Effective Techniques for Performance Enhancement on Embedded Multi-Processor Architectures

Hassan Salamy
Electrical & Computer Engineering
University of Saint Thomas
Minnesota, USA

ABSTRACT
As the complexity of embedded applications is ever increasing, the trend in embedded architecture is to utilize a multi-processor system on a chip (MPSoC). MPSoCs provide the compute power and flexibility to effectively execute complex embedded systems. An embedded system often execute multiple complex embedded applications simultaneously. In this article, we tackle two main problems to further enhance the effective utilization of the embedded MPSoC architecture to reduce the execution time of the applications, namely, resource allocation and scheduling. We first present an effective resource allocator that examines the nature of the applications in the system to fairly allocate the fast on-chip *scratchpad* memory budget and the processing elements. Then this article presents an effective task scheduler that integrates scheduling and on-chip *scratchpad* memory partitioning for the maximum optimization of the system. Results on multiple real and synthetic benchmarks showed the effectiveness of our techniques**.**

Keywords
MPSoC, scratchpad, task scheduling, resource allocation.

1. INTRODUCTION
With the huge demand for processing power from embedded systems to efficiently execute complex embedded applications, the trend in embedded architecture is to deploy multiple processors on a single chip. Multi-processor System-on-a-chip (MPSoC) is an attractive solution to provide the compute power while maintaining a conservative power budget. A typical MPSoC usually utilizes heterogeneous processing elements, a multiple of memory hierarchies all interconnected by a sophisticated communication structure. MPSoCs have come to be an attractive and flexible solution for high performance embedded systems with reduced energy consumption.

Memory access time prediction is usually essential in real-time applications to satisfy the requirement of predictability of execution time. Caches, that are usually the memory of choice, are therefore not suitable for such systems as caches are hardware controlled which makes it extremely hard to model the behavior of a cache memory. An answer to this shortcoming is a software controlled memory known as *scratchpad* memory. Since *scratchpads* are software controlled, their behavior can be predicted with high accuracy. In this article, an MPSoC architecture with scratchpad on-chip memory is assumed to satisfy the predictability requirement for critical real time embedded applications.

An important optimization technique for embedded systems is the generation of optimized schedules to reduce the run time of embedded applications on the system's resources. Given an MPSoC system with multiple embedded applications possibly of different arrival times, an important research question is how to partition the system resources of processing elements and the on-chip *scratch pad* memory among the applications for a schedule with minimum run time. Hence in this paper, we present effective techniques to the two NP problems of proper allocation of system resources to competing embedded applications in the system and the scheduling of each application tasks on the allocated resources with the objective of reduced schedule time.

The problems of task scheduling and memory allocation are usually studied as two separate problems in the literature. However, the decision of scheduling a task on a processor is greatly affected but the amount of on-chip memory allocated to that processor. The main reason is that accessing data from the off-chip memory is usually in the range of hundred times slower than accessing from the on-chip memory. Hence in this article the problem of task scheduling and on-chip memory allocation are studied in an integrated fashion.

In this article, effective techniques to partition the system resources among competing embedded applications as well as effective scheduling of each embedded application tasks on the allocated resources in an integrated fashion with *scratchpad* memory partitioning are presented. Extensive experimental analysis are performed to test the effectiveness of the proposed techniques.

The remainder of this article is organized as follows. Section 2 presents related work and Section 3 introduces the architecture and formally defines the problem. The profiling component from the framework is detailed in Section 4. Sections 5 and 6 presents details about the Resource Allocator and the integrated scheduler, respectively. Detailed experimental results are presented in Section 7 while Section 8 lists the conclusions.

2. RELATED WORK
Many researchers in the literature studied the scheduling and allocation problems. A constraint programming along with integer linear programming for scheduling and memory partitioning was presented in [4]. Ahmed [9] presented a comparison between different scheduling algorithms of task dependence graphs on an architecture with homogenous set of processors. De Micheli et al. [11] studied the scheduling and mapping problem as a hardware/software codesign. Hardware-software partitioning and scheduling with pipelining 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 scratchpad 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 scratchpad 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 task scheduling and memory partitioning on a heterogeneous multiprocessor system on chip with scratch pad memory. They formulated this problem as an integer linear problem (ILP) with the inclusion of pipelining. ILP solutions require long computation time for large applications and hence they are not practical in real life. A technique to effectively divide system resources among competing applications is presented in [19].

Research has been conducted on task scheduling problems for DVS enabled multi-processor real-time embedded systems [21]. In [31] it was shown that the thermal aware task scheduling outperforms the power-aware schemes in terms of maximal and average temperature reduction. In [20], the authors propose an adaptive method which eliminates hot spots in a slightly better way than the load balancing techniques by reducing temporal and spatial temperature variations. Energy efficient for real time task scheduling is presented in [22, 23]. Kanoun et. al [24] presented an online energy efficient task graph scheduling for multicore platforms. Tseng et. al [25] presented energy efficient scheduling on multicore mobile devices.

3. SYSTEM ARCHITECTURE AND PROBLEM DEFINITION

In this article, the underlying system architecture is based on a multiprocessor System-on-a-chip with limited on-chip fast scratchpad memory, a number of processing elements, and unlimited off chip memory all interconnected with a communication bus model. Given such an architecture system and a number of embedded applications utilizing the system possibly arriving at different times, we present effective techniques to allocate the proper resources to each application and generate optimized schedule with minimum overall run time. Allocation and scheduling are essential techniques to optimize the execution of a number of embedded applications in an MPSoC with limited resources.

A main contribution of this paper is that it provides a comprehensive technique to allocation and scheduling where allocation is highly based on the nature of the applications and scheduling is integrated with scratchpad memory partitioning for further reduction in the schedule time. Effective techniques to these sought problems can play a major role in extracting the compute power of multi-core embedded systems. The framework depicted in Figure 1 starts with a profiler that profiles each application the systems receives to generate proper metrics to aid in understanding the nature of each application for optimized allocation and scheduling. The second part is a resource allocator that based on the nature of the applications in the system, it allocates the processing elements and the scratchpad budget among the applications. For instance, a memory intensive application is set to benefit more from additional scratchpad budget compared to a more computationally intensive and parallel application. On the other side, the parallel application has a higher potential to benefit from additional processing elements to exploit parallelism and to tackle the high compute requirements.

Once the resource allocator is done and each application receives its share of the system resources, the integrated scheduler in our holistic framework is invoked. The scheduler is responsible of effectively scheduling the tasks of each application on the resources allocated to such application with the objective of reduced overall schedule time. The three aforementioned parts of the framework are detailed next.

4. THE PROFILER

The first main part of the proposed framework is the profiler. The profiler receives the embedded applications to be executed on the system and studies the nature of each application and generates a number of profiling information.

One main piece of information that the profiling part of the proposed framework establishes is the task dependence graph (TDG). A task dependence graph is a representation of how the embedded applications can be divided into a set of tasks along with the dependencies between such tasks. The profiler will study the structure of the embedded application and generates a set of basic blocks that will translate to tasks in the TDG. The profiler will then study the communication/data flow between the basic blocks (tasks) to establish dependencies between tasks that are represented as edges in the TDG. The edges will be weighted to accommodate for the communication cost between dependent tasks estimated by the profiler.

To further establish the nature of the tasks, the profiler is responsible about generating estimates of how much a task benefits from additional *scratchpad* memory budget. We measure the benefit of a task from additional scratchpad memory by estimating the degree of potential reduction in run time of the task with increased scratchpad budget. To aid in this, the profiler will generate three separate estimate:

1- *Short*: The shortest run time of a task based on assigning all the available *scratchpad* budget to the processor executing task.
2- *Long*: The longest run time of a task based on assigning no *scratchpad* budget to the processor running this task.
3- *Middle*: The middle ground of measuring the run time of a task if the *scratchpad* budget is equally divided among all processors executing an application.

In addition to the aforementioned set of values and estimates generated by the profiler, a list of other values as needed and discussed throughout the rest of this article are also the responsibility of the profiler.

5. THE RESOURCE ALLOCATOR

In this part of our framework, the systems' resources of processing elements and on-chip *scratchpad* memory will be distributed among the embedded applications currently using the system. Due to the assumed limited system resources, allocating such resources should be performed based on the nature of each embedded application. An application could be memory intensive, computational intensive with different levels of possible task parallelism, or a weighted combination of both. Accurately examining and classifying embedded application is a computation intensive problem and hence estimated techniques are developed.

In our approach, resource allocation is highly dependent on the nature of the embedded applications in the. For instance, an application that is memory intensive in nature will benefit more from additional on-chip *scratchpad* budget. This is mainly because a memory intensive application is defined as an application where memory accesses encompass big portion of its computation time and hence it greatly benefits from the ability to access variable elements quickly from the fast *scratchpad* compared to slower off-chip memory. Based on the degree of memory intensiveness, the percentage of overall schedule time reduction is estimated. On the other hand, an application can be classified as computationally expensive with high level of parallelism and hence the proposed resource allocation technique will allocate more processing elements to such application to reduce the schedule time. In such case, the additional processing elements will aid the exploitation of the applications parallelism and will provide the needed computation power for the highest possible reduction in the schedule time.

As such the resource allocation part of our holistic framework receives the profiling information about each application in the system. Then it generates an estimate of the nature of each application to effectively and fairly divide the system resources among competing application concurrently utilizing the embedded system.

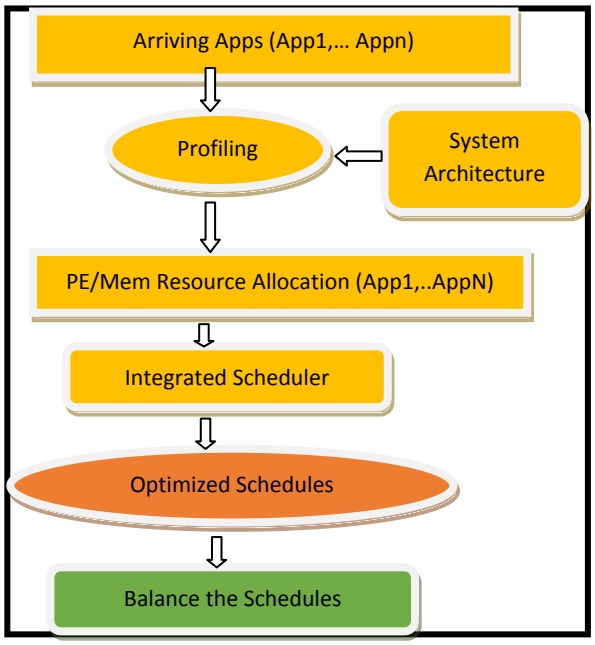

Figure 1: Our holistic framework

Scratchpad Memory Allocation: Once an application is received with all its corresponding profiling information, the resource allocator computes a *flexibility* estimate based on the intrinsic structure of the application. The *flexibility* value of task T_i on processor (P_j) in Equation 1 represents the potential in run time reduction that the current application might exhibit with increased *scratchpad* memory budget. The *Current* variable in Equation 1 is the run time if the remaining *scratchpad* budget in the system is equally partitioned among all the applications that haven't been allocated a *scratchpad* budget whereas *Short* is the run time if all the scratchpad budget in the system is allocated to the processor executing T_i. The *flexibility* is a value between 0 and 1 to track the benefit of increasing the current *scratchpad* budget allocated to the application to the extreme case of allocating all the *scratchpad* budget in the system to such application. A *flexibility* value closer to 1 implies that the application has a high potential of improved run time from higher *scratchpad*.

$$flexibility(T_{ij}) = \frac{Current_{ij} - Short_{ij}}{Current_{ij}} \quad (1)$$

The on-chip *scratchpad* memory budget in the system will be allocated to the embedded applications to reduce the overall schedule time. For each application in the system, a *Memory Potential Reduction* (MPR) in Equation 2 for each application is defined as an estimate of potential reduction in run time of the whole application from allocating additional scratchpad budget and is computed as the average *flexibility* value for all its tasks.

$$MPR(APP_i) = 1/p \cdot \sum_{j \in pT_i \in APP_i} \sum \frac{flexibility(T_{ij})}{t} \quad (2)$$

The memory allocator heuristic in Figure 2 receives as input the on-chip memory size (m) and the number of applications concurrently running (n). It then first determines the the memory requirements of each application in the system (*Mem_Requested*()) through the profiling information and the nature of each application. If the available on-chip *scratchpad* memory budget is less than the total requested memory, each application will receive what it requested. In the most common case where the requested memory is more than what is available, the heuristic will examine the applications in decreasing order of their MPR values. It will then allocate to each application an *scratchpad* budget proportional to its MPR value such that an application with higher MPR value will receive an *scratchpad* budget closer to what it requested compared to an application with a lower MPR value.

Memory_Allocator(n , m)

1.L = Apps in decreasing order of MPR
2.Mem =0 and Total_MPR = 0
3.**For** i = 1 to n **do**:
4. Mem = Mem + Mem_Requested(i)
5. Total_MPR = Total_MPR + MPR(i)
6.**End For**
7.**If** (Mem $\leq$ m)**Then**
8. **For** i = 1 to n
9. Mem_received(i) = MPR_requested(i)
10. **End For**
11.**Else**
12. **While** L not empty
13. i = First application in L.
14. Temp = UpperBound(($\frac{MPR(i)}{Total\_MPR}$)* m)
15. *scratchpad*_ received(i) = MIN(Mem_requested(i),Temp)
16. m = m - Mem_ received(i)
17. Remove i from L.

18. Recompute *Short*, MPR and Total_PRF.
19. Reconstruct the list L.
20. **End While**

Figure 2: Our on-chip memory allocator.

Processing Elements Allocation: Once the profiling information of all the application are received, the available processing element cores in the system will be allocated such that an application with a higher potential of parallelism is allocated more cores so that more tasks can run in parallel. For each application, a potential for parallelism (*PP*) value is computed as in Equation 3. An application with a higher *PP* value implies that it is more parallel in nature, that is, more tasks of such application can run in parallel. The *PP* value is mostly extracted from the task dependence graph (*TDG*). Two tasks in the *TDG* can run in parallel if they are independent. Two tasks are said to be independent if there is no path between the two tasks in the *TDG*. An application with a higher number of independent tasks has a higher potential to run such tasks in parallel and is mostly limited by the number of processing cores allocated to such application.

$$PP(APP_i) = distinct_i + \frac{pairs_i}{distinct_i} \quad (3)$$

Processing_Elements_Allocator(n , p **)**

1.Path = 0 and Path_PP = 0
2.**For** i = 1 to *n*
3. PP(i) = Compute_PP(i)
4.**End For**
5.L = Apps in decreasing order of (1 + 0.1 PP(i)) * path(i)
6.**For** i = 1 to *n*
7. Path = Path + *distinct*(i)
8. Path_PP = Path_PP + (1 + 0.1 PP(i))* d*istinct*(i).
9.**End For**
10.**If** (Path $\leq$ p) **Then**
11. **For** i = 1 to *n*
12. Processor_received(i) = distinct(i)
13. **End For**
14.**Else**
15. **While** L not empty
16. i = First application in L.
17. Temp = UpperBound((1 + 0.1 PP(i)) * distinct(i)/Path_PP * p)
18. Processor_received(i)= MIN(distinct(i),Temp)
19. Path_PP = Path_PP - (1 + 0.1 PP(i)) * distinct(i)
20. Update the number of processors.
21. Update the List L.
22. **End While**

Figure 3: Our processing elements allocator.

As shown in Equation 3, the *PP* value is divided into two main parts. The first part represents the number of distinct paths (*distinct*) in the *TDG*. Two paths are said to be distinct if they have at least one task that doesn't belong to both paths. Compared to more unbalanced paths, two paths in a *TDG* with close number of tasks can benefit more from two processing cores as the idle time of the cores is minimized. This is taken into consideration in the second part of the PP equation. That part counts the number of pairs of tasks on two distinct paths that are eligible to run in parallel and divides that by the total number of distinct paths. The number of distinct paths and the pairs of potential tasks that can run in parallel are used in the *PP* equation to estimate the potential of parallelism in an application. And hence the processing cores allocator will use such information in deciding how to allocate the limited number of processing elements among the concurrently running embedded applications.

The processing elements allocator heuristic in Figure 3 receives as input the number of applications in the systems (*n*) and the number of available processing elements (*p*) and then sorts the applications in decreasing order based on an altered version of the *PP* value since eventhough the *PP* value is an estimation of potential parallelism, it is an exaggeration of realistically the number of tasks that will run in parallel. If the available processing elements is less than the total requested, each application will receive what it requested. In the most common case where the number of requested processing elements is more than what is available, the heuristic will allocate to each application processing elements proportional to the altered potential parallelism calculated values. Hence an application with higher potential for parallelism will receive number of processing elements closer to what it requested compared to applications with lower levels of potential parallelism.

6. THE INTEGRATED TASK SCHEDULER

After the resources in the system have been allocated to the applications, the task scheduler is responsible about scheduling the tasks of each application on the resources allocated to that application. The task scheduler receives the profiling information about each application (App_i) along with its allocated processing elements (PE_i) and its allocated on-chip *scratchpad* memory budget (mem_i).

The majority of the work in the literature have studied scheduling of the tasks of an application on the processors separately from the memory allocation to such processors. However, in this article, we propose that these two problems are highly dependent on each other and should be studied as one integrated problem to generate fully optimized schedules. Unlike the trend in previous research that first schedule the tasks on the available processors and then partition the on-chip memory as a post scheduling step, we opt to the integrated approach that adopt a comprehensive and holistic approach the the scheduling problem. Hence our approach to scheduling of the applications utilizing the system heavily rely on how memory will be partitioned among these applications.

This section details the proposed optimized technique to task scheduling/memory allocation problem to effectively schedule the embedded applications in the system for maximum throughput (Figure 4). The partitioning of the scratchpad memory will be dynamically allocated to the processors integrated in the decision making of mapping tasks to available processors. This is mainly due to the fact that the run time of a task scheduled on a certain processor not only

depends on the features of the processor but also on the on-chip fast *scratchpad* memory allocated to this processor. This is especially more apparent with memory intensive applications that can greatly benefit from the reduced *scratchpad* access time compared to accessing external memory. Hence our task scheduler explicitly considers the changing run time of a task on a processor based on the associated *scratchpad* budget to generate better quality schedules with highest reduction time in run time.

We propose a dynamic algorithm to scheduling that takes into account the varying execution time of a task while building a schedule. The varying execution time is mainly due to the dynamic essence of allocating the available *scratchpad* budget to processors throughput the course of our proposed integrated scheduling heuristic. We first use the profiling information to extract important information about each task to be scheduled. Example of the extracted information are the *Short*, *Middle*, and *Long* values that were introduced and discussed in the previous section as an estimation metric of how much a task can benefit from varying *scratchpad* memory budgets.

Figure 4 presents the proposed dynamic scheduler that begins by sorting the tasks in list L_1 in ascending order of the *As Soon As Possible* (*ASAP*) values. Following the sorted tasks in the list L_1, the scheduling heuristic matches each task to the best processor under the objective of minimal schedule time.

First define equations (4)-(6) below where *Begin* (T_i, PE_j) is the earliest begin time of task T_i on processor P_j detailed in Equation (4) as the maximum between the current finish time of the processing element PE_j (*Finish* (PE_j)) and the biggest finish time of all the parent tasks of T_i extracted from the TDG ($Max(Finish_{T_j \in Parent(T_i)}(T_j)$) with the added communication cost, *Comm*. The finish time of a task T_i scheduled on the processing element PE_j is calculated in Equation 5 whereas the finish time of a certain processing element is calculated as the finish time of the latest task scheduled on this processing element (Equation 6).

$$Begin(T_i, PE_j) = Max(Max(Finish_{T_k \in Parent(T_i)}(T_k) + Comm_{T_k \in Parent(T_i)}(T_k)), Finish(PE_j)) \quad (4)$$

$$Finish(T_i) = Begin(T_i, PE_j) + Time(T_i, Mem_j) \quad (5)$$

$$Finish(PE_j) = Max(Finish_{T_k \in PE_j}(T_k)) \quad (6)$$

Integrated_Scheduler:

1.Receive the Profiling information from the Profiler.

2. Receive the system resources allocated to the application under consideration from the Resource Allocator.

3.Divide the on-chip *scratchpad* memory equally between the processors.

4.Find the ASAP for all the tasks based on *Middle* values.

5.L_1 = List of tasks in increasing order of ASAP.

6.**While** (L1 not empty) **do**:

7. Get the first task T_f from L_1.

8. Find the processing element PE_i to schedule T_f with minimal overall schedule time increase.

9. *min* = Finish time of PE_j with T_f scheduled.

9. **For** each other processor PE_k **do**:

9. Calculate the *flexibility* and *PFR* of PE_k if T_f is mapped to PE_k.

10. Find the minimum Begin time of T_f on PE_k.

11. Find *Finish*(PE_k) if T_f is mapped to PE_k.

12. **if** ((*Finish*(PE_k) < *min* && *PFR*(PE_j) ≥ (1 - δ %)*PFR*(PE_k))||(*Finish*(PE_k) > *min* && *PFR*(PE_k) ≤ (1- δ %)*PFR*(PE_j)) **Then:**

13. min = *Finish*(PE_k)

14. **else if** (*Finish*(PE_k) == min)

15. min = Finish time of processor with the higher *flexibility*.

16. **End For**

17. Assign T_f to *PE* corresponding to *min*.

18. Delete Tf from L_1.

19.**End While**

Figure 4: The integrated scheduler.

In general a task T_i is supposed to be scheduled on the processing element (say PE_j) with the minimal increase in schedule time. However, to keep the dynamic essence of our techniques and to look beyond the current configurations and schedule status, T_i might be scheduled on PE_k with higher schedule increase than if scheduled on PE_j. This is only possible under the condition that the *Predicted Finish Reduction* time (*PFR*(P_k)—Equation 7) of processor PE_K is at least δ % less than that of processor PE_j. The *PFR* value as defined is a guide to the scheduler of the amount of potential overhead reduction due to future *scratchpad* distributions if *T*i is mapped to PE_k. In other words, the *PFR* value is an estimation of finish time reduction of processor PE_k due to possible higher future *scratchpad* budget.

The predicted finish reduction time of a task on a processing element highly depend on the *flexibility* (introduced and explained in Section 5) of the tasks that are mapped to this processing element. As in the definition of *flexibility*, *Current* of a certain task is the time it takes to execute the task on the processor under the current *scratchpad* budget distribution. The *PFR* as defined in Equation 7 is highly dependent on the calculated *flexibility* value and hence it reflects the dynamic essence of our scheduler where the decisions are based not only on the current on-chip *scratchpad* memory allocation but also on an estimated reduction in run time of the tasks due to predicted possible future *scratchpad* distribution though out the life of the task scheduler. We define the flexibility of a processor as the average flexibility value for all the tasks currently allocated to run on this processor.

$$PFR(PE_k) = Finish(PE_k) - \sum_{T_j \in P_k} \left(Current(T_j) - \frac{Current(T_j)}{1 + flexibility(T_j)} \right) \quad (7)$$

Balancing the schedule: The schedule generated by the integrated scheduler will be further balanced in an attempt to reduce the overall schedule time. A balanced schedule is such that the difference between the finish times of all the processor is minimal. To do so, we start with the processor

with the highest Finish time (say PE_k) and perform a set of steps to create a more balanced schedule. This will be achieved by altering the *scratchpad* budgets between the processors of the highest and lowest Finish times and taking the flexibility into consideration. Specifically, it starts by transferring 10 % from the *scratchpad* budget allocated to the processor (PE_j) with the lowest (*Finish * Flexibility*) product to processor PE_k as long as $Finish(PE_j) < Finish(PE_k)$. In the majority of cases, such memory transfer will decrease the Finish time of processor PE_j while increasing that of PE_k and thus decrease the schedule time which is the main objective of our technique. This process of transferring 10 % of the *scratchpad* allocated memory budget between PE_j and PE_k will be repeated multiple times (based on fine tuning) and as long as $Finish(Pj) < Finish(Pk)$ to insure no adverse effect in the overall scheduler.

After every *scratchpad* budget redistribution among the processors and based on the new run time of a task T_i (Equation 9), the $Begin(T_i)$, $Finish(T_i)$ values will be recomputed for each task T_i mapped to a processor whose *scratchpad* budget is changed while balancing the schedule. This is mainly achieved by computing an estimate *Gain* value (Equation 9) for each task T_i on a processor with newly assigned *scratchpad* memory budget Mem_j. The *Gain* of a task represents the reduced execution time of a task due to the new assigned *scratchpad* budget. The *Gain* value is estimated by allocating variables from task T_i to Mem_j in ascending order of $byte_i/freq_i$ with $byte_i$ is the size of the variable V_i and $freq_i$ is the number of times such variable is accessed throughout the course of executing the task Ti. As mentioned earlier, accessing a variable from an external of-chip memory is more expensive in terms of required clock cycles compared to accessing a variable from the on chip *scratchpad* memory. β_1 and β_2 in Equation 9 are the respective assumed cost of accessing from the off-chip and on-chip memory. Even though this is a simple data allocation technique, our experiments showed it is fast and effective. The updated run time of task T_i under the new *scratchpad* memory budget Mem_j in Equation 9, is defined as the difference between the time taken to execute T_i assuming no *scratchpad* memory, $Time(T_i, 0)$ and $Gain(T_i, Mem_j)$.

$$Time(T_i, Mem_j) = Time(T_i, 0) - Gain(T_i, Mem_j) \quad (8)$$

$$Gain(T_i, Mem_j) = \sum_{v_i \in T_i, v_i \in Mem_j} ((\beta_1 - \beta_2) * freq_i). \quad (9)$$

7. EXPERIMENTAL SETUP AND PERFORMANCE ANALYSIS

In this section, the performance of the detailed scheduler, resource allocator, and the holistic approach are studied and examined.

The first step of the presented holistic approach to resource allocation and task scheduling is to extract information about each application utilizing the system through profiling. The profiler will identify the basic computation block in each embedded application along with the control and data flow between these basic blocks. The basic blocks will be vertices in constructing the task dependence graph (TDG) and the data/control flow dictates the dependencies between tasks. The TDG is a weighted graph with weights representing the communication costs between dependent tasks. *Simplescalar* [2] will be used to profile applications. Simplescalar is an architectural simulation to simulate the execution of a task on a processor under different memories allocation. As detailed earlier, the profiler will mainly:

(i) Construct a TDG representing an application.

(ii) Generate the *Short*, *Middle*, and *Long* values for each task on different processors.

(iii) Determine variables sizes and the frequency a variable is accessed throughout the schedule of the application.

In our experiments, we used real life applications extracted from [17], Mediabench [7] and Mibench[10], namely, *enhance*, *lame*, *osdemo*, and *cjpeg* as test benchmarks of which their characteristics are presented in Table 1. We also used Synthetic benchmarks generated using the *TGFF* tool [26].

7.1 Testing The Scheduler

As mentioned earlier, we tested the scheduler and the resource allocator independently first before testing the whole presented holistic approach to resource partition and scheduling of multiple applications on an MPSoC. First, we tested the scheduler detailed in Section 6. The scheduler is responsible about effectively scheduling the tasks of an applications on the resources (processors and *scratchpad* budget) allocated to that application by the resource allocator. For testing and comparisons we implemented the following four different approaches:

1. *EQUAL*: A decoupled approach that tackles scheduling and memory partitioning independently and assuming the *scratchpad* budget is equally divided among the available processors.

2. *ANY*: A decoupled approach based on a tweaked version of [18] that schedules tasks in a TDG dynamically over the available processors and then perform the memory partitioning as a latter and independent step.

3. *INTEG*: Our integrated approach to task scheduling and memory partitioning detailed in Section 6.

TABLE 1 CHARACTERISTICS OF SOME OF OUR BENCHMARKS.

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

For the benchmarks *enhance*, *lame*, *osdemo*, and *cjpeg*, we assumed a microprocessor system with two processors and an on-chip *scratchpad* budget of size varying between 4KB and 4MB. It is critical to chose an underlying microprocessor architecture with the proper number of processors and *scratchpad* budget based on the applications to be tested. This is because too many or too few resources might not project the essence of our presented heuristic and might not properly reflect the effectiveness of our approaches for different embedded applications. We tested each benchmark under three different *scratchpad* budget and presented the average results. In our experiments, we assumed a 100 cycle latency to access data from the off-chip memory and 1 cycle latency to access data from the on-chip *scratchpad* memory. There is no limit on the size of the off-chip memory that is it is assumed

that the off-chip memory is large enough to hold all the data variables in the application.

The first three columns in Figures 5-8 present comparisons between the *EQUAL*, *ANY* [18], and *INTEG* techniques. From the first two columns in Figure 5-8, the *ANY* technique improved over the *EQUAL* technique from almost no improvement to huge improvement of 47% with 7% improvement on average. This expected improvement shows that static allocation of the on-chip memory among processors i.e., dividing the on-chip memory budget equally over the processors without taking into consideration the nature of tasks mapped to each processor fails to effectively utilize the *scratchpad* in the system for overall schedule reduction.

On the other hand, as evident in Figures 5-8, our integrated approach *INTEG* improved over the *ANY* approach up to 22% with 7.9% schedule time reduction on average. As always the improvements greatly depend on the nature and structure of each application. It is clear from the results that our integrated approach greatly reduces the schedule time compared to decoupled approaches that treat scheduling and memory partitioning as two separate problems. Hence, the results clearly showed that decoupled approaches result in schedules of inferior quality compared to techniques that study the task scheduling and on-chip *scratchpad* memory partitioning in an integrated fashion as presented by our technique. The presented schedule time reduction from our integrated approach is mainly due the task nature guidance followed by our technique as the on-chip *scratchpad* configuration of a processor is highly dependent on the nature of the tasks scheduled to be executed on that processor.

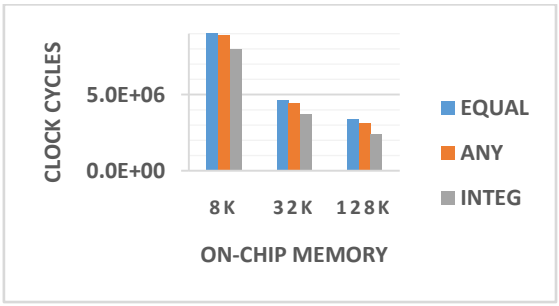

Figure 5 Results for lame benchmark.

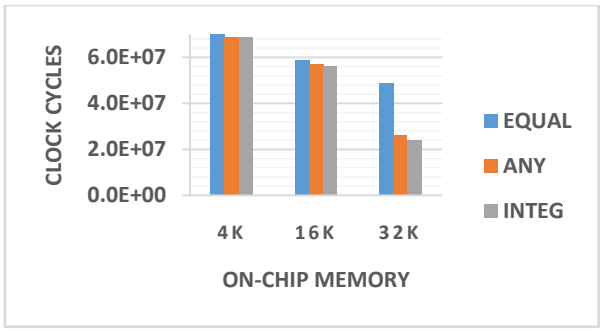

Figure 6 Results for osdemo benchmark.

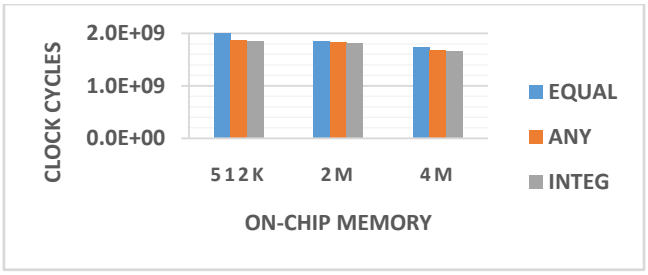

Figure 7 Results for enhance benchmark.

Finally, we tested our integrated scheduler based on synthetic benchmarks generated using *TGFF* [26]. As in Table 2, the benchmarks were divided into 4 different sets with the complexity of the generated Task Dependence Graphs (TDGs) increased as we moved from Set 1 all the way to Set 4. Each set consists of a number of similar complexity benchmarks. The benchmarks in each set were tested under different system resources and their average performances are captured. The results in Table 2 is the average percentage reduction compared to the *EQUAL* technique. As apparent and following similar result pattern on the real-life benchmarks, our integrated scheduler approach improved over the two decoupled approach in all cases.

7.2 Testing The Resource Allocator

After proving the effectiveness of our integrated approach to task scheduling and on-chip memory partitioning, in this part of the experimental results we test the *Resource Allocator* detailed in Section 5. Our resource allocator will be tested against the resource allocator presented by Xue *et al*. [19]. For fair comparisons, we tested the two resource allocator techniques while utilizing our integrated scheduler detailed in Section 6.

For this part of the testing, we utilized two test sets of benchmarks: 1- (*Lame*, *Osdemo*, *Cjpeg*) and 2- (*Lame*, *Enhance* , *Cjpeg*, *Osdemo*). Different scenarios of arrivals times were assumed for the applications in the two sets to mimic real life situations. The two sets were also tested under different system resources and the then the results are averaged out and presented in Figures 9 and 10. Figures 9 and 10 present the average run time among different arrival time scenarios and under the available processing elements and on-chip scratchpad memories labelled in the figures as (# of *PE*, *scratchpad* size). As mentioned earlier, choosing the proper system resources is very essential in fairly testing the effectiveness of our techniques as too little or too many system resources will fail to properly test our techniques. The schedule results are based on our integrated scheduler and presented in terms of system cycles in the figures of results. Clearly, our resource allocator is able to improve over the allocator in [19] in all the tested cases with improvements ranging from 2.3% to 9.4 % and an average reduction of 6.3%.

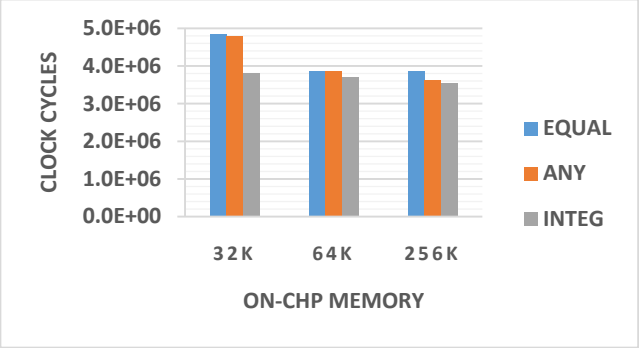

Figure 8 Results for cjpeg benchmark.

TABLE 2 SCHEDULING RESULTS FOR SYNTHETIC BENCHMARK SETS

Benchmark	ANY	INTEG
Set1	9.8%	16.3%
Set 2	8.2%	15.8%
Set 3	12.4%	17.3%
Set 4	10.12%	18.3%

7.3 Testing The Whole Framework

In the two previous sets of experiments, the effectiveness of our integrated scheduler and resource allocator are detailed and presented. For the last set of experiments, we tested our whole approach to allocation and scheduling as one holistic framework. The results were tested against the resource allocator presented in [19] and the decoupled scheduler, *ANY*, based on [18]. For this part of the testing, we also utilized the two test sets of benchmarks: 1- (*Lame*, *Osdemo*, *Cjpeg*) and 2- (*Lame*, *Enhance* , *Cjpeg*, *Osdemo*). Different scenarios of arrivals times were assumed for the applications in the two sets to mimic real life situations.

The system resources along with the cycle count from our holistic framework along with that based on the works in [18] and [19] are detailed in Tables 3 and 4. From the tables, our approaches are able to reduce the overall cycle count in all the tested cases with improvements ranging from 4.2 % to 11.3 % with an average cycle count reduction of almost 8.4 %. This clearly showed the effectiveness of our proposed techniques that are essential in extracting the compute power from multi-core embedded systems.

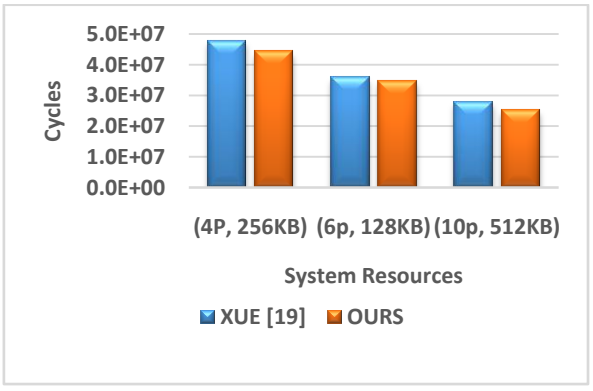

Figure 9 Results for Lame- Osdemo -Cjpeg set.

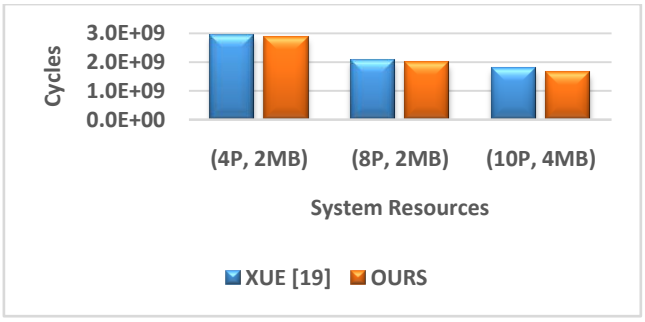

Figure 10 Results for Lame-Enhance-Cjpeg-Osdemo.

We then tested our holistic framework compared to Xue [19] resource allocator and the *ANY* [18] scheduler tested on Synthetic benchmark set generated using TGFF [26]. As in Table 5, the benchmarks were divided into 4 different sets with each test containing a number of benchmarks TDGs of varying complexities. The benchmarks in each set were tested under different system resources and their average performance are captured. The results in Table 5 are the average percentage reduction of our presented approach compared to Xue [19] resource allocator and the *ANY* [18] scheduler. As clearly evidend, our technique performed better than the other technique in all cases with an average improvement of 9.4 %.

TABLE 3 (*LAME, OSDEMO, CJPEG*) CYCLES.

Resources	Ours (cycles)	Xue [19] +[18](cycles)
(4p, 256KB)	44211259	48274652
(6P, 128KB)	34543927	35986540
(10P, 512KB)	25214218	28056288

TABLE 4 (LAME, ENHANCE, CJPEG, OSDEMO) CYCLES.

Resources	Ours (cycles)	Xue[19] +[18](cycles)
(4P, 2MB)	2859838472	3029381901
(8P, 2MB)	1971283798	2098372361
(10P, 4MB)	1636612322	1803677612

TABLE 5 RESULTS FOR SYNTHETIC BENCHMARK SETS

Benchmark	Ours (% improvement)
Set1	9.2%
Set 2	10.8%
Set 3	7.2%
Set 4	10.3%

8. CONCLUSIONS

This article presented effective optimization methods to enhance the performance of a multiprocessor system by providing effective resource allocation and scheduling techniques. The resource allocator carefully examines the structure of each application to fairly allocate the system resources. The task scheduler integrated scheduling and on-chip memory partitioning to enhance the performance of the system. Results on real-life and synthetic benchmarks showed the importance of our proposed techniques.

9. REFERENCES

[1] 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.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. IEEE Computer, 35(2), 2002.

[3] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. ACM Transactions on Embedded Computing Systems, 1(1), 2002.

[4] 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.

[5] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. ACM Trans. Comput. Syst., 28(4), 2010.

[6] CPLEX. Ilog inc., ilog cplex 8.1 reference manual. http://www.ilog.com/products/cplex, 2008.

[7] 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.

[8] 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.

[9] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. Journal of Parallel and Distributed Computing, 59(3), 1999.

[10] 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, pages 330 (335), 1997.

[11] G. D. Micheli, R. Ernst, and W. Wolf. Readings in hardware/software co-design. Morgan Kaufmann, 2002.

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

[13] P. Panda, N. D. Dutt, and A. Nicolau. On chip vs o_ chip memory: the data partitioning problem in embedded processor-based systems. ACM Transactions on Design Automation of Electronic Systems (TODAES), 5(3), 2000.

[14] 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.

[15] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In Proc. Design Automation and Test in Europe (DATE), 2002.

[16] 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.

[17] 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.

[18] V. G. Vaidya, P. Ranadive, and S. Sah. Dynamic scheduler for multi-core systems. In 2nd International Conference on Software Technology and Engineering (ICSTE), 2010.

[19] L. Xue, O. Ozturk, F. Li, M. Kandemir, and I. Kolcu. Dynamic partitioning of processing and memory resources in embedded mpsoc architectures. In Proceedings of the conference on Design, automation and test in Europe (DATE), 2010.

[20] A. K. Coskun, T. S. Rosing, and K. A.Whisnant. Temperature aware task scheduling in mpsocs. In Proceedings of Design, Automate and Test in Europe Conference and Exhibition (DATE), 2007.

[21] Y. Xie and W.-L. Hung. Temperature-aware task allocation and scheduling for embedded multiprocessor systems-on-chip (mpsoc) design. Journal of VLSI Signal Processing, 45:177(189), 2006.

[22] 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.

[23] Q. Tang, S. K.S.Gupta, and G. Varsamopoulos. Energy-efficient thermal-aware task scheduling for homogeneous high performance computing data centers:a cyber-physical approach. IEEE Transactions on Parallel and Distributed Systems, 19:1458-1472, 2008.

[24] K. Kanoun, N. Mastronarde, D. Atienza, and M. V. D. Schaar. Online energy-efficient task-graph scheduling for multicore platforms. IEEE Transactions on Computer Aided Design, 33(8), 2014.

[25] P.-H. Tseng, P.-C. Hsiu, C.-C. Pan, and T.-W. Kuo. User-centric energy-efficient scheduling on multi-core mobile devices. In Design Automation Conference (DAC), 2014.

[26] R. P. Dick, D. L. Rhodes, and W. Wolf, "Tgff: Task graphs for free," in the 6th International Workshop on Hardware/Software Codesign, 1998, pp. 97–101.