Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC

Hoeseok Yang
School of EECS
Seoul National University
Seoul, Korea
hyang@iris.snu.ac.kr

Soonhoi Ha
School of EECS
Seoul National University
Seoul, Korea
sha@iris.snu.ac.kr

***Abstract*—In this paper, we propose a multi-task mapping/scheduling technique for heterogeneous and scalable MPSoC. To utilize the large number of cores embedded in MPSoC, the proposed technique considers temporal and data parallelisms as well as task parallelism. We define a multi-task mapping/scheduling problem with all these parallelisms and propose a QEA(quantum-inspired evolutionary algorithm)-based heuristic. Compared with an ILP (Integer Linear Programming) approach, experiments with real-life examples show the feasibility and the efficiency of the proposed technique.**

I. INTRODUCTION

Insatiable demand of system performance makes it inevitable to integrate more and more processing elements in a single chip, called MPSoC (Multi-Processor System on a Chip), to meet the performance requirement. Recently systems that have a lot of cores are about to appear in market and academia [1][2]. The system with such a high degree of parallelism raises a challenge: how to extract parallelisms from applications and exploit them efficiently.

Parallelisms can be categorized into three types: task, data, and temporal parallelism. Task parallelism is achieved by executing multiple tasks on different cores concurrently. Data parallelism is achieved by instantiating multiple instances of a task and running them with different input data sets simultaneously. By dividing an iteration of a task execution into several pipeline stages, we can exploit temporal parallelism.

We assume that an application task is specified by a task graph which consists of graph nodes and edges as shown in Figure 1(a). Each node represents a computation module, also called a sub-task, while an edge indicates data dependency between two end nodes. The numbers annotated on an edge indicate the number of data samples produced and consumed by two end nodes per each execution. For instance, on edge A-C each invocation of node A produces 2 data samples while node C consumes 1 data sample per execution. Thus node C should be executed twice more frequently than node A in order not to accumulate data samples on the edge unboundedly. A token marked on an edge denotes an initial data sample, which defines a delayed dependency between two end nodes. For instance, a token on edge E-D makes the n'th execution of node D dependent on the n-1'th execution of node E.

This work was supported by BK21 project, System IC 2010 project of Korean Ministry of Knowledge Economy, and Acceleration Research sponsored by KOSEF research program (R17-2007-086-01001-0). The ICT at Seoul National University provided research facilities for this study.

Since the task graph represents the true data dependency between sub-tasks, we can exploit task parallelism from the given specification by scheduling the task graph as shown in Figure 1(b). The horizontal axis represents the elapsed time to run the nodes. After node A is executed, both nodes B and C can be executed, which are scheduled on two different processors. The schedule shows that we have to pay communication overhead between processors to deliver data samples. And, two instances of node C are scheduled. We assume that the application task has a latency or throughput constraint that is marked as a dashed vertical line in Figure 1(b). In this example, the timing constraint cannot be met by exploiting task parallelism only.

We express data parallelism of a task graph in two ways. When multiple data sets are fed into a node and the node can process them in parallel, one invocation of the node can be partitioned into multiple processors. We call this node a data-parallel node and represent it as a shaded node like node B in Figure 1. It is given a priori whether a node is data parallel or not by the programmer of the node. In this example, node B can be mapped into 2 processors to process one input sample on each processor. Exploiting this data parallelism can reduce the latency as shown in Figure 1(c) but yet violating the time constraint. There is another way of exploiting data parallelism: multiple invocations of the same node can be run concurrently with different input data samples, as node C in the graph. If node C does not have any internal state that should be maintained between invocations, two invocations of node C can be concurrently executable, which is not shown in the figure.

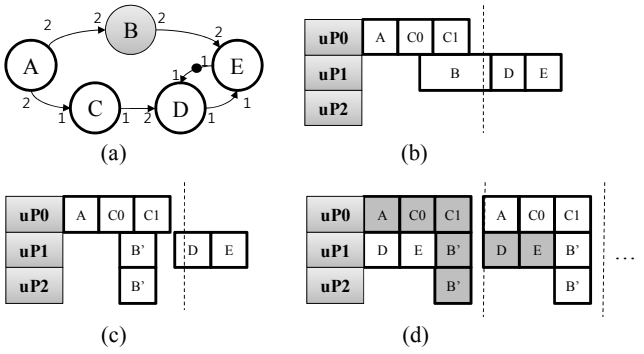


Figure 1. (a) A task graph example, (b) a schedule with task parallelism, (c) a schedule with both task and data parallelism, and (d) a schedule with all three types of parallelisms.

978-3-9810801-5-5/DATE09 © 2009 EDAA

Temporal parallelism can be obtained by pipelining the task graph. Dividing a task graph into several pipeline stages, the current iteration of the task can be overlapped with the previous iteration. For instance, in Figure 1(d) node D and E can be separated to another pipeline stage. For pipelined execution, additional buffers to store the output data of the previous stage should be inserted into edge B-E and C-D. In Figure 1(d), shaded nodes compose a whole iteration of the task, overlapped with the other iterations.

In this paper, a multi-task mapping/scheduling problem which considers three kinds of parallelisms altogether for heterogeneous MPSoC is defined. There is no previous work that tackles this problem before to our best knowledge. We propose a mapping/scheduling technique based on the *Quantum-inspired Evolutionary Algorithm (QEA)*. Compared with an Integer Linear Programming (ILP) technique that produces an optimal solution, the proposed technique shows a near-optimal result in significantly reduced time.

The rest of this paper is organized as follows. In section II, we review the related work and the QEA. The mapping/scheduling problem with three kinds of parallelisms is defined in section III. Section IV explains the proposed QEA solution as well as an ILP based solution. The effectiveness of the proposed technique is demonstrated by experiments in section V. Then, the conclusion is drawn in section VI.

II. RELATED WORK

A. Task Mapping and Scheduling

A multi-processor mapping/scheduling problem is a well-known NP complete problem even for homogeneous processors. Hence many heuristics have been proposed [3]. To cope with the increasing complexity of the problem, systematic approaches such as genetic algorithm based heuristics [4][5] and ILP based solutions [6] have been proposed. Most of them consider only task parallelism of an application.

Temporal parallelism by pipelining has been considered in some researches. Pipelined mapping/scheduling based on list-scheduling was proposed in [7], which maximizes the throughput of a DSP program for homogeneous multi-processor architecture. Another list scheduling based heuristic [8] which separates the component selection and the mapping/scheduling was proposed to minimize the hardware area cost under performance constraints. A branch-and-bound heuristic was proposed in [9] and integer programming based approaches were used in [10] and [11]. Compared with these works, we consider more general problems. In most previous approaches except [9], processors are not shared between separate pipeline stages, while our approach has no such limitation: nodes B and D share uP1 in Figure 1(d) for instance. Moreover, the input task graph may be a multi-rate and cyclic task graph with delays in the proposed approach.

Data parallelism has not been considered in most of the previous works. Recently an ILP based mapping/scheduling technique which considers both data parallelism and task parallelism was proposed in [12]. The proposed technique is the first technique, to our best knowledge, that considers three types of parallelisms altogether targeting general heterogeneous multiprocessor architectures.

B. Quantumn-insprired Evolutionary Algorithm

Evolutionary Algorithms (EA) operate a population of solutions for a given problem, selecting the best in each generation to make a better solution survive to final as in natural adaptation. Like all other EAs, a QEA also consists of the representation of individuals, the evaluation function, and the population dynamics [13]. The only difference is that it uses quantum bits as probabilistic representation for individuals instead of binary representation of genes. The probabilistic representation of quantum bit is described with two values, α and β, where $|\alpha|^2$ and $|\beta|^2$ mean the probability that the corresponding bit becomes 0 or 1 respectively, as shown in the lower part of Figure 2. This probabilistic representation makes QEA overcome some difficulties reported in [4] on fitting EA to mapping/scheduling problem: genetic operators such as crossover or mutation are severely restricted by mapping/scheduling constraint.

For the diversity of generated solutions, we can deploy multiple groups at the same time, each of which has its own quantum stream. Similarly, numerous individuals can be generated in the same group. This redundancy results in a better solution by preventing local optimal solution, even though too excessive duplication may slow down the proposed technique. In the figure, the length of individual stream is *m*, while the number of groups and generated individuals in a group are *n* and *p* respectively.

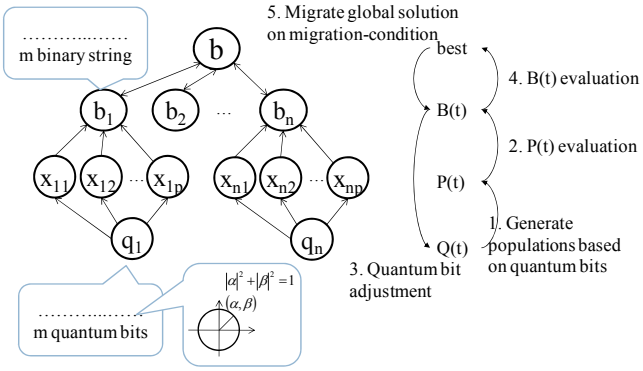


Figure 2. Overall QEA Procedure.

Overall procedure of QEA can be summarized as follows: 1) For each group, p individuals are generated based on the probability that quantum bits of the group implies. Each generated individual is repaired by some application dependent rules to make a valid solution. 2) Then, we evaluate the generated individuals and save the best solution in each group. 3) Based on the best solution of each group, the quantum stream is adjusted so as to make newly generated individuals getting close to the best solution in probability: α and β values are adjusted, which is called "quantum bit rotation". 4) Among the best solutions for all groups, the best one is saved for the highest solution. 5) If the termination condition is met, we stop the evolution process and output the highest solution. The

termination condition is usually defined by the maximum number of generations or by the case that all individuals are converged to the highest solution probabilistically. For the case that neither of above conditions are met, we go back to step 1 for the next generation.

III. PROBLEM DEFINITION

The target architecture assumed in this paper is a heterogeneous multi-processor system that consists of multiple pools of homogeneous processors. The architecture is scalable so that we can increase the number of pools as well as the number of processors in each pool. For instance, IBM CELL™ [1] has 2 processor pools: one pool has only one PowerPC processor (1 PPE) and the other has eight synergistic processors (8 SPEs).

For multiple tasks that have different periods, we expand the task graphs by their hyper-period (least common multiple of all periods). In addition to hyper-period expansion, the multi-rate task graph is expanded to equivalent single-rate task graph in the proposed technique to solve the problem more efficiently. Each node is repeated up to its execution ratio and every multi-rate edge is divided into single-rate edges. In Figure 1(a), node C should be instantiated into 2 nodes in the expanded graph. Note that multiple initial delay tokens are scattered to the corresponding single-rate edges. The conversion into the equivalent single-rate task graph was presented in [14].

In an expanded graph, many nodes may correspond to the multiple instances of the same node in the original task graph. We enforce that they should be mapped to the same processor to keep any internal state consistent if any. If they are mapped to different processors, we have to deliver internal states between processors while keeping the dependency order. So we avoid this situation.

Another restriction we enforce is that data parallel execution should be performed on the same pool of processors, which is quite a reasonable assumption in real situations. If the target architecture has a DSP array for example, a data parallel task is likely mapped solely to the DSP array for data parallel execution.

To evaluate the mapping and scheduling result, we assume that the cost of each processor, the buffer cost of each edge, and the worst case execution time of all nodes on each processor are given. If a data parallel node is mapped to multiple processors, the execution time of the node is simply divided by the number of mapped processors, which is a quite ideal assumption: this assumption can be released easily if the execution time table on the varying number of processors is given. Now we summarize the mapping/ scheduling problem tackled in this paper as follows:

Input:

Target architecture: a heterogeneous MPSoC that consists of multiple pools of homogeneous processors. Each pool has D processors at most.

Application tasks: a set of expanded task graphs with known execution times of each node on all processors and buffer cost of each edge. Data parallel tasks are marked beforehand.

Constraints: For Time constraints, deadline can be set for each task or for an individual sub-task node. In addition, the resource constraint specified as a weighted sum of the processor cost and the pipeline buffer size.

Problem:

For each node of the task graphs, determine the mapped processor (or processors if data parallelism is exploited) and the scheduled time considering the communication overhead between the processors. For each edge of task graphs, determine how many pipeline buffers inserted on it. The objective is to maximize the throughput, which is the same as minimizing the Initiation Interval (II) or to minimize the resource cost such as total processor or pipeline buffer cost while satisfying all deadline constraints in periodic tasks.

IV. PROPOSED SOLUTION

In the proposed technique, mapping and pipelining decisions are made by a QEA-based heuristic, where it is crucial to generate valid candidate solutions and properly evaluate them. Figure 3 shows the proposed Q-stream structure that represents an individual or a solution. It consists of two sections that represent the mapping and pipeline information separately.

For the mapping information, Q-bits are allocated on each sub-task as many as the total processors in the target architecture. In Figure 3, n sub-tasks and m pools exist while $pool_1$ had q processors. In the bit stream, the bits associated with the mapped processors become '1' and all other bits are '0'. On the other hand, p bits are allocated to represent pipeline information. Each bit in this section stands for a possible pipeline, which will be explained in detail later. In short, task and data parallelisms are determined by the mapping bits in the first section, while temporal parallelism is represented by the pipeline bits in the second section in a Q-stream individual.

<Design of Q-stream>

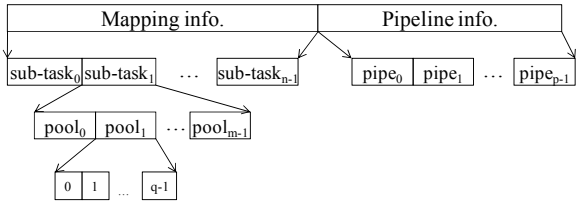


Figure 3. Q-stream structure of the proposed technique.

A. Mapping

When the QEA initially generates a solution by setting each Q-bit according to the associated probability, it may be an invalid mapping. So, it is fed to a repair function to be validated. Generating a valid mapping solution corresponds to exploring task and data parallelisms at the same time.

To be a valid mapping solution, two requirements should be satisfied. One is that at most one processor in each pool can be

selected if the sub-task is not data-parallelizable. If multiple processors are initially chosen for a sub-task, the repair function randomly selects one and invalidates all others. If it is a data-parallel sub-task, it can be mapped to multiple processors. But if the number of mapped processors is larger than the maximum data parallelism of the task, invalidate mapped processor randomly one by one till it becomes valid.

The other requirement is that only one pool should be selected for each sub-task. Every bit except the chosen pool is driven to 0. Pool selection should also be done randomly.

B. *Pipelining*

In contrast to the mapping solution, special care should be taken to enumerate all possible pipeline solutions. We may not select arbitrary edges for pipelines since it may break the task functionality. If we deploy pipeline buffers as in Figure 4(a), for instance, the functionality is broken at sub-task E: the other ports get delayed input samples by 1 iteration cycle while the bottom-most port gets delayed samples by 2 iteration cycles. A valid pipeline should be a cut that separates a graph into two sub graphs without cycle dependency. Various cuts are depicted in Figure 4(b). Cut *a* is not a valid pipeline since it makes a cycle dependency between pipeline stages.

Cuts *b*, *c*, and *d* do not make cycle dependencies between pipeline stages. Among them cuts *b* and *d* are valid while cut *c* is not. Note that cuts *c* and *d* both include delayed feedback edges (C-B and E-B). Pipelining of a task graph with cycles can be performed by retiming after inserting an imaginary feedback edge between the destination node and the source node with infinite number of delay tokens as shown in Figure 4(c). The retiming technique moves the initial data samples on the arcs without breaking the functionality of the task graph [7]. Figure 4(c) illustrates a case when nodes A, B, and C are retimed, where a dotted circle means consumption of an initial sample. After retiming, two pipeline buffers are inserted on edge A-D and C-E. It corresponds to cut d Figure 4(b). No retiming associated with cut c is possible, as it incurs a non-delayed cycle dependency.

In theory, there can be at most *n-1* pipeline cuts on *n* delayed feedback edge [7]. Since edge C-B has only a single delay, it can have zero pipeline cut. On the other hand, edge E-B has two initial samples, so can accommodate one pipeline cut.

Figure 4(d) shows a pipeline schedule associated with the graph of Figure 4(c) assuming that node A is a data parallel node. Nodes D and E are executed independently of A and C respectively. The dashed arrow shows that the 2-delayed data dependency from E to B is still kept.

To enumerate all possible pipelines and find a valid combination of them, we propose a novel data structure, called a Pipeline Ordering Graph (POG). Basically it enumerates all possible topological sorts of nodes starting from the destination node as shown in Fig 5(a). Without loss of generality, we assume that there is a single destination node in the task graph. A POG node is associated with a set of nodes that are partially topological-sorted.

Procedure of POG generation is as follows: Starting from a set that has the destination sub-task as its only element, investigate all predecessors of the elements in the set. If all out-going edges of a predecessor are directed into the set, make a new set adding the sub-task to the set. If there is no POG node associated with the newly made set, make a new POG node as a child node. If there is already a POG node associated with the newly made set, just add an edge to represent parent-child relationship. Repeat this process until a POG node that contains all sub-tasks is made.

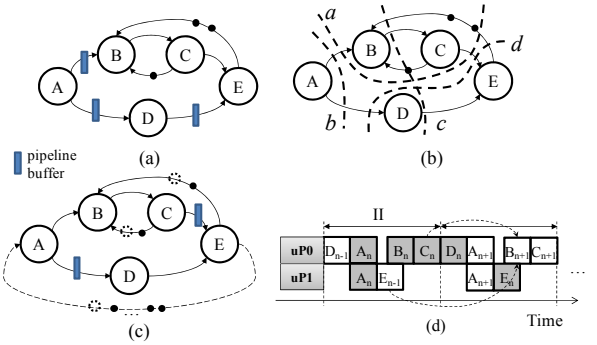


Figure 4. (a) A graph with pipeline buffers randomly inserted, (b) edge cuts, (c) a valid graph with pipeline buffers associated with cut d, and (d) a valid pipelined schedule.

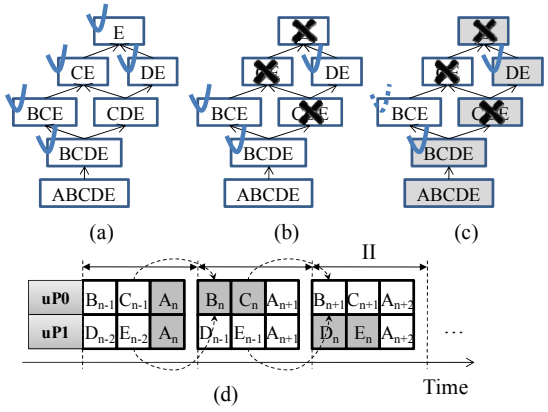


Figure 5. (a) The POG of the task grph of Figure 4, (b) a repaired POG, (c) a valid pipeline solution, and (d) a pipelined schedule associated with (c).

Figure 5(a) is the POG of the task graph of Figure 4. Starting from 'E', two nodes 'CE' and 'DE' are inserted to the POG as children of 'E' since all non-delayed output edges of both sub-task C and D are directed to sub-task E. Similarly, 'BCE' and 'CDE' are added, while 'ADE' cannot.

Each POG node represents a pipeline cut: for example POG node 'DE' denotes the pipeline cut *d* in Figure 4(c). We allocate as many Q-bits as the number of POG nodes to represent task graph's pipelines in Figure 3. Note that some POG nodes correspond to invalid pipelines such as POG node 'CE' associated with cut *c*. So we define a repair function to repair the invalid pipeline solutions.

Figure 5 shows the repairing procedure with an example. Suppose that 5 POG nodes ('V' marked) are chosen initially as shown in Figure 5 (a). The first repair is to invalidate the nodes that violate the feedback edge rule: at most *n-1* pipelines are

chosen for a feedback edge with *n* delay. POG nodes 'CE' and 'CDE' are invalidated, X marked in Figure 5(b) since it cuts an feedback edge C-B with one delay. As feedback edge E-B has two delays, at most one pipeline may exist across it. So, POG nodes 'E' and 'DE' cannot be chosen at the same time. Then we randomly select the victim pipeline that will be invalidated: 'E' is invalidated in this example. Finally, three nodes 'DE', 'BCE', and 'BCDE' remain as valid.

Pipeline cuts that cross each other should not be chosen together: POG nodes 'BCE' and 'DE' should not be selected together in the example of Figure 5. Any two POG nodes do not cross each other if one is an ancestor node of the other, since the child POG node contains all sub-tasks that are associated with the ancestor POG node. So, to be a valid pipeline solution, all chosen POG node should be in ancestor-children relationship. That is, all chosen POG node should be in a path from the terminal POG node to the root node. So, the second repair is to select a path from the terminal POG node to the root node that has the most number of valid pipelines. The other POG nodes which are not in the chosen path are invalidated. ('BCE' in the example) POG nodes 'DE' and 'BCDE' are finally chosen for a valid pipeline solution as depicted in Figure 5 (c).

C. Scheduling

Given a valid solution for mapping and pipelining, we evaluate it by a fixed-priority list scheduling. Note that sub-tasks mapped to separate pipeline stages are scheduled independently as there is no data dependency in-between.

Fig 5(d) shows an evaluation result, where the task graph is divided into 3 pipeline stages for the solution of Fig 5(c). The highlighted nodes form an iteration cycle: During each iteration period, data parallel node A of n-th iteration is parallelized on two processors. And, nodes B and C of (n-1)-th iteration are mapped to uP0 and nodes D and E of (n-2)-th iteration are mapped to uP1.

D. ILP Formulation

To obtain an optimal solution, we extended the ILP solution presented in [12] to take temporal parallelism into account, where the ILP solution considered task and data parallelisms only. In addition to the ILP formulations made there, we formulate the valid pipeline conditions based on the POG. We explain the additional formulas only. For detailed discussion of the ILP formulation, refer to [12].

We define a new variable, $pipe_{ij}$, that indicates the number of pipeline buffers inserted on an edge, e_{ij}, between sub-task i and j. In equation (1), the number of pipeline inserted on each edge (*pipe*) is calculated as the sum of the selected POG nodes where *pog* variable becomes 1 when the POG node is selected. *ff(pog)* is a set of the feed-forward edges included in the cut associated with the POG node. Integer variables *s*, *t*, and *comm* in equation (2) denote start time, execution time, and communication overhead respectively. By equation (2), the data dependency is nullified if at least one pipeline is applied to an edge. The pipeline buffer cost is added to the cost computation in the ILP formulation.

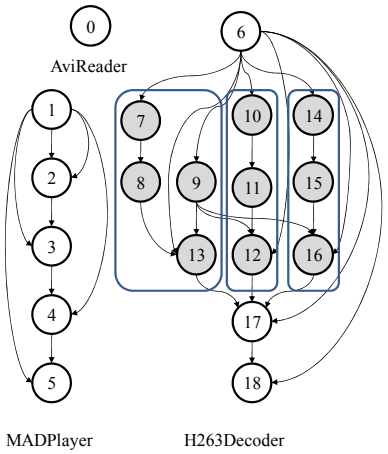
$$\forall e_{ij}, \quad pipe_{ij} = \sum_{k \text{ s.t. } e_{ij} \in ff(pog_k)} pog_k \tag{1}$$

$$\forall e_{ij}, \quad s_i + t_i + comm_{ij} \le s_j + \infty \times pipe_{ij} \tag{2}$$

V. EXPERIMENTS

Experiments with real-life examples proved the effectiveness and feasibility of the proposed technique. All experiments were done on Dual Xeon 3.4GHz machine with 4GB main memory and CPLEX 11.0 was used as the ILP solver. Group size and population size of QEA, *n* and *p* in Figure 2, were adjusted from 50 to 300 according to the problem size, while all other parameters were set to the same as [13]. IPC overhead was modeled as linearly proportional to data size.

The first experiment examines the effectiveness of three parallelisms. A DivX player application consists of 3 tasks: AviReader, MADPlayer, and H263Decoder. The task graphs have 19 sub-tasks as shown in Figure 6. Initially, the original multi-rate task graphs are mapped to 6 ARM926ej-s processors considering task parallelism only. While the original graph does not have any data parallelizable node, there is a set of nodes that can be clustered together to make it as a data parallelizable super node as shown in shaded nodes surrounded by rounded boxes in Figure 6. Then, we consider both task and data parallelism. Lastly, we include temporal parallelism in the mapping/schedule decision.



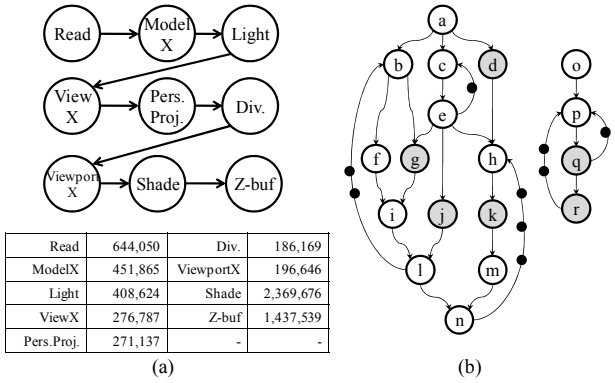
	Exec. time
0	18718
1	139934
2	42368
3	69315
4	226855
5	390186
6	405536
7	2409*99
8	1026*99
9	135*99
10	2409*99
11	1026*99
12	3745*99
13	3745*99
14	2409*396
15	1026*396
16	3745*396
17	237206
18	1076687

Figure 6. Task graphs of DivX player example.

TABLE 1 Throughput maximizing mapping/scheduling result with DivX player example varying the degree of parallelism.

	PL0	PL1	QEA Initiation Interval	QEA Elapsed time	ILP Initiation Interval	ILP Elapsed time
Task	6	0	5,286,894	1.92 s	5,286,894	8.26 s
Task	3	3	2,941,848	2.01 s	2,941,848	9.36 s
+Data	1	5	2,582,808	2.50 s	2,582,808	16.02 s
All	1	5	1,168,099	9.39 s	*1,167,833	*10800 s

The experimental results are summarized in TABLE 1. The row named 'Task' in TABLE 1 shows the result when only task parallelism is considered. 'Task+Data' is the case when data parallelism is additionally considered, while all three parallelisms are considered in 'All'. We divided processors into 2 pools and assumed that data parallelism is only allowed in 'PL1'. As the fourth row shows, we achieve better throughput by allowing more degree of data parallelism even using the same number of processors. As shown in the last row ('All'), significant throughput gain can be obtained by exploiting all three parallelisms. The proposed QEA approach shows optimal or near-optimal result with orders of magnitude speed gain compared to the ILP solution. Note that the * marked result in the last row is not optimal but the best result after 3 hours of computation. Even though it is not optimal, it is close to optimal as it shows 7.8% gap compared to the cut-off value 1,076,687 in the ILP solver.



Read	644,050	Div.	186,169
ModelX	451,865	ViewportX	196,646
Light	408,624	Shade	2,369,676
ViewX	276,787	Z-buf	1,437,539
Pers.Proj.	271,137	-	-

(a) (b)

Figure 7. (a) 3-D rendering task graph and (b) synthetic task graphs.

The second experiment is on 3-D graphic rendering task graph that is shown with performance table in Fig 7(a). It uses Gouraud shading and Z-buffer algorithm. The task graph renders 100 polygons per iteration executing 9 sub-tasks sequentially and all edges deliver 20,800 bytes per execution. All sub-tasks except 'Read' have data parallelism: Each polygon can be rendered independently with others. As all sub-tasks in the task graph are sequential, task parallelism cannot be exploited in this example. The target architecture used in the fourth row (PL0:1, PL1:5) in TABLE 1 is also used here and the objective is set to minimize the cost of pipeline buffer. The proposed technique finds the solution within 5 seconds that inserts a pipeline buffer into edge ModelX-Light and exploits data parallelism in following 7 sub-tasks, which is verified as optimal by ILP solution.

The last experiment is done on two synthetic task graphs which have 18 sub-task nodes. The target architecture has a pool with 6 homogeneous processors and allows data parallel execution. Execution times of all sub-tasks are assumed to be 100 and shaded sub-tasks are data parallel ones. All edges are assumed to deliver 16 bytes token. Without retiming for multiple delayed feedback edges, the best initiation interval (II) is 400 due to the critical path length (b-f-i-l). On the other hand the proposed technique considering retiming for multi-delayed feedback edges, achieves 308 as II, which is very close to the optimal solution, 300, obtained by ILP. The proposed technique deduces the solution in 49.31 seconds on average, while it takes 19551 seconds by ILP.

VI. CONCLUSION

In this paper, we proposed a multi-task mapping/ scheduling heuristic based on the QEA technique considering data and temporal parallelisms as well as task parallelism for MPSoC. In contrast to the previous researches, multi-rate task graphs with cycles are allowed as input task graphs. The target architecture is a heterogeneous multiprocessor system that consists of multiple pools of homogeneous processors.

By exploiting three parallelisms altogether, we could get significantly better result with the same number of processors. The QEA based technique takes significantly less time with negligible penalty on the solution quality than an ILP technique. The proposed approach is scalable and extensible to more complex architectures and/or design constraints. We implemented the proposed technique in a system-level design environment that generates the software for a target MPSoC after the mapping/scheduling decision.

REFERENCES

[1] D. Pham et al., "The design and implementation of a first-generation CELL processor-a multi-core SoC," In Proc. of ICICDT, pp. 49-52, May, 2005.

[2] K. Asnovic et al., "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 18, Dec. 2006.

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

[4] E. S. H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," IEEE Transactions on Parallel and Distributed Systems, vol. 5, no.2, pp.113-120, Feb. 1994.

[5] Robert P. Dick and Niraj K. Jha, "MOGAC: A Multiobjective Genetic Algorithm for the Co-Synthesis of Hardware-Software Embedded Systems," In Proceedings of ICCAD, p. 522, 1997.

[6] P. Arato, S. Juhasz, Z.A. Mann, A. Orban, and D. Papp, "Hardware-software partitioning in embedded system design," IEEE International Symposium on Itelligent Signal Processing, pp. 197-202, Sep. 2003.

[7] P. D. Hoang and J. M. Rabaey, "Scheduling of DSP Programs Onto Multiprocessors for Maximum Throughput," IEEE Trans. On Signal Processing, pp. 2225-2235, June 1993.

[8] S. Bakshi and D. D. Gajski, "Partitioning and pipelining for performance-contrained hardware/software systems," IEEE Transactions on VLSI Systems, vol. 7, no. 4, pp. 419-432, Dec. 1999.

[9] K. S. Chatha and R. Vemuri, "hardware-software partitioning and pipelined scheduling of transformative applications," IEEE Trans. on VLSI, 10(30), 2002.

[10] L. Benini, D. Bertozzi, A. Guerri, and M. Milano, "Allocation and scheduling for mpsocs via decomposition and no-good generation," in Proc. of International Joint Conferences on Artificial Intelligence, 2005.

[11] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures," in Proc. of CASES, 2006.

[12] H. Yang and S. Ha, "ILP Based Data Parallel Multi-task Mapping/Scheduling Technique for MPSoC," in Proc. of ISOCC, pp. 134-137. Nov.2008.

[13] K-H. Han and J-H. Kim, "Quantum-inspired Evolutionary Algorithm for a Class of Combinatorial Optimization," IEEE Trans. on Evolutionary Computation, IEEE Press, Vol. 6, No. 6, pp. 580-593, Dec. 2002.

[14] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," IEEE Trasnsactions on Computers, vol. C-36, no. 1, pp. 24-35, Jan, 1987.