Scalable-Grain Pipeline Parallelization Method for Multi-core Systems

Peng Liu, Chunming Huang, Jun Guo, Yang Geng, Weidong Wang, Mei Yang

Scalable-Grain Pipeline Parallelization Method for Multi-Core Systems*

Peng Liu[1], Chunming Huang[1,2], Jun Guo[1],
Yang Geng[1], Weidong Wang[1], and Mei Yang[1,3]

[1] Department of ISEE, Zhejiang University, Hangzhou 310027, China
liupeng@zju.edu.cn, guojun007@zju.edu.cn, doriru.simon@gmail.com, wdwang@zju.edu.cn

[2] Baidu Co. LTD., Shanghai 201203, China
hcm198611@yahoo.com.cn

[3] Department of ECE, University of Nevada Las Vegas, Las Vegas, USA
meiyang@unlv.edu

Abstract. How to parallelize the great amount of legacy sequential programs is the most difficult challenge faced by multi-core designers. The existing parallelization methods at the compile time due to the obscured data dependences in C are not suitable for exploring the parallelism of streaming applications. In this paper, a software pipeline for multi-layer loop method is proposed for streaming applications to exploit the coarse-grained pipeline parallelism hidden in multi-layer loops. The proposed method consists of three major steps: 1) transform the task dependence graph of a streaming application to resolve intricate dependence, 2) schedule tasks to multiprocessor system-on-chip with the objective of minimizing the maximal execution time of all pipeline stages, and 3) adjust the granularity of pipeline stages to balance the workload among all stages. The efficiency of the method is validated by case studies of typical streaming applications on multi-core embedded system.

1 Introduction

With the continuous advance of semiconductor technology, the enormous number of transistors available on a single chip enables the integration of tens or hundreds of processing cores on a multiprocessor system-on-chips (MPSoCs). These processing cores could be homogeneous or heterogeneous, such as processors, digital signal processor cores, memory blocks, etc. To efficiently utilize these parallel resources available on an MPSoC, one challenge is how to parallelize the legacy sequential programs. However, most research and development efforts in MPSoCs are on the hardware architectural side. Research in application program parallelization and parallel programming for MPSoCs is far more behind.

* This work is supported in part by National Natural Science Foundation of China under the grants 60873112 and 61028004, National High Technology Research and Development Program of China under the grant 2009AA01Z109, and the Huawei Innovation Research Program under the grant YJCB2011033RE.

Existing efforts to exploit pipeline parallelism in C programs are mostly fine-grained [10, 12]. Some approaches partition individual instructions across processors, such as the decoupled software pipeline (DSWP) method [10], while others are dedicated to parallelizing scientific and numerical applications, such as DOALL [1] and DOACROSS [4]. The HELIX project [2] is a generation of the DOACROSS scheme. HELIX satisfies only the necessary loop-carried data dependences. The synchronization required for loop-carried dependences is implemented using a per-thread memory area which resides in the system's shared memory. For streaming applications, these techniques are not sufficient, because the pipeline parallelism in streaming applications is coarse-grained and more complex than that in scientific/numerical applications. To resolve intricate dependency, the fine-grained methods may usually merge those tasks which take part in a dependence cycle. Other work on integration of parallelization techniques [7, 11, 17] includes the speculation DSWP [17], which mainly focuses on cutting dependences caused by loop conditional statements but cannot cut the inter-iteration dependences, and the parallel-stage DSWP [11] which integrates DSWP with DOALL. However, these approaches are not suitable for exploiting course-grained parallelism and cannot manage multi-layer loop structures.

Unfortunately, currently no method can solve the problem of extracting coarse-grained pipeline parallelism well [13]. The method to exploiting coarse-grained pipeline parallelism in C programs proposed in [16] is a language-extension approach, which imposes the burden of parallelism extraction on programmers. The Paralax [18] is a compiler-based parallelization framework which focuses on the dependence analysis but lacks transformation for multi-layer loops. MAPS [3] is a framework for semi-automatic parallelism extraction from sequential legacy code and extended to support parallel dataflow programming.

Programmers have been familiar with sequential programming languages like C for a long time. Instead of using a language/model, an evolutionary parallelization methodology for sequential codes will be more significant. Streaming applications represent a large set of MPSoC applications, such as video, audio, cryptographic, wireless baseband processing, etc. These programs are characterized by heavy use of pointers, multi-layer loop structures, and streaming data input. The parallelism within these programs usually is implicit [13]. Extracting the pipeline parallelism hidden in loops becomes very critical for the rich loop control-flow constructions in the C programs of streaming applications. Our focus is on exploiting coarse-grained pipeline parallelism in the C programs of streaming applications, which are characterized by multi-layer loop structures with intricate dependence relations and fixed data flow.

In this paper, a scalable-grain pipeline parallelization (SPP) method is proposed, which exploits the coarse-grained pipeline parallelism hidden in multi-layer loops existing in streaming applications. These applications are described by a block diagram with a fixed flow of data and the regular communication pattern. We exploit coarse-grained pipeline parallelism by using the source program transformation for embedded applications that can overcome the traditional barriers. The proposed method first resolves intricate dependency by transforming the task dependence graph, which is then scheduled to the target MPSoC to minimize the maximal execution time of a pipeline stage. The experimental results of the parallel programs of the applications on an eight-core platform justify that this method is efficient in parallelizing C sequential programs.

The rest of this paper is organized as follows. The problem to be solved is formulated in Section 2. Section 3 presents the framework of the proposed method. Section 4 presents the experimental result. Section 5 concludes.

2 Problem Formulation

The MPSoC under consideration is modeled as an architectural characterization graph. For a streaming application, its task dependence graph is extracted from the source code of the application.

Definition 1. *An MPSoC architectural characterization graph (ACG), $ACG = (P, L)$, is a undirect graph, where a vertex $p_i \in P$ represents one processing element (PE) in MPSoC, with $m(p_i)$ denoting the memory space of p_i and $t_p(p_j)$ denoting the type of p_i, an arc $l_{ij} \in L$ represents the link between p_i and p_j, with $r(l_{ij})$ denoting the link bandwidth. An MPSoC architecture has $r(l_{ij}) = R$, $\forall l_{ij} \in L$, and $m(p_i) = M$, $\forall p_i \in P$.*

Definition 2. *A task dependence graph (TDG), $TDG = (V, E, W_v, W_e)$, is a directed graph, which represents the dependence relation of a program, where **V** represents the set of nodes, each representing a task of the program, **E** represents the set of edges, each representing the dependence relation between two tasks. The sets W_v and W_e represent the properties of nodes in **V** and edges in **E**, respectively. For each $n_i \in V$, the predecessor node set of n_i is defined as $pre(n_i)$, and the successor node set of n_i is defined as $succ(n_i)$. For an edge $e_{ij} = (n_i, n_j) \in E$, then $n_i \in pre(n_j)$ and $n_j \in succ(n_i)$. There are two types of dependences: data and control dependences. Data dependences are caused by data transfer. Control dependences are evoked by the conditional statements or loop statements.*

These are four types of nodes in set **V**: *Control node set* V_C, *Branch node set* V_B, *Loop node set* V_L, and *Ordinary node set* V_O. The edges in **E** are sorted into three groups: *Control edge set* E_C, *Inter-iteration edge set* E_I, and *Ordinary edge set* E_O. The properties of W_v and W_e are listed in Table 1.

Definition 3. *The dependence distance d_{dep} indicates the number of iterations between two loop nodes forming the inter-iteration dependences. $\forall e_{ij} \in E_I$, $n_i, n_j \in V_L$, if n_i is the kth iteration of the loop and n_j is the lth iteration of the loop, then $d_{dep}(e_{ij}) = l - k$.*

Definition 4. *A strongly-connected component (SCC) of a graph is a maximal strongly connecter sub-graph, in which there exists a path from each vertex to every other vertex in the sub-graph. A directed graph is acyclic if and only if it has no SCC. Given two nodes n_i and n_j in a TDG, the SCC distance $d_s(n_i, n_j)$ indicates the number of SCCs between SCCs.*

Table 1. Properties of Node and Edge

W_v	
l	$\forall n \in V$, $l(n)$ denotes the type label of node n
m	$\forall n \in V$, $\forall p \in P$, $m(n)$ denotes the memory requirement of task n
s	$\forall n \in V$, $s(n)$ denotes the number of strongly-connected components that task n belongs to
i_b	$\forall n \in V_B$, $i_b(n)$ denotes the branch information of task n
i_l	$\forall n \in V_L$, $i_l(n)$ denotes the loop information of task n
t	$\forall n \in V$, $\forall p \in P$, $t(n, p_i)$ denotes the execution time of task n on the PE p_i. If $p_i, p_j \in P$ satisfy $t_p(p_i) = t_p(p_j)$, then $t(n, p_i) = t(n, p_j)$
W_e	
c	$\forall e_{ij} \in E \cap e_{ij} \notin E_C$, $c(e_{ij})$denotes the data traffic amount of edge e_{ij}
d_{dep}	$\forall e_{ij} \in E_I$, $d_{dep}(e_{ij})$ denotes the dependence distance of edge e_{ij}

Definition 5. *A TDG after transformation, $TDG' = (V', E', W'_v, E'_e)$, is an acyclic directed graph, in which each SCC in the TDG is merged into a single vertex and the control dependences are removed. V' is the union of vertex sets V'_B, V'_L, and V'_O. The edge set E' does not include any branch/loop control edge, which is the union of edge sets E'_I and E'_O. The node set W'_v and edge set W'_e properties are defined as the same as the set W_v and W_e in TDG.*

Using these definitions, the problem to be solved can be formulated as below. Given an application $TDG = (V, E, W_v, W_e)$ and a target MPSoC platform $ACG = (P, L)$, find a transformation $T : TDG \rightarrow TDG'(V', E', W'_v, W'_e)$ and a scheduling function $S : V' \rightarrow P$, so that the total execution time of the parallel program is minimized. As streaming applications are executed in pipelined way, the pipeline execution time is bounded by the slowest stage. Hence, the objective is minimizing the maximum runtime (including the execution and communication time) among all pipeline stages, i.e.,

$$\begin{aligned} min(max(&\sum_{i_1} t(n'_{i_1}, p_1) + \sum_{i_1} \sum_{n'_{k_1} \in succ(n'_{i_1})} (c(e_{i_1 k_1})/R), \\ &\sum_{i_2} t(n'_{i_2}, p_2) + \sum_{i_2} \sum_{n'_{k_2} \in succ(n'_{i_2})} (c(e_{i_2 k_2})/R), ..., \\ &\sum_{i_{|p|}} t(n'_{i_{|P|}}, p_{|P|}) + \sum_{i_m} \sum_{n'_{k_{|P|}} \in succ(n'_{i_{|P|}})} (c(e_{i_{|P|} k_{|P|}})/R))) \end{aligned} \tag{1}$$

subject to

$$S(n'_{i_j}) = p_j \tag{2}$$

$$\forall p_j \in P, \forall n'_{k_j}, n'_{i_j} \in V', \forall e'_{i_j k_j} \in E', \sum_S m(n'_{i_j}) \leq m_p(p_j) \tag{3}$$

where $\mathbf{R}$ is the bandwidth of each link between processing elements in $\mathbf{P}$, each sum term calculates the execution time of pipeline stage i, including the execution time of the tasks scheduled on processor p_i and communication time for the traffic send to the successor tasks. Condition 2 restricts that one task is only scheduled to one processing element. Condition 3 ensures that the total memory consumption of all tasks that are assigned to processing elements that should not exceed the memory space of p_j.

3 Framework

Given the C program of a streaming application, the following steps as shown in Figure 1 will be performed under the framework of parallelizing sequential program:

Step1: Select the computing hot-spot region (CHR), which is the procedure with the maximum execution time, of the whole program through dynamic profiling. The source code in computing hot-spot region is scanned to find the relations of parameters, pointers with data structure, data dependence and control dependence between statements through a top-down method using in-house developed tools. Then build the TDG of the computing hot-spot region. The main procedure of a program is usually chosen as the initial computing hot-spot region.

Step2: Transform the TDG to a directed acyclic graph TDG' to eliminate the control dependences and inter-iteration data dependences **(P1)**.

Step3: Allocate and schedule each node in the TDG' to a proper processing element in the ACG to form a thread **(P2)**. A pipeline stage consists of one or more threads.

Step4: Apply split/merge parallelizing technique to balance the workload among all pipeline stages **(P3)**.

Step5: Evaluate the execution time of the scheduled pipeline. The objective of scheduling is to minimize the maximum runtime among all pipeline stages. To further reduce the execution time, choose the bottleneck stage with the largest runtime to be the new computing hot-spot region, then jump to step 1.

Step6: Based on the final scheduling result, generate the parallel program for the application and compile the codes to get the executable files for the hardware platform.

The sub-problems dependence transformation **(P1)**, task scheduling **(P2)**, and workload balancing **(P3)** will be defined and the solutions to these sub-problems will be discussed in the next subsections.

3.1 Dependence Transformation

The sub-problem P1 is defined as: Given the TDG of an application, find a transformation method $T : TDG \rightarrow TDG'$, which removes the redundant dependences to make the output graph satisfying the following requirements: 1) No control dependence edge. 2) No dependence cycle, i.e. no SCC composed of more than one node. Thus TDG' is an acyclic directed graph with only data dependences.

Branch Control Dependence For branch control dependence, it is important to focus on the mutually exclusive branch tasks. The tasks in different branch paths controlled by conflicting conditions are mutually exclusive. Once a branch

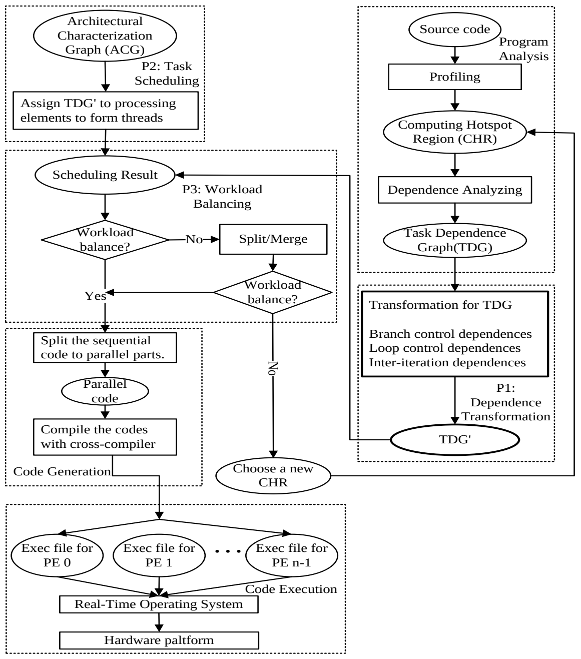

Fig. 1. Framework of parallelizing sequential code.

path is selected to execute, the other branch path will not be executed. For every branch task, the branch exclusive array is updated according to the following steps.

First, scan the branch information i_b of every branch node of which the type label $l(n)$ is $n{\in}V_C$, or $n{\in}V_C{\cap}V_B$, or $n{\in}V_B{\cap}V_L$. Assume $n_j \in V_B$, $i_b(n_j).branch\_level = N$. Search $branch\_label[i]$ and $branch\_condition[i]$ of $i_b(n_j)$ starting from $i = 0$.

Second, traverse the ith level branch control nodes outgoing control edges and find the other nodes controlled by it. Check the $branch\_label$ and $branch\_condition$ of these nodes with those of node n_j to determine whether they are mutually exclusive. If so, add these nodes into the $branch\_exclusive$ array of node n_j.

Third, the ith level branch control node is combined with the branch node n_j. Let $i = i + 1$, repeat steps 2 and 3 until $i = N$.

Finally, after finishing process all the branch nodes, delete all the branch control nodes in TDG.

Through this transformation, each branch control node is merged with its successive branch nodes in different branch paths to form new nodes.

Loop Control Dependence Due to their repeating characteristics, loops are the important structures to explore for parallelizing a program. According to the conventional methods [3, 12, 14] , all tasks in a loop should be merged into one large task to eliminate the loop control dependences. Indubitably this will impede exploiting the parallelism of the tasks in the loop. In addition, the large task is likely to become the bottleneck of parallelization.

We apply the speculation technique to remove the loop control dependences. Provided that the loop runs fixed times or many iterations, the loop is regarded as biased. As a matter of fact, through profiling, many loop behaviors of streaming applications are highly predictable. The following steps are performed to resolve loop control dependences.

First, check the values of *loop_level*, *loop_label*, and *loop_num* to see whether this loop is biased, i.e., if there exists $loop\_num[i] > 0$ (ith level loop is biased). The loop control dependences of this type of loop are seen as highly predictable, and they are chose to speculate.

Second, remove the selected ith level loop control edges. Insert code to detect the mis-speculation.

– Insert an unconditional branch statement, such as while (true), in all the loop tasks that are dependent on this loop control task. If $loop\_num[i]$ is known by profiling, a counter is inserted in every loop task.
– A mis-speculation procedure needs to be inserted in the last task of this loop, when $loop\_num[i]$ is a variable. The mis-speculation detection is achieved by copying and updating the computing results of last iteration of the current iteration which is running. If the predicted loop is not taken, the mis-speculation procedure is responsible for recovering data and jumping to the loop exit path.

Resolving Inter-iteration Dependence The mechanism for resolving inter-iteration dependences is as follows. First, on the transformed TDG, provisionally remove all the inter-iteration dependence edges with their dependence distance greater than 1. Next, merge tasks which belong to a SCC in the transformed TDG into one SCC node. Then, identify the SCC distance d_s of each task.

We can identify the set of inter-iteration edges that satisfy any one of the following rules. Assume an inter-iteration dependence edge e_{ij} exists, e_{ij} representing the dependence from task n_i to task n_j.

– If $d_{dep}(e_{ij}) > d_s(n_i, n_j) \geq 1$, the inter-iteration dependences can be ignored by the pipeline stages built on current SCCs. When the next stage is going to be executed, the inter-iteration dependence relation it relied on has been already satisfied. Thus, removing e_{ij} does not affect the parallel scheduling on the current SCCs.
– If $d_{dep}(e_{ij}) = 1$, check the $d_s(n_i, n_j)$ of this edge. If $d_s(n_i, n_j) = -1$, it means task n_i and n_j are in the same SCC and e_{ij} can be ignored. If $d_s(n_i, n_j) = 0$, actually this is the special case, e_{ij} can be removed.

The inter-iteration dependence edges that satisfy any of the above rules are regarded as redundant and are allowed to be removed. All other provisionally inter-iteration dependences are not allowed to be removed and are added back to the TDG before invoking the transformation.

Finally, grouping the tasks and data dependence edges which are involved in a dependence cycle, then the acyclic directed graph TDG' has been built.

3.2 Scheduling

The scheduling sub-problem is to find a scheduling function $S : V' \rightarrow P$ with the objective of minimizing the maximum run time among all pipeline stages. The constraint of the memory space at each processor need be be satisfied. A heuristic scheduling approach is used here which allocates the tasks in TDG' to the processors such that the workload on each processor is kept balanced.

Two-level priorities are defined to indicate the order of scheduling. A *task batch* is the group of tasks which can be executed in parallel according to their dependence relations. Each task batch has a queue structure to load the tasks, and a corresponding *batch priority* to indicate the execution order of the task batch. Every task in the task batch has a task priority to indicate its order in the same task batch in case of resource confliction.

The batch priority (TP) of a task batch is set according to the dependence relations through a breadth first search method. The smaller its batch priority value, the earlier the task batch can be executed. Once a task is assigned to a processing element, the batch priority values of its successor tasks will be updated. Thus, the batch priority can be seen as a dynamic priority.

The task priority of a task n_i in a task batch queue is defined as a linear function of three major factors. The larger its task priority value, the higher priority of the task is in the task batch. Given $TDG' = (V', E', W'_v, W'_e)$, $ACG = (P, L)$, $\forall n'_i \in V'$

$$TP(n'_i) = \alpha \times bl(n'_i) + \beta \times DMem(n'_i) + \gamma \times SMem(n'_i) \quad (4)$$

where the major factors are defined below.

- $bl(n'_i)$: the bottom level of n'_i is the length of the longest path starting from n'_i [14]. If the $bl(n'_i)$ is high, it implies that n'_i is a critical task and should be given a high priority corresponding to a larger task priority.
- $DMem(n'_i)$: the consumption of the communication buffers for task n'_i. The $DMem(n'_i) = \sum\limits_j c(e'_{ij})$, where edge e'_{ij} originates from n_i, $c(e'_{ij})$ represents the communication traffic of the dependence edge e'_{ij}. If $DMem(n'_i)$ is high, it means that the inter-processor communication traffic may be large. The task with grater value of dynamic memory should be allocated to the processor with a higher priority.
- $SMem(n'_i)$: the memory requirements of instruction and static data, which are obtained at the profiling time.
- The scale coefficients of α, β, and γ are used to normalize the elements. The three coefficients are defined as:

$$\alpha = 1/len(cp), \beta = 1/(R \times min(t(n'_i, p_j))), \gamma = 1/M$$

ALGORITHM 1: Parallel Scheduling (TDG', ACG)

```
Input: Graph $TDG'(V', E', W'_v, W'_e)$, and target MPSoC $ACG(P, L)$.
Output: Schedule tasks to processors in ACG.
Part One:
Calculate the batch priority of each node in $V'$ and insert each node into the
associated task batch queue through a breadth first search procedure;
for each task batch $b_j \in B$ do
    Calculate the TP for each task in $b_i$ according to Equation 4;
    Sort tasks in a task queue in the decreasing order of TP as $n'_{\pi_0}$, $n'_{\pi_1}$,..., $n'_{\pi_{|b_j|-1}}$;
end
Sort the task batches in a task batch queue in the increasing order of batch priority
as $b'_{\pi_0}$, $b'_{\pi_1}$,..., $b'_{\pi_{|b_j|-1}}$;
Part Two:
for ($k = 0; k < |B|; k++$) do
    for ($l = 0; l < |b_k|; l++$) do
        PESelect(task $n'_{\pi_l}$ in $b_{\pi_k}, P$);
        Delete the task $n'_\pi$ from $V'$;
        for each task $n'_i \in succ(n'_{\pi_l})$ do
            Reassign $n'_i$ into a proper task batch $b_j \in B$, according to dependence
            relations in graph $TDG'$;
        end
    end
end
```

where $len(cp)$ indicates the length of the critical path *(cp)*, which is longest path in TDG', R represents the link bandwidth between processors, M represents the memory space of processing element.

The scheduling algorithm is divided into two parts: 1) sort the tasks in the TDG' in task batches and calculate the batch priorities and task priorities, and 2) assign each task to a proper processing element to form a thread according to both batch priority and task priority. Each processing element is responsible for one thread at one time. This approach focuses on good workload balance, and also takes into account mutually exclusive branch tasks identified in earlier phases and data locality optimization. Algorithm 1 outlines the scheduling algorithm. For each task it is necessary to determine which processing element the task should be scheduled to and the time slot the task will be execute on the processing element. The principle of processor selection is according to (5).

$$\begin{aligned} &\forall n'_i \in V', \forall p_j \in P \\ &AvailableFactor(n'_i, p_j) = \quad \lambda \times DL(n'_i, p_j) - max(DRT(n'_i), PEAT(n'_i, p_j)) - t(n'_i, p_j) \end{aligned} \tag{5}$$

where $DL(n'_i, p_j)$ represents the data locality factor which indicates the data reuse time and reduction of communication time, when assigning task n'_i to processing element p_j, λ represents the proportion of DL adjusting factor, $DRT(n'_i)$ represents the data ready time of task n'_i, $PEAT(n'_i, p_j)$ represents the available time of the processing element p_j for task n'_i, and $t(n'_i, p_j)$ represents the execution time of task n'_i on the chosen processing element p_j.

In selecting the suitable processing element for a task, the processing element with the maximum value of *AvailableFactor* is chosen. The larger this value is, the higher available level the processing element has. As described above, the principle for selecting a proper processing element also takes into account the mutual exclusive of the branch tasks. The main difference between scheduling of the branch tasks and that of the ordinary tasks is the computation of the *AvailableFactor* as mentioned before. The computing about $PEAT(n'_i, p_j)$ and $t(n'_i, p_j)$ for the branch tasks should take more attention on the mutual exclusive property.

3.3 Workload Balancing

In this step, the basic software pipeline technique is applied in conjunction with the split and merge technique to further balance the workload. Given the initial scheduling result, split and merge the stages to further reduce the execution time of the computing hot-spot region. It is achieved by assigning more threads to large stages and merging small stages into one stage so that the workload balance and efficiency to the processing elements can be improved.

If the outer-most loop of a program as a computing hot-spot region is split into several task sets in a pipeline style, then each task set is called a *stage*. Usually one stage can be assigned to one or more threads. If the inner loop of a program or a stage of a computing hotspot region is spilt task sets in a pipeline style, then each task set is called a sub-stage. Usually the stage consists of one or more sub-stages. If the minimum execution cost among processing elements account of less than 50% of the maximum one, or the estimated pipeline execution time does not satisfy the user's requirement, the scheduling result is regarded as workload imbalanced, then the granularity of the pipeline stages need be adjusted.

A heuristic approach is used here which operates in three steps. First, check whether the loop in the largest stage can be split into several independent iterations without further profiling. If so, split the largest stage and apply DOALL to assign the iterations to different threads. Second, merge those small stages into one processing element or insert them into the spare time of other working processing elements. Third, after the simple split/merge processing, the stage with largest runtime is selected as the new CHR. Then repeat the steps of transformation and scheduling as shown in Figure 1.

4 Experimental Results

To evaluate speedups of our method, the parallelization framework in Figure 1 is applied to the sequential C programs to produce the parallel codes. Table 2 lists the software tool chains that we have used through the parallelization. Our experiments are conducted on the multi-FPGA-based networks-on-chip emulation

Table 2. Software Tool

Program Profiling	
valgrind [9]	get information about function call and computing cost of sequential code
gcov [6]	get the branch selecting information
Dependence Analysis–developed in-house	
VarAnalyzer	analysis the information of variables in a function, including type, size, define-use chain, and life time.
DepViz	analysis the data dependences and control dependences among functions.
Cross Compiler	
mipsisa32el-gcc	compiler for MIPS32 compatible architecture
compiler option	-O0 -march=4rkc -nostdinc -g -fno-delayed-branch

Table 3. Characteristics of Benchmark

AES	
Characteristic	128-bit plaintext and 128-bit encrypt key
T264 Decoder	
Version	0.14
Sequence	forman, akiyo, container
Input Size	QCIF, 176x144, 300 frames
Characteristic	99 macroblocks/frame, two B frames between P frames, no rate control, deblock, CAVLC entropy coding

platform [8]. Eight 32-bit compatible MIPS4Kc RISC cores were instantiated on the platform, which can be configured to 2, 4, and 8 cores. Each processor core is attached to the advanced microcontroller bus architecture bus in order to connect with peripheral memory, communication, and debugging interfaces. A 3x3 mesh of routers is used to interconnect RISC cores, which implemented with deterministic routing algorithm. Each router consists of five input/output ports, 16-depth first-in first-out buffers, and two virtual channels for each port. The proposed method is applied to two realistic streaming programs: Advanced Encryption Standard (AES) [5] and T264 decoder [15]. The characteristics of programs are shown in Table 3.

T264 decoder [15] is one of the open source video codecs based on H.264 standard. The T264 decoder carries out the complementary processes of decoding, inverse transform, and reconstruction to produce a decoded video sequence. It processes frames of video sequence in units of a macroblock and each macroblock of 16x16 pixels. The outer program loop is responsible for decoding each frame of the video sequence, while the inner loop deals with every macroblock of a frame. T264 *dec_parse_slice* function is responsible for decoding the frames. It is also the computing hot-spot region. In the T264 *dec_parse_slice* function, there is an inner-loop to do the decoding work of macroblocks in the frame. Actually there are complicated dependences among macorblocks and frames. The TDG and TDG' of T264 *dec_parse_slice* function are shown in Figure 2. The arcs for inter-iteration dependences are represented with red solid lines. Control dependences are represented with dashed lines. Data dependences are represented with black solid lines. Every node is recorded the global variable, local variable, and extern parameters and their sizes. The node *for* stands for the loop control node, which computation and storage cost are not accounted.

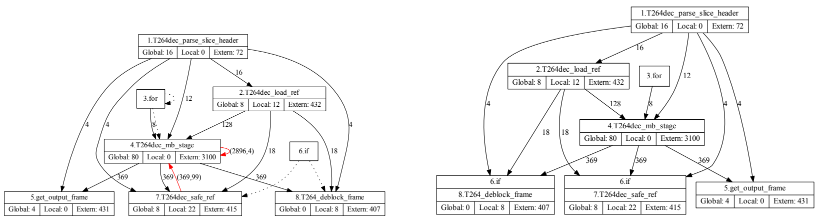

Fig. 2. TDG and TDG' of T264 *dec_parse_slice* function.

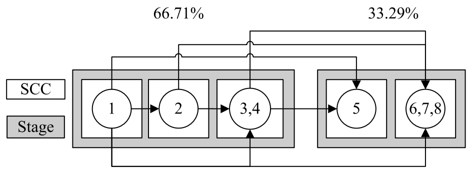

Fig. 3. Scheduling for T264 *dec_parsc_slice* function.

If we use conventional parallelization method to deal with the multi-layer loop structure which makes the inner-loop one large SCC or one large task, the parallelism in the loop cannot be exploited fully. And when the tasks are assigned to stages, the workload balance will be very poor as shown in Figure 3. The T264 *dec_parse_slice* is split into two stages, each is assigned to one thread. The first stage is the inner loop which does the decoding work for macro blocks, while the second stage extracts the prediction information from the current frame for the later frames.

Using the proposed method, the intricate dependences are resolved and the inner-loop is split into 4 sub-stages. At the same time, we partition the data transferred from stage 1 to stage 2 to make sure that each thread of stage 2 is only given the ownership of a dedicated block of data. Each stage 2 thread follows one part of the thread of stage 1 as shown in Figure 4. With one thread assigned to stage 1 and 3 threads assigned to stage 2, the speedup can be improved further. Through analyzing the inner loop we find it that the frame-decoding operation can be partitioned into blocks which consist of several macroblocks. So we gather the dependent macroblocks into one thread and schedule them on the eight-core platform. We assign 8 threads to T264 decoder program which are in a pipeline style, and each thread is mapped to a processing element.

Figure 5 shows that through the proposed method, a speedup can be achieved on the two case studies. The baseline is the conventional parallelization method which merges control dependences into one task and only explores the parallelism of the outer program loops. The 4-core and 8-core represent the parallelization result on a four cores and eight cores platform respectively. Under the proposed parallelization method, the speedup is 5.48x for T264 decocder and 5.12x for

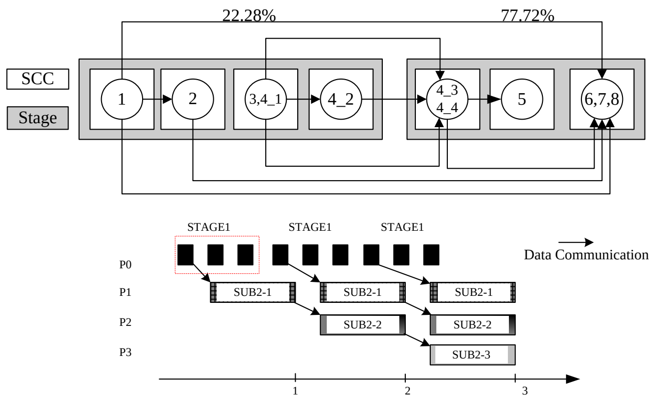

Fig. 4. (a) Scheduling for T264 decoder at inner loop level, (b) Mapping result for stages to processing element. The SUB2-1 represents the first thread of stage2.

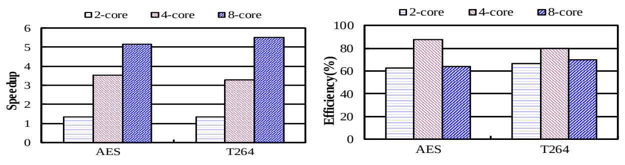

Fig. 5. Speedup and efficiency of the parallelization result.

AES program respectively. As shown in the figure, the parallel scheme on four cores platform makes full use of the hardware of which the efficiency is more than 80%. Since the workload balance of the parallelization scheme is affected by the characteristic of application, the efficiency of processors on eight cores platform is smaller than on four cores platform.

5 Conclusions

In this article, we have proposed method for embedded system applications to exploit the coarse-grained pipeline parallelism hidden in multi-layer loops. The method first resolves intricate dependency by transforming the task dependence graph, which is then scheduled to the target MPSoC in parallel to minimize the maximal execution time of a pipeline stage. The parallel scheme is adjusted in a heuristic way to further improve performance. The experimental results of two typical applications confirm the efficiency of the method in practical. The method will be applied to parallelizing other embedded applications on multicore embedded systems. The sequential program parallelization needs to be integrated with these techniques, such as eliminating redundant dependences, task scheduling, and independent multi-threading to extracting the pipeline parallelism.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
2. Campanoni, S., Jones, T., Holloway, G., Reddi, V.J., Wei, G.Y., Brooks, D.: HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In: Proc. Int'1 Symp. on Code Generation and Optimization, pp. 84–93 (2012)
3. Ceng, J., Castrillón, J., Sheng, W., Scharwächter, H., Leupers, R., Ascheid, G., Meyr, H., Isshiki, T., Kunieda, H.: MAPS: An Integrated Framework for MPSoC Application Parallelization. In: Proc. Design Automation Conf., pp. 754–759 (2008)
4. Cytron, R.: Doacross: Beyond Vectorization for Multiprocessors. In: Proc. Int'l Conf. on Parallel Processing, pp. 836–844 (1986)
5. Specification for the Advanced Encryption Standard (AES) (2001). `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`
6. Gcov - Using the GNU Compiler Collection (GCC) (2012). `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`
7. Huang, J., Raman, A., Jablin, T.B., Zhang, Y., Hung, T.H., August, D.I.: Decoupled Software Pipelining Creates Parallelization Opportunities. In: Proc. Int'l Symp. on Code Generation and Optimization, pp. 121–130 (2010)
8. Liu, Y., Liu, P., Jiang, Y., Yang, M., Wu, K., Wang, W., Yao, Q.D.: Building a Multi-FPGA-based Emulation Framework to Support Networks-on-Chip Design and Verification. International Journal of Electronics **97**(10), 1241–1262 (2010)
9. Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 89–100 (2007)
10. Ottoni, G., Rangan, R., Stoler, A., August, D.I.: Automatic Thread Extraction with Decoupled Software Pipelining. In: Proc. Int'l Symp. on Microarch., pp. 105–116 (2005)
11. Raman, S., Ottoni, G., Rangan, A., Bridges, M.J., August, D.I.: Parallel-stage Decoupled Software Pipelining. In: Proc. Int'1 Symp. on Code Generation and Optimization, pp. 114–123 (2008)
12. Rangan, R., Vachharajani, N., Vachharajani, M., August, D.I.: Decoupled Software Pipelining with the Synchronization Array. In: Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques, pp. 177–188 (2004)
13. Ryoo, S., Ueng, S.Z., Rodrigues, C.I., Kidd, R.E., Frank, M.I., Hwu, W.M.W.: Automatic Discovery of Coarse-grained Parallelism in Media Applications. Transactions on High-Performance Embedded Architectures and Compilers **1**, 194–213 (2007)
14. Sinnen, O.: Task Scheduling for Parallel Systems. John Wiley & Sons (2007)
15. T264 Decoder (2005). `http://sourceforge.net/project/t264`
16. Thies, W., Chandrasekhar, V., Amarasinghe, S.: A Practical Approach to Exploiting Coarse-grained Pipeline Parallelism in c Programs. In: Proc. Int'l Symp. on Microarch., pp. 356–369 (2007)
17. Vachharajani, N., Rangan, R., Raman, E., Bridges, M.J., Ottoni, G., August, D.I.: Speculative Decoupled Software Pipelining. In: Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques, pp. 49–59 (2007)
18. Vandierendonck, H., Rul, S., Bosschere, K.D.: The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In: Proc. Int'l Conf. on Parallel Arch. and Compilation Techniques, pp. 389–400 (2010)