Received 4 February 2024, accepted 27 February 2024, date of publication 5 March 2024, date of current version 12 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3373902

APPLIED RESEARCH

Task Mapping and Scheduling on RISC-V MIMD Processor With Vector Accelerator Using Model-Based Parallelization

SHANWEN WU[1], SATOSHI KUMANO[2], KEI MARUME[2], AND MASATO EDAHIRO[1], (Member, IEEE)
[1]Graduate School of Informatics, Nagoya University, Nagoya 464-8601, Japan
[2]NSITEXE Inc., Tokyo 108-0075, Japan

Corresponding author: Shanwen Wu (wushanwen@ertl.jp)

This work was supported in part by NSITEXE Inc., Tokyo, Japan; and in part by Nagoya University Research Fund. The work of Shanwen Wu was supported by Nagoya University Interdisciplinary Frontier Fellowship funded by Nagoya University and Japan Science and Technology Agency (JST), the Establishment of University Fellowships Towards the Creation of Science Technology Innovation under Grant JPMJFS2120.

ABSTRACT In this paper, we propose a model-based workflow to generate parallel code on a multiple instruction stream, multiple data stream (MIMD) processor with vector accelerator (MIMDV) from a Simulink model. Solving data- and task-parallelism is crucial during this process. For data parallelism, a RISC-V Simulink library written in vector codes is prepared for blocks with sufficient vector or matrix calculations. Moreover, large inputs can be divided, which means that tasks can be executed simultaneously using multiple cores. For task parallelism, integer linear programming (ILP) is designed to deploy tasks on scalar processing elements (SPEs) and a vector processing element (VPE) of MIMDV. The use of a vector library for a task and the number of SPEs a task uses are determined. To reduce the overhead, synchronization is realized by barrier wait, and execution is divided into multiple time intervals called *layer*. We propose a novel one-step ILP that accurately minimizes the parallel time of such a situation. Furthermore, we propose a two-step ILP to achieve reasonable performance in practical time. One step is SPE mapping, and the other is layer scheduling. We tested our methods using random task graphs and real-world applications on DR1000C, a type of RISC-V MIMD processor with a vector accelerator.

INDEX TERMS Task mapping and scheduling, model-based development, multi-core processor, RISC-V, MIMD, vectorization, parallelization.

I. INTRODUCTION

Currently, there is an increasing need to process large numbers of calculations with high performance and low power consumption, even for embedded systems with limited hardware resources. Multi-core processors are widely used because of the bottleneck of single-core performance. In industrial control, various embedded multi-core system-on-chips (SoCs) have been designed to satisfy different requirements. Furthermore, with the development of artificial intelligence (AI) and computer vision, data-intensive tasks such as image processing can be executed on real-time embedded SoCs.

Handling data in a vector, called vectorization, can accelerate data parallelism. For the instruction set, there are AVX-512 [51] for the Intel CPU and Neon [52] for the ARM CPU, but they are not sufficiently flexible for some attributes such as vector length. To solve this problem, RISC-V has become popular. RISC-V [1] is an open-source general-purpose instruction set that contains concise documents that are friendly to developers. Vector extensions are defined in RISC-V, and some SoCs start to have hardware IPs that support these extensions.

Among the recent SoCs, MIMD processors with vector accelerator (MIMDV) have been developed. MIMDV can

The associate editor coordinating the review of this manuscript and approving it for publication was Thomas Canhao Xu.

accelerate applications involving task and data parallelism by simultaneously processing multiple control and data flows. DR1000C [2] of NSITEXE is a type of MIMDV for embedded systems with multiple hardware threads and specialized processing units to handle RISC-V vector instructions.

Model-based development (MBD) is considered a highly efficient method for embedded software design because programmers can easily understand and simulate systems at an early stage. MATLAB/Simulink is a representative toolchain for MBD because it provides numerous block libraries and a well-developed community. Recently, MATLAB/Simulink has been widely used, particularly for autonomous driving. For real-world implementation, a brief and descriptive block diagram in the Simulink model can be automatically translated into a sequential C code for embedded systems using the Embedded Coder [53].

Model-based parallelization (MBP) assigns blocks or subsystems in Simulink models to processors and generates parallel codes. There are two types of MBP: block and code level. In block level MBP [5], blocks or subsystems are tasks and signal lines are edges. It can extract a more simple task graph and parallelize it. However, it cannot resolve data parallelism. Examples include a simple operation (etc. addition) with large input/output vectors, matrix calculation (etc. image processing) in a Mathworks' built-in block, and a loop in an S-function block. In code level MBP [14], functions are tasks and arguments are edges. Data parallelism was resolved by transforming the loops in an S-Function Builder block into CUDA codes and finally SYCL description. However, it cannot cover all situations because the optimized loop must be in a simple format and not a complicated algorithm.

We consider combining the two types of tools that perform task parallelism by [5] and accelerate data parallelism using vectorization. Although compilers such as gcc [54] and clang [55] can perform vectorization automatically, they can only handle simple code and sometimes fail to provide desirable performance. Consequently, libraries for complicated algorithms must be prepared in advance. Reference [47] is a reliable edge AI toolchain based on MIMDV. For MBD using MATLAB/Simulink, the toolchain provides a library with blocks that generate vector codes. We need to further optimize them using MBP. In addition, the blocks using the library are data-extensive; therefore, they can be parallelized using multi-core by partitioning large inputs.

The deployment of software tasks onto hardware resources in MBP is known as task mapping. Task mapping is designed to achieve higher performance, lower communication costs and power consumption, less memory and cache contention, and workload balance. Various methods have been proposed to parallelize software applications on homogeneous [12], [15], [24] and heterogeneous [25], [26] multi-core platforms. However, the program of MIMDV is a mixture of scalar and vector codes, which makes it different from modern CPU-accelerator architecture using OpenCL [56] or CUDA [57]. MIMDV requires its own formulation to select between scalar and vector codes. The execution of tasks using multi-core should also be considered.

Task pairs with data dependency can be assigned to different cores after task mapping; consequently, we must determine a reasonable execution sequence using a scheduling algorithm to ensure the correctness of the parallel program. This process is called synchronization. In general, parallel programs using multi-core are realized using a thread language. *Thread* is an execution path that operates on a single core. [48] considered multi-threaded execution of a task in scheduling and was verified in applications with coarse tasks. However, the fork-join structure involves significant thread creation and destruction with low efficiency. Furthermore, a multi-threaded task is regarded as a batch and cannot parallel sufficiently with other tasks. Therefore, it may fail to achieve a high performance when the task granularity is small. Barrier wait is an easy and common solution to this problem. When using barrier wait, threads with constant numbers are created, and the ended thread must wait for all executing threads to end. Therefore, the program can be considered as multiple time intervals, called *layer*, divided by barrier wait, and various tasks can be parallelized more freely inside layer. The main overhead comes only from the API.

There are a few related studies on this problem. Adapting heuristics such as Best-Fit [4] is possible, but is not optimal. Reference [49] gave an ILP pipeline scheduling among multiple execution stages. Inspired by this, we define the world-first one-step ILP to mathematically describe parallel time using barrier wait. However, with the development of software and hardware, task and core numbers have increased significantly; thus, the problem is regarded as an NP-hard problem and suffers from a long solver time. Therefore, we divided the ILP problem into two steps to achieve high parallel performance and reasonable solver time.

In this study, we propose a task mapping and scheduling ILP formulation for RISC-V MIMDV. Based on this algorithm, we also propose an MBP code generation workflow.

Our contributions are as follows:

- Our workflow integrates the RISC-V Simulink library and generates parallel code using barrier wait that significantly mitigates overhead. It is world-first.
- Proposed one-step ILP is the first formulation to accurately minimize the parallel time of multiple interval execution using barrier wait. Vectorization and multi-core execution of a task are considered. Then, the proposed two-step ILP divides the problem and reduces solver time to enhance the practicability of workflow.

The remainder of this paper is organized as follows:

Section II states several definitions and assumptions. Section III introduces an MBP code generation workflow for MIMDV. Section IV describes the basic one-step ILP and Section V describes the advanced two-step ILP. Section VI introduces the experiments in which we tested randomly generated task graphs using a MATLAB script and codes translated from real-world Simulink models on an FPGA

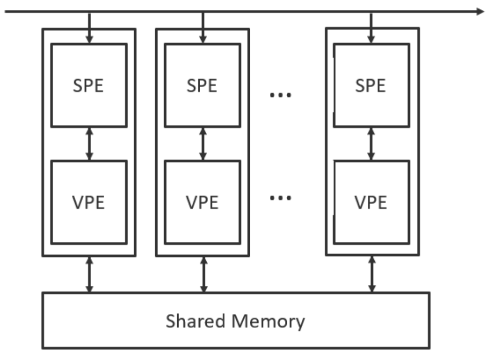


FIGURE 1. The MIMDV architecture.

emulator of DR1000C. Section VII discusses related work. Finally, Section VIII concludes the study and discusses future researches.

II. PRELIMINARY

This section presents several definitions and assumptions:

A. MIMDV ARCHITECTURE

A model of the RISC-V MIMDV architecture assumed in this study is illustrated in Figure 1. There are N_p scalar processing elements (SPEs), and each SPE is associated with a vector processing element (VPE).

When a task is executed on an SPE, RISC-V vector instructions that accelerate data parallelism are executed on the VPE. We make the following assumptions:

- A task can contain both SPE and VPE execution using RISC-V and its vector extension instructions, which means that data transfer from SPE to VPE is contained in task execution time.
- All the SPE/VPE pairs access a shared memory, and a task reads/writes values in it during execution.
- IO ports of the shared memory are more than SPE/VPE pairs, which means latency is zero when concurrent memory access happens.
- Data coherence and correctness are guaranteed by barrier wait with constant *synchronization overhead (SO)* controlling read/write order to shared memory. Data transfer between different SPEs/VPEs is that a task writes data to shared memory before the barrier wait, and the success task reads it after the barrier wait.

B. MODEL-BASED PARALLELIZATION FOR MIMDV

1) MODEL-BASED PARALLELIZATION

Model-based parallelization (MBP), such as [5], reconstructs the sequential code generated from the Simulink model to parallel one for a multi-core processor. We adapt MBP to leverage RISC-V MIMDV, and the inputs are as follows:

- Simulink model
- Sequential code generated from the model
- MIMDV parameters like synchronization overhead
- RISC-V vector library for Simulink

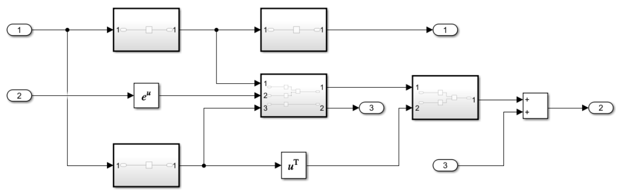
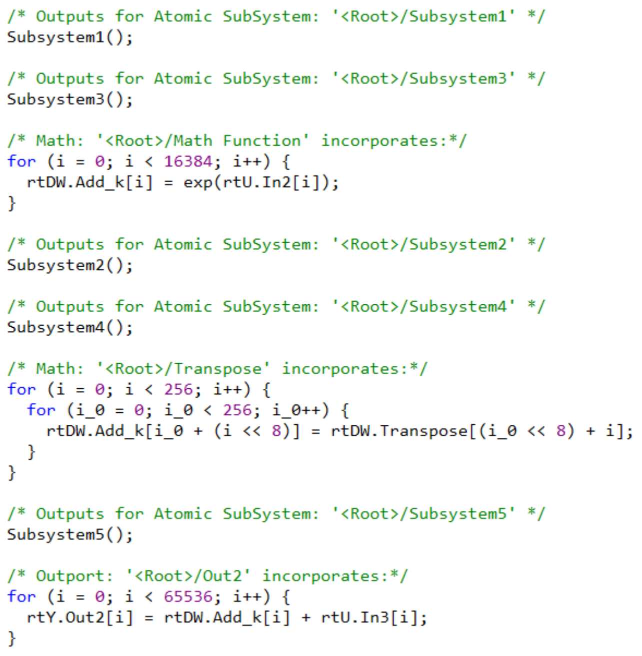


FIGURE 2. Simulink model example.



```
/* Outputs for Atomic SubSystem: '<Root>/Subsystem1' */
Subsystem1();

/* Outputs for Atomic SubSystem: '<Root>/Subsystem3' */
Subsystem3();

/* Math: '<Root>/Math Function' incorporates:*/
for (i = 0; i < 16384; i++) {
  rtDW.Add_k[i] = exp(rtU.In2[i]);
}

/* Outputs for Atomic SubSystem: '<Root>/Subsystem2' */
Subsystem2();

/* Outputs for Atomic SubSystem: '<Root>/Subsystem4' */
Subsystem4();

/* Math: '<Root>/Transpose' incorporates:*/
for (i = 0; i < 256; i++) {
  for (i_0 = 0; i_0 < 256; i_0++) {
    rtDW.Add_k[i_0 + (i << 8)] = rtDW.Transpose[(i_0 << 8) + i];
  }
}

/* Outputs for Atomic SubSystem: '<Root>/Subsystem5' */
Subsystem5();

/* Outport: '<Root>/Out2' incorporates:*/
for (i = 0; i < 65536; i++) {
  rtY.Out2[i] = rtDW.Add_k[i] + rtU.In3[i];
}
```

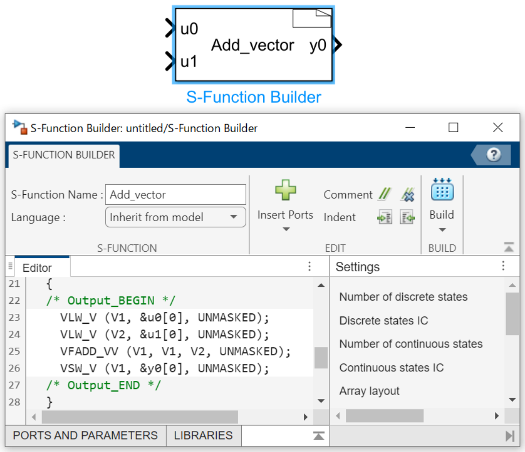
FIGURE 3. C Code generated from Figure 2 using embedded coder.

The output is vectorized threaded code for MIMDV. An example of the Simulink model is shown in Figure 2. Part of the C code generated from this Simulink model using Embedded Coder [53] is shown in Figure 3. In the example, we assume that *SO* is 10. RISC-V Simulink vector library is described in the next section.

2) SIMULINK VECTOR LIBRARY

An example of the Simulink vector library for Add is shown in Figure 4. According to [14], an S-Function Builder exhibits good performance, internal state support, C/C++ language support, and generation support. Therefore, the library is S-Function Builder blocks using inline assembly instructions in the RISC-V vector extension. In the generated code, the loop of the add operation in Figure 3 is translated into vector codes, such as the vector length setting and vector add, and the execution cycles can be reduced sharply.

The library can also handle complex MathWorks' built-in blocks. For example, vector codes for the MPC toolbox using a quadratic programming algorithm, QPKWIK, and matrix QR decomposition, Householder, were realized in [45]. In this study, library blocks for state-space equation, Sobel edge detection, and FIR image smoothing are prepared.



```
{
/* Output_BEGIN */
  VLW_V (V1, &u0[0], UNMASKED);
  VLW_V (V2, &u1[0], UNMASKED);
  VFADD_VV (V1, V1, V2, UNMASKED);
  VSW_V (V1, &y0[0], UNMASKED);
/* Output_END */
}
```

FIGURE 4. S-Function Builder block and vector code for vector addition.

C. TASK MAPPING AND SCHEDULING PROBLEM IN MBP CODE GENERATION FOR MIMDV

1) TASK GRAPH

Task graph $G(V, E, A)$ is extracted from Simulink model:

- Task node set V: Blocks or subsystems
- Dependency edge set E: Signal lines
- Graph adjacency matrix A

The following information is associated with each task for MIMDV:

- SPE execution time
- VPE execution time
- Code fragment, which is accessed by $task.code()$. A task is a class containing attributes and methods.
- Vectorization attribute, which represents a task is associated with a vector library block.
- Parallel attribute, which represents a task can be executed on multiple SPEs.

Figure 5 presents the task graph extracted from the Simulink model in Figure 2. Table 1 lists the execution times associated with each task shown in Figure 5. Before task mapping and scheduling, we further modify the task graph above:

- Parallel attribute, which represents a task contains loop iterations sufficiently larger than N_p.

Tasks with parallel attributes are partitioned into Np tasks. The execution times of partitioned tasks can generally be estimated by profiler. For simplicity, we assume in this paper they are $1/Np$ of the original one.

Figure 6 presents the resultant task graph after partitioning the tasks when $N_p = 4$. Table 2 lists the execution times associated with each task shown in Figure 6. None of the tasks in the resulting task graph have parallel attributes after the process is conducted.

From the modified task graph above, we can extract the following relations among the tasks:

- Dependent relation D: The set of task pairs with control/data dependencies. We calculate D using function $DependentRelation()$.

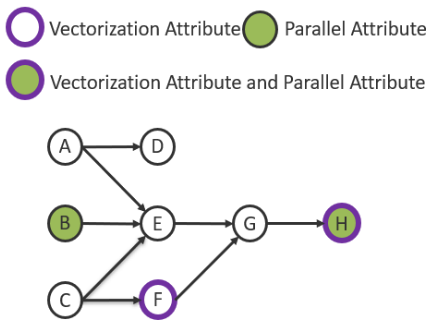


FIGURE 5. Task graph extracted from Figure 2.

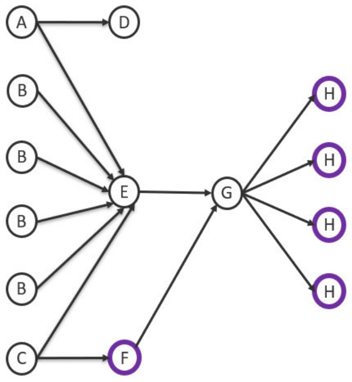


FIGURE 6. Task graph after partitioning parallelizable tasks.

TABLE 1. Execution Times of Each Task in Figure 5.

Task	SPE Execution Time	VPE Execution Time
A	200	inf
B	360	inf
C	110	inf
D	180	inf
E	80	inf
F	1000	70
G	120	inf
H	2000	520

TABLE 2. Execution Times of Each Task in Figure 6.

Task	SPE Execution Time	VPE Execution Time
A	200	inf
B1 ~B4	90	inf
C	110	inf
D	180	inf
E	80	inf
F	1000	70
G	120	inf
H1 ~H4	500	130

- Parallel relation P: The set of task pairs, which can be executed in parallel. We calculate P using function $ParallelRelation()$.
  As one solution, tasks i and j are parallel pairs if there is no dependency between them and if task j does not exist

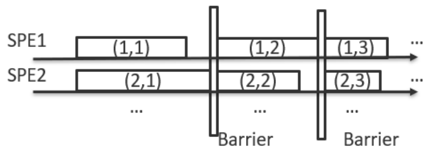


FIGURE 7. Parallel execution in this study.

on any path from start nodes to task i or on any path from task i to end nodes.

2) TASK MAPPING AND SCHEDULING PROBLEM

In this section, the task mapping and scheduling problem in MBP code generation on MIMDV is described.

To illustrate the parallel execution of MIMDV, Figure 7 shows an example of a program that uses barrier wait. From the figure, we define the following two things:

- Thread: Execution sequence on a SPE.
- Layer: Time interval divided by barrier wait.

The execution of each task is defined by (a, b), where a represents the SPE number to be deployed and b represents the layer number to be deployed. Note that, in our code generation method, a single thread executes on a single SPE, and thread number is equal to SPE number.

We aim to reduce overall parallel execution time of task graph on MIMDV. The inputs of task mapping and scheduling for MIMDV are as follows:

- Task graph adjacency matrix A
- SPE time set T_S and VPE time set T_V
- Synchronization Overhead SO

The outputs are as follows:

- Task mapping result Map:

$$[map_1, map_2, \ldots, map_{i-1}, map_i, \ldots]$$

$map_i = [m_i, n_i]$ is the deployment of task i. m_i represents the SPE number that task i is deployed on. Here, $1 \leq i \leq |V|$, and $1 \leq m_i \leq Np$. n_i is 1 when task i uses vector code and 0 when it uses scalar code. How many SPEs a task uses can be known from the deployment of partitioned tasks.

- Scheduling result $LayerInfo$:

$$[[tasks\ in\ Layer_1], [tasks\ in\ Layer_2], \ldots, [tasks\ in\ Layer_{l-1}], [tasks\ in\ Layer_l]]$$

A task is deployed into a layer. For a simple method, we can use $LayerbyDfs()$ which performs depth first search on task graph and put tasks with the same depth into the same layer. In Figure 5, $LayerInfo = [[A, B, C], [D, E, F], [G], [H]]$. After layer division, tasks in the same layer have no dependency and can be parallelized freely using heuristics such as Best-Fit.

III. MBP CODE GENERATION WORKFLOW FOR MIMDV

This section introduces the MBP code-generation workflow for MIMDV. This is illustrated in Figure 8, and the details of each step are as follows:

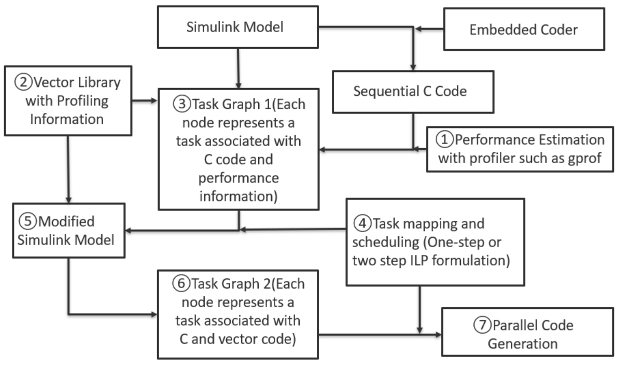

Simulink Model
Embedded Coder
②Vector Library with Profiling Information
③Task Graph 1(Each node represents a task associated with C code and performance information)
Sequential C Code
①Performance Estimation with profiler such as gprof
⑤Modified Simulink Model
④Task mapping and scheduling (One-step or two step ILP formulation)
⑥Task Graph 2(Each node represents a task associated with C and vector code)
⑦Parallel Code Generation

FIGURE 8. MBP workflow for MIMDV.

① SPE time is estimated using a profiler (etc. gprof [44]). It is able to handle complex blocks generating conditional branches and irregular loops to obtain performance close to real machine execution time.

② VPE time is obtained from coalescent simulation using the target processor in model design phase or code fragment profiling using a special compiler tool for the target processor after sequential code generation.

③ Task graph 1 is extracted by an MBP tool. Sequential C code is partitioned into fragments and associated with tasks that are blocks or subsystems. SPE and VPE times are also associated with these tasks.

④ ILP formulation described in later sections finds a solution for deploying tasks on SPEs and VPEs of MIMDV and determining execution sequence. Such a process is called task mapping and scheduling.

⑤ Modified Simulink model is created by replacing ordinary blocks with vector library blocks according to vectorization selection.

⑥ The modified model can generate vectorized C code, and tasks can contain vector code. Task graph 2 is extracted, in which code fragments of tasks contain both C and vector code.

⑦ Vectorized threaded program for MIMDV can be generated by code fragment reconstruction.

The details of the parallel code generation are described in Algorithm 1. The generated code is presented in Figure 9.

Code fragments are placed into multiple threads created in advance according to the task mapping result. These threads are executed on different SPEs.

The barrier wait API is inserted into two adjacent layers according to the scheduling result, synchronizing all threads.

$*\_vector$ is function generated from the S-Function Builder library block containing the vector codes.

One task can run on multiple threads that handle different parts of inputs and run in parallel. In this example, Exponent and Add are multi-threaded executions.

IV. ONE-STEP ILP FORMULATION

First, we propose a novel one-step ILP formulation to minimize the parallel execution time using barrier wait. The

Algorithm 1 Parallel Code Generation

```
Data: Tasks with Code, Task Mapping Result $Map$,
      Scheduling Result $LayerInfo$
Result: Parallel Code
Initialize $CodeGroup$ list based on SPE number;
for each Layer L in LayerInfo do
    for each Task i in Layer L do
        for each SPE j do
            if $m_i \in map_i == j$ then
                $CodeGroup[j].append(i.code())$;
            end
        end
    end
    $CodeGroup[j].append(barrier\ wait)$;
end
for each Code Group j do
    $print(``threadj.create();'')$;
    for each Code Fragment C in Code Group j do
        $print(C)$;
    end
end
```

```
void thread1(void)
{
  Subsystem2();
  Transpose_vector(&rtDW.Saturation[0], &rtDW.Transpose[0]);
  barrier wait1;
  barrier wait2;
  Add_vector(&rtDW.Add[0], &rtU.In2[0], &rtY.Out2[0]);
}
void thread2(void)
{
  exp_new(&rtU.In1[0], &rtDW.Transpose[0]);
  exp_new(&rtU.In1[4096], &rtDW.Transpose[4096]);
  barrier wait1;
  barrier wait2;
  Add_vector(&rtDW.Add[16384], &rtU.In2[16384], &rtY.Out2[16384]);
}
void thread3(void)
{
  exp_new(&rtU.In1[8192], &rtDW.Transpose[8192]);
  exp_new(&rtU.In1[12288], &rtDW.Transpose[12288]);
  barrier wait1;
  Subsystem4();
  Subsystem5();
  barrier wait2;
  Add_vector(&rtDW.Add[32786], &rtU.In2[32786], &rtY.Out2[32786]);
}
void thread4(void)
{
  Subsystem1();
  barrier wait1;
  Subsystem3();
  barrier wait2;
  Add_vector(&rtDW.Add[49152], &rtU.In2[49152], &rtY.Out2[49152]);
}
```

FIGURE 9. The example of parallel code generated from Figure 2.

notations are listed in Table 3. $1 \leq i$ and $k \leq |V|$ for task numbers i and k, and $1 \leq j \leq Np$ for SPE number j.

A. FORMULATION OF ONE-STEP ILP

The objective function is shown in (1). We want to minimize the finish time $finish_m$ of the task graph.

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

There are some constraints in task mapping.

Each task can only be executed on a single SPE:

$$\sum_{j=1}^{N_P} x_{ij} = 1 \tag{2}$$

Each task is executed using either vector or scalar code:

$$z_{i,V} + z_{i,S} = 1 \tag{3}$$

Each task is assigned to a single layer:

$$\sum_{l} s_{i,l} = 1 \tag{4}$$

The execution time of task i, T_i is determined by the selection between the vector and scalar code. $T_{i,S} \in T_S$ is SPE execution time and $T_{i,V} \in T_V$ is VPE execution time:

$$T_i = z_{i,V} * T_{i,V} + z_{i,S} * T_{i,S} \tag{5}$$

The relationship among the start time $start_i$, execution time T_i, and finish time $finish_i$ of task i is as follows:

$$finish_i = start_i + T_i \tag{6}$$

If task i is assigned to layer l, the task should be executed after the start time of layer $start_l$ and before the finish time of layer $finish_l$, as expressed in (7) and (8), respectively. $Maxvalue$ is a very large value, and the inequality should be satisfied when $s_{i,l}$ is 1. Furthermore, the next layer $l+1$ starts when layer l and the barrier wait API are completed, as shown in (9). SO denotes the synchronization overhead:

$$start_i + (1 - s_{i,l}) * Maxvalue \geq start_l \tag{7}$$
$$finish_i \leq finish_l + (1 - s_{i,l}) * Maxvalue \tag{8}$$
$$start_{l+1} = finish_l + SO \tag{9}$$

w_{ijk} is calculated as the following inequality: if both x_{ij} and $x_{kl}(\exists l \neq j)$ are 1, w_{ikj} is 1; otherwise, it is 0. Notably, w_{ikj} is calculated only when tasks i and k have dependent relation:

$$x_{ij} + \sum_{l \neq j} x_{kl} - 1 \leq w_{ijk} \leq \frac{x_{ij} + \sum_{l \neq j} x_{kl}}{2} \tag{10}$$

If task k consumes data from task i, they should obey the execution sequence regardless of the mapping result:

$$finish_i \leq start_k \tag{11}$$

For tasks i and k with a dependent relation, task k can be in the same or latter layer as task i as described in (12). g_{ilk} is calculated using the inequality (13), which implies that if both $s_{i,l}$ and $\sum_{l0 \geq l} s_{k,l0}$ are 1, it is 1; otherwise, it is 0. However, if two tasks are assigned to different SPEs, task k can exist only in the layers after task i as described in (14). h_{ilk} is calculated using the inequality (15), which means that

TABLE 3. Notations.

notations	Meaning
x_{ij}	1 if task i is assigned to SPE j, otherwise 0.
$z_{i,V}$	1 if task i is executed with vector code, otherwise 0.
$z_{i,S}$	1 if task i is executed with scalar code, otherwise 0.
w_{ijk}	1 if task i is assigned to SPE j while task k is not, otherwise 0.
y_{ijk}	1 if both tasks i and k are assigned to SPE j, otherwise 0.
$s_{i,l}$	1 if task i is assigned to layer l, otherwise 0.
g_{ilk}	1 if task i is assigned to layer l and task k is assigned to layer l or latter layers, otherwise 0.
h_{ilk}	1 if task i is assigned to layer l and task k is assigned to latter layers, otherwise 0.
r_{ik}	1 if task i is executed earlier than task k, otherwise 0.
o_{ik}	1 if task i is assigned to the same SPE with task k, and executed earlier than task k. otherwise 0.

if both $s_{i,l}$ and $\sum_{l0>l} s_{k,l0}$ are 1, it is 1; otherwise, it is 0.

$$\sum_{l} g_{ilk} = 1 \tag{12}$$

$$s_{i,l} + \sum_{l0\geq l} s_{k,l0} - 1 \leq g_{ilk} \leq \frac{s_{i,l} + \sum_{l0\geq l} s_{k,l2}}{2} \tag{13}$$

$$\sum_{l\neq l_{max}} h_{ilk} = 1 \tag{14}$$

$$s_{i,l} + \sum_{l0>l} s_{k,l0} - 1 \leq h_{ilk} \leq \frac{s_{i,l} + \sum_{l0>l} s_{k,l0}}{2} \tag{15}$$

y_{ijk} is calculated as the following inequality: if both x_{ij} and x_{kj} are 1, y_{ijk} is 1; otherwise, it is 0. Notably, y_{ijk} is calculated only when tasks i and k have parallel relation:

$$x_{ij} + x_{kj} - 1 \leq y_{ijk} \leq \frac{x_{ij} + x_{kj}}{2} \tag{16}$$

For two tasks i and k without dependency (with parallel relation), task i begins either earlier or later than k, as shown in (17). If they are mapped onto the same SPE. They cannot overlap, obeying the execution sequence shown in (18). *Maxvalue* is a very large value, and the inequality should be satisfied when o_{ik} is 1. o_{ik} is calculated using the inequality (19), which means that if both r_{ik} and $\sum_j y_{ijk}$ are 1, o_{ik} is 1; otherwise, it is 0.

$$r_{ik} + r_{ki} = 1 \tag{17}$$

$$finish_i \leq start_k + (1 - o_{ik}) * Maxvalue \tag{18}$$

$$r_{ik} + \sum_j y_{ijk} - 1 \leq o_{ik} \leq \frac{r_{ik} + \sum_j y_{ijk}}{2} \tag{19}$$

B. RESULT OF ONE-STEP ILP

Figure 10 presents the result of the one-step ILP for the task graph shown in Figure 5. With this task mapping and scheduling, the execution time of the task graph is 200 + 10 + 80 + 120 + 10 + 130 = 550, achieving a 7.36x performance improvement compared with the sequential C code (execution time of 4050) and a 2.98x improvement compared with the vector sequential code using one SPE/VPE pair (execution time of 1640). It is the optimal solution under the assumptions in Section II.

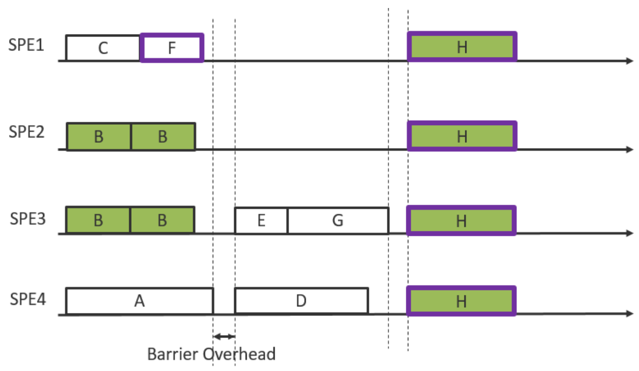


FIGURE 10. The result of one-step ILP for the task graph in Figure 5 with the execution times in Table 1.

```
Algorithm 2 Two-Step ILP
Data: Task Graph Adjacency Matrix $A$, SPE
      Execution times $T_S$, VPE Execution times $T_V$,
      Synchronization Overhead $SO$
Result: Task Mapping and Scheduling Result
$LayerInfo1 = LayerbyDfs(A)$;
$P = ParallelRelation(A)$;
$D = DependentRelation(A)$;
$[Map, T] = TaskmappingILP(T_s, T_V, P, D, SO)$;
$Speedup1 = \frac{time(Map, LayerInfo1)}{sequentialtime}$;
$LayerInfo2 =$
  $SchedulingILP(Map, T, length(LayerInfo1))$;
$Speedup2 = \frac{time(Map, LayerInfo2)}{sequentialtime}$;
```

V. TWO-STEP ILP FORMULATION

One-step ILP may suffer from a long solver time because its objective function and constraints are complex. In this section, to achieve high parallel performance in a practical range and time, we partition the problem into a two-step ILP containing task mapping and scheduling, as shown in Algorithm 2. *LayerInfo*1, P and D are calculated as mentioned before. These are prerequisites of the two-step ILP, and then we execute task mapping ILP and scheduling ILP in sequence. We explain the two ILPs in detail using model in Figure 2 as a example.

TABLE 4. Notations.

notations	Figure 14	Figure 15
n_A	1	1
n_B	1	1
m_C	2	2
$\sum_j y_{AjB}$	0	1
$\sum_j w_{AjC}$	0	0
$\sum_j w_{BjC}$	1	0
SO	5	5

A. TASK MAPPING ILP

The objective function of *TaskmappingILP*() is given in (20), as shown at the bottom of the next page. n_i is the input degree of task i and m_i is the output degree of task i. The inputs of the task mapping ILP are as follows:

- SPE time set T_S and VPE time set T_V
  Vectorization attributes can be judged by them. Only tasks with vectorization attribute have effective VPE time, or it is significantly large, thus, the task cannot be assigned to VPE by ILP formulation.
- Dependent pairs D and parallel pairs P
- Synchronization Overhead SO

The output is Task mapping result *Map*.

The first term minimizes the synchronization overhead, and the second term determines whether vectorization is worth using to accelerate the task. The third term attempts to optimize the overall parallel execution time.

The task mapping ILP obeys the constraints (2), (3), (5), (10) and (16).

Figures 11 and 12 present examples to explain the effect of the third term when parallelizable tasks with execution times of 1s, 1s, 1s, and 3s run on two SPEs. All task pairs are assumed to be parallelizable on different SPEs. Figure 11 shows the optimal solution in which the two SPEs achieve the shortest execution time. Three 1s-1s pairs are executed sequentially such that the objective function returns the best value of $1 + 1 + 1 = 3$. Figure 12 performs slightly worse because one 1s-1s pair and one 1s-2s pair are sequentially executed. In this case, the objective function returns a larger value of $1 + 3 = 4$. Therefore, minimizing the third term achieves a load balance to reduce time.

The first term combined with the third term attempts to determine the utilization of barrier synchronization to parallelize tasks. Figure 13 shows an example of a simple graph in which task C processes data from tasks A and B. According to these definitions, the A-B task pair ($\in P$) is a parallel pair, the A-C task pair ($\in D$) is a dependent pair, and the B-C task pair ($\in D$) is also a dependent pair. Two possible execution situations exist, as shown in Figures 14 and 15. Tasks A and B can be parallelized as shown in Figure 14 or sequentially executed without barrier synchronization as shown in Figure 15. The parameters of each figure are listed in Table 4.

The effectiveness of our proposed formulation is shown with two examples of task execution time. In these examples, synchronization overhead is set to 5s. When each of the three tasks takes 4s, Figure 14 is completed in $4 + 5 + 4 = 13s$, whereas Figure 15 takes a shorter time in $4 + 4 + 4 = 12s$. In this case, the sum of the first and third terms in Figure 14 is $5 * 5/(2 - 1) = 25$, whereas that in Figure 15 becomes a smaller value of $4 * 4 = 16$.

When each of the three tasks takes 6s, Figure 14 is completed in $6 + 5 + 6 = 17s$, whereas Figure 15 takes a longer time in $6 + 6 + 6 = 18s$. In this case, the sum of the first and third terms in Figure 14 is $5*5/(2-1) = 25$, whereas that in Figure 15 becomes a larger value of $6 * 6 = 36$.

B. RESULT OF TASK MAPPING ILP

Figure 16 shows the result of the task mapping ILP for the task graph in Figure 5. The number of SPEs is 4.

We can observe that task pairs without dependency are prone to run on the same SPE to realize load balance, whereas task pairs with dependency are prone to run on different SPEs to avoid synchronization. We resolved the following items:

- Whether to use vector library:
  In the example, tasks F and H benefit from vectorization and use vector codes executing on VPE.
- How many SPEs a task uses:
  It could be suitable to use more SPEs and each SPE deals with less data considering taking full advantage of hardware resources. In the example, task H uses all the four SPEs. However, it could also be suitable to use fewer SPEs and each SPE deals with more data considering leaving resources for other tasks. Task B uses two SPEs and runs concurrently with other tasks.

Parallel execution uses *LayerInfo*1, which is calculated using the node depth mentioned previously. The load balance considered in task mapping does not work because the tasks are not in a reasonable layer and require further scheduling among the layers.

C. SCHEDULING ILP

We assign tasks to the layers to minimize the load difference value among SPEs (idle time) in each layer. The objective function *SchedulingILP*() is given as (21). The inputs of scheduling ILP are as follows:

- Task mapping result *Map*
- Task time set T: Whether to use vector library has been determined in task mapping, vectorization attribute can be deleted and we can get a specific execution time for each task, as shown in (5).
- Layer number of *LayerInfo*1

Here, the output is Scheduling result *LayerInfo*2.

$$min \sum_{l} \sum_{p_1,p_2} \left| \sum_{i} T_i * s_{i,l} * x_{i,p1} - \sum_{i} T_i * s_{i,l} * x_{i,p2} \right| \tag{21}$$

Scheduling ILP obeys the constraints (4), (12), (13), (14), and (15).

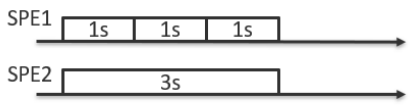


FIGURE 11. Optimal task mapping.

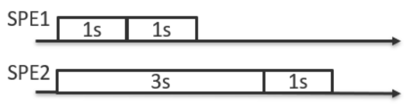


FIGURE 12. Worse task mapping.

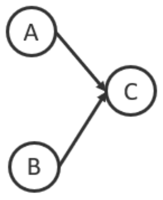


FIGURE 13. Simple example of a task graph.

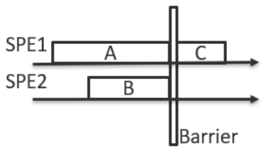


FIGURE 14. Parallel execution using Barrier wait.

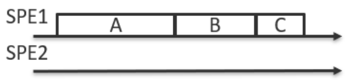


FIGURE 15. Sequential execution Merging tasks.

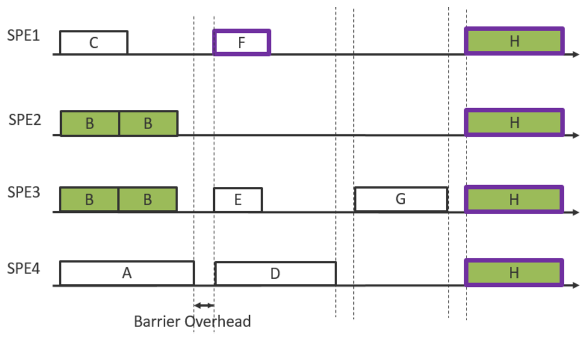


FIGURE 16. The result of the task mapping ILP for the task graph in Figure 5 with execution times in Table 1.

D. FINAL RESULT OF TWO-STEP ILP

Figure 17 depicts the final result of the two-step ILP for the task graph shown in Figure 5. This result is the same as the

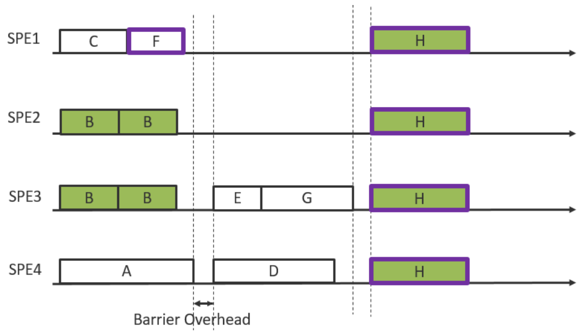


FIGURE 17. Final result of two-step ILP for the task graph in Figure 5 with the execution times in Table 1.

result of one-step ILP in Figure 10, which means that two-step ILP also finds out the optimal solution.

Two situations can occur for a task pair with a dependency. It can be executed on the same SPE in the same layer with a restricted order, or on different SPEs in different layers resolved by barrier wait. The execution sequence can vary, and scheduling ILP makes more adjustments. We resolve the following items:

- Task movement:
  Idle time can be reduced by moving task F from layer 2 to layer 1 (its node depth) to be merged with task C. Idle time can also be reduced by moving task G from layer 3 (its node depth) to layer 2 to be merged with task E and executed in parallel with task D.
- Layer reduction:
  Not all layers contain tasks, which means that some layers can be removed. Therefore, $LayerInfo2$ is fewer than $LayerInfo1$.

VI. EXPERIMENTS

In this section, we evaluated one- and two-step ILP formulations using random task graphs and real-world Simulink models. First, a simple simulation of the random graph execution times on two or four SPEs ($N_p = 2 \text{ or } 4$) was performed. Subsequently, the state-space control and image processing models were evaluated using an FPGA emulator of the DR1000C [2].

In all the experiments, we compared the strictly defined one-step ILP and the two-step ILP formulation. Moreover, because the two-step ILP included task mapping and scheduling, we calculated the speedup of the task mapping ILP, as shown in Algorithm 2, to determine the influence of each step in the two-step ILP.

To the best of our knowledge, no task mapping or scheduling methods have been proposed for MIMDV. Hence, we implemented the Best-Fit algorithm for MIMDV to

$$min\left(\sum_{(i,k)\in D}\frac{SO^2 * \sum_j w_{ijk}}{n_i * m_k - 1} + \sum_i T_i^2 + \sum_{(i,k)\in P}\left(T_iT_k * \sum_j y_{ijk}\right)\right) \tag{20}$$

compare with our ILP formulation. First, the task graph was divided into multiple layers based on node depth. Thereafter, tasks in the same layer were sorted by execution time from large to small and assigned to SPEs individually, choosing the SPE with the longest spare time.

A. RANDOM TASK GRAPH TEST

In this experiment, random DAGs were generated using a MATLAB script. As mentioned above, our ILPs can merge and move tasks among layers. Therefore, they were robust to the granularity of tasks and the overhead of barrier wait. To prove this, the execution times of the tasks were varied from 1000 to 9000 cycles, and small/large synchronization overheads ($SO = 500\ or\ 4000$) were utilized. The main objective was to parallelize tasks when $SO = 500$, whereas that was to compromise between merging and parallelizing tasks when $SO = 4000$. The number of parallel pairs $|P|$ and dependent pairs $|D|$ were counted based on the previous definitions. Finally, ILP methods using the optimization solver CPLEX on an Intel(R) Core(TM) i9-7900X CPU @3.30GHz with up to 20 threads were applied to the task graphs.

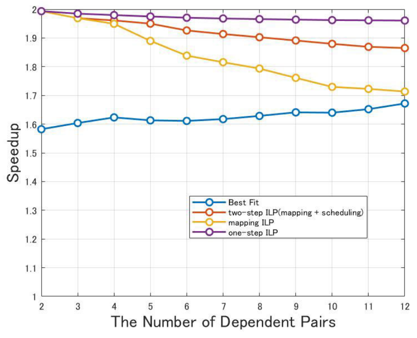
1) SMALL TASK GRAPH TEST

We performed two experiments on speedup by changing the number of dependent pairs and number of parallel pairs. The number of tasks was fixed at 12 and 200 task graphs were generated for each experiment. Each experiment generated four figures, where the number of threads and synchronization overhead were 2 and 500, 2 and 4000, 4 and 500, and 4 and 4000, respectively. In each figure, the blue line represents the Best-Fit algorithm, the red line represents the two-step ILP with task mapping and scheduling, the yellow line represents the task mapping ILP, and the purple line represents the one-step ILP.

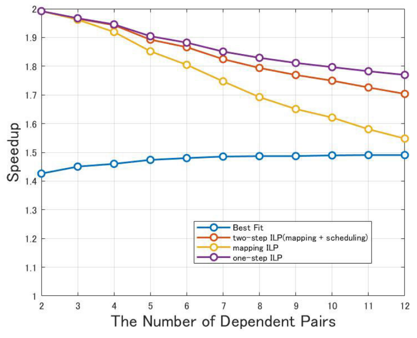
In the dependent pair experiment, the value when $The\ Number\ of\ Dependent\ Pairs = M$ was calculated as the average speedup when the x-axis ranged from $M - 3$ to $M + 3$, and Figure 18 was obtained. As we focused only on the parallel effect, the vectorization ratio was set to 0. For ILP methods, the performance decreases with an increase in the number of dependent pairs, particularly when the number of threads and synchronization overhead are larger.

In the parallel pair experiment, the value when $The\ Number\ of\ Parallel\ Pairs = N$ was calculated as the average speedup when the x-axis ranged from $N - 3$ to $N + 3$, and Figure 19 was obtained. The vectorization ratio was set to 0. The performance increases with an increase in the number of parallel pairs, particularly when the number of threads and synchronization overhead are larger.

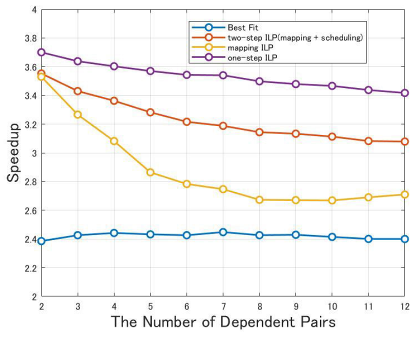
In the two experiments above, the one-step ILP determines the optimal solution. Even when using only task mapping, ILP outperforms Best-Fit because it performs optimization by merging some tasks. The two-step ILP further improves performance by optimizing the structure of multiple layers and scheduling tasks among them. In small task graphs, the



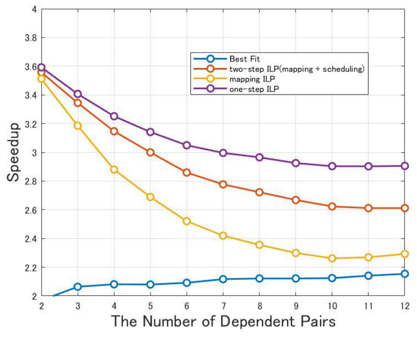
(a) Thread Number:2, Synchronization Overhead:500



(b) Thread Number:2, Synchronization Overhead:4000



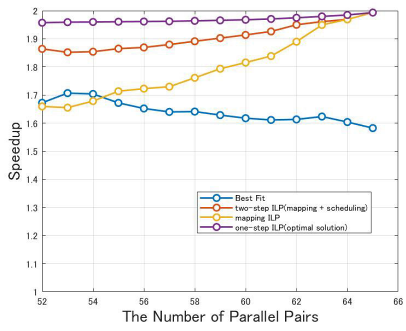
(c) Thread Number:4, Synchronization Overhead:500



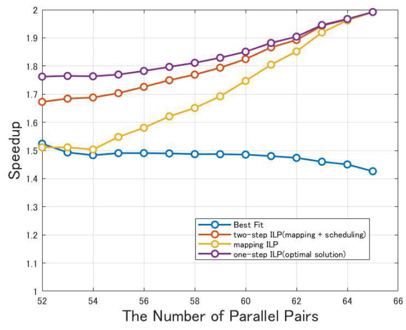
(d) Thread Number:4, Synchronization Overhead:4000

FIGURE 18. Relationship between Speedup and number of dependent pairs under different numbers of threads and barrier overheads.

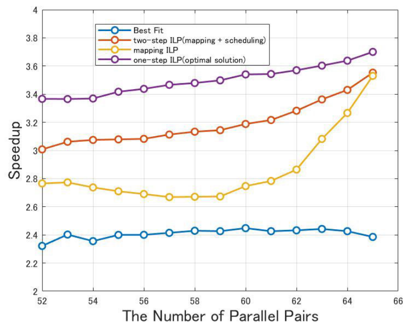
two-step ILP is not able to reach the optimal solution but is close to it when the number of dependent pairs is smaller and the number of parallel pairs increases. Best-Fit only provides



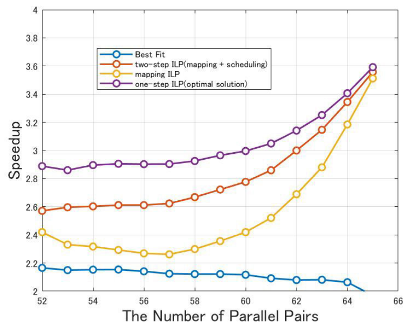
(a) Thread Number:2, Synchronization Overhead:500



(b) Thread Number:2, Synchronization Overhead:4000



(c) Thread Number:4, Synchronization Overhead:500



(d) Thread Number:4, Synchronization Overhead:4000

FIGURE 19. Relationship between speedup and number of parallel pairs under different numbers of threads and barrier overheads.

a simple layer division and performs parallelization inside each layer, which does not change significantly with changes in the dependent and parallel pairs.

TABLE 5. Average Solver Times of One-step and two-step in Different Task Numbers (Thread Number:2, Synchronization Overhead:500).

	One-step ILP	Two-Step ILP			Best Fit
		Whole	Task mapping	Scheduling	
16	6.78s	0.15s	0.04s	0.11s	<0.01s
25	300.00s	0.46s	0.11s	0.35s	<0.01s
36	300.00s	3.29s	0.13s	3.16s	<0.01s
49	—	9.80s	0.36s	9.44s	<0.01s
64	—	53.86s	2.76s	51.10s	<0.01s

2) LARGE TASK GRAPH TEST

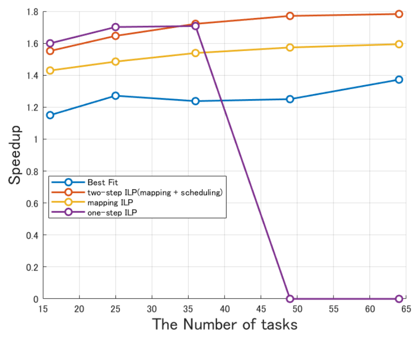
We evaluated the speedup and solver time of the one-step ILP and two-step ILP, changing the number of tasks selected from 16, 25, 36, 49, and 64. The number of threads and synchronization overhead were set to 2 and 500, 2 and 4000, 4 and 500, 4 and 4000, and 2 and 400, respectively. 60 task graphs were tested, and the average speedup and solver time were calculated for each task number. The max time limit of the CPLEX solver was set to 300s. In some cases, the one-step ILP problem was too large, and the solver cannot give any solution within time limit. In this situation, the speedup was set to 0 and the solver time did not exist. In some cases, the solver time reached the 300s time limit before the one-step ILP finds the optimal solution.

We plot speedup into Figure 20. When the number of tasks is less than 36, the two-step ILP can obtain a relatively high-quality solution which approximates or even exceeds the one-step ILP when the number of tasks increases. The two-step ILP is closer to the one-step ILP when the number of threads is higher. When the number of tasks is over 36, the range of the one-step ILP problem is too large for the solver under the experimental settings. By contrast, the two-step ILP can still find a reasonable solution, and the performance increases with the number of tasks. Best Fit increases with the number of tasks but fails to obtain great speedup.

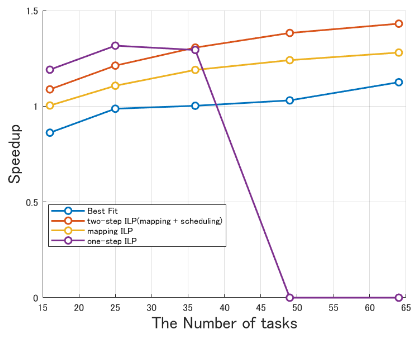
The solver times are listed in Table 5, 6, 7 and 8. In the two-step ILP, we calculate the whole average time, average task mapping and scheduling time, respectively. We can observe that using the two-step ILP can reduce the solver time dramatically compared with the one-step ILP. Two-step ILP can even solve large-scale problems that one-step ILP cannot solve. For the proportion of task mapping and scheduling time, task mapping accounts for a greater percentage when the number of threads is greater because the workload balance among different threads is more difficult. Scheduling accounts for more percentages when the number of threads is small because further adjustment among layers is more important. Best Fit solves problem in a moment but it is not a great choice considering its performance.

B. REAL-WORLD APPLICATION ON DR1000C

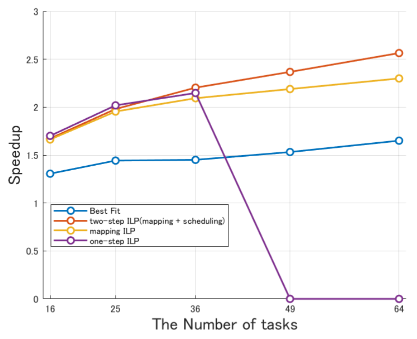
To demonstrate the effectiveness of the workflow illustrated in Figure 8, we performed experiments using two real-world Simulink models on the FPGA emulator of the DR1000C, which is an actual RISC-V MIMDV architecture processor. One model in Figure 21 used a state-space [50] controller



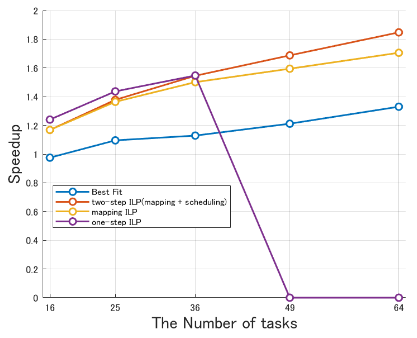
(a) Thread Number:2, Synchronization Overhead:500



(b) Thread Number:2, Synchronization Overhead:4000



(c) Thread Number:4, Synchronization Overhead:500



(d) Thread Number:4, Synchronization Overhead:4000

FIGURE 20. Relationship between Speedup and the number of tasks under different numbers of threads and barrier overheads.

TABLE 6. Average Solver Times of One-step and two-step in Different Task Numbers (Thread Number:2, Synchronization Overhead:4000).

	One-step ILP	Two-Step ILP			Best Fit
		Whole	Task mapping	Scheduling	
16	14.89s	0.15s	0.04s	0.11s	<0.01s
25	300.00s	0.46s	0.11s	0.35s	<0.01s
36	300.00s	2.82s	0.13s	2.69s	<0.01s
49	—	10.14s	0.37s	9.77s	<0.01s
64	—	44.81s	2.58s	42.24s	<0.01s

TABLE 7. Average Solver Times of One-step and two-step in Different Task Numbers (Thread Number:4, Synchronization Overhead:500).

	One-step ILP	Two-Step ILP			Best Fit
		Whole	Task mapping	Scheduling	
16	3.73s	0.09s	0.02s	0.08s	<0.01s
25	300.00s	0.33s	0.17s	0.16s	<0.01s
36	300.00s	1.97s	0.41s	1.57s	<0.01s
49	—	10.40s	1.50s	8.91s	<0.01s
64	—	46.04s	12.47s	33.57s	<0.01s

TABLE 8. Average Solver Times of One-step and two-step in Different Task Numbers (Thread Number:4, Synchronization Overhead:4000).

	One-step ILP	Two-Step ILP			Best Fit
		Whole	Task mapping	Scheduling	
16	8.69s	0.09s	0.01s	0.08s	<0.01s
25	300.00s	0.34s	0.17s	0.17s	<0.01s
36	300.00s	1.92s	0.41s	1.51s	<0.01s
49	—	9.53s	1.59s	7.94s	<0.01s
64	—	44.23s	13.28s	30.95s	<0.01s

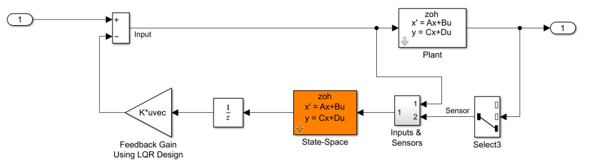


FIGURE 21. State-space control model.

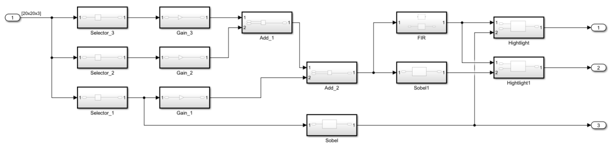


FIGURE 22. Image-processing model.

to control a state-space plant. The other model in Figure 22 used blocks in the Simulink Computer Vision Toolbox [46] to highlight the edges of a picture.

Although DR1000C had four scalar processing units (SPUs), each SPU had four hardware threads. In our experiments, we used one hardware thread per SPU to avoid resource contention. Two SPUs were utilized, indicating that the number of threads was 2. Before the experiments, we tested the overhead of the barrier wait API using two threads on DR1000C, which was considered sufficiently small to parallelize applications in fine granularity.

After converting the Simulink model into a task graph, we performed one-step ILP, two-step ILP, task mapping ILP, and Best-Fit. We measured the execution times of the following four types of programs on DR1000C. *sequentialCcode*

TABLE 9. Speedup of State-space Model using four Methods.

Metric	One-step ILP	Two-step ILP	Task Mapping ILP	Best Fit
VS	4.09	4.09	4.09	4.09
P1S	1.85	1.85	1.85	1.72
P2S	1.44	1.44	1.44	1.21
TS	5.90	5.90	5.90	4.93

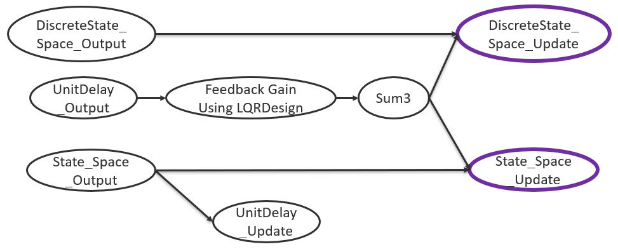


FIGURE 23. Task graph of state-space model.

was generated by the Embedded Coder, *vectorsequentialcode* was generated by replacing some matrix operation blocks with vector library blocks, *threadCcode* was generated by reconstructing fragments of the sequential C code, and *threadvectorcode* was generated by reconstructing fragments of the vector sequential code corresponding to each task based on the mapping and scheduling results.

We tested the following metrics:

Vectorization Speedup(VS): $\frac{TimeofSequentialCCode}{TimeofVectorSequentialCode}$

Parallelization Speedup1(P1E): $\frac{TimeofSequentialCCode}{TimeofThreadCCode}$

Parallelization Speedup2(P2E): $\frac{TimeofVectorSequentialCode}{TimeofTreadVectorCode}$

Total Speedup(TS): $\frac{TimeofSequentialCCode}{TimeofTreadVectorCode}$

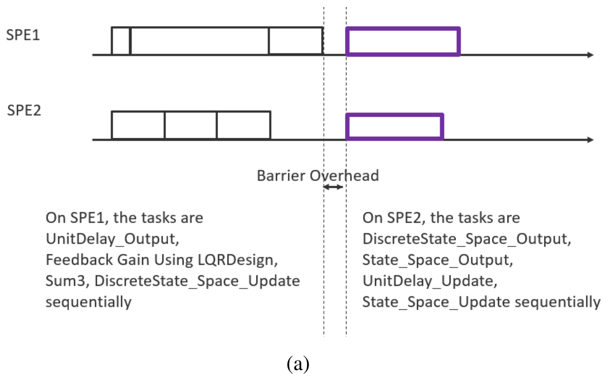
1) STATE-SPACE CONTROL MODEL

Although Figure 21 shows the control model, the two state-space blocks of the controller and plant can be replaced by vector library blocks because the differential equation is solved through matrix multiplication. The dependencies of the input of the state-space and unit-delay can be deleted to increase the possibility of parallelization because our tool can handle the output and update parts of the block, respectively.

In Table 9, three ILP methods outperform the Best-Fit, and they obtain a total speedup of 5.9x (4.1x by vectorization and 1.4x by parallelization). In the control model, although the granularity of the blocks varies and some blocks have execution times similar to the synchronization overhead, applications still benefit from using MIMDV.

As mentioned earlier, the processing of some blocks is divided into the output and update parts. Therefore, the extracted task graph differs slightly from the original Simulink diagram. To explain this, a task graph extracted from the state-space equation model using our tool is shown in Figure 23. DiscreteState_Space, State_Space, and UnitDelay are divided into two tasks. The output executes first, and the update executes later. There are 8 tasks in total.

Figure 24(a) shows that two threads parallel execution in the state-space control model using the three ILP methods



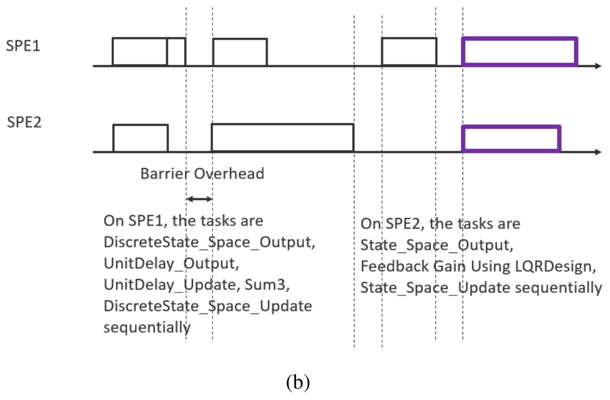


FIGURE 24. (a) Task mapping and scheduling result of two-step ILP, one-step ILP and task mapping ILP in state-space model. (b) Task mapping and scheduling result of the best-fit in state-space model.

is the same. Figure 24(b) shows two threads parallel execution using Best-Fit. All tasks with vector attributes select vectorization. In Best-Fit, the number of layers (four layers) divided by the depth of the node is large, and parallelization within the layer results in a long execution time because of the large idle time. In task mapping ILP, the load difference is reduced by merging tasks into a coarse granularity for parallelization, resulting in a shorter execution time. Optimal parallel execution can be achieved by merging tasks through task mapping, eliminating the need for task movement through further scheduling; consequently, the speedup of task mapping ILP is the same as that of two-step ILP and one-step ILP, the optimal one.

2) IMAGE PROCESSING MODEL

In Figure 22, Sobel and FIR blocks perform convolution operations on the image matrix and can achieve high efficiency when written into vector codes. Furthermore, other basic blocks, that is, Selector, Add, and Highlight (vector comparison), with large inputs can be vectorized. In addition to data parallelism, this model also includes task parallelism.

In Table. 10, our two-step ILP can reach the optimal parallel solution as a one-step ILP, outperforming the task mapping ILP and Best-Fit. The total speedup is 17.7× (10.9× by vectorization and 1.6x by parallelization). Although we assume that the size of the image is 20×20 considering the memory limitation of DR1000C, the speedup of vectorization

TABLE 10. Speedup of Image Processing Model using four Methods.

Metric	One-step ILP	Two-step ILP	Task Mapping ILP	Best Fit
VS	10.89	10.89	10.89	10.89
P1S	1.61	1.61	1.29	1.32
P2S	1.63	1.63	1.05	1.06
TS	17.74	17.74	11.53	11.54

is prominent, which means MIMDV is a good choice for dataflow applications such as image processing.

The tasks and dependencies of the image-processing model correspond to the blocks and signal lines in Figure 22. Consequently, the task graph is omitted here.

Analyzing the depth of the nodes, one Sobel block is in layer 2 and the other is in layer 5. Figure 25 shows two threads' parallel execution of the image processing model with one-step and two-step ILP, task mapping ILP, and Best-Fit, respectively. All tasks use vectorization.

In Best-Fit, the number of layers (five layers) divided by the depth of the node is large, and parallelization within each layer results in a long execution time owing to the large idle time. In task mapping ILP, tasks are merged while considering synchronization overhead; however, load balancing is not desirable in each layer because tasks are not assigned to appropriate layers. The idle times in both cases are almost the same, resulting in similar execution times. However, Best-Fit can freely parallelize within layers, whereas task mapping ILP considers a trade-off with dependencies, resulting in a slightly lower performance. Task mapping ILP and Best-Fit are unable to parallelize two Sobel blocks in the same layer. In contrast, two-step ILP, which involves scheduling, achieves more efficient parallelization by moving tasks among the layers.

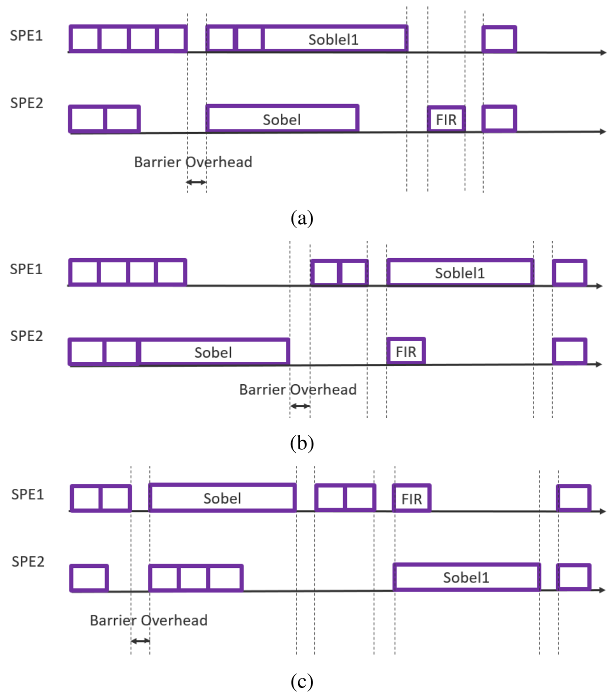


FIGURE 25. (a) Task mapping and scheduling result of the one-step and two-step ILP in image processing model. (b) Task Mapping and scheduling result of task mapping ILP in image processing model. (c) Task mapping and scheduling result of best-fit in image processing model.

VII. RELATED WORKS

For homogeneous multi-core processors, a number of methods have been proposed.

Reference [12] optimized the parallelization on multiple CPUs using mixed-ILP (MILP). References [15] and [16] generated multi-threaded codes using ILP mapping and scheduling to optimize the communication among processors and threads. An ILP formulation was proposed to find a task mapping and scheduling solution to minimize the overall throughput and latency [24]. ILP is a general algorithm that can handle many complex situations, such as data cache [27] or power consumption [28] by adjusting the objective function and constraints. In vehicle control, [29] used ILP to map tasks on multiple ECUs and schedule tasks in the slots of FlexRay.

Some heuristics are available, such as earliest finish time [30] and global fair lateness [31]. Reference [6] adapted an MBP tool to a Kalray MPPA2-256 processor with a cluster architecture. Reference [10] parallelized blocks in function level using multi-threads. Reference [13] partitioned the runnable graph of AMALTHEA model into tasks and then assigned the tasks to a multi-core platform. Reference [20] introduced a tool called MiniSIGNAL generating multi-task codes from multi-layer task graphs, in which tasks are merged considering granularity.

Various search-based algorithms are also available. Reference [33] used a genetic algorithm (GA) to map tasks on MPSoC. Reference [34] scheduled tasks using a whale optimization algorithm (WOA) in a cloud computing environment and compared it with a particle swarm optimization (PSO). Artificial intelligence-based load balancing and task scheduling combined with GA have also been proposed [35]. Reference [36] proposed answer set programming optimization in a Simulink-to-MPSoC design flow. Reference [37] proposed a scheduling algorithm using a satisfiability modulo theory solver.

These methods are difficult to apply to heterogeneous multi-core processors. For heterogeneous multi-core processors, a number of methods have also been proposed.

ILP formulations were adapted for ARM big.LITTLE architecture to reduce communication costs among different cores [25] and CPU-GPU architecture to reduce the execution time of the critical path [26]. Reference [32] modified the basic heuristic to consider the heterogeneous ratio, task graph structure, data partition, and data transfer time. Reference [38] found a solution for less memory contention and energy consumption on heterogeneous MPSoCs by deleting solutions with small fitness values and updating the solution set. Reference [7] used the tool to parallelize a PID control model with multiple rates on MPSoC with FPGA.

Reference [18] generated parallel codes for heterogeneous dataflow process networks (DPNs).

However, the program of MIMDV is a mixture of scalar and vector codes, which makes it different from modern CPU-accelerator architecture using OpenCL or CUDA. Therefore, these heterogeneous methods cannot be used directly to MIMDV, and MIMDV requires its own formulation to select between scalar and vector codes. We realized more for MBP of MIMDV to get higher performance.

Reference [8] used the MBP tool to parallelize a self-driving system using Simulink blocks for a robot operating system (ROS). It partitioned For-Iterator blocks to get higher performance. We further considered loop partition into ILP formulation. Reference [17] synthesized SIMD codes to accelerate the computing-sensitive model. We used RISC-V vector library rather than automatic generation to obtain higher performance.

For other parallelism enhancements, [9] adjusted the feedback structure and assigned tasks using an operating system. Reference [11] analyzed data dependencies in detail to enhance the parallel potential. We also realized the bottleneck of parallelism in Simulink control models and enhanced its parallelism.

Except the MATLAB/Simulink, [19] generated efficient C and Java codes using DFSynth, and actors are divided into multiple layers in schedule analysis. Reference [21] generated parallel codes from Lustre or Scade models. Reference [22] generated VHDL codes from RVC-CAL language. Reference [23] introduced a tool called SESAM/ParIVAll that generates efficient code for MPSoC.

There have also been some studies on barrier wait in multi-threaded programming, such as OpenMP, Pthread, and CUDA. Reference [39] summarized the benefits of barrier synchronization without mutual exclusion and proposed a ring algorithm to realize barrier wait. Reference [40] compared different barrier wait algorithms and used butterfly barrier with shared memory reduction to reduce latency. Reference [41] researched about barrier on ARMv8 multi-core architectures. The barrier wait was also used for GPU calculation synchronization [42], [43]. In this study, we used the barrier wait API of DR1000C whose overhead is smaller than ordinary CPUs, consequently, the generated code is effective and fit well for real-time systems.

VIII. CONCLUSION AND FUTURE WORKS

In this study, we proposed an Model-Based Parallelization (MBP) workflow for MIMD processors with vector accelerator (MIMDV) that inputs a MATLAB/Simulink model and outputs parallel code that contains multiple layers divided by barrier wait, significantly mitigating thread overhead. In the workflow, a one-step integer linear programming (ILP) was proposed to represent the parallel execution time and determine the optimal solution, and a two-step ILP was proposed to ensure both respectable performance and practical solver time. In the experiments, we tested our methods using random task graphs in computer simulations and real-world applications in FPGA emulation of DR1000C, which is an actual RISC-V MIMDV architecture processor.

Because each step in our workflow is achieved using existing tools or scripts, we will conduct research on fully automatic parallel code generation using MBP and extend it to other heterogeneous multi-core embedded systems with GPU or Sycl accelerators. The multi-threaded program can be not only DR1000C hardware threads but also other languages such as pthread. For the optimal algorithm, although we designed ILP formulations to obtain a shorter execution time for a new situation on MIMDV, the solver time can increase when the number of tasks and threads increases. Therefore, we will propose a heuristic solution to this problem and consider other elements such as resource contention.

REFERENCES

[1] RISC-V. *The Free Open RISC Instruct. Set Archit*. Accessed: Mar. 6, 2024. [Online]. Available: https://github.com/riscv

[2] NSITEXE. *DR1000C User Manual*. Accessed: Mar. 6, 2024. [Online]. Available: https://www.nsitexe.com/en/download/

[3] MathWorks. *S-Function Builder*. Accessed: Mar. 6, 2024. [Online]. Available: https://www.mathworks.com/help/simulink/s-function-builder.html

[4] C. Kenyon, "Best-fit bin-packing with random order," *SODA*, vol. 96, pp. 359–364, Jan. 1996.

[5] ESOL. *Model-Based Parallelizer for High Performance Code Generation on Multi/Manycore Systems*. Accessed: Mar. 6, 2024. [Online]. Available: https://www.esol.com/embedded/product/embp_overview.html

[6] Y. Kobayashi, K. Honda, S. Kojima, H. Fujimoto, M. Edahiro, and T. Azumi, "Mapping method usable with clustered many-core platforms for simulink model," *J. Inf. Process.*, vol. 30, pp. 141–150, Jan. 2022.

[7] R. Yamamoto, M. Oinuma, and M. Kondo, "Multirate model parallelization for MPSoC with FPGA in model-based development: A case study," in *Proc. Asia Pacific Conf. Robot IoT Syst. Develop. Platform*, 2021, pp. 29–30.

[8] R. Yoshinaka and T. Azumi, "Model-based development considering self-driving systems for many-core processors," in *Proc. 25th IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2020, pp. 337–344.

[9] T. Kumura, Y. Nakamura, N. Ishiura, "Model-based parallelization from simulink models and their sequential C code," in *Proc. 17th Workshop Synthesis System Integr. Mixed Inf. Technol.*, 2012, pp. 186–191.

[10] M. Cha, K. H. Kim, C. J. Lee, D. Ha, and B. S. Kim, "Deriving high-performance real-time multicore systems based on simulink applications," in *Proc. IEEE 9th Int. Conf. Dependable, Autonomic Secure Comput.*, Dec. 2011, pp. 267–274.

[11] Z. Han, G. Qu, B. Liu, and F. Zhang, "Exploit the data level parallelism and schedule dependent tasks on the multi-core processors," *Inf. Sci.*, vol. 585, pp. 382–394, Mar. 2022.

[12] C. E. Tuncali, G. Fainekos, and Y.-H. Lee, "Automatic parallelization of simulink models for multi-core architectures," in *Proc. IEEE IEEE 17th Int. Conf. High Perform. Comput. Commun. 7th Int. Symp. Cyberspace Saf. Secur., IEEE 12th Int. Conf. Embedded Softw. Syst.*, Aug. 2015, pp. 964–971.

[13] R. Höttger, L. Krawczyk, and B. Igel, "Model-based automotive partitioning and mapping for embedded multi-core systems," in *Proc. Int. Conf. Parallel, Distrib. Syst. Softw. Eng.*, 2015, vol. 2, no. 1, p. 888.

[14] P. Xu, M. Edahiro, and K. Masaki, "Code generation from simulink models with task and data parallelism," *IJCT*, vol. 21, pp. 1–13, Apr. 2021.

[15] K. Huang, M. Yu, X. Zhang, D. Zheng, S. Xiu, R. Yan, Z. Liu, and X. Yan, "ILP-based multi-threaded code generation for simulink models," *IEICE Trans. Inf. Syst.*, vol. 97, no. 12, pp. 3072–3082, 2014.

[16] K. Huang, M. Yu, R. Yan, X. Zhang, X. Yan, L. Brisolara, A. A. Jerraya, and J. Feng, "Communication optimizations for multithreaded code generation from simulink models," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 3, pp. 1–26, May 2015.

[17] Z. Su, Z. Yu, D. Wang, Y. Yang, Y. Jiang, R. Wang, W. Chang, and J. Sun, "HCG: Optimizing embedded code generation of simulink with SIMD instruction synthesis," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, Jul. 2022, pp. 1033–1038.

[18] O. Rafique and K. Schneider, "Synthesis of parallel software from heterogeneous dataflow models," *Social Netw. Comput. Sci.*, vol. 3, no. 3, pp. 1–34, May 2022.

[19] Z. Su, D. Wang, Y. Yang, Y. Jiang, W. Chang, L. Fang, W. Li, and J. Sun, "Code synthesis for dataflow-based embedded software design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 1, pp. 49–61, Jan. 2022.

[20] Z. Yang, S. Yuan, J.-P. Bodeveix, M. Filali, T. Wang, and Y. Zhou, "Multi-task ada code generation from synchronous dataflow programs on multi-core: Approach and industrial study," *Sci. Comput. Program.*, vol. 207, Jul. 2021, Art. no. 102644.

[21] A. Graillat, M. Moy, P. Raymond, and B. D. de Dinechin, "Parallel code generation of synchronous programs for a many-core architecture," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1139–1142.

[22] N. Siret, M. Wipliez, J.-F. Nezan, and A. Rhatay, "Hardware code generation from dataflow programs," in *Proc. Conf. Design Architectures Signal Image Process. (DASIP)*, Oct. 2010, pp. 113–120.

[23] N. Ventroux, T. Sassolas, A. Guerre, B. Creusillet, and R. Keryell, "SESAM/Par4All: A tool for joint exploration of MPSoC architectures and dynamic dataflow code generation," in *Proc. Workshop Rapid Simul. Perform. Eval., Methods Tools*, Jan. 2012, pp. 9–16.

[24] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, "An ILP formulation for task mapping and scheduling on multi-core architectures," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, Apr. 2009, pp. 33–38.

[25] Z. Zhong and M. Edahiro, "Model-based parallelizer for embedded control systems on single-ISA heterogeneous multicore processors," in *Proc. Int. SoC Design Conf. (ISOCC)*, Nov. 2018, pp. 117–118.

[26] Z. Zhong and M. Edahiro, "Model-based parallelization for simulink models on multicore CPUs and GPUs," in *Proc. Int. SoC Design Conf. (ISOCC)*, Oct. 2019, pp. 103–104.

[27] V. A. Nguyen, D. Hardy, and I. Puaut, "Cache-conscious offline real-time task scheduling for multi-core processors," in *Proc. 29th Euromicro Conf. Real-Time Syst.*, 2017, pp. 1–22.

[28] S. Yang, S. Le Nours, M. Mendez Real, and S. Pillement, "0–1 ILP-based run-time hierarchical energy optimization for heterogeneous cluster-based multi/many-core systems," *J. Syst. Archit.*, vol. 116, Jun. 2021, Art. no. 102035.

[29] G. Han, M. Di Natale, H. Zeng, X. Liu, and W. Dou, "Optimizing the implementation of real-time simulink models onto distributed automotive architectures," *J. Syst. Archit.*, vol. 59, no. 10, pp. 1115–1127, Nov. 2013.

[30] F. Lumpp, S. Aldegheri, H. D. Patel, and N. Bombieri, "Task mapping and scheduling for OpenVX applications on heterogeneous multi/many-core architectures," *IEEE Trans. Comput.*, vol. 70, no. 8, pp. 1148–1159, Aug. 2021.

[31] J. P. Erickson, J. H. Anderson, and B. C. Ward, "Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling," *Real-Time Syst.*, vol. 50, no. 1, pp. 5–47, Jan. 2014.

[32] Z. Li, Y. Zhang, A. Ding, H. Zhou, and C. Liu, "Efficient algorithms for task mapping on heterogeneous CPU/GPU platforms for fast completion time," *J. Syst. Archit.*, vol. 114, Mar. 2021, Art. no. 101936.

[33] H. M. G. de A. Rocha, A. C. S. Beck, S. M. D. M. Maia, M. E. Kreutz, and M. M. Pereira, "A routing based genetic algorithm for task mapping on MPSoC," in *Proc. 10th Brazilian Symp. Comput. Syst. Eng. (SBESC)*, Nov. 2020, pp. 1–8.

[34] R. Asif, K. A. Alam, K. M. Ko, and S. U. R. Khan, "Task scheduling in a cloud computing environment using a whale optimization algorithm," in *Proc. 1st Int. Workshop Intell. Softw. Automat.* Singapore: Springer, 2021, pp. 37–52.

[35] A. Asghari, M. K. Sohrabi, and F. Yaghmaee, "Task scheduling, resource provisioning, and load balancing on scientific workflows using parallel SARSA reinforcement learning agents and genetic algorithm," *J. Supercomput.*, vol. 77, no. 3, pp. 2800–2828, Mar. 2021.

[36] A. Cilardo, D. Socci, and N. Mazzocca, "ASP-based optimized mapping in a simulink-to-MPSoC design flow," *J. Syst. Archit.*, vol. 60, no. 1, pp. 108–118, Jan. 2014.

[37] I. Amari, A. Rebaya, K. Gasmi, and S. Hasnaoui, "An optimal scheduling algorithm for data parallel hardware architectures," in *Proc. Int. Conf. Internet Things, Embedded Syst. Commun. (IINTEC)*, Oct. 2017, pp. 111–117.

[38] H. Ali, U. U. Tariq, Y. Zheng, X. Zhai, and L. Liu, "Contention & energy-aware real-time task mapping on NoC based heterogeneous MPSoCs," *IEEE Access*, vol. 6, pp. 75110–75123, 2018.

[39] A. Aravind, "Barrier synchronization: Simplified, generalized, and solved without mutual exclusion," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 773–782.

[40] A. K. Mohamed E. Maarouf, L. Giraud, A. Guermouche, and T. Guignon, "Combining reduction with synchronization barrier on multi-core processors," *Concurrency Comput., Practice Exper.*, vol. 35, no. 1, p. e7402, 2022.

[41] W. Gao, J. Fang, C. Huang, C. Xu, and Z. Wang, "Optimizing barrier synchronization on ARMv8 many-core architectures," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2021, pp. 542–552.

[42] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang, and C. Xu, "Barrier-aware warp scheduling for throughput processors," in *Proc. Int. Conf. Supercomputing*, Jun. 2016, pp. 1–12.

[43] L. Gao, J. Wang, and W. Zhang, "Adaptive contention management for fine-grained synchronization on commodity GPUs," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 4, pp. 1–21, Dec. 2022.

[44] IBM. *Gprof Command*. Accessed: Mar. 6, 2024. [Online]. Available: https://www.ibm.com/docs/en/aix/7.2?topic=g-gprof-command

[45] S. Wu, S. Kumano, K. Marume, and M. Edahiro, "An ILP task mapping for MIMD processor with vector accelerator in model-based development," in *Proc. Int. Conf. Electr., Comput. Energy Technol. (ICECET)*, Jul. 2022, pp. 1–7.

[46] *MathWorks Computer Vision Toolbox*. Accessed: Mar. 6, 2024. [Online]. Available: https://www.mathworks.com/products/computer-vision.html

[47] NSITEXE. *NSITEXE, OTSL, Kyoto Microcomputer, AXELL, Collaborate To Develop RISC-V Based Reliable Edge AI Platform*. Accessed: Mar. 6, 2024. [Online]. Available: https://www.nsitexe.com/en/news/archives/20220830/

[48] K. Shimada, I. Taniguchi, and H. Tomiyama, "Communication-aware scheduling of data-parallel tasks on multicore architectures," *IPSJ Trans. Syst. LSI Design Methodol.*, vol. 12, pp. 65–73, 2019.

[49] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst.*, Oct. 2006, pp. 401–410.

[50] MathWorks. *State-Space*. Accessed: Mar. 6, 2024. [Online]. Available: https://www.mathworks.com/help/simulink/slref/statespace.html

[51] Intel. *Intel Intrinsics Guide*. Accessed: Mar. 6, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[52] Arm Developer. *Intrinsics*. Accessed: Mar. 6, 2024. [Online]. Available: https://developer.arm.com/architectures/instruction-sets/intrinsics/

[53] MathWorks. *Embedded Coder*. Accessed: Mar. 6, 2024. [Online]. Available: https://ww2.mathworks.cn/products/embedded-coder.html?requestedDomain=zh

[54] GCC. *GCC, GNU Compiler Collection*. Accessed: Mar. 6, 2024. [Online]. Available: https://gcc.gnu.org/

[55] Clang. *Clang: A C Language Family Frontend for LLVM*. Accessed: Mar. 6, 2024. [Online]. Available: https://clang.llvm.org/

[56] OpenCL. *Open Standard for Parallel Programming of Heterogeneous Systems*. Accessed: Mar. 6, 2024. [Online]. Available: https://www.khronos.org/opencl/

[57] CUDA Toolkit. *Develop, Optimize and Deploy GPU-Accelerated Apps*. Accessed: Mar. 6, 2024. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

SHANWEN WU received the master's degree in information engineering from Xi'an Jiaotong University, in 2020, and the master's degree in informatics from the Graduate School of Informatics, Nagoya University, in 2023, where he is currently pursuing the Ph.D. degree.

He is also supported by "Nagoya University Interdisciplinary Frontier Fellowship." His research interest includes the optimization of software for multi-many-core systems.

SATOSHI KUMANO is currently a Project Manager of Development Platform Section, Semiconductor IP Research and Development Unit, NSITEXE Inc. He joined NEC Corporation, in 2002, and he engaged in software development for mobile and automotive SoCs. He moved to TOSHIBA Corporation, in 2015, and moved to NSITEXE Inc., in 2020. He engaged in software development platform for RISC-V based domain-specific processors.

KEI MARUME is currently a Manager of Development Platform Section, Semiconductor IP Research and Development Unit, NSITEXE Inc. He joined Toshiba Corporation, in 1999, and he engaged in SoC Hardware Design, a System Engineer and a Functional Safety Manager of ADAS SoCs. He moved to DENSO Corporation, in 2017, and engaged in functional safety expert and software development platform for RISC-V based domain-specific processors.

MASATO EDAHIRO (Member, IEEE) received the bachelor's and master's degrees in mathematical engineering from The University of Tokyo, in 1983 and 1985, respectively, and the master's and Ph.D. degrees in computer science from Princeton University, in 1993 and 1999, respectively.

He joined NEC Corporation, in 1985, and moved to Nagoya University, in 2011, where he is currently a Professor with the Department of Computing and Software Systems, Graduate School of Informatics. His research interests include graph algorithms and software for multi-many-core systems. He is a member of IPSJ, IEICE, and ORSJ.