essential as too little SPM or too much SPM for a certain embedded application may not reflect the effectiveness of our heuristic. 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 first three columns in Figures 8–11 shows the comparison between \textit{TSMP\_EQ}, \textit{TSMP\_ANY}, and \textit{TSMP\_INTEG}. All of these results are based on $\alpha$ value of 10 meaning that 10% of SPM memory is being transferred between two processors at a time in the \textit{Balance()} procedure. The improvement greatly depends on the structure of the embedded application. \textit{TSMP\_ANY} improved over \textit{TSMP\_EQ} from little improvement close to 0% to dramatic improvement of 47% (7% on average). Such improvements show that static memory allocation, that is, partitioning the SPM budget equally among the processors limits the effectiveness of SPM memories as it does not consider the characteristics of the tasks assigned to a processor in its memory partitioning decision.
Our integrated approach TSMP_INTEG further improved the results over the decoupled approach, TSMP_ANY. TSMP_INTEG improved over TSMP_ANY from little improvement close to 0% in some cases to dramatic improvement of 22% (7.9% on average). This improvement is due to the guidance that our integrated approach uses to partition the memory based on the fact that the SPM configuration of a certain processor depends on the tasks mapped to that processor.

In order to show the effectiveness of our memory-aware scheduler, TSMP_INTEG, we compared it to an optimal integer linear formulation (ILP) based on the ILP formulation of this problem in [16] without pipelining, ILP. The ILP solver is stopped after 35 minutes in some cases due to the long execution time taken by the ILP to produce optimal results. Our scheduler took only few seconds (worst case was 5 seconds) to produce the results. Following the same assumptions concerning the MPSoC system model and SPM memory budget, our TSMP_INTEG heuristic is in the range of 0% to 13% (7.45% on average) off the optimal solution in a negligible amount of time. This shows the effectiveness of our proposed scheduler technique as in most of the cases our solution was close to that of the ILP. We also compared our integrated approach to an ILP optimal version of TSMP_ANY and found that on average our integrated approach is 4.76% better than that of the optimal TSMP_ANY. The last columns in Figures 8–11 present the stage time interval in cycles for the case of pipelining. Note that in pipelining, the objective is not to minimize the run time of the application but rather minimize the stage time interval that is minimize the time between the start times of two consecutive iterations.
partitioning techniques were able to reduce the overall schedule time in all cases from 2.2% to 9.1% with an average reduction of 6% compared to [19] while utilizing our scheduler. Moreover, our resource partitioner was able to reduce the overall schedule time in all cases with an average reduction of 8.2% against equally partitioning the resources.

In the previous sets of experiments, we showed the effectiveness of our scheduler and resource partitioner in isolation. In this last set of experiments, we tested our holistic resource partitioning and memory-aware task scheduling framework in comparison to a technique where the resources are partitioned based on [19] and the scheduler is based on partitioning the resources of an application in any ratio among application’s tasks as discussed in the first set of experiments (TSMP\_ANY). We utilized the same embedded application combinations as in the previous set of experiments that is combinations (Lame, Osdemo, Cjpeg) and (Lame, Enhance, Cjpeg, Osdemo).

Tables II and III show the average cycle count from running our framework under different arrival and departure times compared to the unoptimized technique where we were able to reduce the overall cycle count in all cases from 4.1% to 11.2% with an average reduction of 8.3% compared to the unoptimized scenario which clearly show the importance and effectiveness of our techniques compared to the resource partitioner in [19] with decoupled task scheduling techniques.


table II (LAME, OSEDMO, CJPEG) CYCLES.

<table>
<thead>
<tr>
<th>Resources</th>
<th>Ours (cycles)</th>
<th>Unoptimized + Xue Scheme <a href="cycles">19</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>(4P, 256KB)</td>
<td>44211259</td>
<td>48274652</td>
</tr>
<tr>
<td>(6P, 128KB)</td>
<td>34543927</td>
<td>35986540</td>
</tr>
<tr>
<td>(10P, 512KB)</td>
<td>25214218</td>
<td>28056288</td>
</tr>
</tbody>
</table>


table III (LAME, ENHANCE, CJPEG, OSEDMO) CYCLES.

<table>
<thead>
<tr>
<th>Resources</th>
<th>Ours (cycles)</th>
<th>Unoptimized + Xue Scheme <a href="cycles">19</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>(4P, 2MB)</td>
<td>2859838472</td>
<td>3029381901</td>
</tr>
<tr>
<td>(8P, 2MB)</td>
<td>1971283798</td>
<td>2098372361</td>
</tr>
<tr>
<td>(10P, 4MB)</td>
<td>1636612322</td>
<td>1803677612</td>
</tr>
</tbody>
</table>

VIII. CONCLUSIONS

In this article, we presented a holistic approach to resource partitioning and task scheduling of multiple applications on an MPSoC. The resource partitioning techniques effectively divide the processor cores and the memory budget among competing applications based on the structure of each application. The presented scheduling technique of the tasks of an application on the resources allocated to that application is memory-aware to reduce the schedule time. Future work would include tweaking the proposed approaches for power and thermal-aware safety.

REFERENCES


