IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 16, 2019, accepted April 8, 2019, date of publication April 16, 2019, date of current version May 6, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2911653

Fine-Grained Communication-Aware Task Scheduling Approach for Acyclic and Cyclic Applications on MPSoCs

KAI HUANG[1], XIAOWEN JIANG[1], HAITIAN JIANG[1], XIAOMENG ZHANG[1], MIN YU[1], RONGJIE YAN[2], AND XIAOLANG YAN[1]

[1]Institute of VLSI Design, Zhejiang University, Hangzhou 310027, China
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China

Corresponding author: Xiaowen Jiang (xiaowen_jiang@zju.edu.cn)

This work was supported in part by the National Key R&D Program of China under Grant 2018YFB0904900 and Grant 2018YFB0904902.

ABSTRACT Fine-grained task models can exploit parallelism to achieve high performance for multiprocessor system-on-chip (MPSoC). However, fine-grained models face the issues of high-communication overhead and difficult scheduling decisions, and the two challenges are inter-dependent. To address the issues, this paper gives a full analysis of the fine-grained communication optimization technique and communication pipeline, from both time and topology perspectives, and proposes a static fine-grained communication-aware task scheduling (FCATS) approach, which integrates scheduling with communication pipeline for acyclic and cyclic applications based on the fine-grained Simulink model. The approach contains search-based scheduling with high-quality solutions utilizing genetic algorithm-integer linear programming (GA-ILP) and hybrid GA-heuristic scheduling with short solving time to meet different demands for users. The experimental results with both synthetic and real-life benchmarks on the 4/8/16-CPU platform demonstrate the efficiency of the approach on performance improvements compared to previous works.

INDEX TERMS Communication pipeline, genetic algorithm, integer linear programming, MPSoC, task scheduling.

I. INTRODUCTION

The increasing complexity of Multiprocessor System-on-Chip (MPSoC) drives the needs for system software development. To exploit the computation capability of MPSoC, fine-grained task models like Intel's TBB [1], Cilk++ [2], Fine-grain MPI(FG-MPI) [3] and Simulink [4] have been proposed to expose the computation parallelism, which provides more chances for system performance optimization, including easier load balancing, greater potential for overlapping communication and computation, and improved platform-independence [5].

However, fine-grained models face great challenges when exploring the parallelism. First, as the granularity of tasks becomes finer, the amount of inter-task communication becomes larger. The increasing number of processors in an MPSoC provides more opportunities for exposing the communications, which requires the consideration of inter-processor communication optimizations. Next, the decreasing of the granularity of tasks increases the number of tasks and the complexity of task inter-dependencies. Thus, how to allocate and determine the execution sequence of tasks of various applications on MPSoC, i.e. schedule tasks, has become a critical issue for performance improvements and delicate scheduling approaches are necessary for fine-grained models. Moreover, the above two challenges are inter-dependent: an efficient scheduling approach should take the communication optimizations into account to achieve better performance while applying communication optimizations may affect the scheduling sequence, which calls for the consideration of both factors.

To address the communication challenge, two representative optimization techniques, communication pipeline and message aggregation techniques, have been proposed in [6]. Communication pipeline [6] can hide the receiving overhead by overlapping communication with the subsequent

The associate editor coordinating the review of this manuscript and approving it for publication was Juan Touriño.

computation. Message aggregation [6], [7] combines the messages with the same sources and destinations to reduce the communication startup time. Nevertheless, the discussions of communication pipeline in the previous work [6] are not sufficient. From the view of time, the description about communication pipeline lacks the consideration of the increasingly common situation that communication time is larger than its related computation time and the cases where using communication pipeline cannot save time. From the view of topology, the previous work [6] introduces the method to employ communication pipeline on both acyclic and cyclic applications by preprocessing tasks. However, it is unclear which tasks and how many cycles of each task to be preprocessed, which requires theoretical guidance for users. For the scheduling challenge, existing scheduling approaches on fine-grained models [1]–[3], [5] mainly focus on runtime implementation, but design-time (i.e. static) approaches are also important. They have the advantages of efficiently utilizing the overall system information for large chances of optimizations and low runtime overhead. Therefore, the static approaches can be used as the initial scheduling for the dynamic scheduling, or integrated into multithreaded code generators to prepare for the high-quality performance at run time. Although not designed for fine-grained models, existing static multiprocessor task scheduling approaches can be utilized for the models. From the application level, most of the scheduling approaches try to reduce communication through allocating heavily communicated tasks on same processors [8], [9] or duplicating key tasks to overlap communication and computation [10], [11], but neither of the dynamic or static scheduling approaches considers the application of the fine-grained communication optimization techniques for MPSoC, which requires fine-grained communication-aware task scheduling approaches.

This paper aims at addressing these challenges by proposing a fine-grained communication-aware task scheduling approach (FCATS) for MPSoC performance improvements. In this work, scheduling refers to determining the scheduling sequence of each task on each processor and we assume the process of profiling and mapping tasks on processors has been done before. There have been plenty of literature of task profiling [12], [13] and task mapping [14] and can be exploited to obtain the input for the proposed scheduling approach. The approach works on the fine-grained Simulink model where communications are modeled as sending tasks and receiving tasks with each task including communication startup and transfer operations, and thus it schedules computation tasks, sending tasks, and receiving tasks together to efficiently utilize processors, as well as exploits communication pipeline to reduce communication transfer overhead. To better use communication pipeline, we give a thoroughly analysis of it from both time and topology perspectives based on [6]. After that, we introduce FCATS, which integrates scheduling with communication pipeline. FCATS further contains a search-based fine-grained communication-aware task scheduling (S-FCATS) for high quality and a heuristic-based fine-grained communication-aware task scheduling (H-FCATS) for short running time. S-FCATS utilizes a Genetic Algorithm-Integer Linear Programming (GA-ILP) to obtain scheduling results and communication pipeline allocation. Since ILP-based methods consume exponential time and memory when problem instances get larger especially for the scheduling problem, H-FCATS is further developed utilizing a GA-heuristic algorithm to obtain scheduling results with low time consumption. Moreover, the proposed scheduling approach has been integrated into the LESCEA (Light and Efficient Compiler for Embedded Application) multithreaded code generation flow [15] and experiments on both synthetic task graphs and real-life applications have demonstrated its efficiency.

The main contributions of this paper are listed as follows.

- We give a full analysis of the fine-grained communication optimization technique, communication pipeline. We consider all-around application scenarios from both time and topology perspectives, and provide a theoretic guidance for users.
- We propose a static task scheduling approach FCATS for pre-mapped fine-grained tasks. The approach integrates scheduling with communication pipeline to improve system performance, and can provide scheduling results with better performance and less communication overhead. Moreover, FCATS has two distinct advantages which provides a practical guidance for users with different demands.
  - It contains S-FCATS for high solution quality and H-FCATS for low running time.
  - It can handle acyclic and cyclic graphs, which is reflected in both S-FCATS and H-FCATS.

The rest of the paper is organized as follows. Section II gives some related work. Section III presents the modeling background. Section IV analyzes communication pipeline and gives some suggestions for later users. Section V introduces the proposed scheduling approach. Section VI discusses the experiments and results. Section VII concludes the whole paper and gives some future work.

II. RELATED WORK

To exploit the parallelism of MPSoCs, an application can be decomposed into a number of fine-grained tasks (i.e. threads). To manage the fine-grained tasks, runtime libraries including Intel TBB [1], Cilk++ [2], and Fine-grain MPI(FG-MPI) [3] are developed for parallel programming, but they cannot model the system functionality. The Simulink model [15] provides an easy and efficient way to model the functionality of the whole MPSoC. It can model both the hardware architecture and software applications from the complicated MPSoC system to simple functions in the application.

As fine-grained models incur large amount of communication overhead, fine-grained communication optimization techniques have been developed. For Network-on-Chip (NoC) based MPSoC, the techniques [16], [17] consider

the optimizations of data packet transmission and routing mechanisms. For bus-based MPSoC, message aggregation and communication pipeline have been proposed to reduce inter-thread communication overhead [6]. Between them, communication pipeline utilizes Distributed Memory Server (DMS) or Direct Memory Access (DMA) and preprocesses receiving tasks to make the execution of computation and communication overlap to hide the communication transfer time. However, the communication pipeline is not fully analyzed in both time and topology perspectives in their works. Moreover, [6] only introduces communication pipeline and lacks its usage during system software design, which will be covered in this work.

Task scheduling should also be a focus for fine-grained models. From the view of scheduling timing, task scheduling can be divided into dynamic scheduling or static scheduling. Dynamic scheduling can adjust the scheduling results according to the dynamic environments, but has high runtime overhead. Many of the fine-grained scheduling [3], [5], [18] are designed for multithread packages which need the ability to handle dynamics because of their dynamic application scenarios and unawareness of the system information. On the contrary, static scheduling can utilize all the system information to determine the scheduling results at design time with low runtime overhead and high optimization chances, but cannot deal with dynamics.

From the view of scheduling policies, static scheduling can be classified as search-based scheduling and heuristic-based scheduling. Search-based scheduling includes mathematical methods like ILP [19]–[23] or meta-heuristic algorithms such as Genetic Algorithm(GA) [24]–[26], Ant Colony Optimization(ACO) [27] and Particle Swarm Optimization(PSO) [28]. Among these algorithms, ILP-based methods have the advantages of reachable optimality, easy modeling, and easy access to various solving tools. However, its time consumption grows exponentially as the input large scale gets larger. GA-based methods are widely used and have been shown to outperform several algorithms in the task scheduling problem [26]. Although its time complexity is theoretical exponential, much optimizations [25], [26] can be conducted to accelerate its computation speed. On the contrary, heuristic-based scheduling policies, including list-based scheduling [29]–[31], clustering-based scheduling [9], [32], [33], and duplication-based scheduling [10], have lower time complexity but the quality of solutions cannot be guaranteed. As the most widely used scheduling policy, list-based scheduling first determines the priority of each task and assigns tasks according to descendant priorities to their appropriate processors. Well-known algorithms include HEFT [29], CPOP [29], PEFT [30], LDCP [31] and etc. Besides, there are works [24], [34] utilizing two or more above algorithms, which can exploit the advantages of several algorithms to achieve high-quality scheduling and optimize several performance metrics as well.

As scheduling and communication optimizations have inter-dependency, almost all of the recent works consider communications during scheduling. Most of the works consider communications from the application level. Search-based scheduling [20], [25], [28] and list-based scheduling [29], [30] take communication overhead into account in their model but have not further explored the performance benefit of communication optimization. Clustering-based scheduling [8], [9] mostly reduced communication overhead by allocating heavily-communicated tasks to the same cluster, and duplication-based scheduling [10], [11] duplicates key tasks to eliminate the inter-processor communication. However, as these approaches are not specifically designed for fine-grained models, none of the works exploit fine-grained techniques to reduce communication overhead.

Wang et al. [35] proposed an ILP-based task scheduling approach to totally remove communication overhead by employing the retiming technique. The technique can preprocess tasks and transform inter-processor communications of the same iteration into different iterations, thus computation and communication can be overlapped for streaming applications on MPSoC. However, their work requires an initial schedule as a basis, which greatly affects the final performance. Our approach in this work can give the scheduling results based on the input application and the hardware without an initial schedule. Moreover, our approach exploits a more realistic model including the separation of sending and receiving tasks during one communication transfer and using DMS. Not only computation and communication can be overlapped but communications can also be overlapped. While the approach in [35] takes one communication transfer as a whole and mapped on the bus thus communications must be serialized. Other scheduling works [36], [37] consider communication overhead from the architecture level such as bus contention, memory access contention, and etc., which is not the optimization level of this paper.

In this paper, we first analyze communication pipeline thoroughly and then apply communication pipeline with the task scheduling process. As it is impossible for NP-hard problem to obtain optimality and fast solving time simultaneously, the proposed scheduling approach FCATS consists of S-FCATS using a GA-ILP scheduling to obtain optimal or near-optimal results and H-FCATS using a GA-heuristic scheduling to achieve short solving time.

III. BACKGROUD

A. SYSTEM MODEL

This work exploits the fine-grained Simulink model as shown in Figure 1. A Simulink model represents the functionality of a target system, including software threads and hardware architecture. The functional modeling of an application is based on an Abstract Clock Synchronous Model (ACSM [38]), which can easily express parallelism and pipeline by partially ordered intra- and inter-dependencies. Details about Simulink models can be found in [15], [38].

A Simulink model is made up of the following three basic components.

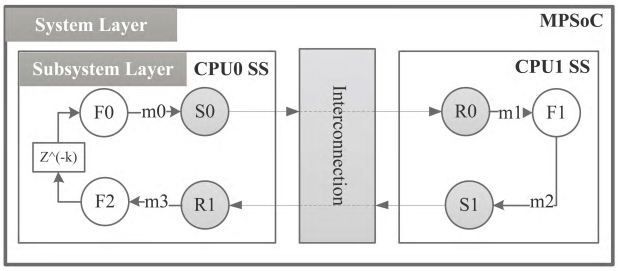


FIGURE 1. An example of the Simulink model.

- Simulink Block represents a function that takes n inputs and produces certain outputs. User-defined functions (S-function), discrete delays, and predefined blocks such as mathematical operations can be classified as Simulink blocks. Basic Simulink blocks contain functional blocks (white circles in Figure 1) used to compute data and communication (sending and receiving) blocks (grey circles in Figure 1) used to explicitly model communications and allow optimizations. Besides, discrete delay blocks (white square in Figure 1) are inserted to avoid deadlocks when building the model.
- Simulink Link is a one-to-many link, which connects one output port of a block to one or more input ports from other blocks, and represents a dependency relation between different blocks. If there is a link from $B0$ block to $B1$ block, we say that $B1$ depends on $B0$, denoted by $B0 \rightarrow B1$. A Simulink link starting from a sending block S and ending with a receiving block R from different processors is referenced as a communication vector, denoted by $S \rightarrow R$.
- Simulink Subsystem can contain blocks, links, and other subsystems to represent hierarchical composition.

In this paper, we consider a typical MPSoC architecture which consists of multiple processors. Each processor has its own local memory, and all processors perform inter-processor communication by a high bandwidth shared bus to access the main memory. The bus controller implements a given bus protocol and assigns bus access rights to individual processors. The communications are supposed to perform at the same speed without contentions and each processor has independent I/O unit that allows for communication and computation to be performed simultaneously. Note that the real communication cost occurs only in inter-processor communications where dependent tasks mapped on different processors.

Having combined mapping and hardware information in the system model, an MPSoC Simulink model can be represented by a two-layered hierarchical structure. The system layer describes a system architecture that is made up of CPU subsystems and inter-subsystem communication channels between CPUs. The subsystem layer describes a CPU subsystem architecture that includes a set of Simulink blocks and links between blocks. In this model, an application is executed for many cycles, and a cycle means from some point all blocks have been executed once. Moreover, both unrelated communications, and communications and unrelated computations can be executed in parallel in this work.

B. APPLICATION MODEL

A software application running on the MPSoC can be represented by a directed graph, denoted by $G(T, E)$. Each node represents one task of the application ($t \in T$) and each edge represents one precedent dependency between a pair of tasks ($e \in E$). Since this work is based on the assumption that task mapping has been determined, inter-processor communications have been modeled by communication tasks including sending and receiving tasks where communication time translates to task execution time and the communication relation translates to task dependency. Therefore, the application graph has weighted nodes and non-weighted edges. Mapping the graph/application on the Simulink model, nodes/tasks equal to Simulink blocks, and edges equal to Simulink links. In the whole paper, equal representations can be interchanged.

A directed graph may be acyclic or cyclic, determined by if there is a path starting from some node and ending with the same node. After mapping application graphs onto Simulink models, we can notice two kinds of cyclic dependency. (To distinguish the ''cycle'' of the abstract clock from the ''cycle'' of the topological dependency, we keep the name ''cyclic graph/application/topology'' but call the ''cycle'' in the graph as ''loop''.)

- Block loop. If there is a path starting from a block and ending with the same block in the application, then the blocks and the links along the path constitute a block loop, as $F0 \rightarrow S0 \rightarrow R0 \rightarrow F1 \rightarrow S1 \rightarrow R1 \rightarrow F2 \rightarrow F0$ shows in Figure 1.
- Inter-processor loop. If there are communication vectors with opposite directions between two different processors, then the blocks and the links related to the communication vectors constitute an inter-processor loop. As shown in Figure 1, the block loop is also an inter-processor loop.

Furthermore, we extend the definitions of ''entry task'' and ''exit task'' from acyclic graphs to cyclic graphs. For cyclic graphs, if a task has no predecessors or its only predecessor is the delay block, then it is an entry task; if a task has no successors or its only successor is the delay block, then it is an exit task.

IV. ANALYSIS OF COMMUNICATION PIPELINE

This section gives a quantitative analysis of communication pipeline and provides guidance for its usage. In Section IV-A, we first introduce the concept of communication pipeline proposed in [6]. To have a deeper understanding of communication pipeline and provide a more general view of how to apply communication pipeline, we give analyses in Section IV-B and Section IV-C on communication pipeline from time and topology perspectives respectively.

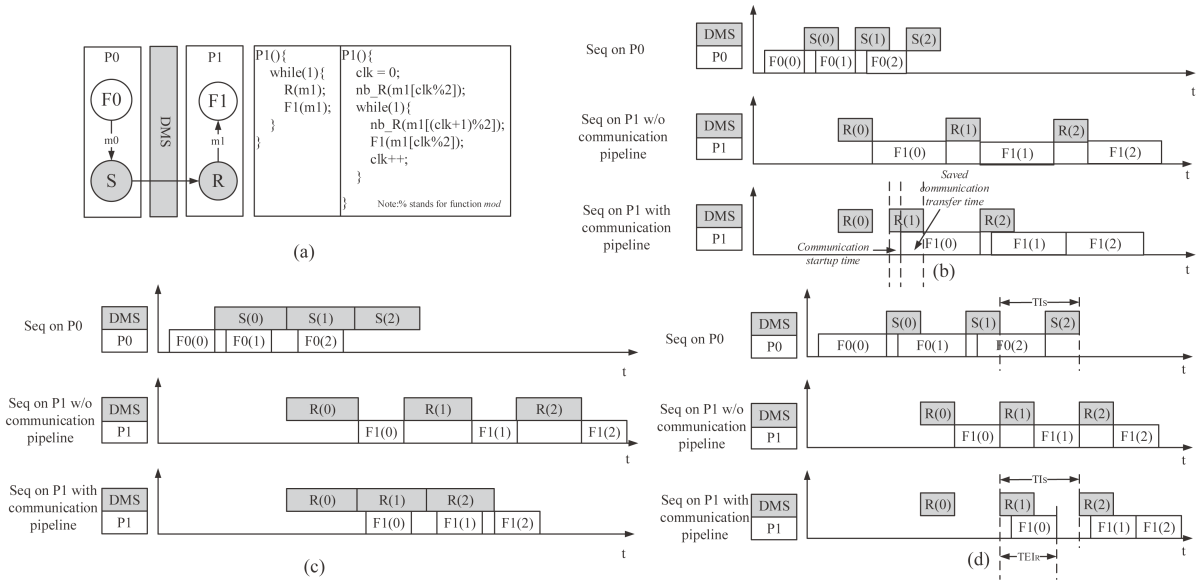


FIGURE 2. An example of communication pipeline.

A. CONCEPT OF COMMUNICATION PIPELINE

Communication pipeline was proposed to overlap communication with its subsequent computation in order to hide the communication time [6] utilizing the hardware feature of DMS or DMA that after initiated by CPU, the hardware can complete the communication transfer without the intervention of CPU. A typical example is shown in Figure 2, where the system has a sending task, a receiving task and two computation tasks on two processors, and the number in parentheses after the task type name denotes the cycle of data which is currently processed. In Figure 2(a), the sending task S sends data of the computation task $F0$ to the receiving task R for its later usage by $F1$. Figure 2(b) demonstrates an example of communication pipeline described in [6], where each task is executed for 3 cycles. Without communication pipeline, $F1(i)(i = 0, 1, 2)$ can only start after R finishes and the waiting time becomes the communication overhead. With communication pipeline where R is preprocessed by one cycle, $R(i + 1)$ and $F1(i)(i = 0, 1)$ can execute concurrently to hide the communication overhead. Communication overhead can be divided into communication startup time and communication transfer time, and the former is constant for each communication and the latter is proportional to the communication transfer size. From this example, we can see that applying communication pipeline on R can hide all of the communication transfer time of R of all cycles except for the first cycle, and the total performance is optimized. In addition, although forming communication pipeline requires preprocessed tasks out of pipelined iterations which may also affect the system performance, as the application is executed for many cycles, the improvements largely outperform the side effects and we do not discuss this factor.

B. ANALYSIS FROM TIME PERSPECTIVE

Figure 2(b) discussed in [6] is a typical example of communication pipeline, but there are other cases related to the time characteristics of task graphs that [6] has not mentioned. By further investigating the communication pipeline, we find its optimization efficiency is affected by two important factors.

1) The relative amount of the receiving task R time T_R and its successor computation task F time T_F with F dependent on R.
   - When $T_R \leq T_F$, the communication transfer overhead in pipelined stages can be totally reduced, and it has been discussed in our previous work [6], as shown in Figure 2(b).
   - When $T_R > T_F$, the communication transfer overhead in pipelined R and F can only be partly reduced, and the reduction amount is T_F, as shown in Figure 2(c). As the amount and complexity of communications are increasing in current MPSoCs, this situation is becoming common.

   This factor indicates that communication pipeline is sensitive to the computation and communication features of the applications. Though, under both cases, communication pipeline can hide communication overhead.
2) The relative amount of the finishing time interval TI_S between successive cycles for the sending task S and the effective pipelined iteration time TEI_R containing the corresponding receiving task if using communication pipeline.
   When $TI_S < TEI_R$, using communication pipeline can improve the performance. Otherwise, using

communication pipeline cannot improve the performance. Taking Figure 2(d) as an example, TI_S is the finishing time interval between $S(1)$ and $S(2)$, and TEI_R is the time when R and $F1$ execute in one pipelined iteration. A pipelined iteration is when the code in *while*(1) executes once if applying communication pipeline as shown in the right most square of Figure 2(a) and tasks in one pipelined iteration may execute in different cycles. As there may be idle time that tasks do not execute in one pipelined iteration, we name TEI_R as the effective pipelined iteration time. In this example, TI_S is larger than TEI_R and there is idle time in the pipelined iteration if communication pipeline is applied. Therefore, although using communication pipeline can overlap communication and computation, the idle time still brings overhead. For Figure 2(b), TI_S is smaller than TEI_R and there is no idle time in each pipelined iteration. Therefore, the overlap efficiently reduces the communication overhead. This factor indicates that communication pipeline should be applied appropriately instead of as much as possible to achieve most performance improvements.

C. ANALYSIS FROM TOPOLOGY PERSPECTIVE

In our previous work [6], communication pipeline can be applied on acyclic graphs without any constraints and on cyclic graphs under the constraint of inter-processor loops. For cyclic graphs, to create chances for overlap between communication and computation tasks, we preprocess not only the receiving tasks but also some related computation tasks and sending tasks as long as there are enough delays in the loop. However, there is no quantitative analysis on the preprocessing and on how to exploit communication pipeline on each task to reach this target in [6].

To analyze communication pipeline precisely, we give definitions for different topologies. *Communication pipeline-aware acyclic topology (CPAT):* If there are no inter-processor loops after the application is mapped on the Simulink model, or there are inter-processor loops but none of them are part of block loops, we call the application as CPAT.

Communication pipeline-aware cyclic topology (CPCT): If there are inter-processor loops after the application is mapped on the Simulink model, and part or all of the inter-processor loops are part of block loops, we call the application as CPCT.

Next, we use the retiming theory to explain and quantify the application of communication pipeline for both CPAT and CPCT. The retiming technique [39] was first proposed to minimize the cycle period of a synchronous circuit by redistributing registers. We extend it and define it as follows: *Retiming:* Given a directed graph $G = (V, E, RT)$, retiming RT of G is a function that maps each task t_i ($t_i \in V$) to an nonnegative integer rt_i. rt_i is the retiming value of t_i. rt_i is initially 0 and if t_i is preprocessed N cycles, $rt(t_i) = N$.

Algorithm 1 Function *calc_rt*().

Input: Task t_i for retiming calculation.
Output: Retiming value of t_i.

```
1: if t_i is the entry task then
2:     return 1;
3: end if
4: while pred_i ∈ t_i'spredecessorsPRED do
5:     rt_pred_i=max (rt_pred_i, rt_t_i + cp_alloc);
6:     calc_rt(pred_i);
7: end while
8: return 0;
```

Using retiming, we can calculate how many cycles of each task can be preprocessed based on the idea that communication pipeline preprocesses the receiving task by one cycle. Therefore, once communication pipeline is applied on one R with its retiming value rt_R, its predecessors' retiming values are all rt_R, and the corresponding F and F's successors all have retiming value $rt_F = rt_R - 1$, until other communication pipelines are encountered. The rationale for the calculation is that when communication pipeline is applied to a receiving task, it is preprocessed one cycle beforehand, and to make this happen, its predecessors should all be preprocessed one cycle beforehand, which translates to the corresponding retiming values.

The idea can be implemented recursively. First, the retiming values of all tasks are initialized as 0. Then, the exit task is found and fed into the recursive function *calc_rt*() as shown in Algorithm 1. In *calc_rt*(), if the input task is the entry task, then the recursion ends which means the calculation has been performed on the whole graph; if not, the retiming values of the predecessors of the input task are determined based on the input task's retiming value, the communication pipeline allocation condition, and the predecessor's own retiming value. This ensures that if the predecessor has multiple successors, its retiming value is calculated based on the successor with the largest retiming value.

For CPAT, as communication pipeline can be applied to any receiving tasks in theory, retiming values can be calculated as mentioned above from exit tasks to entry tasks. Figure 3(a) and Figure 3(b) show the retiming value calculation example of CPAT. For Figure 3(a) with no inter-processor loops, the retiming value calculation begins with exit tasks $F0$, $F3$, and $F4$ and their retiming values are 0. Going upward, we assume $R0$ and $R1$ are applied by communication pipeline, and thus their retiming values are 1. Going upward on, $S0$ has the same retiming value as $R0$ and $F1$ has the same retiming value as $S0$, and the retiming values of $S1$ and $F2$ are calculated in the same way. For Figure 3(b) with inter-processor loops but no block loops, the exit task is $F0$ and its retiming value is 0. We assume $R0$ and $R1$ are applied by communication pipeline. Thus, going upward, $R1$, $S1$, $F3$, and $F2$ have retiming value 1, and $R0$ and $S0$ have retiming value 2. Since $F1$ has two successors $F0$ and $S0$, and $S0$

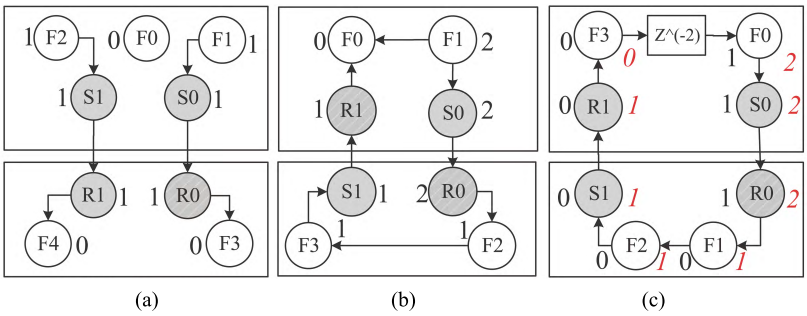


FIGURE 3. An example of calculating the retiming value of typical CPAT and CPCT. (a) CPAT with no inter-processor loop. (b) CPAT with inter-processor loop. (c) CPCT.

has larger retiming value than $F0$, $F1$ has the same value as $S0$ as 2. After calculating retiming values, we find that $F0$ is preprocessed by two cycles earlier than $F1$. However, this preprocessing as well as the task dependency can be implemented by data buffers and poses no limitations on communication pipeline.

For CPCT, however, the number of communication pipelines that can be applied in each loop is limited by the number of delay cycles and receiving tasks. An example in Figure 3(c) elaborates this issue. Assuming that there are two receiving tasks $R0$ and $R1$ in the loop, and the number of delay cycles in this loop is 2, i.e. $F0$ of cycle(i) processes data from $F3$ of cycle($i-2$). We consider the following two cases.

1) Communication pipeline is only used to one of the receiving tasks ($R0$ in the example). The retiming values calculated by the above method are denoted by black numbers. As $F0$ is preprocessed by one cycle, $F0$ of cycle($i-1$) processes data from $F3$ of cycle($i-2$), which ensures no deadlock occurs in the loop.
2) Communication pipeline is used to both receiving tasks ($R0$ and $R1$ in the example). The retiming values calculated by the above method are denoted by red italic numbers. As $F0$ is preprocessed by two cycles, $F0$ of cycle($i-2$) processes data from $F3$ of cycle($i-2$), and meanwhile $F0$ of cycle($i-2$) sends data to $F3$ of cycle($i-2$) through the loop, which forms a deadlock in the loop.

Therefore, the communication pipeline is successfully applied in case 1). In face, it can be used to at most $min(N_D-1, N_R)$ times in each loop, where N_D denotes the number of delay cycles and N_R denotes the number of receiving tasks. This is because the application of communication pipeline is to assign the delay cycles to the receiving task, and require at least one delay cycle to break the loop to avoid deadlock. From the above analysis, once the receiving tasks to be communication pipelined are determined, the retiming values of all tasks can be obtained, i.e. the number of preprocessed cycles for each task can be obtained, which can help determine the scheduling sequence. Moreover, using retiming to explain communication pipeline unifies its application on CPAT and CPCT with the same calculation under different constraints.

V. FCATS: THE PROPOSED SCHEDULING APPROACH

FCATS provides scheduling results for pre-mapped tasks on MPSoC, considering the application of communication pipeline. The approach contains S-FCATS and H-FCATS for different requirements of solution quality and time, which will be described in Section V-A and Section V-B respectively.

A. S-FCATS

As ILP-based methods can achieve optimal performance and have been used in many previous scheduling approaches, we exploit ILP to obtain scheduling results. Communication pipeline allocation is manipulated on one task of different cycles while the scheduling problem is conducted on all tasks in one cycle. Therefore, it is difficult to build them into one ILP model since they are from two dimensions. Even though the ILP model can be built by expanding all tasks from all cycles into a large set of tasks, the problem scale is too large for ILP even for small-scale graphs. GA is a widely used meta-heuristic and it is suitable for the communication pipeline allocation problem as the binary nature of the solution component. As shown in Figure 4, we use GA to find the allocation results and integrate ILP into the GA process for its fitness evaluation. In this way, ILP deals with the large scheduling solution space while GA deals with the small communication pipeline allocation solution space. As scheduling has larger impact on performance than communication pipeline allocation, S-FCATS focuses on the optimality of scheduling with ILP and try to find optimal or near-optimal communication pipeline allocation. With S-FCATS, we can obtain optimal or near-optimal solutions with high quality at a cost of solving time. Therefore, it is recommended for small-scale graphs or users without solving time limit. In the following subsections, we introduce the GA-based flow in Section V-A.1, and the ILP-based scheduling in Section V-A.2.

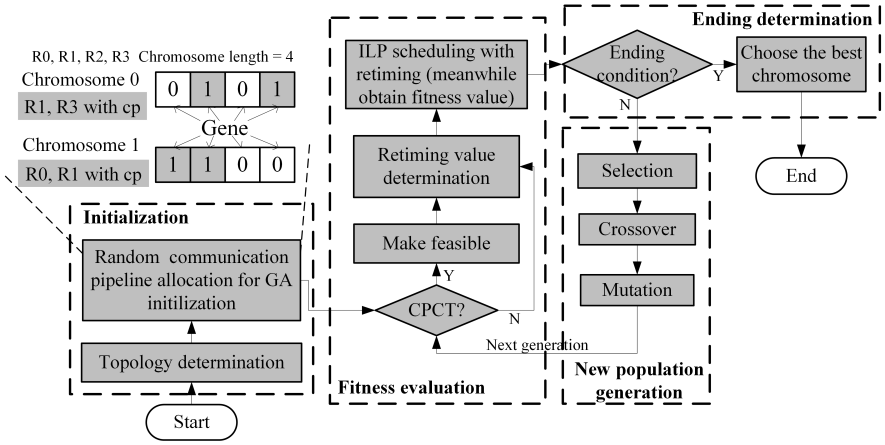


FIGURE 4. S-FCATS working flow.

1) GA-BASED FLOW

In our GA algorithm, a chromosome (i.e. solution) is a permutation of binaries representing the allocation of communication pipeline on all available receiving tasks which have computation tasks dependent on them. Each binary is a gene, and the value "1" represents communication pipeline is allocated on this receiving task and "0" otherwise. Thus, the chromosome length is constant and set as the number of such receiving tasks. As illustrated in Figure 4, the GA algorithm contains four phases: initialization, fitness evaluation, new population generation, and ending determination.

a: INITIALIZATION

The initialization phase first decides if the input application is CPAT or CPCT. Then, predefined number of chromosomes of the initial populations are generated containing random genes.

b: FITNESS EVALUATION

For chromosomes to be evaluated, if the application is CPCT, the generated genes are judged and adjusted for the cyclic constraints considering the maximum number of communication pipeline $\min(N_D - 1, N_R)$. For infeasible chromosomes, their "1" genes are traversed and randomly turned into "0" to reduce the number of communication pipeline until all cyclic constraints are met. If the application is CPAT, it goes straightly to be evaluated. As our problem is to minimize the schedule length, the fitness function is expressed as $fitness = sched\_length_{max} - sched\_length$, where $sched\_length_{max}$ is the maximum schedule length observed in current population and $sched\_length$ is the schedule length under its communication pipeline allocation. To find the optimal scheduling under this allocation, we first use Algorithm 1 to obtain the retiming value of each task. We then develop a set of ILP formulations considering the retiming values to solve the scheduling problem, which will be described in details in Section V-A.2.

c: NEW POPULATION GENERATION

If the ending condition is not met, a new population is generated based on the old population through the roulette selection and elite strategy, one-point crossover, and random mutation phases. As making chromosomes feasible increases the chances of generating identical chromosomes, we set the mutation probability relatively high. After enough chromosomes are generated, chromosomes are made efficient corresponding to the topology of the input application.

d: ENDING DETERMINATION

The larger the fitness value, the smaller the scheduling length, and the corresponding best chromosome is retained in each iteration. Therefore, the optimal chromosome obtained after the final iteration is definitely the smallest schedule length in all generated chromosomes. The whole algorithm ends when the number of iterations reaches the user-defined values. To set the number of iterations, we can first set an acceptable time limit for the GA process. As the most time-consuming part is to solve the ILP formulations in each iteration, actually the time limit is for the maximum times of ILP performances. Therefore, we simply use the result of the time limit divided by the estimated ILP solving time once as the iteration number.

2) ILP-BASED SCHEDULING

We exploit the ILP-based method to find the optimal scheduling under the allocated communication pipeline. As we have utilized retiming values to reflect task dependencies as well as the application topology, the ILP method can be applied for both CPAT and CPCT. Moreover, to better describe the task dependencies considering retiming values, we define a new concept called retimed cycle.

Retimed cycle: For any two tasks t_i and $t_j (i \neq j)$ with retiming value rt_i and rt_j respectively, if the starting order of original tasks are maintained for retiming tasks, we regard the retimed tasks within a retimed cycle.

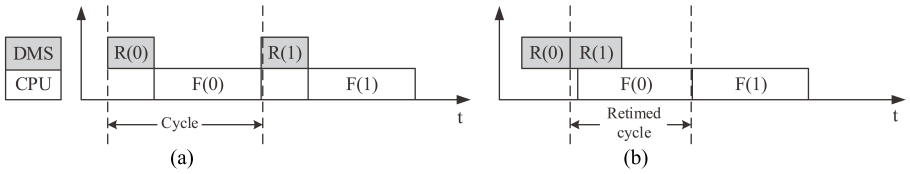


FIGURE 5. A example of retimed cycle. (a) Original cycle. (b) Retimed cycle.

TABLE 1. Constants and variables.

Constants	Representation
$T/T_F/T_S/T_R$	Whole/computation/sending/receiving task set
P	Processor set
$CYCS$	Number of total cycles
ET_i	Execution time of task $t_i (t_i \in T)$
$M_{i,k}$	Task t_i mapped on processor $p_k (t_i \in T, p_k \in P)$
$D_{i,j}$	Task dependency. Integer constant, the value is 1 if task t_j depends on t_i and 0 if t_j does not depend on t_i. If there is a k-cycle delay between t_i and t_j, the value is $-$k $(t_i, t_j \in T)$
rt_i	Retiming value of task t_i $(t_i \in T)$
SUT	Startup time of one communication transfer
MAX	Large enough integer for constraint linearization
Variables	**Representation**
$total$	Total execution time
$start_{i,k}$	Start time of task t_i $(t_i \in T)$ in cycle k $(k \in [1, CYCS])$
$exe_{i,j}$	Execution order between any two tasks. Boolean variable, the value is 1 only if task t_i is executed before task t_j $(t_i, t_j \in T)$

An example of retimed cycle is shown in Figure 5. Supposing F can only start after R and the process goes twice. In Figure 5(a), $F(0)$ should start after $R(0)$ finishes, and $F(0)$ and $R(0)$ are both in the original cycle0. In Figure 5(b), $rt_R = 1$ and $rt_F = 0$. Therefore, $F(0)$ should start after $R(1)$ finishes setup (of course after $R(0)$ finishes) and the starting order between $R(0)$ and $F(0)$ in Figure 5(a) is kept for $R(1)$ and $F(0)$. In this case, $R(1)$ and $F(0)$ are in a retimed cycle.

To better illustrate the ILP formulations, we list the constants and variables in Table 1. The objective function is to minimize the total execution time denoted by Equation (1).

$$min(total) \tag{1}$$

The constraints can be categorized into 7 sets and represented by Equation (2) to Equation (14).

- Total execution time constraint: total execution time should be not smaller than the finish time of any task in the last cycle. For any task $t_i \in T$,

$$total \geq start_{i,CYCS} + ET_i \tag{2}$$

- Successive cycle constraint: any task in next cycle can only start after it finishes in current cycle. For any task $t_i \in T$ in cycle $k \in [1, CYCS)$,

$$start_{i,k} + ET_i \leq start_{i,k+1} \tag{3}$$

- Task order constraint: the execution order between any two tasks should satisfy the task dependencies. For any two tasks $t_i, t_j \in T$,

$$exe_{i,j} \geq D_{i,j} \tag{4}$$

- Delay constraint between tasks: delay is only added between computation tasks or computation tasks and sending tasks. As the retiming values are determined based on the delay cycles, therefore, the execution of the tasks should meet the delay constraint regardless of their retiming values. For tasks $t_i (t_i \in T_F)$ and $t_j (t_j \in T_F$ or $t_j \in T_S)$ mapped on the same processor, if there is delay between the two tasks, then for $k \in [1, CYCS + D_{i,j}]$,

$$start_{i,k} + ET_i \leq start_{j,k-D_{i,j}} \tag{5}$$

- Dependency constraint between any two sending and receiving tasks in one communication vector: the receiving task can only start after the sending task finishes in each cycle. As they must have the same retiming value, rt_i and rt_j can be not considered in this constraint. For any corresponding sending and receiving tasks $t_i \in T_S$, $t_j \in T_R$,

$$start_{i,k} + ET_i \leq start_{j,k}, k \in [1, CYCS] \tag{6}$$

- Non-overlap constraint: for task t_i and t_j mapped on the same processor with no delay between, the execution of the two tasks cannot be overlapped in the following two cases.
  - Within one (retimed) cycle: if $exe_{i,j} = 1$ and $t_i \in T_F$, no matter if t_j is a computation task, sending task or a receiving task, t_j can only start after t_i finishes. A special case is for $t_i \in T_R$, $t_j \in T_F$, and t_j depends on t_i. In this case, if communication pipeline is not applied on t_i, then t_j can only start after t_i finishes receiving and they cannot be overlapped. For $k \in [1, CYCS - max(rt_i, rt_j)]$,

$$start_{i,k+rt_i} + ET_i \leq start_{j,k+rt_j} + (1 - exe_{i,j}) * MAX \tag{7}$$

$$start_{j,k+rt_j} + ET_j \leq start_{i,k+rt_i} + exe_{i,j} * MAX \tag{8}$$

  Otherwise, if communication pipeline is applied on t_i, t_j can only start after t_i starts up and then they can be overlapped in this retimed cycle. For $k \in [1, CYCS - max(rt_i, rt_j)]$,

$$start_{i,k+rt_i} + SUT \leq start_{j,k+rt_j} + (1 - exe_{i,j}) * MAX \tag{9}$$

$$start_{j,k+rt_j} + SUT \leq start_{i,k+rt_i} + exe_{i,j} * MAX \tag{10}$$

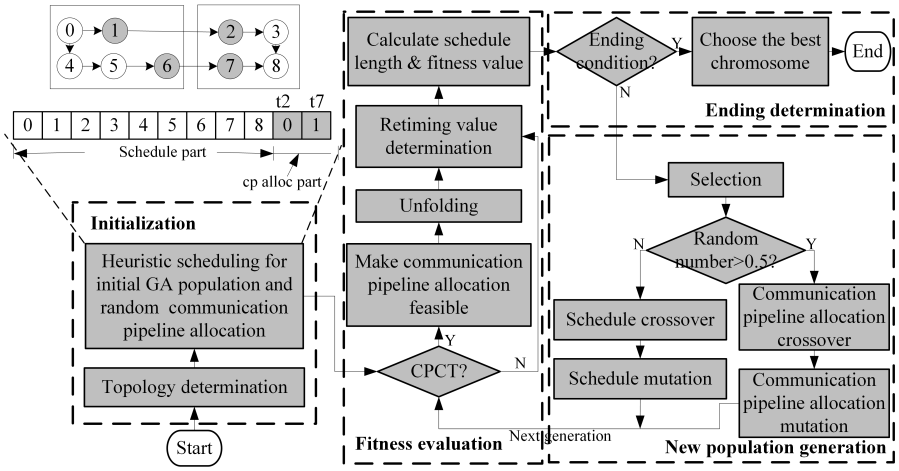


FIGURE 6. H-FCATS working flow.

– For successive (retimed) cycles: if $exe_{i,j} = 1$ and $t_j \in T_F$, no matter if t_i is a computation task, sending task or a receiving task, t_i of next cycle can only start after t_j of current cycle finishes. For $k \in [1, CYCS - max(rt_i, rt_j)]$,

$$start_{j,k+rt_j} + ET_j \leq start_{i,k+1+rt_i} + (1 - exe_{i,j}) * MAX \tag{11}$$

$$start_{i,k+rt_i} + ET_i \leq start_{j,k+1+rt_j} + exe_{i,j} * MAX \tag{12}$$

Otherwise if $exe_{i,j} = 1$ and $t_j \in T_S$ or T_R, no matter if t_i is a computation task, sending task or a receiving task, t_i of next cycle can only start after t_j of current cycle starts up. For $k \in [1, CYCS - max(rt_i, rt_j)]$,

$$start_{j,k+rt_j} + SUT \leq start_{i,k+1+rt_i} + (1 - exe_{i,j}) * MAX \tag{13}$$

$$start_{i,k+rt_i} + SUT \leq start_{j,k+1+rt_j} + exe_{i,j} * MAX \tag{14}$$

For non-overlap and overlap constraints, if $min(rt_i, rt_j)$ is larger than 1, the cycles preprocessed should also be taken into account. Then the $start_{i,k+rt_i}$ variable should be $start_{i,rt_i-k}$ and $k \in [0, min(rt_i, rt_j))$.

Solving these ILP formulations, we can obtain the optimal scheduling results under the defined communication pipeline allocation as well as the fitness value of the GA algorithm.

B. H-FCATS

Although ILP-based methods can give optimal or near-optimal solutions, it consumes considerable time and memory, and even the solutions cannot be obtained when problem instances grow large. Therefore, we have developed H-FCATS with a similar flow as S-FCATS but utilizing a hybrid task scheduling based on GA and heuristic scheduling algorithms, as shown in Figure 6.

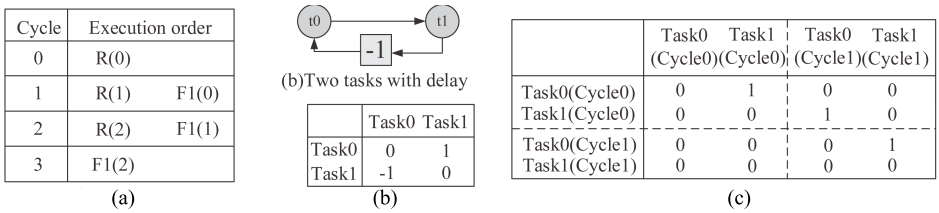
The H-FCATS solution is represented by a chromosome with two parts as the example shows in Figure 6. The *schedule* part represents the scheduling order of the whole application, i.e. the descending priorities of tasks, and each gene stands for a task number. The *cp alloc* part represents the communication pipeline allocation with binary genes just as the chromosome of S-FCATS. As the two parts are somehow independent, we refer the GA flow in [26] to separately deal with each part in the new chromosome generation process in successive iterations. The flow also contains four steps as S-FCATS but each with different implementations.

a: INITIALIZATION

This phase also begins with topology determination and block loops report. To have a high-quality population to accelerate the GA process, the well-known heuristics, uprank and downrank are applied [30]. The heuristics can determine the priorities of tasks of the *schedule* part and inherently generate schedules satisfying the task precedent dependencies. Uprank calculates the path length between the exit task and each task, and downrank calculates the path length between each task and the entry task. The recursive formula of each heuristic is listed by Equation 15 and Equation 16. Note that the original definitions of uprank and downrank involve the communication cost as a separate parameter in the formula, while our approach models communications as tasks and unifies the parameter with ET_i. Based on the two chromosomes, other chromosomes of the population are generated using the method described in the crossover and mutation processes below, and the whole population are kept diverse. The *cp alloc* part is generated randomly as long as the allocation is feasible.

$$\begin{aligned} rank_u(t_i) &= ET_i + \max\left(rank_u(t_j)\right)(t_j \in succ(t_i)); \\ rank_u(t_{exit}) &= ET_{exit} \end{aligned} \tag{15}$$

$$rank_d(t_i) = \max\left(rank_d(t_j) + ET_i\right)(t_j \in pred(t_i)) \tag{16}$$



Cycle	Execution order	
0	R(0)	
1	R(1)	F1(0)
2	R(2)	F1(1)
3	F1(2)	

(a)

	Task0	Task1
Task0	0	1
Task1	-1	0

(b)

	Task0 (Cycle0)	Task1 (Cycle0)	Task0 (Cycle1)	Task1 (Cycle1)
Task0(Cycle0)	0	1	0	0
Task1(Cycle0)	0	0	1	0
Task0(Cycle1)	0	0	0	1
Task1(Cycle1)	0	0	0	0

(c)

FIGURE 7. Retiming and unfolding. (a) Example of task exection order after retiming. (b) Two tasks with delay. (c) Dependency matrix before unfolding. (D) Dependency matrix after unfolding.

b: FITNESS EVALUATION

The fitness of each chromosome is also represented by the difference between the maximum schedule length of this generation and the current schedule length as S-FCATS. However, the way of calculating the schedule length is different. After the schedule order and the communication pipeline allocation are obtained from a GA chromosome, the schedule length for CPAT and CPCT can be calculated respectively.

- For CPAT, we can just determine the retiming values of all tasks based on the communication pipeline allocation using Algorithm 1. Based on the retiming value of each task, we can decide the task appearance and execution order in each cycle. Recall the example in Figure 2, the task appearance and execution order in each cycle is shown in Figure 7(a). Then with the mapping relation, the schedule order and the retiming values of tasks, the schedule length of the graph on the system can be obtained from Equation 2. The start time of each task in each cycle can be recursively calculated by the following Equation 17.

$$start_{i,k} = \max\left(start_{m,k} + ET_m, \max\left(start_{n,k} + ET_n\right)\right) \tag{17}$$

  t_m is the task executed before t_i on the same processor, and $t_n \in pred(t_i)$.
- For CPCT, the retiming values are calculated neglecting the edges with delay, and then the delays are eliminated utilizing the unfolding technique [40]. The unfolding technique was first proposed to convert cyclic SDF graphs into acyclic graphs by distributing delays into different cycles. As our model also has the concept of delays and cycles, we exploit the unfolding technique as well by extending the dependency matrix into all cycles. The unfolding technique tries to transform any two tasks T_a and T_b with k cycle delays into the dependency between $T_a(i)$ and $T_b(i+k)$, i.e. turns intra-cycle dependency into inter-cycle dependency. An example is shown in Figure 7(b) and (c). Task0 and task1 are executing for 3 cycles and they have inter-dependency with 1 cycle delay to avoid the deadlock, i.e. task0 of cycle1 receives data from task1 of cycle0. By unfolding the matrix by listing the dependencies in all cycles, the "-1" dependency between task0 and task1 can be transformed to task0 in cycle1 dependent on task1 on cycle0. In this way, all delays can be eliminated. Therefore, after unfolding the execution order is $t0(0)$, $t1(0)$, $t0(1)$, $t1(1)$, $t0(2)$ and $t1(2)$. With retiming, the retiming value of each task is applied to all cycles of this task. Assuming $rt_0 = 1$ and $rt_1 = 0$, $t0(0)$, $t0(1)$, and $t0(2)$ all should be executed one cycle before, so the execution order is $t0(0)$, $t0(1)$, $t1(0)$, $t0(2)$, $t1(1)$ and $t1(2)$. After obtaining the execution order, the schedule length can also be calculated by Equation 17.

c: NEW POPULATION GENERATION

The new population is generated through selection (the same with S-FCATS), crossover and mutation. To generate new chromosome of both parts, before generation in each iteration, we first generate a random float number between 0 and 1. If it is smaller than 0.5, then the new chromosome of this iteration is generated based on changing the *schedule* part. Otherwise, the new chromosome is based on changing the *cp alloc* part. For the *schedule* part, to keep the task dependency when generating new schedules, the one-point crossover and random mutation are done based on the theorems described in [25]. As the crossover example in Figure 8(a) shows, for two parents p_a and p_b, a random index r_i is first generated to cut p_a and p_b into two parts respectively. The two children c_a and c_b inherent the left parts of p_a and p_b separately. For the right part, c_a (c_b) fills its right part by traversing p_b (p_a) and picking the genes that have not appeared in its left part. As the mutation example in Figure 8(b) shows, for a chromosome in the old population, we first randomly select an index i to mutate. Then we find the locations of its nearest predecessor and successor and this defines the index range that gene i can be moved. Thus, we choose a random index j in the range except for i. If j is before i, then i is inserted before j. Otherwise, i is inserted after j. The crossover and the mutation can keep the task dependency and avoid the large body of infeasible solutions obtained from the standard one-point crossover and random mutation. As for a certain schedule, some communication tasks have been inherently overlapped with their successive computation tasks. Thus for the *cp alloc* part, it is not necessary to apply communication pipeline on these communication tasks. Therefore, before performing crossover and mutation on the second part, we examine such communication tasks and set their corresponding gene values as 0. Then

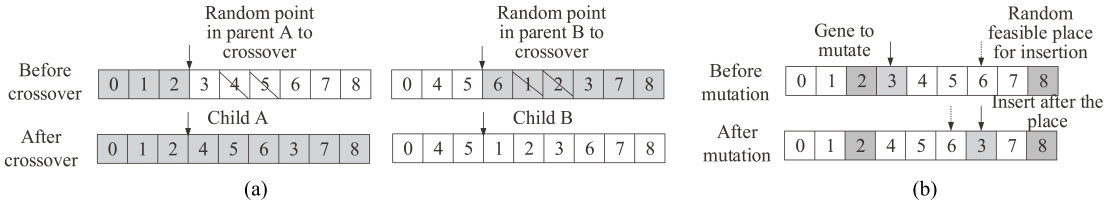


FIGURE 8. H-FCATS schedule new population generation. (a) Schedule crossover. (b) Schedule mutation.

the left undetermined genes go through the crossover and mutation processes just as that in S-FCATS.

d: ENDING DETERMINATION

The whole algorithm ends when the number of iterations reaches the user-defined value.

C. EXTENSION TO HETEROGENEOUS MPSOC

As heterogeneous MPSoC is becoming prevalent with the increasing demands for high performance, the proposed approach can also be applied to the heterogeneous platforms with the execution time of a task varies on different processors. Firstly, since the task-to-processor mapping is already known, the execution time of all tasks can be determined, which unifies the approach on homogeneous and heterogeneous MPSoC. Secondly, the usage of communication pipeline is irrelevant to task execution time but relevant to adjusting the cycles of tasks. The proposed approach can be effective as long as the hardware supports the parallelism of computation and communication. Therefore, the approach can be applied to both homogeneous and heterogeneous MPSoC.

VI. EXPERIMENTS

To show the efficiency of the proposed scheduling approach, we exploit both synthetic and real-life benchmarks, including actual application task graphs from Standard Task Graphs (STG) [41], Task Graph For Free (TGFF) [42]-generated task graphs, and an actual H.264 baseline decoder application. All applications are designed to execute on a 4/8/16-CPU platform (denoted by 4/8/16P).

The objective of the evaluation is to compare the proposed approach with previous works on schedule length. The proposed S-FCATS is compared with the previous work [21] (denoted by ILPTS), where an ILP-based task scheduling method was proposed based on the fine-grained Simulink model with the objective of minimizing schedule length without communication optimization. H-FCATS is compared with the first step of HEFT [29], i.e. directly using uprank to determine priority (denoted by URK) and a recent GA-based task scheduling approach (denoted by GATS) [26]. All scheduling approaches are tested under the same deterministic mapping for each benchmark. In the following subsections, we first introduce the experimental platform in Section VI-A and then discuss the experimental results in Section VI-B.

A. PLATFORM

1) HARDWARE PLATFORM

The whole approach is running on a 64-bit Windows10 with Intel Core i5 CPU at 2.3G Hz and 4GB RAM. The experimental MPSoC platform is with flexible configurations as shown in Figure 9(a). The platform contains at most 16 CPU subsystems, a memory subsystem, a peripheral subsystem and an interconnection subsystem. Each CPU subsystem uses a 32-bit local bus matrix to connect one processor with other local components. The processor type is configured as a 32-bit 7-stage pipeline CKCore RISC processor [43]. The Memory subsystem uses a 64-bit local bus matrix to connect on-chip global SRAM and off-chip DDR2 SDRAM. These three subsystems are connected with DMS interconnection subsystems respectively through a Memory Service Access Point(MSAP) [44]. The DMS acts as a server that provides the communication and synchronization services to the clients. Each MSAP delivers data transfer requests issued by its corresponding subsystem to other MSAPs via the control network. In this paper, architecture-level memory-related constraints (e.g. memory bandwidth constraint) are not considered for now.

2) SOFTWARE PLATFORM

The ILP formulations are solved by Cplex. Other programs are implemented by C language, and compiled and linked by gcc. For GA parameters, we set the crossover probability as 0.8 and the mutation probability 0.2. The population size is 20. The variables are set empirically and users can adjust according to the actual situations.

The proposed scheduling strategy has been integrated with the Simulink-based MPSoC design platform-LESCEA multithreaded code generator [15]. LESCEA takes a Simulink-modeled application as an input, generates a set of multithreaded C codes and builds software stacks on targeting hardware architecture. As shown in Figure 9(b), the general multithreaded code generation flow contains four main steps: task mapping, task scheduling, thread code generation and hardware dependent software (HdS) adaption.

Task mapping: Tasks are allocated to processors. In this work, each task represents a thread and there are multiple threads on each processor. We use the task assignment algorithm in [21] to assign each task to the MPSoC.

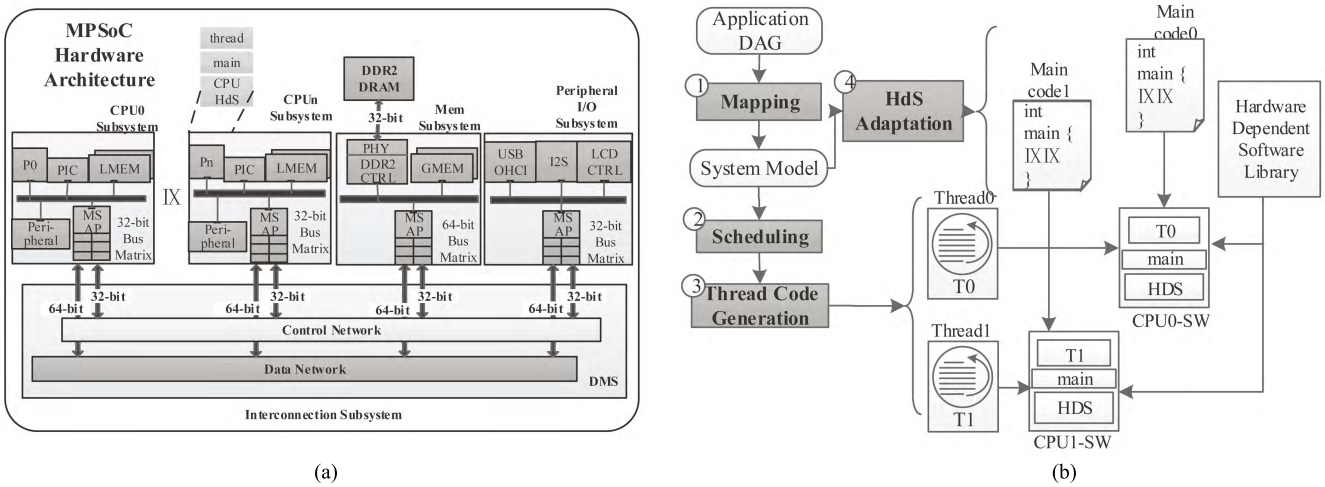


FIGURE 9. Hardware and software platform. (a) Hardware architecture. (b) LESCEA code generation flow.

Task scheduling: The execution sequence of tasks on each processor is determined after this step. This work mainly deals with this step.

Thread code generation: After determining the mapping and scheduling result, this step generates a set of C codes, including memory declarations and function calls related to the task scheduling results, and maps the memory space and function parameters.

HdS adaption: This step generates main function code for threads and initializes communication channels for CPU subsystem, as well as generates Makefile which links threads and HdS library.

B. EXPERIMENTAL RESULTS

1) SYNTHETIC GRAPHS

TGFF is a widely used DAG generator, which can generate various acyclic task graphs with different computation and communication relations, especially useful for theoretic analysis for task mapping and scheduling problems. In this work, we use TGFF to generate a series of graphs with different computation and communication features. To test both CPAT and CPCT inputs, we first generate acyclic task graphs for CPAT and then randomly add edges with delays onto the task graphs to make CPCT. We assume each application is executed for 10 times and the delay number is regarded maximum as 8. We have generated 3 small-scale and 3 large-scale task graphs with different Communication-to-Computation Ratio (CCR) for both acyclic and cyclic graphs. We use the same computation time, the same communication relations (including delay edges) and only vary the communication time of each edge. Small-scale graphs each have 16 tasks, 22 edges, and CCR is among 0.3 (denoted by SMSP), 1 (denoted by SMEP), and 3 (denoted by SMLP). Large-scale graphs each have 93 tasks, 138 edges, and CCR is also among 0.3 (denoted by LMSP), 1 (denoted by LMEP), and 3 (denoted by LMLP).

TABLE 2. Synthetic benchmark configurations.

Benchmark	No. of tasks	No. of edges	CCR
SMSP	16	22	0.3
SMEP	16	22	1
SMLP	16	22	3
LMSP	93	138	0.3
LMEP	93	138	1
LMLP	93	138	3

Table 2 shows the detailed information of each synthetic graph.

a: RESULTS FOR ACYCLIC TGFF

The results of acyclic TGFF are shown in Figure 10. The proposed approach shows better performance on small-scale and large-scale task graphs with different CCRs.

The performance is enhanced gradually with the increasing number of processors, which demonstrates the proposed scheduling approach can efficiently exploit the parallelism of the applications. Compared to ILPTS, S-FCATS achieves at most 26.08% improvement for 4P, at most 23.55% for 8P, and 15.58% for 16P, which is mostly contributed to the application of communication pipeline. Compared to URK, H-FCATS achieves at most 36.62% improvement for the 4P, at most 24.73% for 8P, and 20.23% for 16P. The large improvement is contributed to the adjustment of scheduling towards optimality and the application of communication pipeline in H-FCATS. Compared to GATS, H-FCATS achieves at most 26.15% improvement for 4P, at most 24.83% for 8P, and 27.59% for 16P, with also considerable improvement. H-FCATS and GATS both uses GA-based algorithm, but H-FCATS exploits the communication pipeline technique to reduce the communication overhead. Note that in some cases like SMLP-4P, SMLP-16P, and LMEP-8P, S-FCATS/H-FCATS shows the same performance as ILPTS/GATS. The reason is that for some graphs,

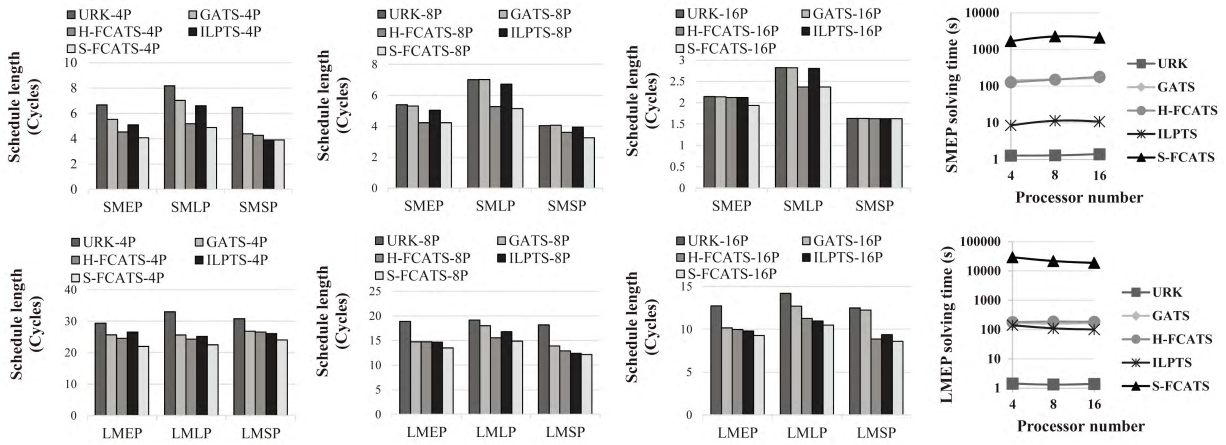


FIGURE 10. Performance results of acyclic TGFF.

after scheduling some communications have been overlapped with their previous computations and the others are not fit for the application of communication pipeline, therefore not using communication pipeline gives the best performance. Compared S-FCATS and H-FCATS, for small-scale graphs, the latter is 8.16% worse on average than the former for 4P, 4.13% for 8P, and 2.96% for the 16P. For large-scale graphs, the average percentage is 9.15% for 4P, 6.16% for 8P, and 5.61% for 16P. The results illustrate that while S-FACTS can give optimal or near-optimal solutions, the proposed H-FCATS also has acceptable solution quality. Moreover, the percentage decreases as the number of processors increase due to the decreasing number of tasks on each processor.

For small-scale graphs or large-scale graphs with different CCRs, the improvement percentage is proportional to CCR with high probability due to the increasing amount of communication. However, it is not absolute because for SMLP or LMLP, communication pipeline can reduce the major transfer time, but there are other types of minor communication overheads like startup time or synchronization time, which communication pipeline cannot handle. However, the high probability of performance improvement indicates the efficiency of using communication pipeline.

Furthermore, we have also plotted the solving time under all five approaches. As the time consuming trend is similar for the same scale with different CCRs, we only plot the time for SMEP and LMEP. For both small and large graphs, URK can obtain solutions in about 1s, but it gives the worst performance. H-FCATS and GATS have almost the same solving time because they go through similar GA process with the same iterations, but H-FCATS can bring better performance. Although H-FCATS takes much larger time than URK due to the GA iterations, but it is totally acceptable that we can obtain 20%-30% performance improvement with a tradeoff of about 1min time, and even for large graphs, the time consumption has not grown much. The solving time of ILPTS and S-FACTS grow with the scale of graphs. S-FCATS takes hundreds times of solving time of ILPTS due to the GA iterations. However, for small-scale graphs, it is acceptable as well to obtain the best performance by trading off some time.

b: RESULTS FOR CYCLIC TGFF

As ILPTS, URK and GATS cannot deal with cyclic graphs, to make the comparison fair, we add the delay constraints in the ILP model, and unfolding with URK and GATS so all results can be obtained for cyclic TGFF. As it takes unacceptable time for ILP-based approaches to obtain solutions for large-scale graphs, only results for small-scale graphs of S-FCATS are shown in Figure 11. For both S-FCATS and H-FCATS, the proposed approach yields better performance on task graphs with different CCRs.

S-FCATS can give up to 12.14% performance improvements than ILPTS for SMEP, up to 25.05% for SMLP, and up to 5.36% for SMSP. Although the application of communication pipeline requires more constraints for cyclic graphs, we can still achieve performance improvements like the trend of acyclic TGFF, which demonstrates the efficiency of S-FCATS on cyclic graphs. During the experiment, we have found that there is possibility that no solutions, i.e. no effective fitness values, can be found under some communication pipeline allocations. The reason is that the cyclic constraints and the retiming constraints may be conflict. Therefore, to overcome this problem, if the no-solution case appears in the initial generation, we regenerate feasible chromosomes until all the initial chromosomes have solutions; if the no-solution case appears in other generations, we set the schedule length under the no-solution case as a maximum value, i.e. set the fitness value as 0, to avoid the selection in next generations.

H-FCATS can produce up to 35.23% better performance than URK for SMEP, 24.75% for SMLP, and 34.27% for SMSP. The improvements are at most 7.14% for LMEP, 8.58% for LMLP, and 28.61% for LMSP. For GATS, H-FCATS achieves up to 29.48% improvement for SMEP,

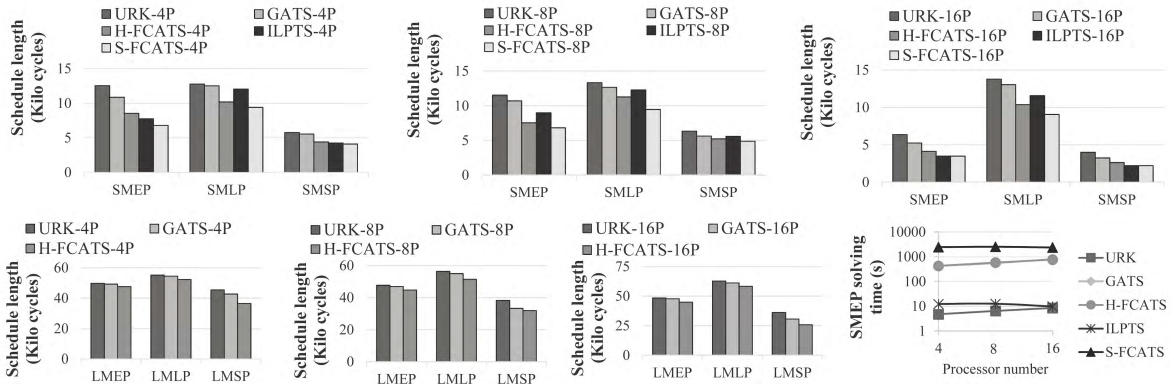


FIGURE 11. Performance results of cyclic TGFF.

20.45% for SMLP, 20.71% for SMSP, 5.82% for LMEP, 6.28% for LMLP, and 15.80% for LMSP. The solution quality trend between S-FCATS and H-FCATS is also similar to that of acyclic TGFF. A main difference is that for cyclic TGFF, H-FCATS utilizes unfolding, leading to the increasing scheduling scale and the solving time in each GA iteration grows, as seen in the representative time curve for SMEP. Though, the advantage of solving time of H-FCATS is still obvious than S-FCATS and it is suitable to handle the large-scale graphs.

2) ACTUAL APPLICATION GRAPHS

Actual application graphs contain STG and H.264 baseline decoder. STG is a benchmark for evaluation of multiprocessor scheduling algorithms. It includes both random DAGs and actual application DAGs. In this work, we exploit the actual application task graphs: robot control (denoted by RBT), sparse matrix solver (denoted by SPS), and a part of fpppp in the SPEC benchmarks (denoted by F4P) as the representations of CPAT. There are three communication time under different CCRs for the task graphs, including 0.25 (denoted by MSP), 1 (denoted by MEP), and 2.25 (denoted by MLP). Each application is executed for 10 times. Robot control has 88 tasks and 131 edges. Sparse matrix solver has 96 tasks and 67 edges. SPEC fpppp has 334 tasks and 1145 edges. H.264 baseline decoder can be modeled as a cyclic task graph with 262 tasks and 680 task relations by applying the tool from [15], [38]. It adopts a 100-frame CIF H.264 format Foreman data stream as the input. The execution time and communication size of tasks can be obtained from profiling before executed, and the CCR can be calculated out as 0.18. Table 3 shows the detailed information of each actual application graph.

The experimental results for STG with ILPTS, S-FCATS, URK, GATS and H-FCATS are shown in Figure 12 and Figure 13. As SPS and F4P consume too much time in our limited experimental environment, we have not obtained the results of S-FCATS and ILPTS for these two graphs. From Figure 12, the average communication cost of URK and

TABLE 3. Actual benchmark configurations.

Benchmark	No. of tasks	No. of edges	CCR
RBT-MSP	88	131	0.25
RBT-MEP	88	131	1
RBT-MLP	88	131	2.25
SPS-MSP	96	67	0.25
SPS-MEP	96	67	1
SPS-MLP	96	67	2.25
F4P-MSP	334	1145	0.25
F4P-MEP	334	1145	1
F4P-MLP	334	1145	2.25
H.264	262	680	0.18

GATS on 4/8/16P are reduced by 24.74% and 11.23% for RBT, 16.17% and 5.90% for SPS, and 26.92% and 19.61% for F4P. This shows that our proposed method can effectively utilize the scheduling algorithm and communication pipeline technique to reduce communication and improve system performance. One can see that from Figure 13, for all the three task graphs, performance improvements are obtained for different CCRs, and the optimizing trend is similar to that of acyclic TGFF on schedule length. S-FCATS can give up to 36.36% performance improvements than ILPTS for RBT. H-FCATS can produce up to 46.51% better performance than URK and 42.64% than GATS for RBT, 17.83% than URK and 6.54% than GATS for SPS, and 50.66% than URK and 15.41% than GATS for F4P. These data indicate that the proposed approach is effective for different kinds of task graphs, but the improvement percentage for each task graph can be quite different due to its own feature of computation and communication time, task dependencies and mapping relations. Take SPS as an example. It has 96 tasks but only 67 edges and after mapping only 2 edges left for inter-processor communication for 4P. Therefore, the results for H-FCATS, URK and GATS are almost the same.

H.264 is a real application which goes through the LESCEA code generator and finally runs on the real MPSoC platform. As H.264 also has too many tasks for our

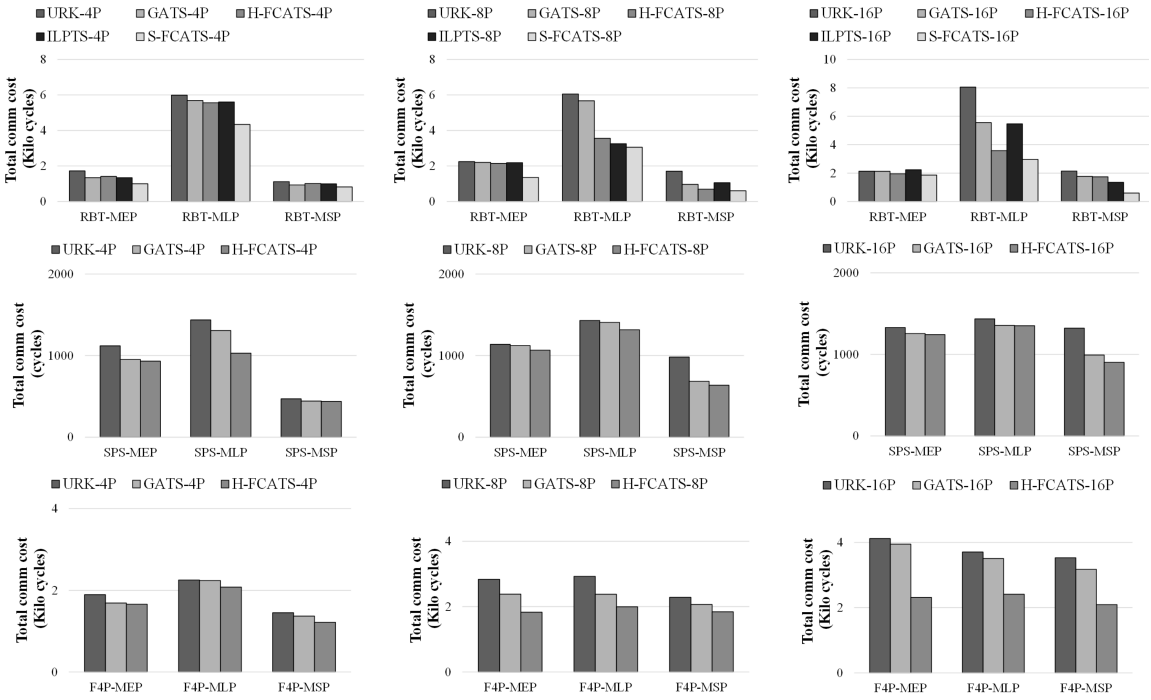


FIGURE 12. Total communication cost of STG under different methods.

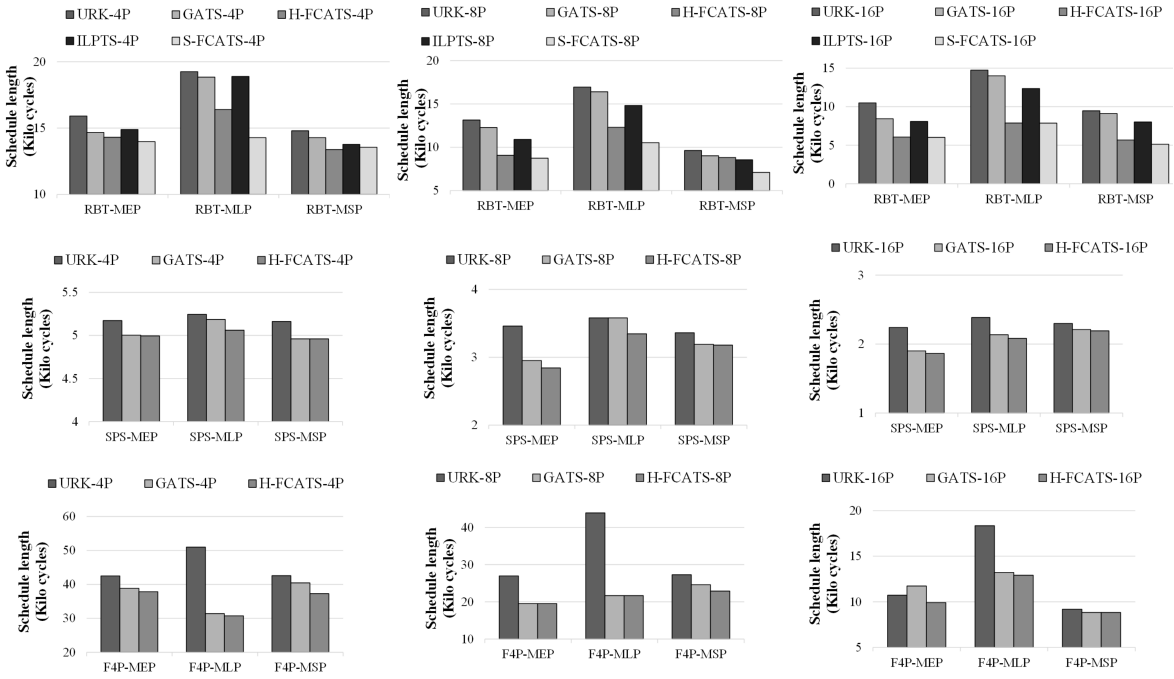


FIGURE 13. Scheduling length of STG under different methods.

experimental environment for S-FCATS, we apply H-FCATS onto H.264 and compare the result with URK and GATS. From Figure 14, the experimental result for H-FCATS shows 2.15% better performance than URK for 4P, 13.92% for 8P, and 7.01% for 16P. Compared with GATS, the percentage is 0.47% for 4P, 11.73% for 8P, and 3.78% for 16P. The performance improvement is not as high as TGFF and STG due to the small percentage of communications and the complex cyclic dependencies. However, we can still observe the communication cost reduction by analyzing the total

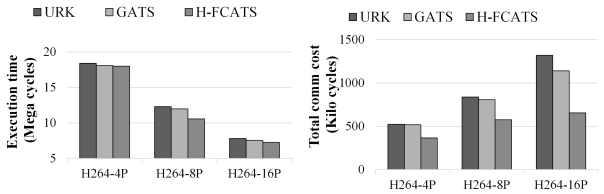


FIGURE 14. Performance results of H.264.

communication cost for all 4/8/16P cases as shown in the figure. Although the total communication cost increases with the number of processors, we have found that an average of 37.33% and 33.62% communication cost reduction on 4/8/16P for URK and GATS respectively can be obtained, contributed to the application of communication pipeline and the high-quality scheduling with much overlapping, which demonstrates the efficiency of the proposed approach.

VII. CONCLUSION

This work gives a full analysis of communication pipeline and proposes a fine-grained communication-aware task scheduling approach FCATS for pre-mapped acyclic and cyclic applications on MPSoC based on the fine-grained Simulink model. The approach integrates communication pipeline with the scheduling process to achieve performance improvements. Experimental results on both synthetic and real-life benchmarks demonstrate its efficiency.

In the future, further study can be conducted based on this work. Firstly, many important variables in the current GA algorithm are determined by experience, which calls for further optimizations. In addition, the GA algorithm can be optimized for parallel execution, which can accelerate the GA-based computation. Secondly, as the first part of the general scheduling process, task mapping, i.e. allocate application tasks on multiple processors, is also important and has great impacts on the scheduling and the communication pipeline allocation in this work, we will develop effective mapping approaches combined with the proposed scheduling approaches for further performance improvements.

REFERENCES

[1] A. D. Robison, "Intel tregistered threading building blocks (TBB)," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA, USA: Springer, 2011, pp. 955–964.

[2] C. E. Leiserson, "The cilk++ concurrency platform," *J. Supercomput.*, vol. 51, no. 3, pp. 244–257, Mar. 2010.

[3] H. Kamal and A. Wagner, "An integrated fine-grain runtime system for MPI," *Computing*, vol. 96, no. 4, pp. 293–309, 2014.

[4] (2018). *Simulink-Simulation and Model-Based Design*. [Online]. Available: https://www.mathworks.com/products/simulink.html

[5] G. W. Price and D. K. Lowenthal, "A comparative analysis of fine-grain threads packages," *J. Parallel Distrib. Comput.*, vol. 63, no. 11, pp. 1050–1063, Nov. 2003.

[6] K. Huang *et al.*, "Communication optimizations for multithreaded code generation from simulink models," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 3, p. 59, 2015.

[7] L. Brisolara *et al.*, "Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC," in *Proc. 10th Int. Workshop Softw. Compil. Embedded Syst.*, Apr. 2007, pp. 81–89.

[8] Y.-L. Tsai, H.-C. Liu, and K.-C. Huang, "Adaptive dual-criteria task group allocation for clustering-based multi-workflow scheduling on parallel computing platform," *J. Supercomput.*, vol. 71, no. 10, pp. 3811–3831, Oct. 2015.

[9] H. Kanemitsu, M. Hanada, and H. Nakazato, "Clustering-based task scheduling in a large number of heterogeneous processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3144–3157, Nov. 2016.

[10] J. Mei, K. Li, and K. Li, "A resource-aware scheduling algorithm with reduced task duplication on heterogeneous computing systems," *J. Supercomput.*, vol. 68, no. 3, pp. 1347–1377, Jun. 2014.

[11] X. Tang, K. Li, G. Liao, and R. Li, "List scheduling with duplication for heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 70, no. 4, pp. 323–329, Apr. 2010.

[12] B. P. Railing, E. R. Hein, and T. M. Conte, "Contech: Efficiently generating dynamic task graphs for arbitrary parallel programs," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, p. 25, Jul. 2015.

[13] Y. Xue and P. Bogdan, "Scalable and realistic benchmark synthesis for efficient NoC performance evaluation: A complex network analysis approach," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Oct. 2016, pp. 1–10.

[14] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proc. 50th ACM/EDAC/IEEE Design Automat. Conf. (DAC)*, Jun. 2013, pp. 1–10.

[15] S. I. Han *et al.*, "Memory-efficient multithreaded code generation from Simulink for heterogeneous MPSoC," *Des. Automat. Embedded Syst.*, vol. 11, no. 4, pp. 249–283, Dec. 2007.

[16] L. Wesolowski *et al.*, "Tram: Optimizing fine-grained communication with topological routing and aggregation of messages," in *Proc. 43rd Int. Conf. Parallel Process.*, Sep. 2014, pp. 211–220.

[17] L. Yang, W. Liu, W. Jiang, M. Li, J. Yi, and E. H.-M. Sha, "Application mapping and scheduling for network-on-chip-based multiprocessor system-on-chip with fine-grain communication optimization," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 10, pp. 3027–3040, Oct. 2016.

[18] P. E. Hadjidoukas, G. C. Philos, and V. V. Dimakopoulos, "Exploiting fine-grain thread parallelism on multicore architectures," *Sci. Program.*, vol. 17, no. 4, pp. 309–323, 2009.

[19] S. Venugopalan and O. Sinnen, "ILP formulations for optimal task scheduling with communication delays on parallel systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 142–151, Jan. 2015.

[20] Y. Wang, Z. Shao, H. C. B. Chan, D. Liu, and Y. Guan, "Memory-aware task scheduling with communication overhead minimization for streaming applications on bus-based multiprocessor system-on-chips," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 7, pp. 1797–1807, Jul. 2014.

[21] K. Huang *et al.*, "ILP based multithreaded code generation for Simulink model," *IEICE Trans. Inf. Syst.*, vol. 97, no. 12, pp. 3072–3082, 2014.

[22] W. Liu *et al.*, "Thermal-aware task mapping on dynamically reconfigurable network-on-chip based multiprocessor system-on-chip," *IEEE Trans. Comput.*, vol. 67, no. 12, pp. 1818–1834, Dec. 2018.

[23] A. Minaeva, B. Akesson, Z. Hanzálek, and D. Dasari, "Time-triggered co-scheduling of computation and communication with jitter requirements," *IEEE Trans. Comput.*, vol. 67, no. 1, pp. 115–129, Jan. 2017.

[24] C. Wang, J. Gu, Y. Wang, and T. Zhao, "A hybrid heuristic-genetic algorithm for task scheduling in heterogeneous multi-core system," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.*, Sep. 2012, pp. 153–170.

[25] Y. Xu, K. Li, J. Hu, and K. Li, "A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues," *Inf. Sci.*, vol. 270, pp. 255–287, Jun. 2014.

[26] Y.-S. Jiang and W.-M. Chen, "Task scheduling for grid computing systems using a genetic algorithm," *J. Supercomput.*, vol. 71, no. 4, pp. 1357–1377, Apr. 2015.

[27] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, "Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 29, no. 6, pp. 911–924, Jun. 2010.

[28] A. A. Badawi and A. Shatnawi, "Static scheduling of directed acyclic data flow graphs onto multiprocessors using particle swarm optimization," *Comput. Oper. Res.*, vol. 40, no. 10, pp. 2322–2328, Oct. 2013.

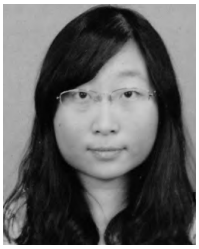
[29] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002.

[30] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 682–694, Mar. 2014.
[31] M. I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 68, no. 4, pp. 399–409, Apr. 2008.
[32] H. Croubois and E. Caron, "Communication aware task placement for Workflow scheduling on daas-based cloud," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, Jun. 2017, pp. 452–461.
[33] A. Yoosefi and H. R. Naji, "A clustering algorithm for communication-aware scheduling of task graphs on multi-core reconfigurable systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2718–2732, Oct. 2017.
[34] N. Kumar and D. P. Vidyarthi, "A novel hybrid PSO—GA meta-heuristic for scheduling of DAG with communication on multiprocessor systems," *Eng. Comput.*, vol. 32, no. 1, pp. 35–47, Jan. 2016.
[35] Y. Wang, D. Liu, Z. Qin, and Z. Shao, "Optimally removing intercore communication overhead for streaming applications on MPSoCs," *IEEE Trans. Comput.*, vol. 62, no. 2, pp. 336–350, Feb. 2013.
[36] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato, "Cache-hierarchy contention-aware scheduling in CMPs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 581–590, Mar. 2014.
[37] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 4, p. 8, 2010.
[38] S. I. Han, S. I. Chac, and A. A. Jcrraya, "Functional modeling techniques for efficient sw code generation of video codec applications," in *Proc. Asia South Pacific Conf. Design Automat.*, Jan. 2006, pp. 935–940.
[39] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, nos. 1–6, pp. 5–35, Jun. 1991.
[40] K. K. Parhi and D. G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Trans. Comput.*, vol. 40, no. 2, pp. 178–195, Feb. 1991.
[41] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *J. Scheduling*, vol. 5, no. 5, pp. 379–394, 2002.
[42] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. 6th Int. Workshop Hardw./Softw. Codesign.*, Mar. 1998, pp. 97–101.
[43] (2018). *C-SKY Ip Authorization*. [Online]. Available: http://www.c-sky.com/solution/CPU-IP-shou-quan.htm
[44] S. I. Han, A. Baghdadi, M. Bonaciu, S. I. Chae, and A. A. Jerraya, "An efficient scalable and flexible data transfer architecture for multiprocessor SoC with massive distributed memory," in *Proc. 41st Annu. Design Automat. Conf.*, Jun. 2004, pp. 250–255.

KAI HUANG received the B.S.E.E. degree from Nanchang University, Nanchang, China, in 2002, and the Ph.D. degree in engineering circuit and system from Zhejiang University, Hangzhou, China, in 2008. From 2009 to 2011, he was a Postdoctoral Research Assistant with the Institute of VLSI Design, Zhejiang University. In 2010, he was a Collaborative Expert with the VERIMAG Laboratory, Grenoble, France. Since 2012, he has been an Associate Professor with the Department of Information Science and Electronic Engineering, Zhejiang University. His current research interests include embedded processors and SoC system-level design methodology and platforms.

XIAOWEN JIANG received the Ph.D. degree from the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China, in 2018. His current research interests include multiprocessor software exploration, real-time systems, energy-efficient scheduling, and fault tolerance.

HAITIAN JIANG is currently pursuing the master's degree with the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China. His current research interests include multiprocessor software exploration and artificial intelligence.



XIAOMENG ZHANG received the Ph.D. degree from the Institute of VLSI Design, Zhejiang University, Hangzhou, China, in 2018. Her current research interests include multiprocessor software exploration and multi-thread code generation.

MIN YU received the B.S. and Ph.D. degrees from the College of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China, in 2009 and 2014, respectively, where he is currently a Research Assistant. His current research interests include performance estimation, high-performance software exploration on multiprocessor, and performance-oriented automatic code generation on MPSoC.

RONGJIE YAN received the Ph.D. degree from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2007, where she is currently an Assistant Researcher. She has spent two years with VERIMAG, Grenoble, France, where she focused on compositional and incremental verification methodology, and correctness-by-construction of component-based systems. Recently, she worked on extra-function analysis of embedded systems. Her current research interests include modeling and formal verification of embedded systems.

XIAOLANG YAN received the B.S.E.E. and M.S.E.E. degrees from Zhejiang University, Hangzhou, China, in 1968 and 1981, respectively. From 1993 to 1994, he was a Visiting Scholar with Stanford University, Palo Alto, CA, USA. From 1994 to 1999, he was a Professor and the Dean of the Hangzhou Institute of Electronic Engineering, Hangzhou. Since 1999, he has been a Professor, the Dean of the Information Science and Engineering College, and the Director of the Institute of VLSI Design, Zhejiang University. His current research interests include embedded CPU design, SoC design methodology, and design for manufacturability.