A List Scheduling Heuristic with New Node Priorities and Critical Child Technique for Task Scheduling with Communication Contention

Pengcheng Mu, Jean François Nezan, Mickaël Raulet

A List Scheduling Heuristic with New Node Priorities and Critical Child Technique for Task Scheduling with Communication Contention

Pengcheng Mu, Jean-François Nezan, and Mickaël Raulet

Abstract Task scheduling is becoming an important aspect for parallel programming of modern embedded systems. In this chapter, the application to be scheduled is modeled as a Directed Acyclic Graph (DAG), and the architecture targets parallel embedded systems composed of multiple processors interconnected by buses and/or switches. This chapter presents new list scheduling heuristics with communication contention. Furthermore we define new node priorities ***(top level and bottom level)*** to sort nodes, and propose an advanced technique namely ***critical child*** to select a processor to execute a node. Experimental results show that the proposed method is effective to reduce the schedule length, and the runtime performance is greatly improved in the cases of medium and high communication. Since the communication cost is increasing from medium to high in modern applications like digital communication and video compression, the proposed method is well-adapted for scheduling applications over parallel embedded systems.

1 Introduction

The recent evolution of digital communication and video compression applications has dramatically increased complexities of both the algorithm and the embedded system. To face this problem, System-on-a-Chip (SoC), which embeds several cores (e.g. multi-core DSPs) and several hardware accelerators (e.g. Intellectual Properties), becomes the basic element to build complex multiprocessor embedded sys-

Pengcheng Mu
Ministry of Education Key Lab for Intelligent Networks and Network Security, School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, P.R. China, e-mail: pengchengmu@gmail.com

Jean-François Nezan · Mickaël Raulet
IETR/Image and Remote Sensing Group, CNRS UMR 6164/INSA Rennes, 20, avenue des Buttes de Coësmes e-mail: {jnezan, mraulet}@insa-rennes.fr

tems. This kind of system is also known as Multiprocessor System-on-Chip (MPSoC) [12]. Meanwhile, the language used for multiprocessor programming is changing from traditional C to dataflow language such as SystemC [1] and CAL [2]. This kind of program is usually modeled as dataflow graph during compilation of multiprocessor programming [10].

Task scheduling of a dataflow description over a multi-component embedded system is becoming more and more important due to the strict real-time constraints and growing complexities of applications. It usually consists of assigning tasks to components, specifying the order in which tasks are executed on each component, and specifying the time at which they are executed. However, task scheduling is not straightforward; when performed manually, the result is usually a suboptimal solution. Scheduling on general parallel computer architectures has been actively researched, but task scheduling on parallel embedded systems is different from the general scheduling problem [18]. Communications between components have a very important impact on the scheduling performance and the hardware resources' utilization. Therefore, it is necessary to find new task scheduling methodologies which produce optimal results for programming on parallel embedded systems.

This chapter aims at tackling the task scheduling problem for programming on parallel embedded systems. The program is represented as a task graph modeled by Directed Acyclic Graph (DAG) [13, 18], where nodes represent tasks (i.e. computations) and edges represent data flows (i.e. communications) between tasks. The objective of task scheduling is to respectively assign computations and communications to processors and buses (communication links) of the target system in order to get the minimum schedule length (makespan). The scheduling could be done at compile time namely static or done at run time namely dynamic. Static scheduling is more suitable than dynamic scheduling for deterministic applications in parallel embedded systems by leading to lower code size and higher computation efficiency. This chapter focuses on the static scheduling; all the task scheduling heuristics in the following parts are done at compile time.

The general task scheduling problem is proven to be NP-hard [3, 13]; therefore, many works try to find heuristics to go up to the optimal solution. Early task scheduling heuristics as in [1, 6] do not consider communication costs. As the communication increases in modern applications, many scheduling heuristics [13, 5, 21, 22, 8] have to take communications between tasks into account. Most of these heuristics use fully connected topology network in which all communications can be concurrently performed. Different arbitrary processor networks are then used in [14, 7, 4, 17, 20] to accurately describe real parallel systems, and the task scheduling takes communication contentions on communication links into account.

Most of the above heuristics are based on the approach of list scheduling. Some basic techniques are given in [15] for list scheduling with communication contention. This chapter will provide more advanced techniques. Firstly, three new groups of node priorities will be defined and used to sort nodes in addition to the

[1] http://www.systemc.org/

two existing groups; secondly, a technique of using a node's ***critical child*** will be proposed to improve the performance of selecting a processor for this node. This chapter will finally combine these two techniques giving efficiency runtime performances.

The rest of this chapter is organized as follows: Section 2 firstly introduces necessary models and definitions, then the task scheduling problem with communication contention is described in this section. The node levels used for sorting nodes are defined in Sect. 3. Our advanced list scheduling heuristic is proposed in Sect. 4, and its time complexity is analyzed in Sect. 5. Section 6 gives experimental results, and the chapter is concluded in Sect. 7 at last.

2 Models and Definitions

The program to be scheduled is called algorithm and is modeled as a DAG in this chapter. The multiprocessor target system is called architecture and is modeled as a topology graph. These models are detailed as follows.

2.1 DAG Model

A DAG is a directed acyclic graph $G = (V, E, w, c)$ where V is the set of nodes and E is the set of edges. A node represents a computation meaning that a node in the graph can be a subprogram specified in another language (C, fortran). Between two nodes $n_i, n_j \in V$, e_{ij} denotes the edge from the origin node n_i to the destination node n_j and represents the communication between these two computations. The weight of node n_i (denoted by $w(n_i)$) represents the computation cost; the weight of edge e_{ij}(denoted by $c(e_{ij})$) represents the communication cost. In this model, the set $\{n_x \in V : e_{xi} \in E\}$ of all the direct predecessors of node n_i is denoted by $pred(n_i)$; the set $\{n_x \in V : e_{ix} \in E\}$ of all the direct successors of node n_i is denoted by $succ(n_i)$. A node n with $pred(n) = \emptyset$ is named a source node, where $\emptyset$ is the empty set. A node n with $succ(n) = \emptyset$ is named a sink node.

The execution of computations on a processor is sequential and a computation cannot be divided into several parts. A computation cannot start until all its input communications are finished, and all its output communications cannot start until this computation is finished. In general, communications are sequentially scheduled on a communication link between two processors; however, when a switch is used, communications can be simultaneously scheduled respecting the input and output constraints above.

Figure 1 gives a DAG example which consists of 9 nodes and 12 edges, weights of nodes and edges are also shown in this figure. This DAG has been used in [9] to illustrate performances of different scheduling heuristics, and it will be also used in Sect. 6.1 to show the performance of our method.

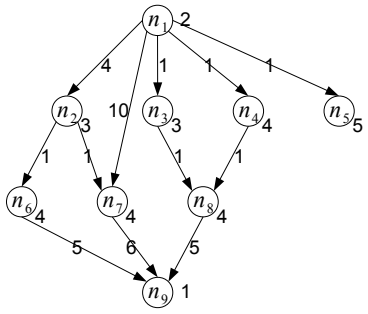


Fig. 1 A DAG example

2.2 Topology Graph Model

A topology graph $TG = (N, P, D, H, b)$ has been used to model a target system of multiple processors interconnected by communication links and switches [17]. N is the set of vertices, P is a subset of N, $P \subseteq N$, D is the set of directed edges, H is the set of hyperedges, and b is the relative data rate of edge. The union of the two edge sets D and H designates the link set L, $L = D \cup H$, and an element of this set is denoted by l, $l \in L$.

The topology graph is denoted as $TG = (N, P, L, b)$ in this chapter, and directed edges are not used in a target system. A vertex $p \in P$ represents a processor, and a vertex $n \in N, n \notin P$ represents a switch. Since directed edges are not used, a link $l \in L$ is actually a hyperedge h, which is a subset of two or more vertices of N, $h \subseteq N, |h| > 1$. A hyperedge connects multiple vertices and represents a half-duplex multidirectional communication link (e.g. a bus). The positive weight $b(l)$, associated with a link $l \in L$, represents its relative data rate.

Differing from the vertex of processor, a switch is a vertex used only for connecting communication links, and no computation can be executed on it. Switches are assumed to be ideal.

Ideal Switch For a switch s, let $l_1, l_2, \ldots, l_n$ be all the communication links connected to s. If two links l_{i_1} and l_{i_2} of them are not used for the moment, a communication can be transferred on l_{i_1} and l_{i_2} without any impact from/to communications on other communication links connected to s.

Switches are contention-free according to the above description. Separate communication links connected to the same switch can be used for different communications at the same time; however, a new communication could not begin on a link if this link is busy. Communication links are considered homogeneous in this chapter, but processors can be heterogeneous. Therefore, the relative data rate is assumed to be 1 for all the links, $b(l) = 1, \forall l \in L$, but a computation usually needs different execution durations on different types of processors.

Figure 2 gives three architecture examples: (a) three processors sharing a bus; (b) eight processors connected to a switch by eight buses; and (c) six processors interconnected by buses and switches. Figure 2(c) models the C6474 Evaluation Module (EVM)[2] which includes two C6474 multicore DSPs.

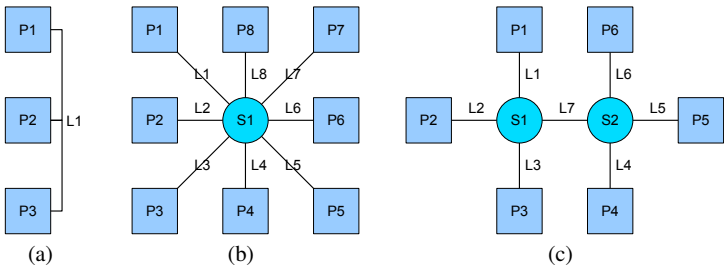


Fig. 2 Architecture examples

A route is used to transfer data from one processor to another in the target system. It is a chain of links connected by switches from the origin processor to the destination processor. For example, $L1 \rightarrow L7 \rightarrow L4$ is a route from $P1$ to $P4$ in Fig. 2(c). Routing is an important aspect of task scheduling. Since the scheduling is static, a route between two processors is also considered as static and is determined at compile time. It is possible to determine routes once and to store them in a table, then the routing during the scheduling becomes looking up the table.

2.3 Task Scheduling with Communication Contention

A schedule of a DAG is the association of a start time and a processor with each node of the DAG. When the communication contention is considered, a schedule also includes allocating communications to links and associating start times on these links with each communication. A communication needs the same duration on each link because of the homogeneity of links. However, a computation usually needs different durations on different processors because processors are heterogeneous. Therefore, the average duration of a computation on different types of processors is used to present the node weight.

Following terms describe a schedule S of a DAG $G = (V, E, w, c)$ over a topology graph $TG = (N, P, L, b)$. The start time of a node $n_i \in V$ on a processor $p \in P$ is denoted by $t_s(n_i, p)$; the finish time is given by $t_f(n_i, p) = t_s(n_i, p) + w(n_i, p)$, where $w(n_i, p)$ is the execution duration of n_i on p. A node can be constrained to some processors of the target system. The set of processors on which n_i can be executed is denoted by $Proc(n_i)$, and the processor on which n_i is actually allocated is denoted

[2] http://focus.ti.com/docs/toolsw/folders/print/tmdxevm6474.html

by $proc(n_i)$. The finish time of a processor is the maximum finish time among all the nodes allocated on this processor, $t_f(p) = \max_{proc(n_i)=p} \{t_f(n_i, proc(n_i))\}$, and the schedule length of S is the maximum finish time among all the processors in the system, $sl(S) = \max_{p \in P} \{t_f(p)\}$.

The communication represented by an edge exists only when its origin node and destination node are not allocated on the same processor. The start time of an existing edge $e_{ij} \in E$ on a link $l \in L$ is denoted by $t_s(e_{ij}, l)$; the finish time of e_{ij} is given by $t_f(e_{ij}, l) = t_s(e_{ij}, l) + c(e_{ij})$. A node (computation) can start on a processor at the time when all the node's input edges (communications) finish. This time is called the Data Ready Time (DRT) and is denoted by $t_{dr}(n_j, p) = \max_{e_{ij} \in E} \{t_f(e_{ij}, l)\}$, where l is a link on which e_{ij} is allocated. DRT is the earliest time when a node can start. If n_j is a node without input edge, $t_{dr}(n_j, p) = 0, \forall p \in P$.

Node Scheduling Condition For a node n_i, let $[A, B], A, B \in [0, \infty]$ be an idle time interval on the processor p. n_i can be scheduled on p within $[A, B]$ if

$$\max\{A, t_{dr}(n_i, p)\} + w(n_i, p) \leq B$$

The start time of n_i on p is given by

$$t_s(n_i, p) = \max\{A, t_{dr}(n_i, p)\}$$

Communications are handled in the way of cut-through on a route because of the circuit switching. Therefore, an edge e_{ij} is aligned on all the links of the route $l_{R_1} \to l_{R_2} \to \ldots \to l_{R_k}$ with $t_s(e_{ij}, l_{R_1}) = t_s(e_{ij}, l_{R_2}) = \ldots = t_s(e_{ij}, l_{R_k})$. The start time and finish time of e_{ij} on all the links of the route are uniformly denoted by $t_s(e_{ij})$ and $t_f(e_{ij})$ with $t_f(e_{ij}) = t_s(e_{ij}) + c(e_{ij})$.

Edge Scheduling Condition For a DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$, let $l_{R_1} \to l_{R_2} \to \ldots \to l_{R_k}$ be a route for an edge $e_{ij} \in E$ and let $[A, B], A, B \in [0, \infty]$ be a common idle time interval on all the links of this route. e_{ij} can be scheduled on this route within $[A, B]$ if

$$\max\{A, t_f(n_i, proc(n_i))\} + c(e_{ij}) \leq B$$

The start time of e_{ij} on this route is given by

$$t_s(e_{ij}) = \max\{A, t_f(n_i, proc(n_i))\}$$

3 Node Levels with Communication Contention

The top level and bottom level are usually used as node priorities which are important for DAG scheduling as in [14, 16]. The top level of a node is the length of the

longest path from any source node to this node, excluding the weight of this node; the bottom level of a node is the length of the longest path from this node to any sink node, including the weight of this node.

3.1 Existing Node Levels

Two groups of top and bottom levels have been used in task scheduling heuristics, which are respectively named as computation top/bottom levels (tl_{comp} and bl_{comp}) and top/bottom levels (tl and bl). Figure 3 illustrates the dependences between nodes for the two existing groups of top levels and bottom levels, where the red dotted nodes and edges are used to recursively define the top levels and bottom levels of n_i.

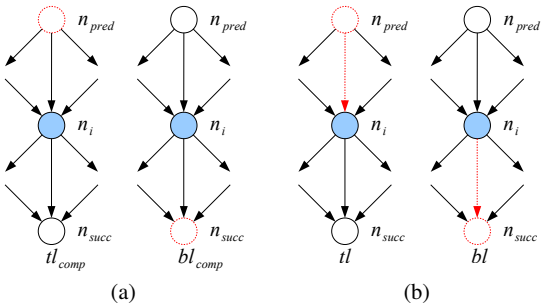


Fig. 3 Two existing groups of node levels

- **Computation top level and bottom level** (Fig. 3(a))
  The computation top level of a node is the length of the longest path from any source node to this node including only the weights of nodes; the computation bottom level of a node is the length of the longest path from this node to any sink node including only the weights of nodes. The weights of edges are not taken into account in the computation top level and bottom level. They are recursively defined as follows:

$$tl_{comp}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \left\{ tl_{comp}(n_k) + w(n_k) \right\}, & \text{otherwise} \end{cases}$$

$$bl_{comp}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \left\{ bl_{comp}(n_k) \right\} + w(n_i), & \text{otherwise} \end{cases}$$

- **Top level and bottom level** (Fig. 3(b))
  The top level and bottom level additionally take the weights of edges on the path into account by contrast with the computation top level and bottom level. They are recursively defined as follows:

$$tl(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \{tl(n_k) + w(n_k) + c(e_{ki})\}, & \text{otherwise} \end{cases}$$

$$bl(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \{bl(n_k) + c(e_{ik})\} + w(n_i), & \text{otherwise} \end{cases}$$

3.2 New Node Levels

Besides the two existing groups, this chapter proposes three new groups. The dependences between nodes for the three new groups of top levels and bottom levels are shown in Fig. 4. The formalized definitions of top levels and bottom levels are given as follows.

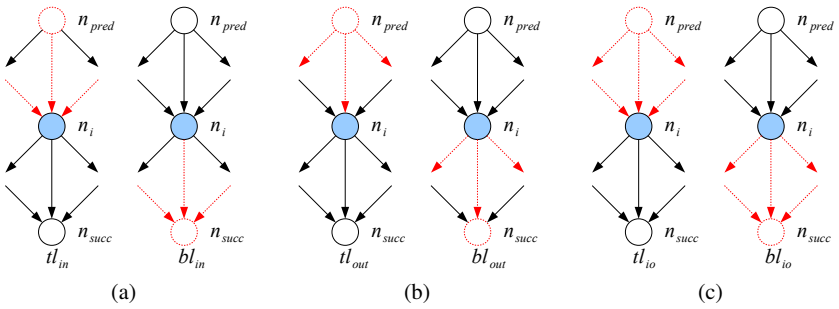


Fig. 4 Three new groups of node levels

- **Input top level and bottom level** (Fig. 4(a))
  The input top level and bottom level take into account weights of nodes on the path as well as weights of all the input edges of a node on the path. They are recursively defined as follows:

$$tl_{in}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \{tl_{in}(n_k) + w(n_k)\} + \sum\limits_{e_{li} \in E} c(e_{li}), & \text{otherwise} \end{cases}$$

$$bl_{in}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \left\{ bl_{in}(n_k) + \sum\limits_{e_{lk} \in E} c(e_{lk}) \right\} + w(n_i), & \text{otherwise} \end{cases}$$

- **Output top level and bottom level** (Fig. 4(b))
  The output top level and bottom level take into account weights of nodes on the path as well as weights of all the output edges of a node on the path. They are recursively defined as follows:

$$tl_{out}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \left\{ tl_{out}(n_k) + w(n_k) + \sum\limits_{e_{kl} \in E} c(e_{kl}) \right\}, & \text{otherwise} \end{cases}$$

$$bl_{out}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \{ bl_{out}(n_k) \} + \sum\limits_{e_{il} \in E} c(e_{il}) + w(n_i), & \text{otherwise} \end{cases}$$

- **Input/output top level and bottom level** (Fig. 4(c))
  The input/output top level and bottom level take into account weights of nodes on the path as well as weights of all the input and output edges of a node on the path. They are recursively defined as follows:

$$tl_{io}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max\limits_{n_k \in pred(n_i)} \left\{ tl_{io}(n_k) + w(n_k) + \sum\limits_{e_{kl} \in E} c(e_{kl}) - c(e_{ki}) \right\} \\ \quad + \sum\limits_{e_{li} \in E} c(e_{li}), & \text{otherwise} \end{cases}$$

$$bl_{io}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max\limits_{n_k \in succ(n_i)} \left\{ bl_{io}(n_k) + \sum\limits_{e_{lk} \in E} c(e_{lk}) - c(e_{ik}) \right\} \\ \quad + \sum\limits_{e_{il} \in E} c(e_{il}) + w(n_i), & \text{otherwise} \end{cases}$$

The three new groups take into account the communication contention between nodes in comparison with the two existing groups which are usually used in the list scheduling without communication contention. Table 1 gives all the five groups of top levels and bottom levels for the DAG given in Fig. 1.

Table 1 Different node levels

	tl_{comp}	bl_{comp}	tl	bl	tl_{in}	bl_{in}	tl_{out}	bl_{out}	tl_{io}	bl_{io}
n_1	0	11	0	23	0	41	0	35	0	55
n_2	2	8	6	15	6	35	19	16	19	36
n_3	2	8	3	14	3	26	19	14	19	26
n_4	2	9	3	15	3	27	19	15	19	27
n_5	2	5	3	5	3	5	19	5	19	5
n_6	5	5	10	10	10	21	24	10	24	21
n_7	5	5	12	11	20	21	24	11	34	21
n_8	6	5	8	10	9	21	24	10	25	21
n_9	10	1	22	1	40	1	34	1	54	1

4 List Scheduling Heuristic

Algorithm 1 gives the commonly used static list scheduling heuristic. This algorithm is composed of three procedures of `Sort_Nodes()`, `Select_Processor()` and `Schedule_Node()`. This section describes improvements for the first two procedures compared with the classic methods given in [17].

Algorithm 1: **List_Scheduling**(G, TG)

Input: A DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$
Output: A schedule of G on TG

```
1 NodeList ← Sort_Nodes(V);
2 for each n ∈ NodeList do
3 |   p_best ← Select_Processor(n, P);
4 |   Schedule_Node(n, p_best);
5 end
```

4.1 Sorting Nodes with Five Groups of Node Priorities

Nodes are firstly sorted into a static list by the procedure of `Sort_Nodes()` in the heuristic. Since the order of nodes in the list affects much the schedule result, many different priority schemes have been proposed to sort nodes [14, 8]. Experiments in [16] show that list scheduling with static list sorted by bottom level outperforms other compared contention aware algorithms. Our list scheduling heuristic uses the bottom level and top level to sort nodes, the procedure of `Sort_Nodes()` sorts nodes into a list of *NodeList* according to the following rule:

Rule for Sorting Nodes Nodes are sorted by the decreasing order of their bottom levels; if two nodes have equal bottom levels, the one with greater top

level is placed before the other; if both the bottom level and the top level are equal, these nodes are sorted randomly.

According to the five groups of top levels and bottom levels given in Table 1, the resulting static lists are shown in Table 2.

Table 2 Different static node lists

Node priority	Static node list	No.
bl_{comp} & tl_{comp}	$n_1, n_4, n_3, n_2, n_8, n_7, n_6, n_5, n_9$	(1)
bl & tl	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$	(2)
bl_{in} & tl_{in}	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$	(2)
bl_{out} & tl_{out}	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$	(3)
bl_{io} & tl_{io}	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$	(3)

4.2 Processor Selection

The classic list scheduling heuristic selects the processor allowing the earliest finish time for a node. This rule probably gives a locally optimized result. In fact, this rule usually gives bad results for the join structure of a DAG especially in the case of great communication cost and communication contention. Figure 5(a) shows such an example; Figure 5(b) gives the schedule result with the classic processor selection method, which selects a new processor for each one of n_1, n_2 and n_3 to provide the earliest finish time. Therefore, the execution of node n_4 has to wait until the communications from n_2 and n_3 finish, and the schedule length is 6 at last. By contrast, the schedule of all nodes on the same processor is shown in Fig. 5(c) and has a schedule length of 4.

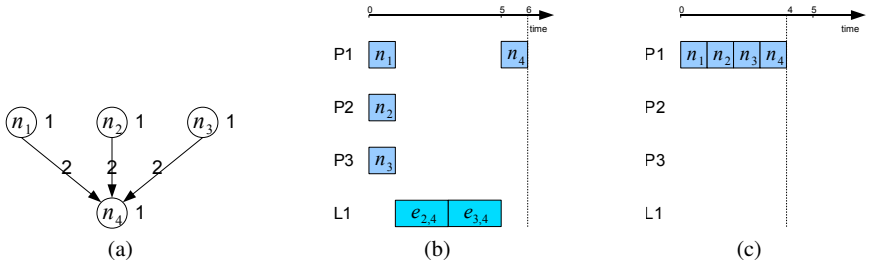


Fig. 5 A join DAG and two different schedule results

In [8], the critical child of a node is defined as one of its successors that has the smallest difference between the absolute latest possible start time (ALST) and the absolute earliest possible start time (AEST). It is used for scheduling in the case

of unbounded number of processors and without communication contention. This chapter uses the concept of critical child for list scheduling in the case of bounded number of processors and with communication contention. The critical child is differently defined as follows.

Critical Child Given a static node list *NodeList*, the critical child of node n_i is denoted by $cc(n_i)$ and is one of n_i's successors which emerges firstly in *NodeList*.

According to this definition, the critical child of n_i may be different if *NodeList* differs. This is the major difference between our critical child and that in [8]. Table 3 shows the critical children according to the different static node lists given in Table 2.

Table 3 Critical children according to different static node lists

No.	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
(1)	n_4	n_7	n_8	n_8	null	n_9	n_9	n_9	null
(2)	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
(3)	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null

Using critical child makes the processor selection take into account not only the predecessors of a node, but also its most important successor. Our method of using the critical child to select processor is given in Algorithm 2. Since it is possible that $cc(n_i)$ is not a free node with all its predecessors scheduled during the processor selection for n_i, the scheduling of $cc(n_i)$ only takes into account its scheduled predecessors in the procedure of `Select_Processor()` for n_i.

4.3 Node and Edge Scheduling

The method of scheduling a node n_i onto a processor p is given in Algorithm 3, and Algorithm 4 gives the method for edge scheduling. Since an edge e_{ij} is scheduled only when its origin node n_i has been scheduled, the scheduling of this edge needs additionally the processor p on which the destination node n_j of e_{ij} to be scheduled.

5 Analysis of Time Complexity

In [15] the time complexity of the classic list scheduling heuristic is given as $O\left(PE^2O(routing)+V^2\right)$, where P, V and E are the number of processors, the number of nodes and the number of edges, respectively. The time complexity increases by a factor of P by using the critical child, but the combination with different node

Algorithm 2: **Select_Processor**(n_i, P)

Input: A node $n_i \in V$ and the set P of all processors
Output: The best processor p_{best} for the input node n_i

Choose the critical child $cc(n_i)$;
$BestFinishTime \leftarrow \infty$;
for *each* $p \in Proc(n_i)$ **do**
  $FinishTime \leftarrow$ **Schedule_Node**(n_i, p);
  $MinFinishTime \leftarrow \infty$;
  if $cc(n_i) \neq null$ **then**
    for *each* $p' \in Proc(cc(n_i))$ **do**
      $FinishTime \leftarrow$ **Schedule_Node**($cc(n_i)$, p');
      if $FinishTime < MinFinishTime$ **then**
        $MinFinishTime \leftarrow FinishTime$;
      end
      Unschedule the input edges of $cc(n_i)$;
      Unschedule $cc(n_i)$ from p';
    end
  else
    $MinFinishTime \leftarrow FinishTime$;
  end
  if $MinFinishTime < BestFinishTime$ **then**
    $BestFinishTime \leftarrow MinFinishTime$;
    $p_{best} \leftarrow p$;
  end
  Unschedule the input edges of n_i;
  Unschedule n_i from p;
end

Algorithm 3: **Schedule_Node**(n_i, p)

Input: $n_i \in V$ and a processor $p \in P$
Output: The finish time of n_i on p

for *each* $n_l \in pred(n_i), proc(n_l) \neq p$ **do**
  Schedule_Edge(e_{li}, p);
end
Calculate DRT of node n_i;
Find the earliest idle time interval for node n_i on processor p respecting the node scheduling condition;
Calculate the finish time of n_i on p;

priorities does not increase the time complexity. The time complexity of our proposed list scheduling heuristic is analyzed as follows.

The route can be determined (calculated or looked up) in $O(1)$ time in the procedure `Schedule_Edge()` for static routing. If the route contains $O(routing)$ links, it takes $O(EO(routing))$ time to find the earliest common idle time interval on all links of the route. Thus, the complexity of `Schedule_Edge()` is $O(EO(routing))$.

Algorithm 4: **Schedule_Edge**(e_{ij}, p)

```
   Input: e_ij ∈ E and a processor p ∈ P on which the node n_j is to be scheduled
   Output: None
1  if n_i is scheduled then
2      if proc(n_i) ≠ p then
3          Determine the route R from proc(n_i) to p;
4          Find the earliest common idle time interval on all the links of R respecting the edge
           scheduling condition;
5      end
6  end
```

The procedure `Schedule_Node()` firstly needs to use $O\left(\frac{E}{V}\right)$ times of the procedure `Schedule_Edge()` on average, then it takes $O\left(\frac{E}{V}\right)$ time to calculate the DRT, and it takes $O\left(\frac{V}{P}\right)$ time to find an idle time interval for a node on average. At last, it takes $O(1)$ time to calculate the finish time of the node. Therefore, the total complexity of the procedure `Schedule_Node()` is $O\left(\frac{E^2 O(routing)}{V}+\frac{V}{P}\right)$ on average.

As to the procedure `Select_Processor()`, it firstly takes $O(V)$ time to find the critical child $cc(n_i)$. When $cc(n_i)$ is found, given a specific processor p, it needs at most $O(P)$ times of `Schedule_Node()` for the scheduling of n_i and $cc(n_i)$. Hence, the complexity in the outer for-loop is $O\left(P\left(\frac{E^2 O(routing)}{V}+\frac{V}{P}\right)\right)$, and the total complexity of `Select_Processor()` is $O\left(P\left(\frac{PE^2 O(routing)}{V}+V\right)\right)$.

In Algorithm 1, sorting nodes has the complexity of $O(V\log V+E)$ (computing node levels in $O(V+E)$ + sorting in $O(V\log V)$). Our new definitions of top level and bottom level do not change the complexity of computing node levels; therefore, the complexity of sorting nodes is always $O(V\log V+E)$. Since the procedure `Select_Processor()` is more complicated than the procedure `Schedule_Node()`, the complexity in the for-loop is equal to that of the procedure `Select_Processor()`. Finally, the total complexity of the proposed list scheduling heuristic is given by $O\left(P\left(PE^2 O(routing)+V^2\right)\right)$.

6 Experimental Results

This section gives experimental results of our proposed list scheduling heuristic compared to the classic one given in [17]. The architecture in Fig. 2(a) and 2(b) are used for the comparison in Sect. 6.1 and 6.2, respectively.

6.1 Comparison with an Example

The DAG given in Fig. 1 is used in this section to show that using the critical child and different priorities improves the schedule performance. Table 1 has given all the five groups of top levels and bottom levels; the resulting static lists are given in Table 2; and the critical child for each node is obtained according to these static lists in Table 3.

Figure 6 gives the schedule result of the classic heuristic with nodes sorted by bl and tl, where the schedule length is 21. Using the critical child technique with the three different node lists in Table 2 gives different schedule results. The schedule result for the node list sorted by bl_{comp} and tl_{comp} is shown in Fig. 7(a) with the schedule length of 18. Since the node list sorted by bl and tl is same as that sorted by bl_{in} and tl_{in}, the same schedule result is obtained and shown in Fig. 7(b) with the schedule length of 18. Figure 7(c) gives the schedule result for the same node list sorted by bl_{out} and tl_{out} and by bl_{io} and tl_{io}. The schedule length is 17 and is better than the two former schedule lengths of 18. All the three schedule results of using the critical child technique are better than that of the classic heuristic.

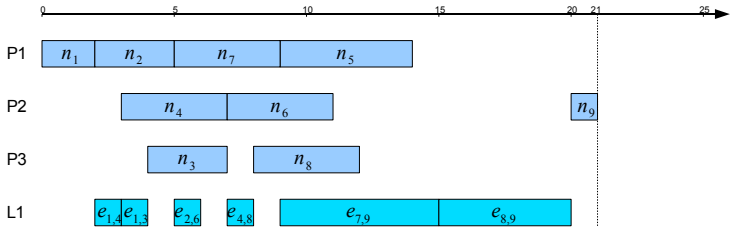


Fig. 6 Schedule result of classic heuristic

6.2 Comparison with Random DAGs

Random graphs are commonly used to compare scheduling algorithms in order to get statistical results which are more persuasive than the result for a particular graph. We implement a graph generator based on SDF^3 [19] to generate random SDF graphs [11] except that the SDF graphs are constrained to be DAGs (same rate between two operations, no cycles). A random DAG is described in five aspects: the number of nodes, the average in degree, the average out degree, the random weights of nodes and the random weights of edges. The average in degree and out degree are assumed to be same. The weights of nodes vary randomly from w_{min} to w_{max}. The communication to computation ratio (*CCR*) is used to generate random weights of edges. The *CCR* is defined as the average weight of edges divided by the aver-

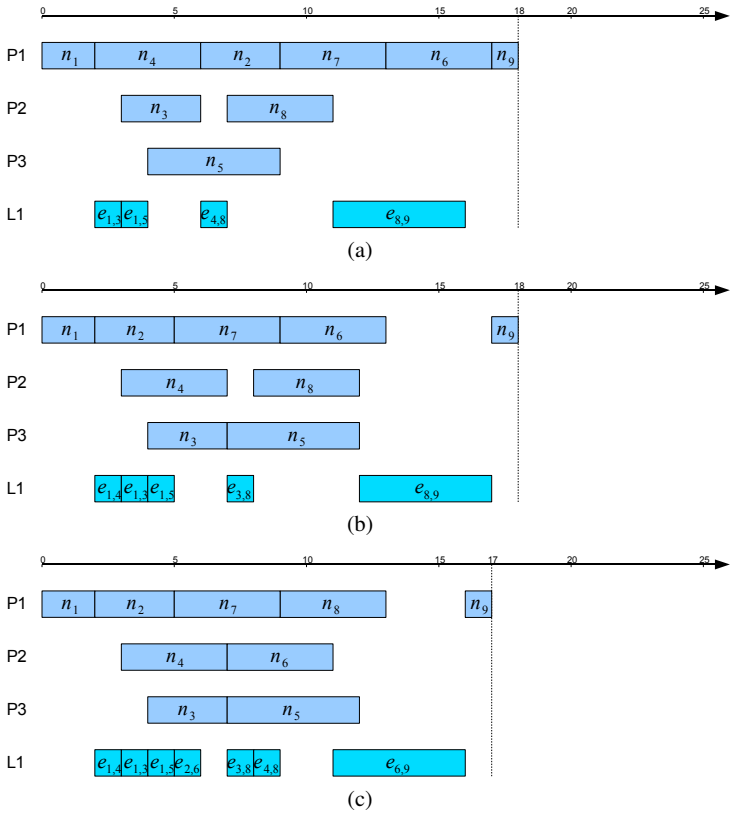


Fig. 7 Schedule results with critical child

age weight of nodes in this chapter, that is $CCR = \frac{\frac{1}{|E|}\sum_{e\in E} c(e)}{\frac{1}{|V|}\sum_{n\in V} w(n)}$. The weights of edges are generated randomly from $w_{min} \times CCR$ to $w_{max} \times CCR$. The CCR's typical values of 0.1, 1 and 10 represent the low, medium and high communication situations, respectively.

A list scheduling heuristic can use all the five groups of node priorities to get different results. We combine the five groups of node priorities with a heuristic and choose the best result; the whole process is called a combined heuristic. The schedule length of the combined heuristic is compared to the classic list scheduling heuristic with nodes sorted by bl and tl. The acceleration factor (acc) is defined as $acc = \frac{sl_{classic}}{sl_{compared}}$ to show the speed-up of the compared heuristic.

Figure 8 gives the average acc of the combined heuristic with critical child. Weights of nodes are generated randomly from 100 to 1000, and 1000 random DAGs for each group are tested to obtain the statistical results.

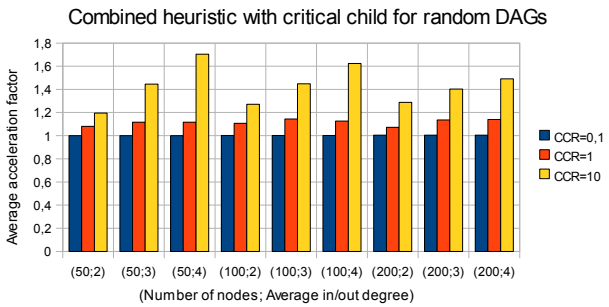


Fig. 8 Average *acc* of combined heuristic with critical child

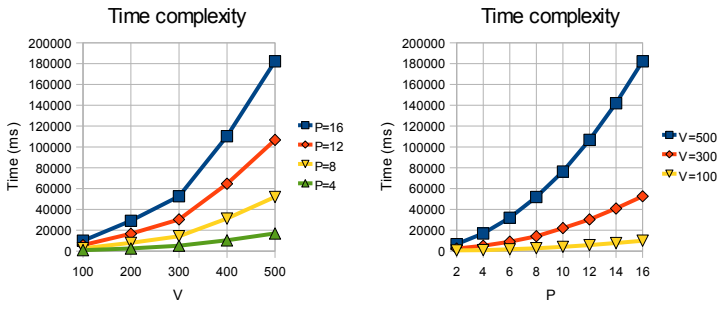
The average *acc* increases as *CCR* increases, and the schedule result is sped up by using the combined heuristic in the cases of $CCR = 1$ and $CCR = 10$. The average *acc* also increases as the number of average in/out degree increases when $CCR = 10$. The reason for this phenomenon is that the critical child technique helps to select better processors for nodes with multiple predecessors. The greater the in/out degree is, the better the critical child works. Since the modern applications like digital communication and video compression usually have $CCR > 1$, our method will be suitable for scheduling these applications on parallel embedded systems.

6.3 Time Complexity

Figure 9 shows the time used to schedule different sizes of DAGs on architectures with different numbers of processors by our combined heuristic. All the DAGs have the average in/out degree of 4, and all the processors are connected to a switch. As shown in Fig. 9(a) and Fig. 9(b), the time increases as the square of V and also as the square of P. We run our heuristic on a Pentium Dual-Core PC at 2.4GHz, and it takes about 3 minutes to schedule a DAG with 500 nodes on an architecture of 16 processors.

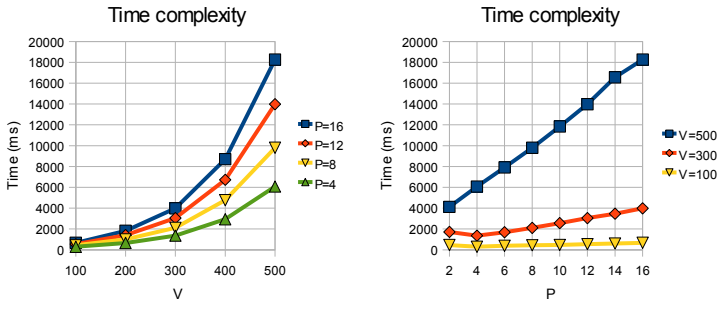
By contrast, Figure 10 shows the time used by the classic heuristic. As shown in Fig. 10(a) and Fig. 10(b), the time increases with the square of V, but it increases linearly with P.

In fact, a complicated embedded application usually has less than 500 nodes in models of coarse and medium grain, and P is usually much smaller than V and E in a parallel embedded system. Therefore, the increase of time complexity is reasonable and acceptable for rapid prototyping.



(a) (b)

Fig. 9 Time complexity of the proposed heuristic



(a) (b)

Fig. 10 Time complexity of the classic heuristic

7 Conclusions

This chapter presents three new groups of node priorities (top level and bottom level) and a technique of critical child for list scheduling with communication contention. The new priorities take the communication contention into account and are used to sort nodes in order to get different node lists. The technique of critical child helps to select a better processor for a node. The combination of different node lists and the critical child technique gives different schedule results for a given DAG, and the best one is chosen at last. Experimental results show that using different node lists and the critical child technique is effective to shorten the schedule length for most of the randomly generated DAGs in the cases of medium and high communication. Since the communication cost is increasing from medium to high in modern digital communication and video compression applications, our method will work well for scheduling these applications on parallel embedded systems.

References

1. Adam, T.L., Chandy, K.M., Dickson, J.R.: A comparison of list schedules for parallel processing systems. Commun. ACM **17**(12), 685–690 (1974). DOI http://doi.acm.org/10.1145/361604.361619
2. Eker, J., Janneck, J.W.: CAL Language Report. Tech. rep., ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley (2003)
3. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman (1979)
4. Grandpierre, T., Lavarenne, C., Sorel, Y.: Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In: Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99. Rome, Italy (1999). URL http://www-rocq.inria.fr/syndex/pub/codes99/codes99.pdf
5. Hwang, J.J., Chow, Y.C., Anger, F.D., Lee, C.Y.: Scheduling precedence graphs in systems with interprocessor communication times. SIAM J. Comput. **18**(2), 244–257 (1989). DOI http://dx.doi.org/10.1137/0218016
6. Kasahara, H., Narita, S.: Practical multiprocessor scheduling algorithms for efficient parallel processing. IEEE Trans. Comput. **33**(11), 1023–1029 (1984). DOI http://dx.doi.org/10.1109/TC.1984.1676376
7. Kwok, Y.K., Ahmad, I.: Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In: SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing, p. 36. IEEE Computer Society, Washington, DC, USA (1995)
8. Kwok, Y.K., Ahmad, I.: Dynamic critical-path scheduling: An effective technique for allocating task graphs onto multiprocessors. IEEE Transactions on Parallel and Distributed Systems **7**(5), 506–521 (1996). DOI 10.1109/71.503776
9. Kwok, Y.K., Ahmad, I.: Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Computing Surveys **31**(4), 406–471 (1999). URL citeseer.ist.psu.edu/kwok99static.html
10. Lee, E., Parks, T.: Dataflow process networks. Proceedings of the IEEE **83**(5), 773–801 (1995). DOI 10.1109/5.381846
11. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proceedings of the IEEE **75**(9), 1235–1245 (1987)
12. Martin, G.: Overview of the mpsoc design challenge. In: Proceedings of the 43rd annual conference on Design automation. San Francisco, CA, USA (2006)
13. Sarkar, V.: Partitioning and Scheduling Parallel Programs for Multiprocessors. The MIT Press (1989)
14. Sih, G., Lee, E.: A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. IEEE Transactions on Parallel and Distributed Systems **4**, 175–187 (1993). DOI 10.1109/71.207593
15. Sinnen, O.: Task Scheduling for Parallel Systems. Wiley (2007)
16. Sinnen, O., Sousa, L.: List scheduling: Extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. Parallel Computing **30**(1), 81–101 (2004)
17. Sinnen, O., Sousa, L.: Communication contention in task scheduling. IEEE Transactions on Parallel and Distributed Systems **16**(6), 503–515 (2005)
18. Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors - Scheduling and Synchronization. Marcel Dekker, Inc. (2000)
19. Stuijk, S., Geilen, M., Basten, T.: SDF^3: SDF For Free. In: Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings, pp. 276–278. IEEE Computer Society Press, Los Alamitos, CA, USA (2006). DOI 10.1109/ACSD.2006.23. URL http://www.es.ele.tue.nl/sdf3
20. Tang, X., Li, K., Padua, D.: Communication contention in apn list scheduling algorithm. Science in China Series F: Information Sciences **52**(1), 59–69 (2009)

21. Wu, M.Y., Gajski, D.: Hypertool: A programming aid for message-passing systems. IEEE Transactions on Parallel and Distributed Systems **1**(3), 330–343 (1990). URL citeseer.ist.psu.edu/wu90hypertool.html
22. Yang, T., Gerasoulis, A.: Dsc: scheduling parallel tasks on an unbounded number of processors. IEEE Transactions on Parallel and Distributed Systems **5**(9), 951–967 (1994). DOI 10.1109/71.308533