IEICE TRANS. INF. & SYST., VOL.E97–D, NO.12 DECEMBER 2014

PAPER Special Section on Parallel and Distributed Computing and Networking

ILP Based Multithreaded Code Generation for Simulink Model

Kai HUANG[†a)], ***Member***, **Min YU**[†b)], **Xiaomeng ZHANG**[†c)], **Dandan ZHENG**[†d)], **Siwen XIU**[†e)], **Rongjie YAN**[††f)], **Kai HUANG**[†††g)], **Zhili LIU**[††††h)], ***and*** **Xiaolang YAN**[†i)], ***Nonmembers***

SUMMARY The increasing complexity of embedded applications and the prevalence of multiprocessor system-on-chip (MPSoC) introduce a great challenge for designers on how to achieve performance and programmability simultaneously in embedded systems. Automatic multithreaded code generation methods taking account of performance optimization techniques can be an effective solution. In this paper, we consider the issue of increasing processor utilization and reducing communication cost during multithreaded code generation from Simulink models to improve system performance. We propose a combination of three-layered multithreaded software with Integer Linear Programming (ILP) based design-time mapping and scheduling policies to get optimal performance. The hierarchical software with a thread layer increases processor usage, while the mapping and scheduling policies formulate a group of integer linear programming formulations to minimize communication cost as well as to maximize performance. Experimental results demonstrate the advantages of the proposed techniques on performance improvements.

key words: *code generation, ILP, task mapping, scheduling, Simulink*

1. Introduction

The demands on high performance from emerging embedded applications drives MPSoC as an attractive solution with strong flexibility and concurrent capability. The increasing complexity of embedded systems makes programming on MPSoC more difficult. This process involves laborious effort, such as to extract communication between concurrent tasks and avoid deadlock among them, to adapt software code to different types of processors and communication protocols manually, and to distribute code and data among processors. Therefore, it is necessary to take advantage of automatic techniques to shorten this process and improve the efficiency of software design exploration.

Manuscript received January 9, 2014.
Manuscript revised April 10, 2014.

[†]The authors are with Institute of VLSI Design, Zhejiang University, China.
[††]The author is with the Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences, China.
[†††]The author is with the Department of Informatics VI, Technical University of Munich, Germany.
[††††]The author is with the Hangzhou C-SKY Co. Ltd, China.
a) E-mail: huangk@vlsi.zju.edu.cn
b) E-mail: yumin@vlsi.zju.edu.cn
c) E-mail: zhangxm@vlsi.zju.edu.cn
d) E-mail: zhengdd@vlsi.zju.edu.cn
e) E-mail: xiusw@vlsi.zju.edu.cn
f) E-mail: yrj@ios.ac.cn
g) E-mail: huangk@in.tum.de
h) E-mail: zhili_liu@c-sky.com
i) E-mail: yan@vlsi.zju.edu.cn
DOI: 10.1587/transinf.2014PAP0015

Recently, functional models have been used to express parallelism in target applications and can be easily transformed into multithreaded code automatically [1]. Some high-level modeling languages, such as Khan Process Network (KPN) [2], dataflow [3] and Simulink [4], have been used for system specification and system implementation with automatic hardware and software code generators [5]–[9]. However, automatic code generation may lead to low performance of the whole system if mishandling the performance factors.

With the increasing size of applications and the number of tasks mapped on the same processor, tasks on different processors must collaborate with each other to complete their work, which necessitates synchronization amongst them. Such synchronization may result in a blocked processor and further lead to processor utilization reduction. Multithreaded software is a solution to the problem, which allows non-blocking execution of processors even though some threads are blocked.

However the increasing number of processors drives software designers to use fine-grained multithreaded software, which makes it harder to manage the frequent and explicit inter-thread communication for better performance. An obvious problem in fine-grained multithreaded code generation is that breaking a task into finer grained subtasks incurs higher overhead in terms of scheduling and synchronizing the sub-tasks. Fundamentally, the system performance and scalability will be limited by these overhead [10]. So the techniques of fine-grained multithreaded code generation have to face the following challenges to reduce the impacts of these overheads on system performance:

(1) Improve processor utilization. To achieve good scalability with increasing number of processors in software, a sufficient number of threads, capable of running in parallel, need to be available so all processors can be fully utilized. Finer-grained threads may cause more frequent synchronization and lead to longer synchronization time in processor usage. It is necessary to make threads work concurrently to improve processor utilization.

(2) Reduce communication cost. Automatic code generation method based on fine-grained specification can provide more optimization opportunities such as exploiting fine-grained parallelism, more efficient partitions, and fine-grained memory optimization. However, the fine granularity obtained from the specification after thread mapping may introduce a large number of messages among threads and

processors, which ultimately increases the communication overhead. This overhead increases the required execution time and limits the benefits that could be obtained with the target MPSoC.

To address the first challenge, carefully designed scheduling algorithms are required to improve processor utilization, and communication-aware mapping algorithms are needed to handle the second challenge. In this paper, we propose an ILP based design-time mapping & scheduling method, where performance factors of multithreaded software are formulated to get an optimal solution for performance, and integrate these techniques into our automatic code generation flow from Simulink model.

2. Related Work

Generally, task scheduling is implemented statically or dynamically according to the moment when tasks are scheduled.

Run-time scheduling (RTS) is called dynamic scheduling that requires fast heuristics to explore system resources. It includes ASAP/ALAP scheduling, list scheduling, fore-directed scheduling. Heuristic scheduling schedules tasks one by one and finally obtains feasible solutions [11].

Based on work stealing [12], some software-only, hardware-only and software-hardware combined schedulers are proposed in [13]–[15]. These run-time schedulers work only when they are called, e.g., when some threads run out of available tasks or some tasks become ready. These schedulers concentrate on how to manage queues of ready tasks and how to steal tasks from queues of other threads.

LESCEA (Light and Efficient Simulink Compiler for Embedded Application) [1] is an automatic code generation tool based on Simulink models. Abstract Clock Synchronous Models (ACSM) is proposed as its functional model, where applications are represented with tasks and data links. Derived from ACSM, a Combined Algorithm and Architecture Models (CAAM) is used as its system model, where both task level and system level are visible. LESCEA is a memory-oriented code generation tool with two main techniques: Copy Removal and Buffer Sharing. Copy Removal minimizes the size of data memory for each thread. Buffer Sharing allows two pieces of buffer in the same thread to share the same memory space if their lifetimes do not overlap. However, it has little consideration on processor utilization and communication optimization. Its processor and thread mapping are manually determined, and the workload balance and communication are always non-optimal for performance enhancement. Meanwhile, it includes a run-time scheduler which is based on FIFO policy, and the scheduler works when some threads are blocked.

Although schedulers introduced above can make the best decision according to the status of the system during run-time, they may obtain local optimization results rather than global optimization results, as they cannot determine when to be called.

Design-time scheduling (DTS) is a static scheduling strategy. The execution sequences of tasks are determined during design-time, and tasks are executed according to the scheduling results during run-time, regardless of the system status. Design-time scheduling is application-sensitive which should be applied once the application is changed, so it is less general compared with run-time scheduling. Furthermore, design-time scheduling requires complex heuristics to explore the system resources and balance workloads. As the scheduling cost does not affect the system performance, techniques requiring huge amount of computation can work well, such as ILP based approaches.

Armin Bender proposes a design-time scheduling approach based on Mixed Integer Linear Programming (MILP) for real-time systems [16]. Communication costs, execution time of tasks, and real-time constraints are considered in this MILP model. The task mapping and scheduling are determined by the MILP formulations. In order to obtain optimal throughput, tasks are mapped and executed in sequence according to the result of MILP formulations.

Based on above works, Yi et al. present a similar ILP based approach for loop level task partitioning, task mapping and scheduling while taking communication into account [17]. The difference between the approaches of Armin Bender and Yi is that in the former, the computation of a task in the $(i + 1) - th$ iteration can be executed only if all tasks in the $i - th$ iteration are finished, while the latter eliminates this limitation and generates more efficient solutions.

All these ILP based algorithms are performance oriented approaches, which generate solutions with optimal throughput based on their hardware & software systems. However, all of the above mapping and scheduling algorithms are based on a two-layered hierarchical software (2LHS) system. In the software systems of Armin Bender and Yi, the thread layer does not exist and all tasks mapped on the same processor are executed sequentially, which may limit the throughput as the application size grows.

In this paper, we propose an ILP based design-time scheduling approach on the three-layered hierarchical software (3LHS) system to optimize the throughput of a multithreaded system.

3. Background

A Simulink model represents the functionality of a target system with software function and hardware architecture. Detailed concepts on Simulink models have been introduced in [18]–[20]. Usually, a Simulink model has the following three types of basic components.

- *Simulink Block* represents a function that takes n inputs and produces certain outputs. User-defined (S-function), discrete delay, and pre-defined blocks such as mathematical operations are examples of Simulink blocks. Besides the functional blocks (cycles in white in Fig. 1), our Simulink model also has communication (sending and receiving) blocks (cycles in gray in Fig. 1) to explicit model communication.

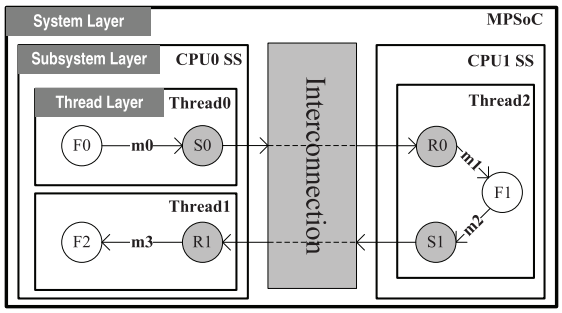


Fig. 1 Simulink model hierarchical architecture.

- *Simulink Link* is a one-to-many link, which connects one output port of a block to one or more input ports from the corresponding blocks. If there is a link from block $F0$ to block $F1$, we say that $F1$ depends on $F0$, denoted by $F0 \rightarrow F1$. That is, a Simulink link is a dependency relation between different blocks. For a Simulink link starting from a sending block S and ending with a receiving block R from different threads, we call it a communication vector, denoted by $S \rightarrow R$.
- *Simulink Subsystem* can contain blocks, links, and other subsystems to represent hierarchical composition and conditionals such as for-loop iteration or if-then-else structure.

In our design, an MPSoC Simulink model can be specified as a three-layered hierarchical architecture after ILP based mapping, including system layer, subsystem layer and thread layer, as presented in Fig. 1. The system layer describes a system architecture that is made up of CPU subsystems and inter-subsystem communication channels between them. The subsystem layer describes a CPU subsystem architecture that includes a set of threads and intra-subsystem communication channels between them. Finally, the thread layer describes a software thread that consists of Simulink blocks and links between them.

Based on the above model, tasks mapped to the same thread are executed sequentially according to a static order, and the execution order is determined by the task dependencies and the intra-thread scheduling policy. Tasks in different threads are executed statically according to the scheduling results if design-time scheduling has been applied. Otherwise, the order of tasks in different threads is determined dynamically according to the policy of scheduler and the status of system.

4. Combination of Three-layered Hierarchical Software and Design-time scheduling

In this section, we illustrate the advantages of the combination of 3LHS and DTS with some examples. We take the run-time scheduler applied in LESCEA as a comparison, which takes FIFO as its scheduling policy and works when the threads being executed are blocked.

Compared with the DTS system, thread switching occurs only when the threads being executed are blocked in

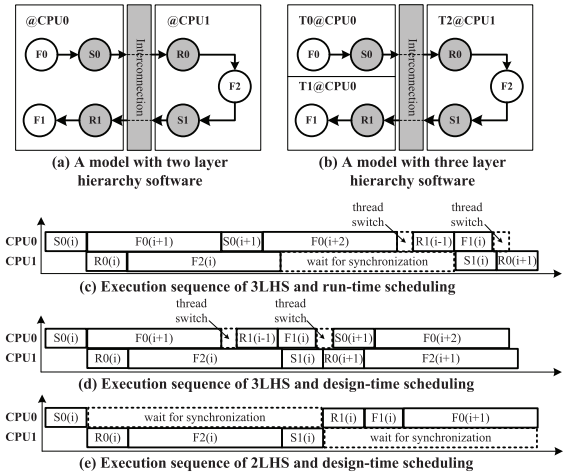


Fig. 2 Advantage of combining 3LHS and DTS.

the RTS system. However, this run-time strategy may lead to un-optimal thread switching timing. We present a 3LHS model in Fig. 2(b), while Fig. 2(c) and Fig. 2(d) present the execution sequence of the 3LHS model combining RTS and DTS respectively. In RTS system, after $F0(i+1)$ is executed by $CPU0$, $S0(i+1)$ is still executable and a new iteration of $T0$ is executed. Meanwhile, $CPU1$ is idle because $T2$ is waiting for the synchronization of $R1(i-1)$ in $T1$. In the DTS system, thread switching occurs after $F0(i+1)$ is executed in order to reduce the synchronization cost of $CPU1$.

We can observe from the comparison that, the main difference between DTS and RTS is the time of thread switching. A DTS system is able to catch the optimal thread switching timing from a global view. However, this advantage of DTS may be reduced if applied to a 2LHS system. Compared with 3LHS system, all tasks mapped to the same processor are executed sequentially in the 2LHS system, which hinders the effectiveness of DTS, as thread switching does not occur in the 2LHS system. Furthermore, the processor utilization in a 2LHS system is much lower than that in the 3LHS system. The reason is that once one task is blocked, the whole processor is also blocked even though other tasks on the processor are still executable. As shown in Fig. 2(e), ignoring thread layer increases the synchronization waiting time dramatically.

Based on above discussions, we can observe that combining 3LHS and DTS is a more optimal solution to reduce synchronization waiting time.

5. ILP based Design-time Mapping and Scheduling on Three-layered Hierarchical Software

In this section, we first propose ILP based task mapping on the 3LHS system. According to the complexity of ILP formulations, we divide task mapping into two steps: processor mapping and thread mapping, which determines the relationship between tasks and processors, tasks and threads respectively. During processor mapping, we consider interprocessor communication and workload balance of the sys-

tem and make a compromise between them. Then during thread mapping, we try to minimize intra-processor communication to reduce synchronization between threads.

Next we propose our intra-thread scheduling algorithm to determine the invocation order of tasks in each thread. In order to reduce synchronization cost of each processor, we concentrate on inter-processor communications of each thread. The inter-processor sending blocks are invocated as early as possible while the inter-processor receiving blocks as late as possible. Furthermore, we propose a combination of global and local scheduling in the algorithm to speed up scheduling as well as avoid deadlock.

Finally, we propose an ILP based inter-thread scheduling approach. In our approach, we introduce performance factors such as thread switching cost, intra-processor communication cost and inter-processor communication cost to describe the features of the 3LHS system.

5.1 Processor Mapping

As the overall performance is determined by the longest execution time of all processors, minimizing the workload gaps between processors is required to improve performance. Furthermore, inter-processor communication increases communication cost, and may make some processors blocked (all threads of these processors are blocked) during run-time. In order to obtain better performance, we need to minimize inter-processor communications.

According to above reasons, during processor mapping, we consider both workload balance and inter-processor communication. We propose a group of ILP formulations to satisfy the above two requirements. Before listing the components of ILP, we first present the notation of constants.

- P be the set of processors.
- T be the set of tasks.
- N_{ij} be the message size between task $t_i, t_j \in T$.
- D_{ij} be boolean constants indicating the dependency relationship between task $t_i, t_j \in T$, D_{ij} is 1 only if task $t_j \in T$ depends on task $t_i \in T$.
- AV_{ik} be boolean constants, the value is 1 only if processor $p_k \in P$ can execute task $t_i \in T$.
- ET_{ik} be the execution time of task $t_i \in T$ on processor $p_k \in P$.
- AT_i be the average execution time of task $t_i \in T$ on all available processors.

Variables

- Mapping relationship between tasks and processors: Let M_{ik} be boolean variables, the value is 1 only if task $t_i \in T$ is mapped to processor $p_k \in P$.
- Mapping relationship between tasks: Let $B_{ij,k}$ be boolean variables, the value is 1 only if task $t_i \in T$ and $t_j \in T$ are mapped to processor $p_k \in P$.

Object

An objective function is given below to minimize the total inter-processor communication size (CS) and workload gap (WG). WG is a normalized variable indicating the gap between the workload of each processor and the average workload of all processors. The weights $k1$ and $k2$ of the costs CS and WG can be tuned by the designer.

- $$min(k_1 \times CS + k_2 \times WG) \quad (1)$$

$$CS = \Sigma_{i\le|T|}\Sigma_{i<j\le|T|}(1 - \Sigma_{k\le|P|}B_{ij,k}) \times N_{ij} \quad (2)$$

$$WG = \Sigma_{k\le|P|}|\frac{\Sigma_{i\le|T|}M_{ik} \times ET_{ik} \times |P|}{\Sigma_{i\le|T|}AT_i} - 1| \quad (3)$$

Constraints

- Each task must be mapped to a single processor.

$$\Sigma_{k\le|P|}M_{ik} = 1 \quad (4)$$

- Tasks can only be mapped to available processors.

$$M_{ik} \le AV_{ik} \quad (5)$$

- If either tasks t_i or t_j is not mapped to p_k, then $B_{ij,k}$ is 0.

$$B_{ij,k} \le (M_{ik} + M_{jk})/2 \quad (6)$$

5.2 Thread Mapping

After determining processor mapping with the above constraints and objective function, we implement thread mapping to decide the relationship between tasks and threads, where the results of processor mapping are used.

As a processor may be blocked by inter-processor communication, in order to decrease blocking probability, we try to map inter-processor communication tasks to different threads to reduce interaction between different inter-processor communications on the same processor. However, if too many inter-processor communications exist, it is impossible to map one inter-processor communication task to one thread respectively, as too many threads will increase intra-processor communication cost. Therefore, we need to find a trade-off between processor utilization and communication cost. In our design, the number of threads in each processor is the minimum value between inter-processor communication number and $four$, where $four$ is an empirical value.

Additionally, intra-processor communications bring communication cost and thread switching, so we try to minimize the number of intra-processor communication channels during thread mapping. The components of ILP are listed below.

- TH_k be the set of threads in $p_k \in P$.
- T_{inter} be the set of inter-processor communication tasks.

Variables

- Mapping relationship between tasks and threads: Let

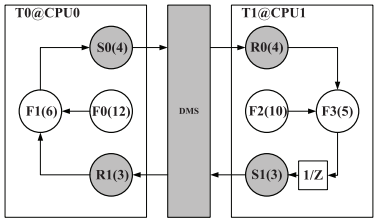


Fig. 3 A scheduling example.

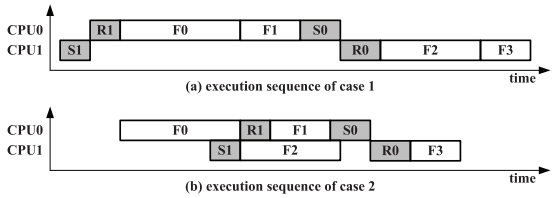


Fig. 4 Execution sequences for different scheduling results.

A_{ijk} be boolean variables, the value is 1 only if task $t_i \in T$ is mapped to thread $th_{jk} \in TH_k$.

- Mapping relationship between tasks: Let $S_{ij,kl}$ be boolean variables, the value is 1 only if task $t_i \in T$ and $t_j \in T$ are mapped to thread $th_{kl} \in TH_l$.

Object

- During thread mapping, we try to minimize the number of intra-processor communication channels.

$$min(\Sigma_{i\le|T|}\Sigma_{i<j\le|T|}(1-\Sigma_{l\le|P|}\Sigma_{k\le|TH_l|}S_{ij,kl})\times D_{ij}) \quad (7)$$

Constraints

- Each task must be mapped to a single thread.

$$\Sigma_{j\le|TH_k|}A_{ijk} = M_{ik} \quad (8)$$

- Each thread must contain at least one inter-processor communication task.

$$\Sigma_{i\le|T_{inter}|}A_{ijk} \ge 1 \quad (9)$$

- If either tasks t_i or t_j is not mapped to thread th_{kl}, then $S_{ij,kl}$ is 0.

$$S_{ij,kl} \le (A_{ikl} + A_{jkl})/2 \quad (10)$$

After tasks have been mapped to different threads, the three-layered hierarchical architecture is formed.

5.3 Intra-thread scheduling

Intra-thread scheduling is an important issue in code generation, which determines the invocation order of blocks in each thread statically. Inappropriate scheduling results can increase processor synchronization waiting time thus decreasing performance dramatically.

We take the model in Fig. 3 as an example, where the numbers in brackets indicate the execution time of each block. Two feasible scheduling results and their corresponding execution sequences are shown in Fig. 4.

The scheduling result of case 1 is $T0(R1 \to F0 \to F1 \to S0)$ and $T1(S1 \to R0 \to F2 \to F3)$, and the scheduling result of case 2 is $T0(F0 \to R1 \to F1 \to S0)$ and $T1(S1 \to F2 \to R0 \to F3)$. The execution sequences of one iteration for both cases are illustrated in Fig. 4, where the performance of case 2 is much better than that of case 1.

From the above example, we can observe that intra-thread scheduling is an important factor for system performance. Meanwhile, the invocation time of inter-processor communication blocks affects processor utilization significantly. So managing inter-processor communication is the focus of our scheduling algorithm. Inter-processor sending blocks should be invocated as early as possible, while inter-processor receiving blocks should be invocated as late as possible.

In order to avoid deadlock, global scheduling [1], [21], [22] is the most general strategy, where all blocks in the model are scheduled together, no matter whether they are in the same thread. Global scheduling concerns more between threads, so it may decrease the scheduling speed. On the contrary, local scheduling strategy schedules threads one by one independently, which can get scheduling results fast. If thread dependency cycle does not exist, local scheduling can work efficiently. Otherwise, it may cause deadlock. Therefore, to reduce scheduling time as well as avoid deadlock, we propose a global and local combined scheduling flow. For threads not in a thread dependency cycle, local scheduling is employed. Otherwise, the sending/receiving blocks in the thread dependency cycles are scheduled globally, and the rest blocks are scheduled locally.

To illustrate the algorithm, we first present two definitions.

Definition 1: Given two blocks A and B, if there is a finite path from A to B, we say that A is a predecessor of B, and B is a successor of A. To simplify the description, the set of successors of block A is denoted by $suc(A)$, and the set of predecessors of A is denoted by $pre(A)$.

Definition 2: Given two blocks A and B, block A is a successor of block B. If there is a finite path between B and any predecessor of A, we say that A is a pure successor of block B. The set of pure successors of block A is denoted by $ps(A)$.

We take the model in Fig. 3 as an example to explain Definition 2. $S0$ is a pure successor of $F1$. But $F1$ is not a pure successor of $R1$, because $F0$ is a predecessor of $F1$ and the path between $R1$ and $F0$ does not exist.

The scheduling algorithm is given in Algorithm 1. First, all communication blocks in thread dependency cycles are scheduled globally. Their invocation order is determined according to their dependency relation in case of deadlock (Line 1). After that, all threads are scheduled locally. Second, inter-processor sending blocks of each thread which do not depend on inter-processor receiving blocks are scheduled (Line 2 to 7). They are scheduled one by one, and are sorted according to the total execution time of their predecessors and themselves, with less total execution time

Algorithm 1: *intra_thread_scheduling*

input : a set of threads $\mathcal{T} = \{T_i = (B_i, E_i)\}_{1 \le i \le n}$, where $B_i = S_i \cup R_i \cup F_i$ is the union of the sets of sending blocks, receiving blocks on inter-processor communication and the rest blocks including function blocks and blocks for intra-processor communication
output: n lists of scheduled blocks
begin
1 $\mathcal{S}$= *thread_dependency_cycle*($\mathcal{T}$);
for $\forall S_c \in \mathcal{S}$ **do**
sorting communication vectors in S_c according to their dependency relation;
for $\forall T_i \in \mathcal{T}$ **do**
// deal with blocks in S_i that do not depend on R_i
do
$Sum = \infty$; $mark = \emptyset$;
for $\forall s \in S_i$ **do**
2 **if** $pre(s) \cap R_i == \emptyset$ **then**
3 **if** $\Sigma_{b \in pre(s) \setminus B_{l_i}} cost(b) < Sum$ **then**
4 $mark = s$;
5 $Sum = \Sigma_{b \in pre(s) \setminus B_{l_i}} cost(b)$;
6 $l_i.add(pre(mark))$; $l_i.add(mark)$;
$S_i.remove(s)$; $F_i = F_i \setminus pre(s)$;
7 **if** $S_i == \emptyset \vee \forall s \in S_i.pre(s) \cap R_i \neq \emptyset$ **then**
break;
while;
// deal with blocks in F_i that do not depend on R_i
for $\forall f \in F_i$ **do**
8 **if** $pre(f) == \emptyset$ **then**
9 $l_i.add(f)$; $l_i.add(ps(f, B_{l_i}))$;
10 $F_i.remove(f)$; $F_i = F_i \setminus ps(f, B_{l_i})$;
// deal with blocks in S_i that depend on R_i
do
$Sum = \infty$; $mark = \emptyset$;
for $\forall s \in S_i$ **do**
11 **if** $\Sigma_{b \in pre(s) \setminus B_{l_i}} cost(b) < Sum$ **then**
12 $mark = s$;
13 $Sum = \Sigma_{b \in pre(s) \setminus B_{l_i}} cost(b)$;
$R_m = \{r | r \in pre(mark) \wedge r \in R_i\}$;
do
$Sum = 0$; $mark' = \emptyset$;
for $\forall r \in R_m$ **do**
14 **if** $\Sigma_{b \in ps(r, B_{l_i})} cost(b) > Sum$ **then**
15 $mark' = r$;
16 $Sum = \Sigma_{b \in ps(r, B_{l_i})} cost(b)$;
17 $l_i.add(mark')$; $l_i.add(ps(mark', B_{l_i}))$;
18 $S_i = S_i \setminus ps(mark', B_{l_i})$;
19 $F_i = F_i \setminus ps(mark', B_{l_i})$;
20 $R_m.remove(mark')$;
21 **if** $R_m == \emptyset$ **then**
break;
while;
if $S_i == \emptyset$ **then**
break;
while;
// deal with the rest blocks in R_i
do
$Sum = 0$; $mark = \emptyset$;
for $\forall r \in R_i$ **do**
22 **if** $\Sigma_{b \in ps(r, B_{l_i})} cost(b) > Sum$ **then**
23 $Sum = \Sigma_{b \in ps(r, B_{l_i})} cost(b)$;
$mark = r$;
24 $l_i.add(mark)$; $l_i.add(ps(mark, B_{l_i}))$;
25 $R_i.remove(mark)$; $R_i = R_i \setminus ps(mark, B_{l_i})$;
26 $F_i = F_i \setminus ps(mark, B_{l_i})$;
if $R == \emptyset$ **then**
break;
while;

scheduled earlier. If one sending block is scheduled, its predecessors and itself are added to the scheduled list according to their dependency relation. Third, the unscheduled functional blocks of each thread which do not depend on inter-processor receiving blocks are added to the scheduled list (Line 8 to 10). Fourth, the remaining inter-processor sending blocks are scheduled together with the inter-processor receiving blocks which they depend on (Line 11 to 21). The inter-processor sending blocks are chosen using the same principle in the second step (Line 11 to 13). Then, the inter-processor receiving blocks being the predecessors of the chosen inter-processor sending blocks are sorted according to the total execution time of their pure successors and themselves. The more the total execution time, the earlier they are scheduled (Line 14 to 16). Next, the scheduled inter-processor receiving block and its pure successors are added to the scheduled list according to their dependency relation (Line 17 to 21). Finally, remaining inter-processor receiving blocks and their pure successors are scheduled as the same principle in the fourth step. (Line 22 to 26).

5.4 Inter-thread Scheduling

After applying intra-thread scheduling algorithm to obtain better thread codes, we now determine the execution flow of thread code to get the optimal performance.

In this subsection, we propose an ILP based DTS. Performance factors of 3LHS system such as thread switching cost, inter-processor communication cost, intra-processor communication cost, etc. are introduced to our ILP model.

We first present the summary of the constants before listing the components of ILP.

- P be the set of processors.
- B be the set of blocks, tasks and sending/receiving ports are collectively called blocks.
- $time_i$ be the execution time of $b_i \in B$ on the mapped processor.
- $time_{switch,k}$ be the thread switching time on $p_k \in P$.
- $time_{startup,k}$ be the start up time of communication on $p_k \in P$.
- $time_{transfer,k}$ be the transfer time for each unit on $p_k \in P$.
- $time_{inter-transfer}$ be the transfer time for inter-processor communication of each unit.
- N_{ij} be the message size between block $b_i, b_j \in B$.
- cyc be the number of total iteration.

Furthermore, the intra-thread scheduling results are also constants in this step.

Variables

- Start time of each block: Let $s_{i,l}$ be the start time of block $b_i \in B$ in the $l-th$ iteration.
- Overall time: Let $time_{overall}$ be the overall time which needs minimization.

Objects

- $min(time_{overall})$ (11)

Constraints

- The overall time is no less than the finishing time of any blocks.

$$time_{overall} \geq s_{i,cyc} + time_i \quad (12)$$

- Blocks mapped to the same thread should be executed according to the order determined by intra-thread scheduling. Assume b_i and b_j are mapped to the same thread, and b_i is invocated before b_j. The start time of b_j in $l-th$ iteration cannot be earlier than the finishing time of b_i in $l-th$ iteration, and the start time of b_i in $(l+1)-th$ iteration cannot be earlier than the finishing time of b_j in $l-th$ iteration.

$$s_{i,l} + time_i \leq s_{j,l} \quad (13)$$

$$s_{j,l} + time_j \leq s_{i,l+1} \quad (14)$$

- For a couple of sending/receiving blocks, the receiving block cannot be invocated before the finishing time of the sending block. However, if the couple of communication blocks are on the same processor, thread switching time should also be considered. Assume a couple of intra-processor communication blocks b_s and b_r are mapped to p_k.

$$s_{s,l} + time_s + time_{switch,k} \leq s_{r,l} \quad (15)$$

$$s_{r,l} + time_r + time_{switch,k} \leq s_{s,l+1} \quad (16)$$

$$time_s = time_r = time_{startup,k} + time_{transfer,k} \times N_{sr} \quad (17)$$

Assume a couple of inter-processor communication blocks, b_s is mapped to p_k and b_r is mapped to $p_{k'}$.

$$s_{s,l} + time_s \leq s_{r,l} \quad (18)$$

$$s_{r,l} + time_r \leq s_{s,l+1} \quad (19)$$

$$time_s = time_{startup,k} + time_{inter-transfer} \times N_{sr} \quad (20)$$

$$time_r = time_{startup,k'} + time_{inter-transfer} \times N_{sr} \quad (21)$$

- Any two blocks in different threads of the same processor cannot be executed at the same time. In other words, their execution time cannot overlap with each other. Assume two blocks b_i and b_j are mapped to different threads in $p_k \in P$. The finish time of b_i in the $l-th$ iteration is earlier than the start time of b_j in the $l'-th$ iteration, or the finish time of b_j in the $l'-th$ iteration is earlier than the start time of b_i in the $l-th$ iteration.

$$s_{i,l} + time_i + time_{switch,k} \leq s_{j,l'} + (1-m) \times MAX \quad (22)$$

$$s_{j,l'} + time_j + time_{switch,k} \leq s_{i,l} + m \times MAX \quad (23)$$

Here boolean variable m is 1 if b_i in the $l-th$ iteration is executed earlier than b_j in the $l'-th$ and 0 otherwise.

With above formulations, the execution sequences of all blocks can be determined.

6. Implementation and Experiment

6.1 Implementation

The proposed techniques are implemented based on LESCEA multithreaded code generator of the Simulink-based MPSoC design platform [1], [21], [22]. Multithreaded code generator takes a Simulink Model as an input and generates a set of software thread codes, and builds software stacks fit to corresponding scheduling results and the target hardware architecture. The Simulink blocks are bound to processors and threads, and scheduled statically within each thread according to data dependency and block features (block type, execution time, etc). Then the generated threads are statically scheduled according to the inter-thread scheduling results.

Figure 5 shows the global flow of the multithreaded code generation that produces a set of efficient threads and a main C code for each CPU subsystem. The whole design flow of automatic multithreaded code generation consists of six steps: 1) Processor mapping. Processor mapping binds each task to one processor. 2) Thread mapping. Based on the first step, it divides tasks on the same processor into multiple threads. After this step, a system model is obtained, which is the starting point of LESCEA. 3) Intra-thread scheduling. The invocation order of blocks in each thread is determined according to the scheduling policy and dependency relationship between blocks. 4) Thread Code Generation. A set of C code is generated for each thread, which includes memory declarations and a sequence of function calls corresponding to the invocation order of blocks, and maps the allocated memory space to the arguments of functions. Moreover, the code also contains primitives for communications. 5) Inter-thread scheduling. All blocks in the system are scheduled simultaneously based on ILP method to achieve the best performance. The invocated time point of each block for each iteration is determined in this step. 6) HdS (Hardware dependent Software) Adaptation. This step generates a main code for creating threads

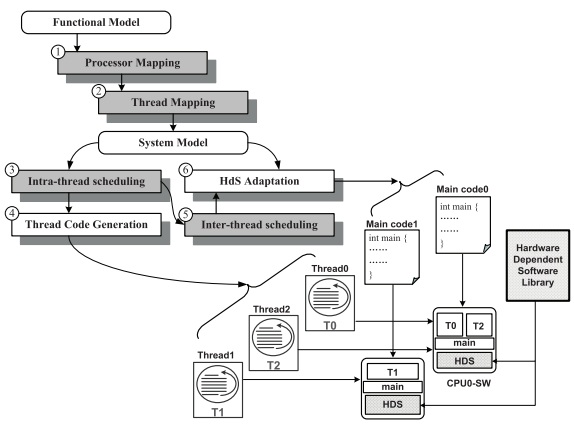


Fig. 5 Design flow of automatic multithreaded code generation tools.

Table 1 Experiments integrating different techniques.

	M1	*M2*	*LESCEA*	*M3*	*M4*
2LHS architecture	√	√			
3LHS architecture			√	√	√
ILP based processor mapping		√		√	√
ILP based thread mapping				√	√
ILP based scheduling		√			√

and initializing channels for the target CPU subsystem, and a thread scheduler according to the inter-thread scheduling results. It also generates a Makefile to link the threads with HdS library.

6.2 System Emulation Platform

In our experiment we have adopted an H.264 baseline decoder as the case study for the proposed techniques. The hardware platform used in the experiment is a flexible MPSoC hardware platform with an efficient interconnection network for processor scalability. This hardware platform consists of 8 CPU subsystems, a Memory subsystem and an Interconnection subsystem. Each CPU subsystem uses a 32-bit high speed bus to connect one processor with its local SRAM and a 32-bit low speed bus to other local peripherals such as timers. The processor type in the CPU subsystem is configured as a 32-bit 3-stage ultra-low power RISC CKCore processor [23] without instruction and data cache. The Memory subsystem uses a 64-bit high speed bus to connect off-chip global DDR SDRAM. These two subsystems are connected with distributed memory server (DMS) Interconnection subsystem respectively through memory service access point (MSAP) [24]. The DMS acts as a server that provides communication & synchronization services to subsystems in the MPSoC. Each MSAP delivers data transfer requests issued by its corresponding subsystem to other MSAP via the control network. It also exchanges synchronization information, which indicates the completions of requests handling, with other MSAP via the control network.

The software platform consists of thread codes, CPU main codes and an HdS library on each target CPU. The H.264 model is mapped to the target 2/4-CPU hardware platform. The inter-processor communication channels are allocated with global SDRAM and implemented by MSAP configuration. The intra-processor thread communication channels are allocated with local SRAM and implemented by shared memory mechanism. The experimental results about time cost and processor utilization are obtained from FPGA emulation using 300-frame CIF H.264 bit stream as input.

To check the effectiveness of the proposed techniques, we have applied them incrementally. As shown in Table 1, there are five sets of experiments with different combination of techniques over the same application. We mainly compare results on memory size, processor usage, and total execution cycles from these experiments.

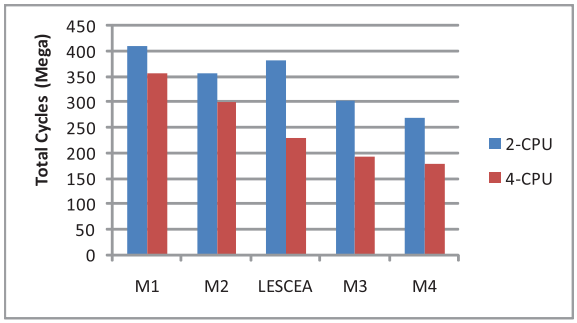


Fig. 6 Total cycles of each case.

6.3 Experimental Results

To evaluate the effectiveness of the proposed techniques, we have applied our multithreaded code generator to an H.264 baseline decoder. The Simulink functional model of the H.264 decoder consists of 83 S-Functions, 24 delays, 286 data links, 43 IASs (if-action subsystems), 5 FISs (for-iteration subsystems) and 101 pre-defined Simulink blocks. This ACSM model is built on a 16x16 macro block index as an abstract clock with good granularity to represent parallelism and communication explicitly. We have implemented different mapping and scheduling strategies to a 2-CPU and a 4-CPU hardware platform.

In our experiment, there are three categories for a processor state: idle state, communication (comm) state and computation (comp) state. The communication state includes communication setup state and communication transfer state, and the idle state consists of synchronization waiting state and thread switching state. Except for computation state, the other two states mainly represent the overhead of the multithreaded codes, which need to be reduced. In the following experimental results, we use average percentage of processor utilization among multiple processors to evaluate the resource usage of the whole system.

The time cost is presented by the number of cycles in Fig. 6. The throughput is enhanced in both 2-CPU and 4-CPU systems with the proposed techniques employed incrementally.

In LESCEA, M3 and M4, the model is divided into 8(16) threads on a 2(4)-CPU platform, and each processor is mapped with 4 threads. While in M1 and M2, all blocks mapped to the same processor are merged into one thread.

The processor mapping and RTS policies of M1 are the same with that of LESCEA, and the ILP based processor mapping results of M2 and M4 are the same. Two pairs of cases are introduced to demonstrate the benefit of 3LHS architecture. The time cost and the percentage of each state are shown in Fig. 7(a) and (b). Compared to the experimental results of M1, the throughput of LESCEA is increased by 25.40%. The processor utilizations (the percentage of computation state) in M1 and LESCEA are 35.63% and 47.77% respectively. The throughput of M4 is raised by 34.61% compared with that of M2, and the processor utilizations in

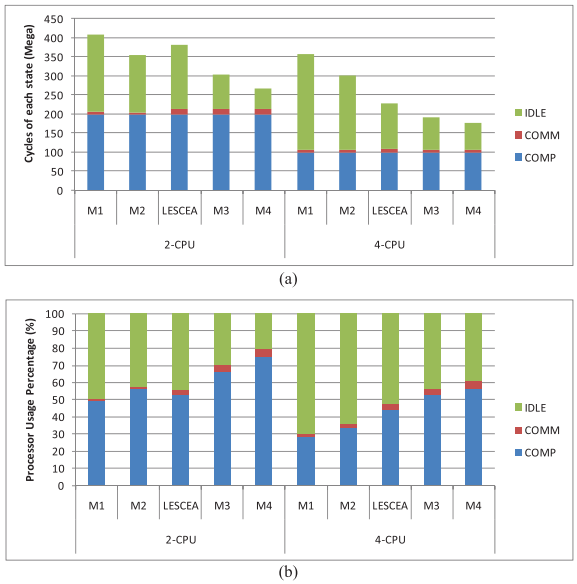


Fig. 7 Cycle count and percentage of each case.

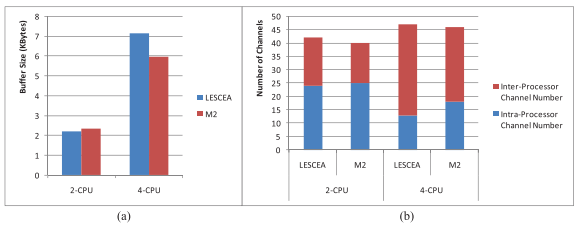


Fig. 8 Buffer size and number of channels of LESCEA and M2.

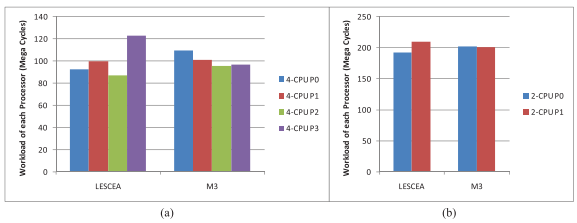


Fig. 9 Processor workload of LESCEA and M3.

M2 and M4 are 41.87% and 64.02% respectively. These results prove that the 3LHS architecture conduces to increasing processor utilization. Furthermore, the throughput increase between LESCEA and M4 is 34.02%, which is higher than that between M1 and M2 (14.89%). This also implies that the 2LHS architecture hinders the effectiveness of ILP based design-time mapping & scheduling.

The results of LESCEA and M3 illustrate the benefit of ILP based mapping. The inter-processor buffer sizes and channel numbers of each case are depicted in Fig. 8. With the ILP based mapping, the communication transfer cost, which is proportional to inter-processor buffer size, is decreased by 11.06%. Meanwhile, the communication setup cost, which is proportional to total number of channels, is decreased by 3.37%, and the total communication cost is

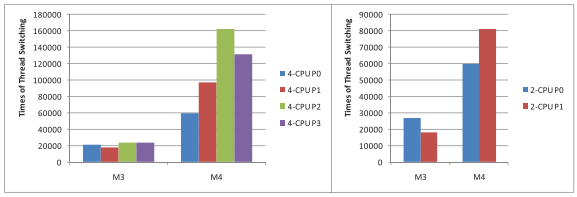


Fig. 10 Times of thread switching of M3 and M4.

reduced by 5.10%. Besides communication optimization, ILP based processor mapping also considers workload balance. The workload of each processor is shown in Fig. 9, the workload gap is reduced from 7.82% in LESCEA to 2.36% in M3. With these optimizations, the time of idle state is reduced by 36.83% and the system throughput is enhanced by 18.08%. These results indicate that ILP based mapping contributes to saving communication cost and raising system performance.

M3 and M4 are implemented to certify the effectiveness of ILP based DTS. The times of thread switching are depicted in Fig. 10. We find that even though the thread switching cost (proportional to the times of thread switching) is increased by 352.28%, the idle state cost is reduced by 23.76%, and the system throughput is increased by 8.91%. These indicate that the timing of thread switching in M4 is better than that of M3, and more frequent thread switching reduces the synchronization waiting cost, so the system throughput is enhanced.

Finally, the throughput of M4 is raised by 44.34% compared with that of M1, which indicates the total throughput enhancement of techniques proposed in this paper.

7. Conclusion

To improve the performance of MPSoC applications, we have proposed techniques to increase processor utilization and reduce communication cost during multithreaded code generation from Simulink models. The most important contribution is the combination of three-layered hierarchical software architecture and ILP based design-time mapping & scheduling, which can minimize workload gap, communication cost and synchronization cost. The experimental results on an H.264 baseline decoder demonstrate the effectiveness of our techniques.

As one of the future work, we will introduce other techniques to raise processor utilization to our design. One possibility is to apply pipeline techniques, such as software pipeline and communication pipeline, to further increase processor utilization so as to improve system performance.

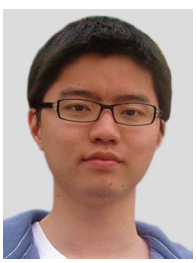
Acknowledgments

This work is supported in part by National Science Foundation of China under Grant No. 61100074.

References

[1] S.I. Han, S.I. Chae, L. Brisolara, L. Carro, R. Reis, X. Guerin, and A.A. Jerraya, "Memory-effient multithreaded code generation from simulink for heterogeneous mpsoc," Design Automation for Embedded Systems, vol.11, no.4, pp.249–283, 2007.
[2] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," 1976.
[3] E.A. Lee and T.M. Parks, "Dataflow process networks," Proc. IEEE, vol.83, no.5, pp.773–801, 1995.
[4] Simulink, Mathworks. http://www.mathworks.com
[5] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," Seventh International Conference on Application of Concurrency to System Design, 2007. ACSD 2007. pp.29–40, 2007.
[6] S. Kwon, Y. Kim, W.C. Jeun, S. Ha, and Y. Paek, "A retargetable parallel-programming framework for mpsoc," ACM Trans. Design Automation of Electronic Systems (TODAES), vol.13, no.3, p.39, 2008.
[7] J. Castrillon, R. Velasquez, A. Stulova, W. Sheng, J. Ceng, R. Leupers, G. Ascheid, and H. Meyr, "Trace-based kpn composability analysis for mapping simultaneous applications to mpsoc platforms," Proc. Conference on Design, Automation and Test in Europe, pp.753–758, European Design and Automation Association, 2010.
[8] A. Canedo, T. Yoshizawa, and H. Komatsu, "Skewed pipelining for parallel simulink simulations," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010, pp.891–896, 2010.
[9] T.h. Shin, H. Oh, and S. Ha, "Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph," Proc. 16th Asia and South Pacific Design Automation Conference, pp.165–170, 2011.
[10] S. Eyerman and L. Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," ACM SIGARCH Computer Architecture News, pp.362–370, 2010.
[11] R.A.Walker and S. Chaudhuri, "Introduction to the scheduling problem," IEEE Des. Test Comput., vol.12, no.2, pp.60–69, 1995.
[12] R.D. Blumofe and C.E. Leiserson, "Scheduling multithreaded computations by work stealing," J. ACM, vol.46, no.5, pp.720–748, 1999.
[13] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," Proc. Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp.21–28, 2005.
[14] S. Kumar, C.J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," ACM SIGARCH Computer Architecture News, pp.162–173, 2007.
[15] D. Sanchez, R.M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," ACM Sigplan Notices, pp.311–322, 2010.
[16] A. Bender, "Design of an optimal loosely coupled heterogeneous multiprocessor system," Proc. European Design and Test Conference, 1996. ED&TC 96. pp.275–281, 1996.
[17] Y. Yi, W. Han, X. Zhao, A.T. Erdogan, and T. Arslan, "An ilp formulation for task mapping and scheduling on multi-core architectures," Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09., pp.33–38, 2009.
[18] S.I. Han, S.I. Chae, and A.A. Jerraya, "Functional modeling techniques for effcient sw code generation of video codec applications," Proc. 2006 Asia and South Pacific Design Automation Conference, pp.935–940, 2006.
[19] K. Huang, S.i. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. Yan, S.l. Chae, L. Carro, and A.A. Jerraya, "Simulink-based mpsoc design flow: case study of motion-jpeg and h. 264," Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE, pp.39–42, 2007.
[20] S.I. Han, S.I. Chae, L. Brisolara, L. Carro, K. Popovici, X. Guerin, A.A. Jerraya, K. Huang, L. Li, and X. Yan, "Simulink-based heterogeneous multiprocessor soc design flow for mixed hardware/software refinement and simulation," Integration, the VLSI Journal, vol.42, no.2, pp.227–245, 2009.
[21] S.I. Han, X. Guerin, S.I. Chae, and A.A. Jerraya, "Buffer memory optimization for video codec application modeled in simulink," Proc. 43rd Annual Design Automation Conference, pp.689–694, 2006.
[22] L. Brisolara, S.i. Han, X. Guerin, L. Carro, R. Reis, S.I. Chae, and A. Jerraya, "Reducing fine-grain communication overhead in multithread code generation for heterogeneous mpsoc," Proc. 10th International Workshop on Software & Compilers for Embedded Systems, pp.81–89, 2007.
[23] C-SKY Inc. http://www.c-sky.com.
[24] S.I. Han, A. Baghdadi, M. Bonaciu, S.I. Chae, and A.A. Jerraya, "An effcient scalable and flexible data transfer architecture for multiprocessor soc with massive distributed memory," Proc. 41st Annual Design Automation Conference, pp.250–255, 2004.

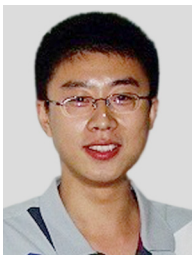
Kai Huang received the B.S.E.E. degree from Nanchang University, Nanchang, China, in 2002, and the Ph.D. degree in engineering circuit and system from Zhejiang University, Hangzhou, China, in 2008. From 2006 to 2006, he was a short-term Visitor with the TIMA Laboratory, Grenoble, France. From 2009 to 2011, he was also a Post-Doctoral Research Assistant with the Institute of VLSI Design, Zhejiang University. In 2010, he also worked as a Collaborative Expert in VERIMAG Laboratory, Grenoble, France. Since 2012, he has been an Associate Professor with the Department of Information Science and Electronic Engineering, Zhejiang University. His current research interests include embedded processors and SoC system-level design methodology and platforms.



Min Yu received the Bachelor degree from Department of Information Science & Electronic Engineering, Zhejiang University, China, in 2009. He is currently pursuing the Ph.D. degree in Institute of VLSI Design, Zhejiang University, China. His current research interests include performance estimation, high performance software exploration on multiprocessor and performance oriented automatic code generation on MPSoC.

Xiaomeng Zhang received the bachelor's degree from the College of Electrical Engineering, Zhejiang University, China, in 2013. She is currently pursuing the Ph.D. degree at the Institute of VLSI Design, Zhejiang University, China. Her current research interests include multiprocessor software exploration and multithread code generation.

Dandan Zheng received the PHD from Department of Information Science & Electronic Engineering, Zhejiang University, China, in 2009. She is currently a Research Assistant of College of Electrical Engineering at the University of Zhejiang.Her research interests are VLSI physical design of nano technology. Currently the main research is the low power physical design of MPSOC.



Siwen Xiu received his Bachelor of Electronic Science and Technology degree from Zhejiang University, Hangzhou, China, in 2009. He is currently pursuing his Ph.D from Institute of VLSI Design, Zhejiang University. His research interests include MPSoC Performance Estimation and Architecture Exploration, Multiprocessor Architecture Design, Digital System Design, etc.

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

Kai Huang received his Ph.D. degree under the supervision of prof. dr. Lothar Thiele in ETH Zurich Switzerland 2010. He received B.Sc. and M.Sc. degree in computer science at Fudan University China 1999 and Leiden University The Netherlands 2005, respectively. His research interests include techniques for the analysis, design, and optimisation of embedded systems. He received Chinese Government Award for Outstanding Self-Financed Students Abroad 2010, Best Paper Awards from Int'l Symposium on Systems, Architectures, Modeling and Simulation (SAMOS), and General Chairs' Recognition Award For Interactive Papers in the IEEE Conf. on Decision and Control (CDC) in 2009.

Zhili Liu received the Master of Engineering degree in electronics and Communications Engineering, Hangzhou Dianzi University, China, in 2007. His current research interests include the application of embedded CPU and high performance low power software exploration.

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