An Effective Technique to Higher Throughput for Streaming Applications on an MPSoC

Hassan Salamy*
Electrical & Computer Engineering, University of Saint Thomas, 2115 Summit Ave, Saint Paul, MN 55105, USA.

* Corresponding author. Tel.: (651) 962-5758; email: hsalamy@stthomas.edu
Manuscript submitted March 22, 2019; accepted May, 2019.
doi: 10.17706/jcp.14.8.507-518

Abstract: In the era of multicore system architectures, it is empirical to develop efficient techniques to utilize the added compute power. High speed schedules are one of the main benefits that can be extracted from such systems. This is especially important for embedded systems that are required to run highly complex applications fast. Embedded systems often employ a multi-processor system-on-a-chip (MPSoC) to enhance performance. Hence in this article, we present an effective technique to scheduling of multiple streaming applications on an MPSoC with the objective of maximizing throughput. The proposed technique is implemented and tested on real and hypothetical systems on a range of benchmarks and the results show the great improvement in the systems' throughput generated from the presented technique.

Key words: MPSoC, pipelining, scheduling, throughput.

1. Introduction

Multi-core architectures whether in high performance computing or embedded systems are now the architecture of choice. They are widely used to get a performance boost and to lift the hard to overcome physical and performance constraints imposed to extract more power from a single core architecture. Multi-core architectures while powerful, require the development of proper techniques to effectively extract their power. Without such effective techniques, the multi-core architecture will not meet the performance expectations. In embedded systems, often a multi-processor system-on-a-chip (MPSoC) is used.

Multi-processor system-on-a-chip (MPSoC) is a computing architecture that utilizes multiple processing elements (possibly heterogeneous), a memory hierarchy, and a communication platform optimized for better performance. MPSoCs often utilize a scratchpad memory in addition or instead of a cache. Cache employs hardware based control that adds to the complexity of the design. Moreover, cache hits and misses often create problems in real time embedded systems where predictability is required for enhanced performance. Scratchpad memories, on the other hand, are software controlled and hence the predictability requirement is satisfied.

With the added potential performance increase from an MPSoC, a heavier burden are now on the shoulders of researchers to devise efficient methods to extract this compute power. Often multiple applications are utilizing an embedded system and in many scenarios these applications are streaming under possibly different set of data inputs. In this case of streaming applications where an application's instances are continuously getting executed in the system, improving the throughput of the system tend to be of higher

performance priority. In this case, the main metric is not how fast the system can execute one instance of the application but rather the reduction of the initiation interval (II) that is defined as the time between two instances of an application. Reducing the initiation intervals of the applications increases the systems throughput as more instances of the applications will finish executing in the system in a shorter period of time. Under this performance metric, executing one instance of the application might not be the fastest but the throughput of the whole system under multiple and continuous streaming of applications' instances will be optimized.

In this article, an optimized and effective heuristic is devised to enhance the throughput of an MPSoC system utilizing an SPM memory component for a stream of applications. The technique captures the importance of integrating memory allocation and scheduling for the applications in the system to improve the system's throughput. Traditionally the two steps of memory allocation and scheduling are dealt with as two decoupled steps. However, an integrated approach is presented as the memory allocated to a processor to execute a task of an applications greatly affects the execution time and hence the decision to allocate a task to a processor should be integrated with the decision of how to partition the on-chip memory in the system.

The rest of the article is organized as follows. Section 2 presents the related work. Section 3 presents the problem formulation with the proposed solution presented in Section 4. Section 5 presents our extensive results to test the proposed techniques. Finally Section 6 presents the conclusions.

2. Related Work

Many research groups have studied the problem of task scheduling of applications on multiple processors where the objective is to minimize the execution time. The authors in [1] 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. 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 are presented in [2]. Micheli *et al.* [3] studied the mapping and scheduling problem onto a set of processing elements as a hardware/software codesign. Neimann and Marwedel [4] used integer programming to solve the hardware/software codesign partitioning problem. A tool for hardware-software partitioning and pipelined scheduling based on a branch and bound algorithm was presented in [5]. Their objective was to minimize the initiation time, number of pipeline stages, and memory requirements. Similarly, in [6] an ILP solution for the partitioning problem with pipeline scheduling is proposed. An accurate scheduling model of hardware/software communication architecture to improve timing accuracy is presented in [7]. March *et al.* [8] presented a power-aware scheduling techniques based on task migration in systems implementing DVFS capabilities.

Many authors have studied the memory allocation problem in MPSoCs. The main focus of their research work is data parallelism in the context of homogeneous multiprocessor systems. Meftali *et al.* [9] formulated the memory allocation problem as an ILP to obtain an optimal distributed shared memory architecture to minimize the global cost to access shared data as well as the memory cost. In [10], the authors presented a memory-centric scheduling technique to improve hard real-time utilization. The scheduling of memory intensive periodic tasks on a partitioned multicore real-time system is studied in [11]. The authors assumed no shared resources among cores. The paper also presented scheduling real-time tasks that comply with the PRedictable Execution Model (PREM). Blagodurov *et al.* [12] presented a contention-aware scheduling algorithm on multicore systems. Vaidya *et al.* [13] 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. Power and energy efficient scheduling on multicore systems has been studied in [14] and [15].

Suhendra *et al.* [16] studied the problem of integrating task scheduling and memory partitioning among 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. Xue *et al* [17] presented techniques to divide the MPSoC system resources among multiple applications in the system based on the internal structure of each application to reduce the schedule time. Kim *et al.* [18] provided an effective scheduling technique based on cluster slack optimization where the tasks are iteratively clustered and each cluster is optimized by using a branch and bound technique. Their technique is applied on a pipelined data stream. Liu *et al.* [19] presented techniques to task assignment and cache partitioning with cache locking to minimize WCET on an MPSoC. They presented a joint task assignment and cache partitioning techniques utilizing cache locking to guarantee a precise WCET.

Many research has been done on task scheduling problems for DVS enabled multi-processor real-time embedded systems [20]. An adaptive method to eliminate hot spots in a slightly better way than the load balancing techniques by reducing temporal and spatial temperature variations is presented in [21]. Zhao and Gu [22] proposed a dynamic task scheduling on an NoC-based multiprocessor design to reduce energy consumption. Their technique consists of an offline part using optimization tools and online part using an efficient heuristic. An online task scheduling policy for energy-aware computing onto large multi-threaded multiprocessors is presented in [23]. In [24], a multi-objective scheduling algorithm based on decomposition for scheduling of the system workflow is presented. In [25], the authors proposed an energy-aware processor merging (EPM) algorithm to select the most effective processor to turn off from the energy saving perspective, and a quick EPM (QEPM) algorithm to reduce the computation complexity of EPM.

3. Problem Definition and System Architecture

Given a number of streaming applications in a multi-processor system with software controlled scratchpad memories (SPM) and large off-chip memory, the presented technique will generate a scheduling of the tasks to maximize the throughput from the system. In the case of streaming applications, each application will be continuously executed on possibly different set of data inputs. Streaming applications are widely used in many fields such as digital signal processing (DSP). The schedule for the streaming applications is referred to as pipelined scheduling since as to be explained later on, tasks from different instances will be executed in parallel within each pipeline stage.

Maximum throughput does not necessarily result in the shortest execution time of one application in the system but rather the overall throughput of the system with multiple streaming applications running continuously is optimized. In such scenario, the main metric is not how fast the system finishes running one instance of an application but rather how short is the initiation interval (II). The initiation interval is defined as the time between running two instances of the same application. As the II interval is reduced, the throughput of the system is improved.

To clarify this point, consider the simple example depicted in Fig. 1. Fig. 1-(a) presents a task dependence graph of a simple application of 4 tasks. A hypothetical run time of each task is noted in the parenthesis. A possible schedule of the original TDG in Fig. 1-(a) is shown in Figure 1-(c) and it requires a total of 14 time units. Fig. 1-(b) represents a possible updated TDG of the streaming application along with a possible schedule in Fig. 1-(d) including tasks from the current instance represented as T_i, the previous instance represented as T'_i, and the next instance represented as T''_i. The schedule in Fig. 1-(d) presents two stages. Each pipeline stage must include all the tasks but they could be from different instances. Since T_1 and T'_2 are from different instance, they can now run in parallel. Please note that in the schedule in Fig. 1-(d) the tasks T_1 T_2 T_3 and T_4 are scheduled within two stages but they still abide by the dependencies in the original TDG in Fig. 1-(a). From the pipelined schedule for a streaming application, it will require the current instance

of the application 22 time units which is higher than the 14 time units in the non-pipelined case. However, since the intention is to improve the throughput of the system in the case of pipelined scheduling, an equivalent of an application instance will finish executing in the system every 11 time units. Hence to finish three instances of an application utilizing the pipelined schedule, 33 time units are needed compared to 42 time units for the case of no pipelining.

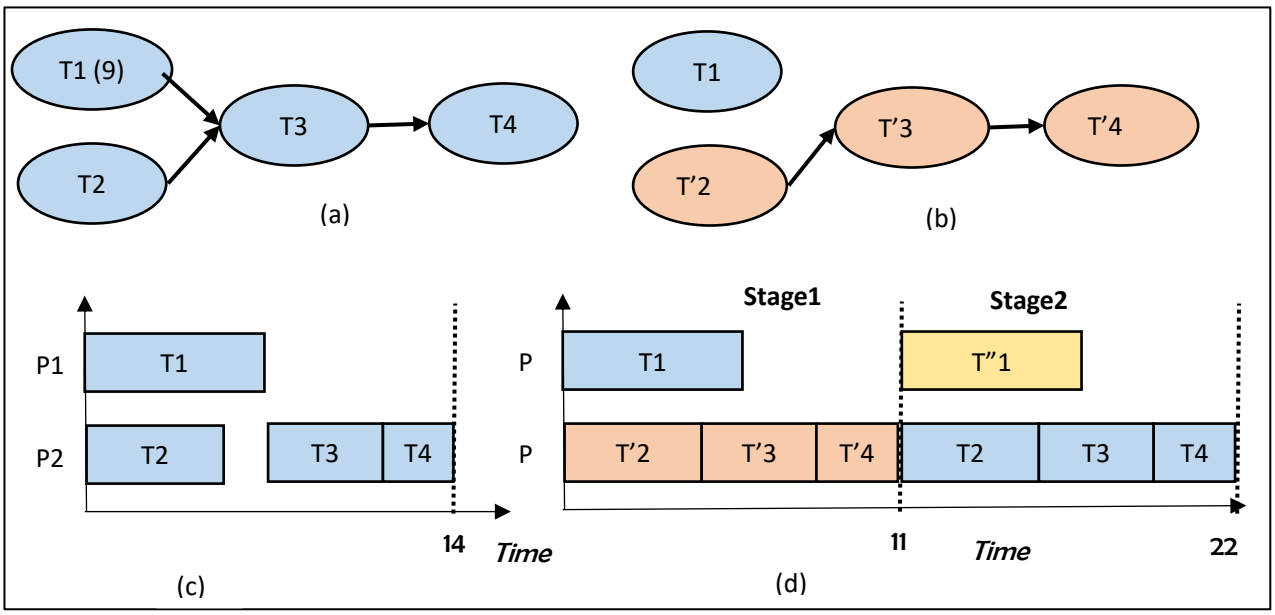

Fig. 1. (a) An example TDG, (c) Schedule with no pipelining, (b) TDG for pipelining, (d) Schedule with pipelining.

The problem to be solved in this article is now defined as follows. Given a set of streaming applications that will continuously run on the same or different set of input data, an MPSoC architecture with SPM, find an optimized pipelined schedule utilizing an integrated approach to task scheduling and memory partitioning to increase the overall system throughput by efficiently using the system resources. An instance of all the applications streaming in our system will be modeled using a Directed Acyclic Graph (DAG) with each application task modeled as vertex and an edge between two tasks represent dependencies and hence might incur communication cost.

The proposed framework will receive a set of applications. The application will them be profiled to extract important information. One of the main outcomes of the profiler is the task dependence graph (TDG) of each application in the system. The task dependence graph is a directed acyclic graph with a vertex to represent a task and an edge between two vertices to represent data dependency. For instance, an edge between two tasks, say *i* and *j*, means that task *j* should wait till task *i* finishes executing before it can start executing. Task *j* will have to also wait for the time required for the dependent data to be communicated from task *i*.

The profiler will examine the structure of each application and will locate the basic blocks within each application. The basic blocks will be called tasks and will be represented as the vertices in the TDG. The profiler will also examine the data and control flow between each two tasks for data dependency. Dependencies will be represented as weighted edges in the task dependence graph with the weight representing the communication cost for the dependent data between two dependent tasks.

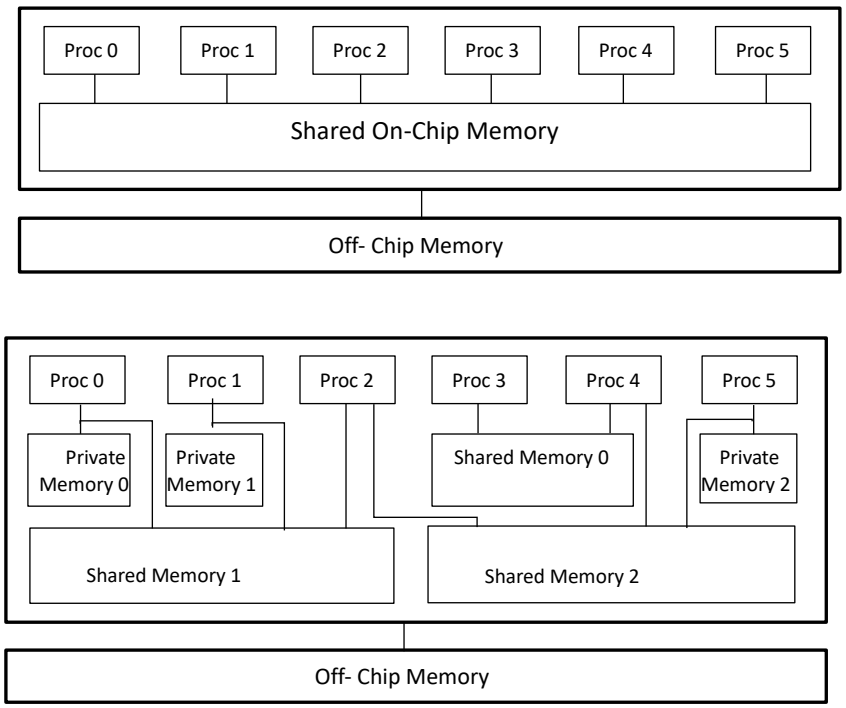

Fig. 2. An MPSoC with (a) pure shared on-chip memory, (b) hybrid on-chip memory.

The underlying architecture considered in this paper is a multi-processor system-on-a-chip that constitutes of multiple processing cores, a memory hierarchy, a communication platforms such as a network-on-a-chip and a set of input/output. Examples of two MPSoCs architectures are depicted in Fig. 2 that show a system with pure shared on-chip memory and another system of hybrid on-chip memory where some memory is shared between a number of processors and some memory elements are pure private.

4. Pipelined Scheduling and Memory Partitioning

The proposed approach starts by trying to reduce critical paths in the task dependence graph by mapping tasks on the critical paths to different instances and hence cutting the critical path into subpaths Fig. 3. Critical paths tend to increase the schedule time due to dependencies and hence reduced possibility of parallel execution of tasks. First, add to the task dependence graph (TDG) a dummy start node (vertex) with outgoing edges to all the nodes with no incoming edges and a dummy end node with ingoing edges from all nodes in the TDG with no outgoing edges. Then the critical paths are to be found. When an edge is removed in a TDG to reduce the critical path, two sub TDGs are created. The tasks in each of the two subgraphs will now belong to two different instances of the application. Any time an edge is removed, all the corresponding edges that connect the two sub graphs will also be removed. The *TDG* can at most be divided into *s* sub graphs where *s* is the number of pipeline stages. The number of pipeline stages is set to be at most the number of processors in the multi-processor system.

The critical paths will be determined based on the estimated execution time of the tasks on that path. The estimated execution time of a task detailed next is highly dependent on the processor allocated to execute the task as well as the on-chip fast scratchpad memory allocated to that processor. Memory and task allocation are often studied as two independent and consecutive problems. Hence usually tasks are first allocated to processors and then the memory budget is divided among the processors following a certain criteria. However, the task allocation should also be dependent on memory allocation as tasks will exhibit different run times based on the on-chip memory allocated. Hence task scheduling and memory allocation are interdependent problems and should be considered in an integrated fashion rather than decoupled

fashion. Hence after dividing the original task dependence graph into subgraphs that is dividing the tasks into stages , the tasks from different instances will then be scheduled based on an integrated task scheduling and memory partitioning approach.

```
Pipeline_Stages()
1.  Profile the applications.
2.  Add a dummy start Node s.
3.  Add outgoing edges from s to vertices with no parents.
4.  Add a dummy end Node e.
5.  Add ingoing edges to e from all vertices with no children.
6.  Find the paths from the start to end.
7.  Find the critical paths.
8.  Divide the critical path into n paths of close execution times.
9.  Create n unconnected subgraphs based on the point of cut of the
    critical path.
10.  Label the tasks in the new TDG to reflect previous, current and next
     instance.
```

Fig. 3. Creating the pipeline stages.

Hence, the estimated execution time (*EET*) depends not only on the current memory allocations on the processors but rather it presents an estimation for the run time for different future memory allocations. This is important since the memory allocation will vary throughout the proposed heuristic and the decision of deciding the critical path as well as the task allocation should look beyond the current memory allocation to possible future SPM budget distributions. In the beginning, the on-chip scratchpad memory is assumed to be equally divided among the available processing cores. The estimated execution time (*EET*) is defined in *Equation 2* and it is dependent on an *estimate*(T_{ij}) (*Equation 1*) that estimates the reduction in execution time due to possible additional scratchpad memory allocated to the processor P_j executing a certain task T_i. *estimate* presents an estimation to the extent the run time of a task will decrease on a processor if higher SPM budget is allocated to it. It is a number between zero and one based on the run time $Run(T_{ij})$ of task T_i under the current memory allocation and the lowest possible ($Low(T_{ij})$) run time if all the SPM memory is allocated to the processor P_j executing task T_i.

$$estimate(T_{ij}) = \frac{Run(T_{ij}) - Low(T_{ij})}{Run(T_{ij})} \tag{1}$$

$$EET(P_j) = End(P_j) - \sum_{T_i \in P_j} \left(Run(T_i) - \frac{Run(T_i)}{1 + estimate(T_i)} \right) \tag{2}$$

The proposed heuristic in Fig. 4 will start with the tasks in the new resultant task dependence graphs (subgraphs) for pipelined scheduling sorted in increasing order of *ASAP* values and stored in a list *List*. The *ASAP* values are calculated based on the execution times of the tasks assuming the SPM budget is equally divided among all the processors. Following the order of the tasks in the *List*, each task will be allocated to be scheduled to the appropriate processor that results in minimal increase in the schedule time.

```
Task Scheduling()
1. Get the new TDGs from Pipeline_Stages().
2. Get the available system resources.
3. Find the minimum run time of each task Ti on processor Pj, Low(Tij ).
4. Divide the on-chip SPM memory equally between the processors.
5. Find the run time of each task based the equally partitioned SPM budget.
6. Find ASAP for all the tasks based on the equally partitioned SPM budget.
7. List = List of tasks in increasing order of ASAP.
8. While (List not empty) do:
9.    T1 = the first task in List.
10.   For every processor Pi:
11.      Find the estimate and EET values of Pi if T1 is executed on Pi.
12.      Find the earliest start time of T1 on Pi.
13.      Find Finish(Pi) if T1 is executed on Pi.
14.      if ((Finish(Pi) < min && EET(Pm) ≥ (1 - δ%)EET(Pi))||
                    (Finish(Pi) > min && EET(Pi) ≤ (1 - δ%)EET(Pm)))
15.         min = Finish(Pi).
16.      else if (Finish(Pi) = min)
17.         min = Finish time of the processor with the higher estimate.
18.    Allocate T1 to processor Pj of min finish time.
19.    Remove T1 from List.
20. End While
```

Fig. 4. The proposed integrated scheduling heuristic.

The *Begin*(T_i, P_j) is the start time of task T_i on processor P_j and is defined in *Equation 3* as the earliest possible start time of the task on the processor while meeting all the dependency constraints from all the parent tasks. The communication cost is also taken into account in *Equation 3*. The Finish time of a task is simply calculated in *Equation 4* as the sum of the *Begin* time and the execution time of the task *Execute* under the current memory allocation. The finish time *Finish* of a processor in *Equation 5* is equal to the finish time of the last task scheduled on that processor.

$$Begin(T_i, P_j) = Max\left(Max\left(Finish_{T_k \in Parent(T_i)}(T_k)\right), Finish(P_j)\right) + \; Max\left(Comm\_Cost_{T_k \in Parent(T_i)}(T_k)\right) \tag{3}$$

$$Finish(T_i) = Begin(T_i, P_j) + Execute(T_i) \tag{4}$$

$$Finish(P_j) = Max\left(Finish_{T_k \in P_j}(T_k)\right) \tag{5}$$

Generally a task T_i is scheduled on processor P_j that results in the minimum increase in the schedule time. However, since the proposed scheduling and allocation algorithm is memory-aware, task T_i may be scheduled on processor P_m if the estimated execution time (*EET*) (*Equation 2*) of P_m is at least 12% less than the EET value of processor P_j. This is because a lower *EET* value is an implication that the schedule time of tasks on a certain processor is susceptible to higher reduction if more scratchpad memory is allocated to that processor in the future beyond the current memory allocation. The 12% value is found through fine tuning. The SPM memory allocation might change after a scheduling of each task. After a task is scheduled, the processor with the highest finish time is allocated more *SPM* memory with the intention to balance the schedule and hence

reduce the schedule time. This is achieved by transferring 10% of the SPM budget allocated to the processor with the lowest *Finish*Estimate* to the processor with the highest *Finish* time provided that this does not result in a longer schedule.

5. Experimental Setup and Results

For the experimental testing, a C based simulator is created under different system architectures to cover a wide range of architecture designs. We mainly utilized an Intel Xscale, ARM Cortex-A53 and hypothetical processors with cores ranging from 2 to 8. In the experimental testing, it is assumed an on-chip fast memory ranging from 20kB to 4 MB. We also utilized hypothetical cores mainly based on AMD Ryzen 7 1800X 8-Core 3.6 GHz desktop processor but with scaled down frequencies to mimic realistic embedded processors.

We implemented and tested the following four methods:

- *Our Approach*: The presented integrated approach to memory-aware scheduling with pipelining.
- *Our Approach_No_Pip*: The presented integrated approach to memory-aware scheduling with no pipelining.
- *Optimal_Dec_Pip*: An optimal ILP solution based on an altered version of [16] of a decoupled solution with pipelining.
- *Optimal_Integ_No_Pip*: An optimal ILP solution based on [16] of an integrated solution with no pipelining.

For the first set of experiments, the proposed pipelined and memory-aware scheduling heuristic is compared against an optimal ILP based pipelined approach where task scheduling and memory partitioning are considered as two decoupled steps, *Optimal_Dec_Pip*. A set of applications were tested multiple times under different processor and scratchpad memory budget and the average results for each application is depicted in Fig. 5. In this set of experiments, it is assumed that only one application is utilizing the system at the same time and is being streamed continuously. The following set of application were tested mainly extracted from *Mediabench* and *Mibench* [26], [27], *Lame* and *Osdemo*. Each application is profiled using Simplescalar [28] to extract important information such as the task dependence graph (TDG). Three semi random benchmarks named *Bench1*, *Bench2* and *Bench3* are also created and tested. The number of processors in the system is varied between 2 and 8 processors and the SPM memory budget is varied between 256KB and 2MB. It is important when testing to choose a proper processor count and memory budget as too few or too many system resources will fail to reflect the quality of the proposed technique. As depicted in Fig. 5, *Our Approach* is able to improve the throughput of the system by up to 17.5% (11.1% on average) compared the optimal decoupled approach *Optimal_Dec_Pip*. As expected, the results clearly show the importance of the integrated approach over the decoupled approach.

For the second set of experiments, we compared our approach against an integrated technique that does not consider pipelining, *Optimal_Integ_No_Pip*. A set of applications were tested multiple times under different processor and scratchpad memory budget and the average results for each applications is depicted in Fig. 6. In this set of experiments, it is also assumed that only one application is utilizing the system at the same time and is being streamed continuously. We tested the same set of real life benchmarks and semi randomly created benchmarks as the same set of experiments namely, *Lame*, *Osdemo*, *Bench1*, *Bench2* and *Bench3*. The number of processors in the system is varied between 2 and 8 processors and the SPM memory budget is varied between 256KB and 2MB. As depicted in Fig. 6, *Our Approach* is able to improve the throughput of the system by up to 26.2% (21.3% on average) compared to the optimal decoupled approach *Optimal_Integ_No_Pip*. As expected, the results clearly show the importance of the pipelining in improving

the throughput of the system.

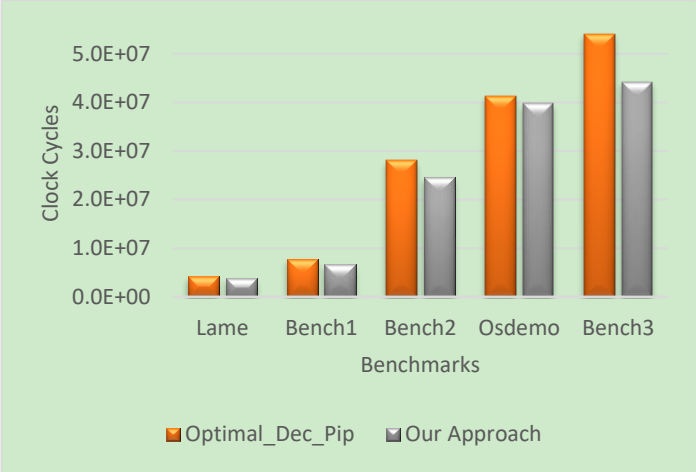

Fig. 5. Results for integrated vs decoupled.

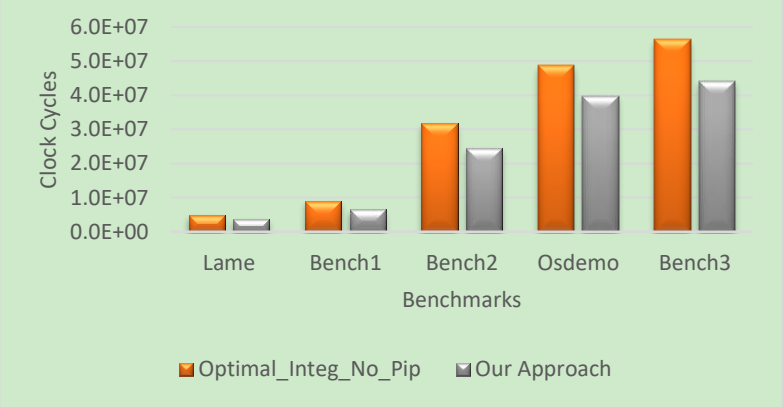

Fig. 6. Results for pipelined vs non-pipelined.

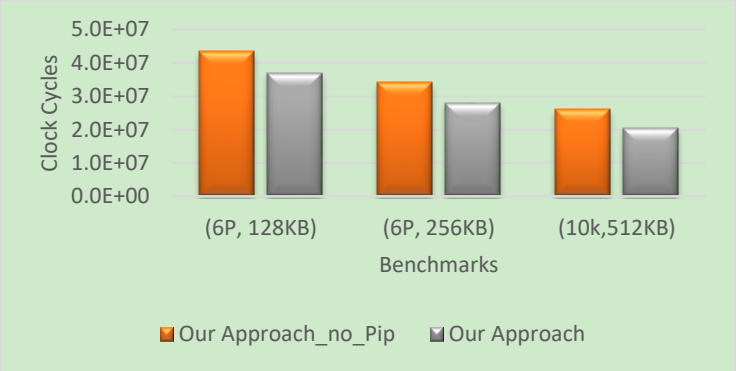

Fig. 7. Results for Set (1) *Lame, Cjpeg* and *Osdemo.*

The last set of experimental testing was a continuation of the second set of experiments but assuming there are multiple applications utilizing the system at the same time that are also to be continuously streaming. Two sets of benchmarks were created and tested: Set (1) *Lame*, *Cjpeg* and *Osdemo* and Set (2) *Lame*, *Cjpeg*, *BM1*, *BM2*. The sets are tested multiple times under different processor count and memory budget and the average results for each set are depicted in Fig. 7 and 8. Due to the size of the applications and the corresponding TDGs, optimal ILP-based solutions are no longer feasible since the run time is prohibitive.

Hence we compared *Our Approach* to our approach altered so that it does not include pipelining, *Our Approach_No_Pip*. The number of processors in the system is varied between 4 and 16 processors and the SPM memory budget is varied between 128KB and 2MB. As the results show, our pipelined schedule technique improved the results up to 25% with 21% on average improvement compared to no pipelining.

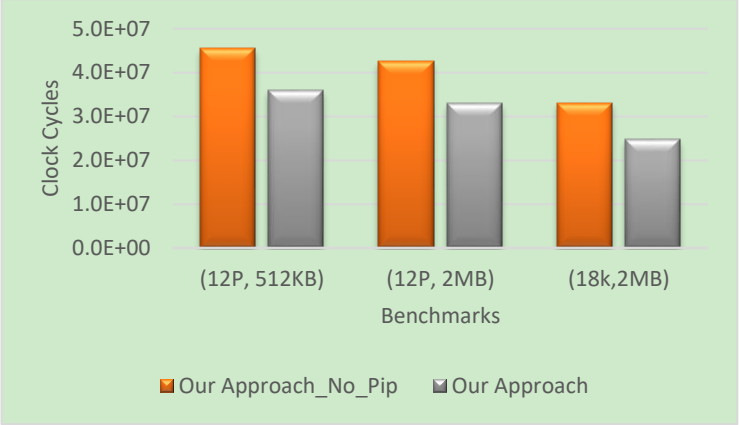

Fig. 8. Results for Set (2) *Lame, Cjpeg, BM1, BM2*.

6. Conclusion

In this article, an effective technique to improve the throughput of a multiprocessor system with a set of streaming applications is presented. The proposed technique integrates task scheduling and memory partitioning to further reduce the schedule time. Results on multiple set of benchmarks showed the effectiveness of our technique compared to decoupled techniques and to techniques with no pipelining.

References

[1] Benini, L., Bertozzi, D., Guerri, A., & Milano, M. (2005). Allocation and scheduling for mpsoc via decomposition and no-good generation. *International Joint conferences on Artificial Intelligence (IJCAI),* 107-121.

[2] Kwok, Y. K., & Ahmad, I. (1999). Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing, 59(3),* 381-422.

[3] Micheli, G. D., Ernst, R., & Wolf, W. (2002). Readings in hardware/software co-design. *Morgan Kaufmann.*

[4] Neimann, R., & Marwedel, P. (1996). Hardware/software partitioning using integer programming. *Design Automation and Test in Europe (DATE),* 473-479.

[5] Chatha, K. S., & Vemuri, R. (2002). Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Transactions on VLSI, 10(3),* 193-208.

[6] Kuang, S. R., Chen, C. Y., & Liao, R. Z. (2005). Partitioning and pipelined scheduling of embedded systems using integer linear programming. *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)* (pp. 37-41).

[7] Cho, Y., Zergainoh, N. E., Yoo, S., Jerraya, A., & Choi, K. (2007). Scheduling with accurate communication delay model and scheduler implementation for multiprocessor system-on-chip. *Design Automation for Embedded Systems, 11(2),* 167-191.

[8] March, J. L., Sahuquillo, J., Petit, S., Hassan, H., & Duato, J. (2012). Power-aware scheduling with effective task migration for real-time multicore embedded systems. *Concurrency and Computation: Practice and Experience, 25(14),* 1987-2001.

[9] Meftali, S., Gharsalli, F., Rousseau, F., & Jerraya, A. (2001). An optimal memory allocation for application-

specific multiprocessor system-on-chip. *Proceedings of International Symposium on Systems Synthesis (ISSS).*

[10] Yao, G., Pellizzoni, R., Bak, S., Betti, E., & Caccamo, M. (2012). Memory centric scheduling for multicore hard real-time systems. *Real-Time Systems, 48,* 1-35.

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

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

[13] Vaidya, V. G., Ranadive, P., & Sah, S. (2010). Dynamic scheduler for multi-core systems. *Proceedings of the 2nd International Conference on Software Technology and Engineering(ICSTE).*

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

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

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

[17] Xue, L., Ozturk, O., Li, F., Kandemir, M., & Kolcu, I. (2012). Dynamic partitioning of processing and memory resources in embedded mpsoc architectures. *Proceedings of the 2nd International Conference on Software Technology and Engineering (ICSTE).*

[18] Kim, J., Lee, S., & Shin, H. (2013). Effective task scheduling for embedded systems using iterative cluster slack optimization. *Circuits and Systems, 4(8).*

[19] Liu, T., Zhao, Y., Li, M., & Xue, C. (2011). Joint task assignment and cache partitioning with cache locking for wcet minimization on mpsoc. *Journal of Parallel and Distributed Computing, 71(11),* 1473-1483.

[20] Xie, Y., & Hung, W. L. (2006). Temperature-aware task allocation and scheduling for embedded multiprocessor systems-on-chip (mpsoc) design. *Journal of VLSI Signal Processing, 45(3),* 177-189.

[21] Coskun, A. K., Rosing, T. S., & Whisnant, K. A. (2007). Temperature aware task scheduling in mpsocs. *Proceedings of Design, Automate and Test in Europe Conference and Exhibition (DATE)* (pp. 1-6).

[22] Zhao, X., & Gu, M. (2012). A novel energy-aware multi-task dynamic mapping heuristic of noc-based mpsocs. *International Journal of Electronics, 100(5),* 603-615.

[23] Ghose, M., Sahu, A., & Karmakar, S. (2016). Energy efficient online scheduling of aperiodic real time task on large multi-threaded multiprocessor systems. *Proceedings of IEEE Annual India Conference (INDICON),* (pp. 1-6).

[24] Yuan, S., Deng, G., Feng, Q., Zheng, P., & Song, T. (2017). Multi-objective evolutionary algorithm based on decomposition for energy-aware scheduling in heterogeneous computing systems. *J-JUCS, 23*, 636-651.

[25] Xie, G., Zeng, G., Li, R., & Li, K. (2017). Energy-aware processor merging algorithms for deadline constrained parallel applications in heterogeneous cloud computing. *Sustainable Computing, IEEE Transactions on, 2(2),* 62-75.

[26] Lee, C., Potkonjak, M., & Mangione-Smith, W. (1997). Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *Proceedings of IEEE International Symposium on Microarchitecture* (pp. 330-335).

[27] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., & Brown., R. B. (2001). Mibench: A free, commercially representative embedded benchmark suite. *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*.

[28] Austin, T., Larson, E., & Ernst, D. (2002). Simplescalar: An infrastructure for computer system modeling. *IEEE Computer, 35(2),* 59-67.

Hassan Salamy is an associate professor of computer and electrical engineering at the University of Saint Thomas in Saint Paul, MN, USA. Before joining the University of Saint Thomas, he was an electrical engineering faculty member at Texas State University. Throughout his academic career, Dr. Salamy taught a wide range of courses in electrical, computer, and software engineering as well as computer science. His research interests are in the areas of embedded systems, high performance computing, computer architecture, renewable technologies, and engineering education.