Model Based Parallelization from the Simulink Models and Their Sequential C Code

Takahiro KUMURA†1†2, **Yuichi NAKAMURA**†2, **Nagisa ISHIURA**†3,
Yoshinori TAKEUCHI†1, **Masaharu IMAI**†1
†1 Osaka University, Yamadaoka 1-5, Suita City, Osaka, 565-0871, Japan
†2 NEC Corporation, Shimonumabe 1753, Nakahara-ku, Kawasaki City, 216-8666, Japan
†3 Kwansei Gakuin University, 2-1 Gakuen, Sanda City, Hyogo, 669-1337, Japan

Abstract— This paper proposes a method to generate parallel C codes suited to pipeline processing from schematic models developed on the Simulink, which is a model-based development tool. The Simulink is widely used in the field of control systems for ranging from algorithm development to code generation for embedded systems. Although there are several researches which focus on parallelization based on Simulink models, they exploit parallelism mainly within one step processing of the models, or among mutiple-step processing by ignoring inter-step data dependencies. Here, one step processing means that a model processes an input signal and calculate an output signal. In order to exploit more parallelism among multiple-step processing while preserving the original semantics of the model, this paper focuses on a pipeline processing based on a way of applying the theory of communicating sequential processes (CSP). Under the parallelization process, the proposed method eliminates loop structures in models and builds directed acyclic graphs (DAGs) suited to a pipeline processing. While data items are transferred through communication on the CSP, they are stored and shared in double buffers on the proposed method. On the experiment of applying the method for an audio processing model, the execution time of the parallelized code could be reduced successfully to 26.3% on a 4-core processor running at 400MHz with a symmetric multi-processing real-time operating system, compared with that of the sequential code.

I. Introduction

Multicore processors have been becoming popular to increase their performance in the field of PCs and embedded systems. For example, communication infrastructures such as mobile base stations work on heterogeneous and/or homogeneous multicore processors to handle radio signal and packets of many users. Furthermore, multicore processors are also used for encoding, decoding, and image processing on high definition televisions, and are evaluated for car navigation systems. Behind this trend focused on multicore, there is a story that multicore is becoming one of key technologies to increase processors' performance since increasing processors' operating frequency is more and more difficult and requires much power. This movement toward multicore is remarkable in the field of communication infrastructures, which require higher processing performance than in other fields.

To exploit the inherent performance of multicore processors, it is very important to parallelize software working on them. However, parallelization makes it difficult to develop software since parallelization requires adequate workload balancing and access controlling of shared resources [3]. To ease parallelizing software, so far there have been many researches of parallel languages, frameworks, and compilers, etc. [1, 2]. Recent examples are GPU-oriented frameworks such as CUDA and OpenCL to exploit data parallelism in which same calculations are performed for many pixels, and the Intel C/C++ compiler that has features such as automatic loop parallelization and automatic vectorization.

To take advantage of task parallelism, which is another aspect for parallelism, parallelization methods based on dataflow and pipeline processing have also been actively researched [4, 6]. For example, StreamIt [4] is a research project on a source-to-source compiler for stream processing, which handles continuous data streams such as audio and video signals. Programmers describe pipelines in the language of the StreamIt for dataflow of stream processing, then the StreamIt compiler generates source codes that make every stage of the pipelines work in parallel. For another example, Molatomium [6] is a research project that also focuses on dataflow.

While these novel frameworks based on dataflow and pipeline processing are very useful for parallelizing software developed from scratch, they have a problem in terms of translation from conventional languages. To cure this problem, we focus on the Simulink, a model development tool, which is widely used in a model-based development method spreading recently in the field of control systems. Models developed on the Simulink are widely used in the field of control systems for raging from algorithm development to code generation for embedded systems. Simulink models are just dataflow graphs themselves, and are suited to parallelization since they represent structural

parallelism visually in them. If parallel software can be generated from existing models developed on the popular Simulink, language translation by hand should never be needed to make software parallel.

There are several activities by the MathWorks and others in terms of Simulink models and parallelization [7–11]. The MathWorks, a developer of the Simulink, provides Parallel Computing Toolbox (PCT) as his own product before, which allows programmers to run large-scale models for simulation in parallel. On the latest version of the C code generation tool from the Simulink models, internal processing of individual blocks such as FFT and filters is parallelized. Although there have been several other researches and products for generating parallel code from Simulink models [9–11], programmers on these researches and products have to take charge of making tasks and allocating tasks to CPUs. One of conventional research extracts parallelism only from one-step processing of models [7]. Here, one step processing means that a model processes an input signal and calculate an output signal. Another research extracts parallelism among multiple-step processing by ignoring inter-step data dependencies, which may cause acceptable numerical errors [8]. The first one has an issue of less parallelism, and the second one has an issue of trade-off between obtained parallelism and numerical error.

This paper proposes a method to generate parallel C code from the Simulink models, in which we aim to extract parallelism from multiple-step processing without ignoring inter-step dependencies. Specifically, the method transforms a model having feedback loops to a directed acyclic graph (DAG) while preserving the original semantics of the model, and executes every node of the DAG concurrently on the basis of the theory of communicating sequential processes (CSP) [12]. While the theory of CSP transfers data items through communication, which may cause data copies from buffer to buffer, the proposed method shares data items by using double buffers placed at shared memories. The contribution of this work is the CSP-based method using double buffering in combination with the loop structure decomposition for pipeline processing, and the evaluation of the proposed method. In the rest of this paper, the proposed method is described, and then parallelization experiment is explained. Finally, this paper shows that as a result of parallelizing an audio equalizer model on a 4-core processor running at 400MHz the proposed method reduces execution time down to 26.3% through parallelization.

II. Parallel C code generation from Simulink models

This paper proposes a method to generate parallel C code from Simulink models on the basis of the theory of CSP and regarding blocks in Simulink models as tasks and lines between blocks in Simulink models as communication channels. Figure 1 shows a process to generate parallel C code from Simulink models. Underling concepts of the proposed method are described below.

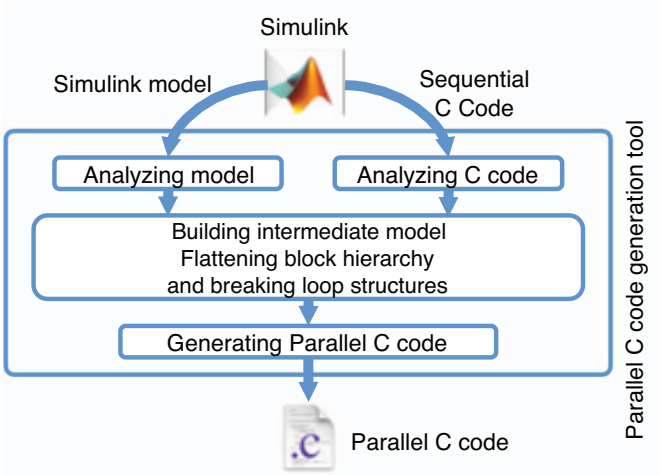


Fig. 1. The process to generate parallel C code from a Simulink model.

Mapping a block in a model to a task: The proposed method regards the processing of a block in a Simulink model as a task, and a dataflow graph represented with blocks and lines of a Simulink model as a data dependency graph of tasks, respectively. Then behavior of each task is retrieved from the sequential C code generated from a Simulink model by the Real-Time Workshop, which is a Simulink component.

Signaling completion of tasks with synchronized task communication: In the theory of CSP, processes running concurrently communicate via synchronized message passing. The proposed method makes the tasks run concurrently while communicating based on the data dependency graph extracted from a Simulink model. The proposed method uses synchronized task communication as a means to notify completion of tasks to each other. An event of completion of task calculation is transmitted to other tasks via synchronized task communication, and the tasks that receive the event begin their own calculation. During synchronized task communication, tasks wait until the communication partners become ready. Figure 2 shows the task execution flow. Each task repeats the following steps: (1) receiving events to know completion of ascending tasks that feed input data to this task, (2) calculating output data, and (3) sending events to notify that output data is ready to descending tasks that uses the output data. Tasks communicate using application programming interfaces (APIs) for handling semaphores, events, or messages available on a target platform. These APIs may cause context switches and another task will become ready to run if synchrnous communication between tasks is suspended because one of tasks is not ready.

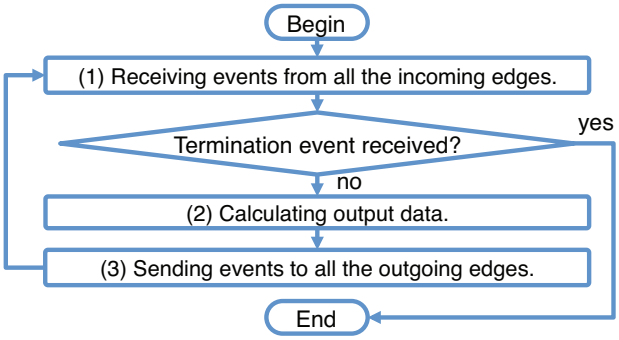


Fig. 2. The flow chart of task execution.

Pipeline parallel processing using double buffers for task output data: On the proposed method, tasks obtain the calculation results of other tasks from shared memories but not from task communication. Although the theory of CSP transfers the calculation results of tasks through task communication, task communication is not adequate for large data transfer since it is necessary to transfer data from the buffer storing the calculation results to the buffer for task communication. The data copies from buffer to buffer could be a problem when large data items are transferred through task communication. To avoid the data copies from buffer to buffer, the proposed method transfers a buffer index via task communication, and shares data items among tasks via double buffers storing calculation results of tasks. The buffer index transferred from task to task represents which buffer to be used. When a large amount of data of a task is fed to multiple descending tasks, the proposed method write the output data to shared memory one time, while the original CSP requires data copy the number of descending tasks. Using double buffers to store task outputs and switching the buffers alternately allow data-producer tasks and data-consumer tasks to work in parallel simultaneously. In this way, costly data copies between buffers can become needless in the proposed method.

Although Simulink can work with various models, the proposed method handles only models subjected to the following restrictions: fixed time step, discrete-time solver, and single signal rate. These restrictions make analyzing C code simple, and are acceptable ones for developing practical Simulink models.

III. Analyzing C code

The proposed method analyzes two files generated from the Real-Time Workshop: *model*`.c` and *model*`.h`. The file of *model*`.c` contains the following three functions:

model`_step()`
executing a single step of the target model.
model`_initialize()`
initializing the internal status of the target model.
model`_terminate()`
terminating the processing of the target model.

In these functions, the function to be parallelized is *model*`_step()`. The proposed method breaks the content of the function of *model*`_step()` into code blocks separated by comment tags that represent which code block comes from which Simulink block. The comment tags are inserted by the Real-Time Workshop for code tracing between Simulink models and C code, and make it possible to find one-to-one relationship between code blocks and model blocks. In the file of *model*`.c`, all variables and arrays to store output data of model blocks are put together into a single structure. Each of these variables and arrays are read also as input data. The aggregated structure makes it easy to duplicate the structure for double buffering. The header file of *model*`.h` contains input and output data structures of model blocks and a list of subsystem names included in the target model. A subsystem is a group of blocks, and is one level of hierarchy in models.

IV. Analyzing model

The proposed method reads the model file *model*`.mdl`, analyzes it, and finds its hierarchical structure and connection of blocks. Simulink models have a hierarchy of blocks. A subsystem, which is a group of blocks, forms one level of hierarchy and can contains other subsystems. In the files of Simulink models, information on block connection is recorded individually for every level of block hierarchy. In order to generate parallel C code, it is necessary to get block connection information between different levels of block hierarchy.

V. Flattening block hierarchy

The proposed method builds an intermediate model, which becomes a task dependency graph to generate parallel C code. In the building process of an intermediate model, to take advantage of the inherent structural parallelism represented in Simulink models that have hierarchical structures, the proposed method flattens the hierarchical structure of model blocks. Target model blocks for this flattening are subsystem blocks that do not have any binding to code blocks extracted from the sequential C code. After the flattening, input and output port blocks included in such subsystem blocks are connected to external model blocks outside the subsystem blocks.

VI. Breaking loop structures

Several Simulink models have loop structures. Here, a loop structure means a feedback loop but neither a for-

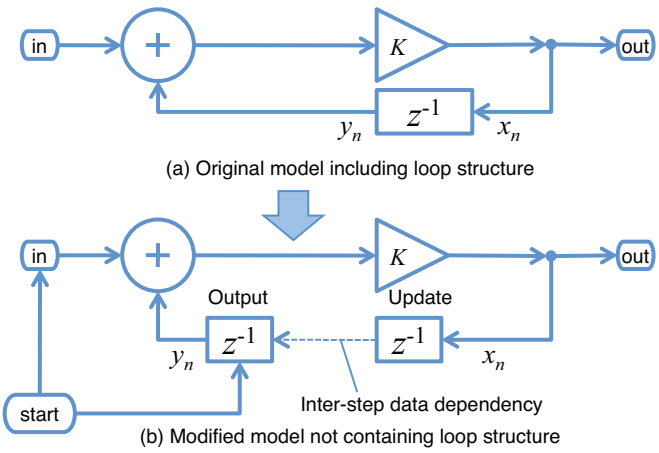


Fig. 3. How to break a loop structure.

loop nor a while-loop. Loop structures prevent the proposed method from determining the execution order of model blocks in the intermediate models, and make parallelization difficult. To cure this problem, the proposed method breaks loop structures that exist in the intermediate models by dividing indirect-feedthrough blocks such as delay element blocks and integral blocks in loop structures while preserving the original semantics of Simulink models. Output data y_n and internal status in an indirect-feedthrough block are calculated as follows:

$$\text{Output:} \quad y_n = f(status_{n-1}) \tag{1}$$

$$\text{Update:} \quad status_n = g(x_n) \tag{2}$$

In a usual design guideline of Simulink models, there must be indirect-feedthrough blocks somewhere in usual loop structures of Simulink models. Taking advantage of the fact that indirect-feedthrough blocks can calculate output data y_n by using only internal status without using their input data x_n, as shown in Figure 3, an indirect-feedthrough block can be divided into two blocks: an output calculation block and a status update block. After this process of breaking loop structures, a DAG in terms of tasks and their data dependency is obtained from the modified intermediate model. The specific steps for dividing an indirect-feedthrough block are as follows.

(1) Dividing a delay element block into two blocks: The proposed method finds indirect-feedthrough blocks, and divides such a delay element block into an output calculation block and a status update block. The status update block takes over the incoming edges of the indirect-feedthrough block, and the output calculation block takes over the outgoing edges of the indirect-feedthrough block.

(2) Adding an edge from the start block to an output calculation block: The proposed method creates the start block that starts the processing of the target model, adds an edge from the start block to the output calculation block. The edge added here means that the output calculation block does not have any data dependency from the other model blocks and it can calculate its output data using its own internal status.

(3) Adding an inter-step data dependency edge from a status update block to an output calculation block: Since the internal status updated at the current time step in the status update block will be used in the output calculation block at the future time step, there is a data dependency from the status update block at time step n to the output calculation block at time step $n+1$. This dependency is one between different time steps, and this paper calls such a dependency as an inter-step data dependency. The proposed method adds an edge for this inter-step data dependency from the status update block to the output calculation block. The inter-step data dependency edge is used only for determining if a model block becomes ready for execution at run time but not for determining the execution order of model blocks before run time. The modified intermediate model in this way becomes a directed acyclic graph by ignoring edges of inter-step data dependency.

VII. Generating Parallel C code

After building an intermediate model for the target model, the proposed method generates parallel C code based on the intermediate model. Nodes and edges in the intermediate model are translated to tasks and task communications, respectively, and the generated parallel code includes the following three functions. These functions correspond to ones in the sequential C code generated from the target Simulink model.

model`_step_parallel()`
model`_initialize_parallel()`
model`_terminate_parallel()`

These functions are to be invoked from the task corresponding to the start block. The function of *model*`_initialize_parallel()` starts up all the tasks for the target model. The function of *model*`_step_parallel()` sends a calculation completion event to the tasks corresponding to the model blocks that are connected to the start block, and then returns without waiting for the completion of the tasks. Tasks receiving the event sent from the start task calculate their output data, and send another event to their descending tasks. The function of *model*`_terminate_parallel()` sends a terminate event to running tasks and returns. Tasks receiving a terminate event immediately come to a halt.

The proposed method does not have any rules in terms of (a) task assignment onto CPUs and (b) task scheduling, and those two features are up to the operating systems. In this paper, (a) and (b) are provided by a symmetric multi-processing (SMP) operating system.

TABLE I
EXPERIMENT RESULTS OF GENERATING PARALLEL C CODE FROM SIMULINK MODELS.

	Model blocks	Tasks	Execution time ratio compared with sequential programs.		
			PC0	PC1	RTOS
Audio equalizing	252	57	85.8%	38.3%	26.3%
Lane detection	302	64	94.9%	44.3%	39%

MATLAB/Simulink version: R2010b
Experiment environment: PC0 and **PC1**
OS: Windows Server 2008
processor: Xeon@1.83GHz (4 cores)
compiler: Intel C++ Compiler XE 12.1
compiler options: `/O2 /Qparallel`
task communication: Win32 API
PC0 does not use the proposed method.
PC1 use the proposed method without compiler option /Qparallel.
∗ Scores of PC0 and PC1 are the execution time ratio against PC0 without option /Qparallel.
Experiment environment: RTOS
OS: eSOL eT-Kernel Multi-Core Edition (SMP) [15]
processor: NaviEngine ARM11 MPCore@400MHz (4 cores)
compiler: ARM RealView Compiler 3.0
compiler options: `-g -O3`
task communication: message buffers (no queues, no time out)

VIII. EXPERIMENT

To investigate how effective the proposed method works, in this section, experiment results are described for generating parallel C code from two Simulink models [13,14]. The target environments to execute generated C code are a PC running the Windows operating system and an embedded system running a real time operating system (RTOS). The details of the environments and results of the experiment are shown in Table I.

A. Audio equalizing

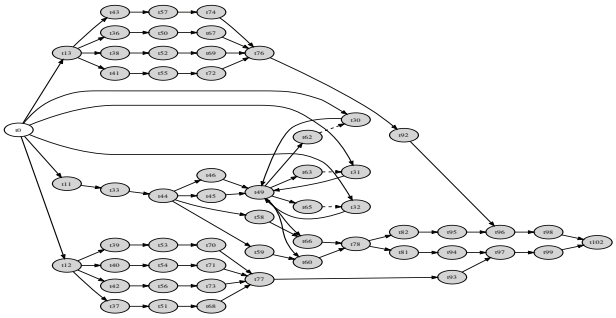
Audio equalizing in the model of [13] is an audio processing that reads audio signals from a file and modifies their waveform in both time and frequency domains. The audio signals are stereo and 16 bits/sample, and their sampling rate is 44.1kHz. Each of signals is represented as a 32-bit floating-point number. The audio equalizing model used in this experiment performs audio processing for a 1024-sample audio frame at a time.

Figure 4 shows a task dependency graph extracted from the audio equalizing Simulink model. Since the PC used in this experiment has 4 CPUs, the execution time of parallelized programs could be ideally decreased down to 1/4=25%. For the experiment on the Windows PC, the execution time of the parallelized program using the proposed method is reduced to 38.3% compared with a sequential program before parallelization while the execution time ratio of the program parallelized automatically by the Intel Compiler is 94.9%. For the experiment on the RTOS system, the execution time of the program parallelized by the proposed method is reduced to 26.3%, which is very close to the ideal case.

B. Lane detection

Lane detection in the model of [14] is an image processing that detects car lanes from roadway images captured by a camera in a car, tracks the detected lanes, and makes alert messages if the car is going to be out of the lanes. The size of the roadway images is 360 x 240 pixels. Hough transformation is used for detecting car lanes, and kalman filtering is used for tracking the detected lanes. Pixels in the image processing of this lane detection are represented as 32-bit floating-point numbers.

Figure 5 shows a task dependency graph extracted from the lane detection Simulink model. For the experiment on the Windows PC, the execution time of the parallelized program using the proposed method is reduced to 35% compared with a sequential program before parallelization, while the execution time ratio of the program parallelized automatically by the Intel Compiler is 85.8%. For the experiment on the RTOS system, the execution time of it has been reduced to 39%. On both of the PC and the embedded system that does not have any display device, any of result images were not displayed for the sake of profiling on different systems under the similar conditions.



- Node tN represents a task.
- Node t0 is a main task, which corresponds to the beginning block and invokes such functions like *model*`_step_parallel()`.
- Edges between nodes represent data dependencies of tasks.
- Dashed edges between nodes represent inter-step data dependencies of tasks, which are the dependencies between different time steps.

Fig. 4. Task dependency graph extracted from the audio equalizing model.

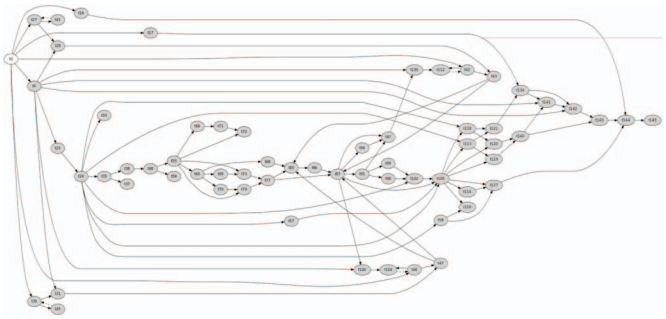
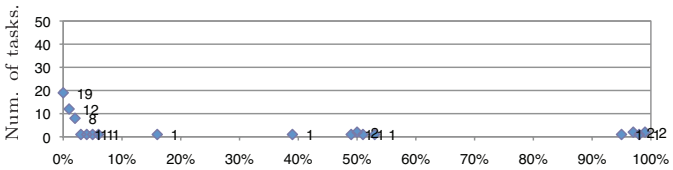
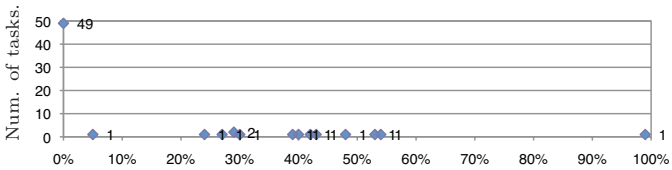


Fig. 5. Task dependency graph extracted from the lane detection model.



Task execution time ratio over the maximum task execution time.

Fig. 6. Histogram of task execution time ratio for the audio equalizing.



Task execution time ratio over the maximum task execution time.

Fig. 7. Histogram of task execution time ratio for the lane detection.

IX. Discussion

As shown in the experimental results in Table I, the proposed method has worked effectively for the audio equalizing model, for which the execution time was reduced to 26.3% on the RTOS system. This result is very close to the ideal case for a 4-core processor. For the lane detection model, on the other hand, the execution time was reduced to 39% on the RTOS system. This section discusses this result.

One problem comes from a structure of task dependency graph. The task dependency graph of the audio equalizing model shown in Figure 4 has a simpler straightforward structure than that of the lane detection model shown in Figure 5. This difference of task dependency graphs in terms of their structure results in the difference of performance in Table I. In Figure 5, there are more long edges than in Figure 4. If there is a long edge between two nodes A and B and there is another path from node A to node B via other nodes, the nodes on the another path are difficult to be parallelized in a pipeline manner because node A cannot finish its task until node B starts its task. Remember each node works in the way shown in Figure 2.

Another problem is task granularity. Figures 6 and 7 show distribution of task execution time ratios for the audio equalizing and the lane detection, respectively. In these figures, a task execution time ratio over the maximum task execution time for the two models respectively is calculated for every task, and the number of tasks within every 1% segment of the ratio is plotted.

While the number of tasks that have lower task execution time ratio than 1% is 19 in the audio equalizing, the number of such tasks is 49 in the lane detection, as can be seen in Figures 6 and 7. The tasks that have lower task execution time ratio than 1% does not practically perform any calculation and does consume task communication overhead. This means that the parallelized lane detection has more workloads in terms of task communication than does the audio equalizing, and it leads to the lower execution time reduction of 39%.

X. Conclusion

This paper proposes a method to generate parallel C code from models developed by the Simulink, which is a model development tool. The proposed method regards blocks in Simulink models as tasks and lines between blocks in Simulink models as communication channels respectively, and then generates parallel C code on the basis of the theory of communicating sequential processes (CSP). Under the process of parallelization, the proposed method breaks loop structures in Simulink models for parallelization while preserving the original semantics of the model. While the theory of CSP transfers data items through communication, the proposed method shares data items by using double buffers placed at shared memories. As a result of parallelizing an audio equalizer model on a four-core processor running at 400MHz the proposed method has reduced execution time down to 26.3% through parallelization.

References

[1] Maurice Herlihy, and Nir Shavit, "The art of multiprocessor programming," Morgan Kaufmann, March 2008.

[2] Takamichi Miyamoto, Saori Asaka, Hiroki Mikami, Masayoshi Mase, Yasutaka Wada, Hirofumi Nakano, Keiji Kimura, and Hironori Kasahara, "Parallelization with automatic parallelizing compiler generating consumer electronics multicore API," IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 600-607, Dec. 2008.

[3] Edward A. Lee, "The problem with threads," Computer, Vol. 39, Issue 5, May 2006.

[4] Michael I. Gordon, William Thies, Saman Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 21-25, 2006.

[5] William Thies, Michal Karczmarek, and Saman P. Amarasinghe, "StreamIt: a language for streaming applications," Proceedings of International Conference on Compiler Construction, p.179-196, April 08-12, 2002.

[6] Motohiro Takayama, Ryuji Sakai, Nobuhiro Kato, and Tomofumi Shimada, "Molatomium: parallel programming model in practice," USENIX Workshop on Hot Topics in Parallelism, June 2010.

[7] Arquimedes Canedo, Takeo Yoshizawa, and Hideaki Komatsu, "Automatic parallelization of simulink applications," Proceedings of International Symposium on Code Generation and Optimization (CGO), pp. 151-159, 2010.

[8] Arquimedes Canedo, Takeo Yoshizawa, and Hideaki Komatsu, "Skewed pipelining for parallel simulink simulations," Proceedings of DATE 2010, pp. 891-896, 2010.

[9] Sang-il Han, Xavier Guerin, Soo-Ik Chae, and Ahmed. A. Jerraya, "Buffer memory optimization for video codec application modeled in simulink," Proceedings of Design Automation Conference (DAC), July 24-28, 2006.

[10] Lisane Brisolara, Sang-il Han, Xavier Guerin, Luigi Carro, Ricardo Reis, Soo-Ik Chae, and Ahmed Jerraya, "Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC," Proceedings of International Workshop on Software and Compilers for Embedded Systems (SCOPES), pp. 81-89, 2007.

[11] dSPACE, " Real-time interface for multiprocessor systems (RTI-MP)," http://www.dspace.jp/en/pub/home/products/sw/impsw/rtimpblo.cfm .

[12] Charles Antony Richard Hoare, "Communicating sequential processes," Prentice Hall, ISBN 0-13-153289-8, April 1985.

[13] Younes Seyedi, "Professional Simulink audio equalizer," available at http://www.mathworks.com/matlabcentral/fileexchange/ .

[14] The MathWorks, "Lane departure warning system," `vipldw_all.mdl`, which is a sample model distributed with Video and Image Processing Toolbox for MATLAB/Simulink.

[15] Masaki Gondo, "Blending asymmetric and symmetric multiprocessing with a single OS on ARM11 MPCore," eSOL Co., Ltd. white paper, Information Quarterly Vol. 6, Num. 2, 2007, available at http://www.esol.co.jp/english/embedded/pdf/esol_Multicore_whitepaper.pdf .