Skewed Pipelining for Parallel Simulink Simulations

Arquimedes Canedo, Takeo Yoshizawa, Hideaki Komatsu
IBM Research - Tokyo

Abstract—Modern automotive and aerospace embedded applications require very high-performance simulations that are able to produce new values every microsecond. Simulations must now rely on scalable performance of multi-core systems rather than faster clock frequencies. Novel parallelization techniques are needed to satisfy the industrial simulation demands that are essential for the development of safety-critical systems. Simulink formalism is the industrial de facto standard, but current state-of-the-art simulation and code generation techniques fail to fully exploit the parallelism in modern multi-core systems. However, closed-loop and dynamic system simulations are very difficult to parallelize because of the loop-carried dependencies. In this paper we introduce a novel skewed pipelining technique that overcomes these difficulties and allows loop-carried Simulink applications to be executed concurrently in multi-core systems. By delaying the forwarding of values for a few iterations, we can break some data dependencies and coarsen the granularity of programs. This improves the concurrency and reduces the high cost of inter-processor communication. Implementation studies to demonstrate the viability of our method on a commodity multi-core system with 2, 3, and 4 processors show a 1.72, 2.38, and 3.33 fold speedup over uniprocessor execution.

I. INTRODUCTION

The development of embedded systems is a complex engineering task that has to be completed within tight cost, time, performance, and reliability constraints. To meet all of these requirements, Model-Based Design tools are heavily used by the industry. Modeling, simulation, verification, and deployment can be done efficiently and certified software for safety-critical applications can be produced. Automotive embedded electronics was a 127-billion-Euro market in 2002 that is expected to grow to 316 billion Euros by 2015 [1]. Simulink formalism is the de facto standard for the development of automotive and aerospace embedded controllers. Typically, the dynamic system under control (the plant), and the controller are modeled, simulated, and refined under Simulink [2]. The next step consists of a software-in-the-loop simulation (SILS) in which the controller and a real-time version of the plant are simulated to verify real-time behavior. Finally, a hardware-in-the-loop simulation (HILS) couples the actual embedded microcontroller with a real-time version of the plant for a real-time simulation. Clearly, generating real-time C programs with response times of microseconds is a very challenging problem. Although commercial code generators can be used [3], [4], they are focused on generating production-quality code and sometimes fail to exploit all the performance available in modern processors. In this paper we introduce a technique that exploits modern multi-core systems to produce faster C programs that target high-performance real-time simulations of dynamic systems.

Chip multiprocessors, also known as multi-core chips, are the current trend in microprocessor design [5]. Physical limits of silicon have been hit and clock speeds are no longer the main source of performance improvements. Instead, multiple processors are laid out in the same chip and using some sort of communication mechanism these designs offer scalable parallelism without increasing the clock speed. Exploiting this kind of parallelism, however, requires novel techniques in the compilers, programming languages, and runtime tools [6], [7]. One of the biggest challenges is obtaining coarse grain computation units (threads) that can hide the still high communication cost. For many applications, organizing computations into threads is very difficult or sometimes impossible. Simulations of dynamic systems very often fall in this category. Closed-loop control systems are tightly coupled subsystems interacting with each other through feedback signals. These feedback signals are represented in programs by loop-carried dependencies that conventional parallelization methods are unable to handle. Without careful code generation, the resulting multi-core application may even be slower than the same application executed in a single processor. The two factors working against the parallelization of these applications are excessive inter-processor communication cost and loop-carried dependencies.

This paper introduces a novel method for transforming Simulink models into concurrent threads that can exploit a form of pipeline parallelism in multi-core systems. Skewed pipelining reduces the communication overhead by delaying some of the data dependencies for a few iterations while concurrently executing independent fragments of the program. We benefit from the very slow rate of change of the states driving a dynamic system simulation to delay the updates of the inputs to some blocks. This allows fine-grain programs to be coarsened for faster simulations in multi-core systems. We implemented a compiler that translates Simulink models into multi-thread C code. Our compiler removes inter-iteration dependency edges from the dataflow graph and builds a concurrent program dependence graph (CPDG). The main contributions of this work are: (1) a new skewed pipelining technique that coarsens the granularity of threads by delaying communication for a few iterations, (2) an automatic compiler transformation that selectively breaks some of the data dependencies in Simulink models and allows tightly coupled applications to be parallelized, and (3) a new Simulink block operator, the Skew block, that gives designers explicit control over parallel simulations in multi-core systems.

The rest of the paper is organized as follows. Section II discusses related work. Section III introduces the Simulink
semantics. Section IV presents our automatic transformation and pipelining technique. Section V shows some experimental results. Section VI concludes and considers future directions.

II. RELATED WORK

In [8], a method to compile Simulink models into multi-threaded code that optimizes communication by using a Message Aggregation technique is proposed. This technique merges individual messages between two processors into a single larger message to reduce communication overhead. While this is useful for data-intensive applications with many communication channels such as MPEG decoder, control-based applications use only a few communication channels whose performance is affected by the frequency of communication. This paper proposes a novel optimization technique suitable for control-based applications that delays communication across several iterations to reduce the communication cost. Unlike [8], we target applications with loop-carried dependencies.

Pipelining is a well known technique that has found new applications in multi-core systems [9]. Like many other well established parallelization techniques, pipelining can only be applied to programs with very special characteristics. Without support for speculative execution [10], [7], pipelining techniques are useless for programs with loop-carried dependencies. In this paper, we use forwarding delays rather than speculative techniques to weaken the data dependencies. We benefit from the very slow changes in dynamic system simulations to delay the inputs and break some data dependencies, thus allowing concurrent execution.

Domain-specific parallelization techniques have been very effective in exploiting multi-core systems. For example, stream computing community has developed languages and techniques that exploit multi-core systems very effectively. Similarly, we apply domain-specific techniques to the Simulink domain to exploit the full potential of modern multi-core systems. The difference between the two is that control-based programs are replete with loop-carried dependencies, which are very rare in stream programs [6].

III. SEMANTICS OF SIMULINK

Simulink is a visual programming language that uses a block-diagram notation to mathematically describe and simulate dynamic systems (that vary over time). A set of predefined primitive blocks is used to describe the algebraic and differential equations governing a system. Some blocks (e.g. integrators, transfer functions) have a state and their output is a function of their current state and input from previous iteration, these blocks will be referred to for the rest of the paper as Mealy blocks.

Time being the independent variable in a dynamic system simulation, Simulink discretizes time into a series of finite time steps in which the model is repeatedly executed until the simulation time is reached. An ordinary differential equation (ODE) solver performs the numerical integration of the simulated model on each time step. The speed and fidelity of Simulink simulations depend on the selection of the time step size and the ODE solver since ODE integration methods trade-off accuracy for computation workload. While variable-step size ODE solvers attempt to gap a fast simulation time with an accurate result, the rule of thumb is that small time steps produce very accurate but long-running simulations.

Regardless of the time parameter selection, a critical step for correct simulation is determining the block execution order in a Simulink model. In blocks without a state, the current input determines the value of each block’s outputs in the current iteration. For example, the result of an addition whose current inputs are 1 and 2 is 3. These blocks are said to have direct-feedthrough inputs. Mealy blocks, on the other hand, produce an output for the current iteration as a function of its state and/or the input in the previous iteration. These blocks are said to have non-direct-feedthrough inputs. Clearly, blocks with direct-feedthrough inputs should be executed after all their inputs are available. Put another way, Mealy blocks should be processed earlier than the direct-feedthrough blocks attached to their outputs. Consider the model in Figure 1 that simulates a mass-spring-dashpot system. The order of execution is shown by the numbers in the circles. Notice that mass, damping, and stiffness gains have direct-feedthrough inputs and therefore the Step, Integrator, and Integrator1 blocks have to be executed earlier. Although there is a line connecting Integrator and Integrator1, the Integrator1 can be executed earlier since its output depends on its current state and its input from the previous iteration. Therefore, the data dependency is not within the current iteration and the order in which these blocks are executed is flexible (See “Determining Block Update Order” in [2]). At the end, the sum block can be executed after all of the blocks driving its direct-feedthrough inputs have been executed.

Regardless of the time parameter selection, a critical step for correct simulation is determining the block execution order in a Simulink model. In blocks without a state, the current input determines the value of each block’s outputs in the current iteration. For example, the result of an addition whose current inputs are 1 and 2 is 3. These blocks are said to have direct-feedthrough inputs. Mealy blocks, on the other hand, produce an output for the current iteration as a function of its state and/or the input in the previous iteration. These blocks are said to have non-direct-feedthrough inputs. Clearly, blocks with direct-feedthrough inputs should be executed after all their inputs are available. Put another way, Mealy blocks should be processed earlier than the direct-feedthrough blocks attached to their outputs. Consider the model in Figure 1 that simulates a mass-spring-dashpot system. The order of execution is shown by the numbers in the circles. Notice that mass, damping, and stiffness gains have direct-feedthrough inputs and therefore the Step, Integrator, and Integrator1 blocks have to be executed earlier. Although there is a line connecting Integrator and Integrator1, the Integrator1 can be executed earlier since its output depends on its current state and its input from the previous iteration. Therefore, the data dependency is not within the current iteration and the order in which these blocks are executed is flexible (See “Determining Block Update Order” in [2]). At the end, the sum block can be executed after all of the blocks driving its direct-feedthrough inputs have been executed.

The first key observation we make is that Mealy blocks “eliminate” some data dependencies in the model during every iteration and break the program into independent fragments that can be executed in parallel. These fragments of instructions are known as strands. The “eliminated” inter-iteration dependencies are actually delayed and satisfied between iterations when the inputs of the Mealy blocks are updated to contain the input value of the previous iteration upon which their outputs depend. Figure 2 shows the execution trace of two iterations (N, N+1) of the sample spring-mass system. Notice that the output of the integrators in iteration N+1 is a function of the input values computed in the previous

![Fig. 1. Spring-mass example](Image 339x247 to 537x334)
iteration ($x'_{old}, x''_{old}$) and their current states, and therefore their execution is independent in the current iteration and can be parallelized together with the direct-feedthrough blocks they drive.

Therefore, the Mealy blocks are natural delays in the current iteration that our technique leverages to enable parallel simulations. In addition to these natural delays in Simulink models, we provide to the user a new block called a Skew block. This block allows the designers to specify the locations in the model where delays are wanted. In many cases, the model designer knows where the non-critical parts of the model are, so they can be delayed for a few iterations and this valuable information can be used to aggressively parallelize the simulations. Our compiler can be configured to partition the model using natural delays and/or user-defined delays. The Skew block is an empty mask subsystem that Simulink builds the multi-threaded C code is produced.

The Skew block is an empty mask subsystem that Simulink builds the model using natural delays and/or user-defined delays. The Skew block is an empty mask subsystem that Simulink builds the model using natural delays and/or user-defined delays. The Skew block is an empty mask subsystem that Simulink builds the model using natural delays and/or user-defined delays.

The dashed edges connecting nodes represent the broken-edge set. Therefore, the Mealy blocks are natural delays in the current iteration and can be parallelized together with the direct-feedthrough blocks they drive.

In this section, we introduce a novel method to parallelize Simulink simulations. We describe how our method is implemented as an automatic compiler transformation using the BlueLink compiler infrastructure [11]. Our compiler takes as input the standard Simulink model files (MDL files) and produces multi-threaded C code as output. Algorithm 1 outlines our parallelization method. The algorithm takes the dataflow graph of the model, the skew factor, and the number of processors to be used as its inputs. The dataflow graph of the model is the result of parsing the MDL file and building an intermediate graph representation of the blocks and the connections between the blocks in the Simulink model. The algorithm first finds the edges that can be broken in the current iteration. This set is obtained by identifying the inputs of the Mealy blocks and/or the Skew blocks. With this information the compiler builds a concurrent program dependence graph. Load balancing and thread assignment is performed by a heuristic algorithm. Nodes in the concurrent program dependence graph are mapped to threads and each thread is mapped to one independent core. The set of broken edges is used to define inter-thread communication to update the inputs for the Mealy blocks for the next iteration. Finally, the multi-threaded C code is produced.

### IV. Parallelizing Simulink Applications

In this section, we introduce a novel method to parallelize Simulink simulations. We describe how our method is implemented as an automatic compiler transformation using the BlueLink compiler infrastructure [11]. Our compiler takes as input the standard Simulink model files (MDL files) and produces multi-threaded C code as output. Algorithm 1 outlines our parallelization method. The algorithm takes the dataflow graph of the model, the skew factor, and the number of processors to be used as its inputs. The dataflow graph of the model is the result of parsing the MDL file and building an intermediate graph representation of the blocks and the connections between the blocks in the Simulink model. The algorithm first finds the edges that can be broken in the current iteration. This set is obtained by identifying the inputs of the Mealy blocks and/or the Skew blocks. With this information the compiler builds a concurrent program dependence graph. Load balancing and thread assignment is performed by a heuristic algorithm. Nodes in the concurrent program dependence graph are mapped to threads and each thread is mapped to one independent core. The set of broken edges is used to define inter-thread communication to update the inputs for the Mealy blocks for the next iteration. Finally, the multi-threaded C code is produced.

**Algorithm 1: Skewed Pipelining Simulink Transformation**

1. **Input**: Program dataflow graph
2. **Input**: Skew factor
3. **Input**: Number of processors
4. **Output**: Multi-threaded C code

#### A. Constructing the Concurrent Program Dependence Graph

The first step is finding the set of edges that can be broken during each iteration and delayed in-between iterations. The compiler performs a depth-first search walk on the dataflow graph and, for every Mealy block and Skew block that is found, its input edges are included in the set. Depending on the compilation flags set by the user, all of the discovered edges will remain in the set, or the inputs to the Mealy blocks or to the Skew blocks may be discarded. The computed broken-edge set is then used to partition the dataflow graph into chains of blocks and edges where the direct-feedthrough blocks are driven by the Mealy blocks and/or the Skew blocks, which are located at the beginning of the strand. Every resulting strand represents a node in the concurrent program dependence graph (CPDG). The edges in the broken-edge set connect the strands in the CPDG. It is important to notice that the edges connecting the strands in the CPDG are inter-iteration data dependencies and all strands in the CPDG can be executed in parallel. Figure 3 shows the partitioned data flow graph of the anti-lock brake system program shipped with Simulink.

The dashed edges connecting nodes represent the broken-edge set edges. Boxes \{S1, ..., S7\} represent the boundaries of the discovered strands, and gray nodes represent the Mealy blocks.

Nodes in the CPDG are weighted according their estimated execution time. This is computed by adding the individual estimated execution times of all of the blocks in the strand. The edges of the CPDG are weighted according to the number of data tokens that are forwarded between iterations. Figure 4 shows the resulting CPDG obtained from the dataflow graph in Figure 3. The strands in the CPDG are annotated with the estimated execution time (in CPU cycles). For example, S3 has an estimated execution time of 24, and S5 of 18. Nodes in gray indicate chains with at least one Mealy block and the white nodes show chains where all blocks have direct-feedthrough inputs. For this example, each edge communicating two strands forwards a single value and therefore all the edges have a weight equal to 1.

Since the execution time and communication latency may differ between architectures, we first profile the execution of individual blocks by executing every block across a large iteration space and averaging the execution time of several runs. Given the clock speed of the host processor, the number of iterations, and the average execution time, we obtain the
block’s workload in CPU cycles per iteration.

![Diagram](image1.png)

**Fig. 3.** Partitioned dataflow graph of an ABS automotive model. Boxes represent the thread-level concurrency in the model exposed by our method (Strands).

![Diagram](image2.png)

**Fig. 4.** CPDG for ABS automotive model. Nodes are annotated with estimated execution time. Edges are annotated with number of forwarded values between iterations.

### B. Skewed Pipelining

Achieving a good balance between computation and communication for every thread is critical in multi-core systems [6]. The main objective of this phase is mapping the nodes in the CPDG into N processors such that each processor has roughly the same workload and communication between iterations is minimized. This problem can be restated as the classic multiprocessor scheduling problem, which is NP-complete. We solve this problem with a heuristic algorithm to solve the bin-packing problem. Our algorithm balances the workload among N processors while trying to minimize the communication among threads. If two nodes connected by an edge in the CPDG are placed in the same processor the communication cost is set to zero, else it is a function of the number of forwarded values and the architecture-dependent inter-core communication cost. Similarly, nodes that are assigned to the same processor arecollapsed into a single node.

Regardless of the load-balancing and scheduling heuristics used, the main problem in parallelizing small Simulink models is that the granularity of the resulting threads is very fine and the execution is dominated by inter-thread communication cost. We define the granularity of a thread as the ratio between computation and communication. Figure 5(a) shows the load-balanced schedule for two processors of the CPDG in Figure 4. Assuming that a very efficient software implementation for inter-core communication is available [12], the forwarding cost between two cores is about 100 cycles. The critical path in the multi-core case is given by the latency of the computations plus the communication cost, for this example $35 + 100 = 135$ cycles. For these cases, the obvious strategy would be to exploit the fine-grain concurrency at the instruction level in a superscalar uniprocessor rather than in a multi-core system. Figure 5 shows the critical path in a uniprocessor is given by the latency of the computations, since the communication cost can be ignored, in this case for 70 cycles. Therefore, executing small models in a uniprocessor results in faster simulations than using multi-core systems.

![Diagram](image3.png)

**Fig. 5.** Critical path for small models is dominated by communication

To overcome this problem and effectively use multi-core systems to improve simulation times, including small Simulink models, we propose a novel skewed pipelining technique. Skewed pipelining delays the inter-thread communication for several iterations while concurrently executing all of the threads. This scheme coarsens the granularity of the threads by enlarging the computation bodies by several iterations and reducing the communication overhead. The number of iterations that are executed without receiving new values is known as the skew factor. Figure 6 depicts the computation and communication patterns of conventional pipelining and our skewed pipelining during the execution of the example in Figure 5(a). Notice that communication cost $(c_1, c_2, \ldots)$ is larger than computation workload $(i_1, i_2, \ldots)$. In the conventional pipelining depicted in Figure 6(a), for every computation there is one communication. In contrast, by delaying communication by a skew factor of 4 as shown in Figure 6(b), the communication occurs only once per 4 iterations. In general, the execution time using skewed pipelining is bounded by:

$$skew \_exec = total \_work + \frac{total \_communication}{skew \_factor}$$

Where total _work is the workload of the largest thread times the number of iterations, and total _communication is the communication cost times the number of iterations. Using Equation 1, our algorithm decides whether it is profitable to apply the transformation and generate multi-thread code by comparing the estimated execution time using skewed pipelining against uniprocessor execution, and otherwise it discards
the transformation and generates uniprocessor code. Since the number of iterations is always known for a Simulink model executed with a fixed-step numerical solver, this analysis can be done at compile-time. Although our current implementation requires the skew factor as an input, we note that the selection of the best skew factor can be delegated to off-line training methods, or use dynamic adjustment based on current simulation conditions.

![CPU diagram](image)

(a) Computation and communication patterns for N iterations.

Let the iteration space for the example in Figure 5(a) be 100 iterations. Conventional pipelining produces an estimated execution time of 13,500 cycles. Our algorithm decides to apply the transformation because using 2 processors with a skew factor of 4 the estimated total execution time is 6,000 cycles. The uniprocessor estimated execution time is 7,000 cycles.

Communication primitives are inserted in the CPDG to forward data between the threads. All forwarded values between a pair of threads are packed into a single communication message using the Message Aggregation technique [8]. The skew factor is also given for the code generator to emit the corresponding code. Threads communicate through software queues similar to [12]. The synchronization policy is to block threads using a busy loop whenever a queue is empty or full. This scheme decouples execution into a pipeline of working threads.

V. EXPERIMENTAL RESULTS

This section presents the results of the experimental evaluation of skewed pipelining on five Simulink programs that have the following characteristics: (1) dynamics described by a set of differential equations (with at least two Mealy blocks), (2) tightly coupled components in a closed-loop control system configuration with at least one feedback signal, and (3) small programs with a few hundred blocks. The lack of standardized Simulink benchmarks narrows the selection to a very few publicly available models that are not protected by corporate IP. Therefore, our selection includes an automotive anti-lock brake system model (ABS) [2], an aeronautic flight control system for a jet fighter (F14) [2], a maritime autopilot system for a cargo ship (Mariner) [13], a biophysical model of the human circulatory system (Physbe) [14], and an electric induction machine (Inductor) [15].

Our multi-core testbed has one Intel Xeon processor with 4 cores running at 3.16 GHz. The host operating system is Linux 2.6.27 and our compiler emits multi-thread code compatible with POSIX threads. Inter-thread communication queues are a custom implementation of [12]. We explicitly control the thread affinity to physical processors to fully exploit the underlying architecture and hierarchical cache memory system.

A. Effects of Skew Factor in Performance

Figure 7 shows the speedup obtained by executing the benchmarks with skewed pipelining. The results are normalized to the performance of the programs executed on a single processor. The baseline programs were obtained by compiling the applications for a uniprocessor using our compiler. In [16], we have shown that our compiler generates faster uniprocessor code than available code generators. Therefore, it is fair to measure the speedup using multiple processors against our uniprocessor code. According to their sizes and workload, the programs were executed in 2, 3, and 4 processors. Being the smallest and having similar sizes, about 70 blocks, ABS and F14 were executed on 2 processors. Mariner, with 150 blocks was executed on 3 processors. Physbe, with 346 blocks, was executed on 4 processors. Although Inductor has 260 blocks, its workload is small and was executed on 2 processors.

A skew factor of 1 represents a case very similar to conventional pipelining techniques in which every iteration includes communication. In this configuration, the granularity of every iteration is not coarse enough for the smaller models (ABS, F14) and using 2 processors is slower than executing the programs on 1 processor by 4% and 13%, respectively. Larger models (Mariner, Physbe), on the other hand, have better granularity and using 3 and 4 processors results in modest speedups of 1.23 and 1.27 times over the uniprocessor.

Increasing the skew factor coarsens the granularity and concurrency of iterations. Interestingly, the performance for all programs is saturated at a skew factor of 16, except for Physbe that is saturated with a skew factor of 32. These results suggest that skew factors between 16-32 produce the maximum possible speedups. The maximum theoretical speedup that can be obtained by parallel execution in N cores is N times, assuming ideal conditions. For the ABS, F14, and Inductor programs executed in 2 processors, the achieved speedups are 1.56, 1.72, and 1.61 times. Mariner executed in 3 processors achieves a speedup of 2.38 times. And Physbe in 4 processors achieves a speedup of 3.33 times. These results show that our skewed pipelining technique can effectively parallelize applications with small bodies by coarsening the granularity of iterations.

B. Numerical Error

Delaying the communication by some iterations has the unwanted effect of introducing numerical errors in the simulation. This occurs because the Mealy blocks that represent differential equations are solved by ODE numerical methods on every iteration. The outputs and new states of these blocks...
are a function of their current state and their previous input. When several inputs are delayed by the skewed pipelining, these blocks are being fed with the same constant input for several iterations when, in practice, these inputs are very similar but not identical. Figure 8 plots the resulting error for the benchmarks using the skewed pipelining, relative to the accuracy of the original simulations. The error grows linearly with respect to the skew factor. Notice that with a skew factor of 8-16, the error is within 0.1%. Even for larger skew factors, the error is within 1%

The numerical error depends greatly on the time step of the simulation because of the discretization of time. All of these programs have a step size of 1 ms, except for Mariner with 10 ms, and Inductor with 10 µs. Notice that Mariner generates larger errors with smaller skew factors and Inductor produces very small errors because the larger step sizes introduce larger errors. Typical automotive and aerospace simulations have step size of 1 ms or smaller. Therefore, the error for these applications can be expected to remain within this range. It is important to notice that larger applications have coarser granularities and therefore very small skew factors may be sufficient to achieve maximum speedups. Figure 7 and Figure 8 show that these benchmarks have their best performance/accuracy configuration with a skew factor of 8-16.

VI. CONCLUSION

In this paper we introduced a novel skewed pipelining technique that applies domain-specific knowledge to speedup dynamic system simulations in multi-core systems. Communication cost can be mitigated by delaying the forwarding of values for a few iterations. An automatic parallelization compiler technique can be effectively combined with user-inserted skew blocks to aggressively parallelize Simulink applications. For five characteristic applications of automotive, aerospace, maritime, biophysical, and electric disciplines, we shown that speedups of 1.72, 2.38, and 3.33 times can be obtained by using 2, 3, and 4 processors on a commodity multi-core system.

Potential applications of our technique include real-time SILS and HLS, code generation for multi-core embedded systems, acceleration of complex systems simulations, and it can also be valuable in reducing the execution time in studies that involve large numbers of simulations. High-performance simulations will be required more than ever for the development of hybrid and electric vehicles. Although our technique aims at automotive and aeronautical simulations, it can be the starting point for the acceleration of simulations of other engineering domains. In our future work, we will combine skewed pipelining technique with parallel ODE solvers to treat specific problems like simulation of stiff systems. We believe that our method can be extended for embedded target code generation by adding constraints of communication minimization and WCET analysis.

REFERENCES