Process Partitioning for Distributed Embedded Systems

Junwei Hou and Wayne Wolf
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544

Abstract

We present a new technique for partitioning processes in distributed embedded systems. Our heuristic algorithm minimizes both context switch and communication overhead under real-time deadline and process size constraints; it also tries to allocate functions to processors which are well-suited to that function. The algorithm analyzes the sensitivity of the latency of the task graph to changes in vertices hierarchical clustering, splitting and border adjusting. This algorithm can be used for initial partitioning during co-synthesis of distributed embedded systems. Synthesis of examples partitioned by our algorithm with implementations synthesized directly from the original example shows that our partitioning algorithm significantly improves the results obtainable by practical co-synthesis algorithms.

1 Introduction

This paper introduces a new technique for process partitioning, which is the first phase of our three-phase strategy for hardware-software co-synthesis of distributed real-time embedded systems. Partitioning has been shown to be an important phase for complicated ASIC design at various level of abstraction. Process partitioning groups the logical processes in a system specification into a new set of physical processes which actually appear as processes in the system’s implementation. Figure 1 summarises the transformation of the system specification as a data flow graph of modules into a set of processes and finally into a hardware-software architecture. Because the process boundaries which make for clean, maintainable specifications may not be the best process boundaries for implementation, partitioning is an important step in co-synthesis.

Our co-synthesis strategy targets a distributed heterogeneous system implementation which consists of several different types of processing engines (PE). A PE can be a general purpose processor, an application specific processor, or an ASIC. We try to isolate the problems of partitioning and allocation, and take a three phase strategy for the co-synthesis of distributed embedded systems:

Phase 1: Allocation independent partitioning

- Analyze the cost of the system and formulate the partitioning objective function, which will minimize both computation and communication costs, and maximize the advantage of each specific PE.
- Partition the graphs towards the objective function, using an optimization algorithm with hierarchical clustering, splitting and border adjusting, while satisfying the performance constraints.

Phase 2: Sensitivity-driven allocation [1]

- Estimate the bounds on the schedulability of the partitioned graphs composed of processes executing on a selected set of PEs.
- Iteratively improve the architecture of the system by using the bounds to drive the sensitivity of the implementation to incremental modification.

Phase 3: Repartitioning and allocation

- Identify the usable idle intervals, which are incurred by the data block and unbalance between processes on a processor.
- Repartition these processes by moving modules between them, and modify the architecture of the system by replacing and/or eliminating some PEs.

Although the system’s cost cannot be accurately determined before allocation, we can estimate it during allocation-independent partitioning by assuming that every PE is fully utilised. Repartitioning after allocation is necessary to balance and optimize the system. This paper concentrates on the allocation-independent partitioning performed in Phase 1.

Many embedded computing systems are implemented as distributed systems [2] in which PEs are connected using point-to-point links and/or system buses, to realize the concurrent and reactive data processing and controlling purposes for an application system, which consists of several relatively independent (different threads of) tasks. Each task needs to accomplish a particular computation and may have its
own deadline, which specifies the time at which the computation must be completed, and its own period, which specifies the rate at which the data must be processed.

Partitioning’s goal is to reduce the cost of the hardware engine, which is equivalent to minimizing the computation time required for the application. Several factors must be considered when choosing a partition of the specification into functions:

1. **Communication cost**
   Communication between PEs is usually expensive. Proper partitioning can greatly reduce the amount of inter-PE communication required during execution.

2. **Diversity of data processing**
   Every off-the-shelf processor has its advantage in some particular data processing. Two modules with different types of data processing should be partitioned into two processes.

3. **Context switch time**
   In a distributed embedded system, several processes in different tasks may share one PE, and different data transactions between different processes may share one communication link. The system has overhead, which is modeled as context switch, for the synchronization between processes.

4. **Data dependencies**
   A process can start only after its proceeding processes are finished and it has higher priority than other processes ready to execute on its PE.

We require that each partition be completely executable—a process formed by a partition may not start until all its input data has been arrived, and will not issue its output data until it completed. A partition of the process graph is completely executable if, for any chosen source and sink of the partition, every node on any path from the source to the sink is in the partition. This requirement preserves the data dependencies between processes and minimizes the system’s overhead. For partition of Figure 2(a), process P2 may not start until P1 is completed, even if P1 and P2 are allocated on two PEs. If the time constraints of this task graph are not critical, the partition of Figure 2(a) can reduce the system’s overhead. If the time constraints of this task graph are critical, however, we may partition the graph into three processes, as show in Figure 2(b), and allocate P1 and P2 on two PEs for parallel processing.

This requirement also distinguishes process partitioning from other algorithms which assume that only one process at a time is executing in the system. For example, in Figure 2(c), the partition of P1 and P2 of this data flow graph may be optimal for hardware partitioning. But in the case of process partitioning, this is illegal because P1 is not completely executable. In distributed real-time systems, this will lead to a bad design because, when P2 is preempted by another process, the processor hosting P1 will idle during the execution of P1.

2 Previous Work

A great deal of recent work has addressed the co-synthesis of systems with a one-CPU-n-ASIC hardware engine architecture. This problem is commonly known as hardware-software partitioning, although partitioning is used in a different sense than ours. Given the exact processor in use, the algorithm of Gupta and De Micheli [3] achieves the hardware-software partitioning by moving operations from hardware of ASIC to software on a processor to reduce system cost, while the algorithm of Ernst et al. [4] moves operations from software to hardware to meet performance goals. Vahid et al. [5] present a binary-search algorithm which minimizes the hardware of ASIC and satisfies other constraints. Eles et al. [6] present a stepwise approach of partitioning with different granularities based on VHDL specification. Adams and Thomas [7] suggest transforming the behaviors between processes to explore simultaneously the benefits of addressing concurrency optimization and hardware/software trade-
off. Vahid and Gajski [8] present an N-way partitioning method using clustering heuristics to improve the system’s performance and cost, while satisfying various constraints. All the above approaches deal with one processor system with single task or synchronized tasks, so neither context switch nor the diversity of data processing is considered.

Recent work [9, 10, 1, 11, 12], addressed distributed embedded systems co-synthesis starting with the descriptions in processes, which are also assumed to be completely executable. Some of these co-synthesis algorithms allow multi-threaded execution and can be severed as the second phase of our co-synthesis strategy.

Huang [13] proposes an allocation-independent software partition modeling which leads to maximising the resource utilisation on a general purpose homogenous distributed computing system. It is actually a modelling for n-way partitioning of the same fixed size to partition a big real-time application software in n-partition to get the best performance on a distributed computer system. This partition modeling is too simple to reflect the heterogeneous property of distributed embedded systems.

Ismail et al. [14] propose an interactive system-level partitioning method and a tool, which support the design of heterogeneous hardware/software systems described in hierarchical data flow graphs with extended FSM as nodes. The designer makes the partitioning decisions according to the statistics and performance estimation provided by the tool.

3 Process Partitioning Model

3.1 Cost Function of the System

The overall purpose of co-synthesis for embedded systems is to minimize the total hardware cost while satisfying the time and memory or chip size constraints. So, we need to exactly formulate the cost function of the system.

For allocation-independent partitioning, we may evaluate the cost of the system in terms of every process’s computation costs, since the system’s hardware architecture information is not available. For example, if a process’s running time is 2s on a $10 PE and its evoking period is $5s$, the cost of this process on the PE is $4$. In addition to the upper bound of its computation time on a PE estimated by some tools like Cinderella [15], the running time of the process should also include the overhead of context switch time associated to the PE. For simplicity, we assume that I/O operations do not consume CPU time.

Our algorithm is based on the following assumptions:

Assumption 1 The system will eventually be balanced and every PE and communication link will be fully utilized.

Assumption 2 There is no data blocking because of resource limitation, and every process or communication will start as soon as all of its preceding data arrive.

Assumption 3 Any two adjacent processes are not on the same PE and the communication cost always exists unless the two are clustered into one.

These assumptions ignore the effects of unbalanced allocation, which means the performance may be overestimated, but pessimistically estimate communication cost. The purpose is to make the partitioning effort clearly towards the goal of minimising the communication and taking advantage of each specific PE.

The system specification is modeled as a set of independent tasks. Based on the above assumptions, we can compute the system’s cost and model the partitioning formulation on a task-by-task basis, while considering that each task has its own deadline $T^h$ and period $T$. A task is represented as a data flow graph $G = (V, A)$, where $V$ is the set of vertices which model modules or processes and $A$ is the set of directed edges which represent communication.

The architecture model has several elements. We let $H$ represent the set of PEs given by the user, $r_{v,h}$ the running time of a vertex $v \in V$ when it executes on a PE $h \in H$, $d_o$ the data volume of communication on edge $a$, and $\delta$ the unit datum transmission time on a communication link.

The total cost of a task is equal to the sum of the computation costs and the communication costs in the task graph $G(V, A)$:

$$C(G) = \sum_{v \in V} r_{v,h} C(h)/r + \sum_{a \in A} d_o \delta C(l)/r \quad (1)$$

The system’s cost is the sum of all the tasks’ costs. For simplicity, here we assume that there is only one type of link to be used for communication among PEs.

3.2 Partitioning Formulation

When clustering modules into processes, we want to find an optimum partition of $G$, so that $C(G)$ is minimum, under time and size constraints. As discussed before, for a heterogeneous system, the computation cost of a process running on different PEs may be quite different. The cheapest implementation of a task is an allocation with every vertex on its minimal cost PE, and the corresponding term of the vertex in Equation 1 is minimum, because the allocation of each vertex is independent under Assumption 3. We define the minimal cost PE of a vertex $v$ as the biasing PE, $b(v)$, of the vertex, that is:

$$t_{v,b(v)} C(b(v)) = \min_{h \in H} t_{v,h} C(h) \quad (2)$$

and $t_{v,b(v)}$ is called the biasing time of the vertex.

The partitioning objective function $f(G)$ can be formulated from Equation 1 as minimising the sum of computation and communication costs:

$$\text{minimize: } f(G) = \sum_{v \in V} t_{v,b(v)} C(b(v)) + \sum_{a \in A} d_o \delta C(l) \quad (3)$$
For process partitioning, it must be guaranteed that every process in the task is completely executable. As discussed in Section 1, in the graph. (In comparison, Huang [13] did not consider all possible clusters.) As discussed in Section 1, for process partitioning, it must be guaranteed that every process in the task is completely executable.

4.1 Legality of clustering vertices

In order to search the global optimal partition, we evaluate all the possible clustering of adjacent vertices in the graph. (In comparison, Huang [13] did not consider all possible clusters.) As discussed in Section 1, for process partitioning, it must be guaranteed that every process in the task is completely executable.

4.2 Clustering sensitivity to the objective function

For optimal partitioning, the first term of objective function 3 means every vertex should take its biasing PE, while the second term shows the vertices should be clustered together to minimize the communication cost. We make the trade-off by exactly evaluating every edge's weight, which is defined as the sensitivity of clustering the vertices of the edge to objective function 3. Let $u$ and $v$ be the two adjacent vertices of a direct-legal edge $a$, and let $G_a$ be the newly updated task graph after clustering vertices of the edge $a$, then the weight $w_a$ of edge $a$ is:

$$w_a = \Delta f_a = f(G) - f(G_a)$$

$$= d_a \Delta C(l) + t_{u,b}(u)C(b(u)) + t_{v,b}(v)C(b(v))$$

$$- t_{u+v,b(u+v)}C(b(u+v))$$  

subject to:

$$\text{sizeof}(u) < \text{sizeof}(b(v)), \forall u \in V \quad (4)$$

$$t_{u,b}(v) < \tau, \forall v \in V \quad (5)$$

$$D^h(G) < T^h \quad (6)$$

For the partitioning constraints, Inequality 4 represents the size constraint, Inequality 5 the period constraint, and Inequality 6 the deadline constraint of the task.

4.1 Legality of clustering vertices

For optimal partitioning, the first term of objective function 3 means every vertex should take its biasing PE, while the second term shows the vertices should be clustered together to minimize the communication cost. We make the trade-off by exactly evaluating every edge's weight, which is defined as the sensitivity of clustering the vertices of the edge to objective function 3. Let $u$ and $v$ be the two adjacent vertices of a direct-legal edge $a$, and let $G_a$ be the newly updated task graph after clustering vertices of the edge $a$, then the weight $w_a$ of edge $a$ is:

$$w_a = \Delta f_a = f(G) - f(G_a)$$

$$= d_a \Delta C(l) + t_{u,b}(u)C(b(u)) + t_{v,b}(v)C(b(v))$$

$$- t_{u+v,b(u+v)}C(b(u+v))$$  

$$(7)$$

For the partitioning constraints, Inequality 4 represents the size constraint, Inequality 5 the period constraint, and Inequality 6 the deadline constraint of the task.

The time constraints of a system are usually dealt with during allocation and scheduling. In allocation-independent partitioning, we estimate a task's execution latency by pre-allocating every vertex of the task graph on its biasing PE, because this tends to be the optimal implementation of the task as discussed above.

Based on Assumption 3, all adjacent successors of a vertex can be executed concurrently. So, the execution time of the task graph is the biasing longest delay, noted as $D^h(G)$ in Inequality 6, of the graph with each vertex having the latency of its biasing time $t_{u,b}(v)$ and each edge having the latency of $d_{u,v}$.

A main concern in system co-synthesis is balancing the system load [1]. As a result, the process in the task graph may not be allocated on its biasing PE after co-synthesis—there may be other tradeoffs which force the process off its biasing PE. In order to make the partitioning results have some extent of flexibility for allocation, our algorithm made a practical modification to the partitioning constraints in Inequality 4, 5 and 6, such that each partitioned process can be allocated on at least two PEs.

4.2 Clustering sensitivity to the objective function

For optimal partitioning, the first term of objective function 3 means every vertex should take its biasing PE, while the second term shows the vertices should be clustered together to minimize the communication cost. We make the trade-off by exactly evaluating every edge's weight, which is defined as the sensitivity of clustering the vertices of the edge to objective function 3. Let $u$ and $v$ be the two adjacent vertices of a direct-legal edge $a$, and let $G_a$ be the newly updated task graph after clustering vertices of the edge $a$, then the weight $w_a$ of edge $a$ is:

$$w_a = \Delta f_a = f(G) - f(G_a)$$

$$= d_a \Delta C(l) + t_{u,b}(u)C(b(u)) + t_{v,b}(v)C(b(v))$$

$$- t_{u+v,b(u+v)}C(b(u+v))$$  

$$(7)$$
between clusters.

lay of the task graph with less cost of increasing partitioning steps. By splitting a vertex into two vertices, be eliminated.

ter, this cluster is checked to be not mistaken by analyzing the structure of the graph. We this method tries to greatly decrease the longest delay.

longest delay is global. Our algorithm is iterative and uses hierarchical clustering, cluster splitting, and border adjusting stage, they can be effectively corrected by splitting.

4.3 Partitioning sensitivity to the longest delay

Analyzing the clustering sensitivity to the delay of the task graph is difficult, because there are several paths which have almost the same delay as the biasing longest delay $D^l(G)$. It is hard to predict those earlier mistakes by analyzing the structure of the graph. We use two methods to compensate for modeling errors which could badly affect the later stages of clustering.

First, while clustering an edge $a$, we need to check the increment of the graph’s longest delay. The increment is defined as $\Delta D_a = D^l(G_a) - D^l(G)$. If $\Delta D_a$ is larger than the biasing time of any vertex in this cluster, this cluster is checked to be not feasible and will be eliminated.

Second, we introduce a splitting method in our algorithm to correct bad clusters created by previous partitioning steps. By splitting a vertex into two vertices, this method tries to greatly decrease the longest delay of the task graph with less cost of increasing $f(G)$. So, only vertices on the longest path are considered and the vertex with the lowest differential coefficient $\Delta f(G)/\Delta D^l(G)$ is split. Our examples show that, if the bad clusters are not blocked in the hierarchical clustering stage, they can be effectively corrected by splitting.

5 Partitioning Algorithm

The partitioning problem under time constraints is much more complicated than that with no time constraints, since the partitioning sensitivity to the task delay is global. Our algorithm is iterative and uses hierarchical clustering, cluster splitting, and border adjusting between clusters.

Hierarchical clustering has long been used for hardware partitioning [16]. We use hierarchical clustering method as a major phase in partitioning because it has good characteristics for sensitivity analysis. But, hierarchical clustering is still a greedy method and may lead to local optimal solutions, since the earlier clustering decision could not predict the later partition’s topographical structure.

The splitting and adjusting steps globally examine the partitioned graph and try to climb out of the local optimum. The splitting method tries to reduce $D^l(G)$ to satisfy the deadline constraint with less sacrifice to $f(G)$, as discussed in Section 4.3. Adjusting method is seeking improvement to $f(G)$ by adjusting the border between clusters, which are clustered earlier but not good because of the later change of the graph structure.

The partitioning algorithm is given in Figure 6. Figure 5 illustrates how the algorithm might apply optimization phases to search the solution space.

The function $\text{BetterFeasible}(a_m, a_p)$ returns a better $\Delta f/\Delta D$ edge if $a_m$ is feasible, or $\phi$ otherwise. $a_p$ is the best of previously checked edges, with the highest $\Delta f/\Delta D$. $\text{BetterFeasible}()$ also allows such $a_p$, which will violate the deadline constraint of Inequality 6, taking into consideration the parallel property of a task graph. That is why some clustering results go exceeding the deadline constraint in Figure 5.

VertexSplit() and BorderAdjust() are based on max-flow min-cut algorithm [17], but with different criteria as proposed above. Because of the lack of space, we refer the reader to [17] for the details of the algorithm.
6 Experimental Results

Figure 7.(a) gives four examples for partitioning. The initial set of PEs is given by the user and the computation time of each module on each processing engine is estimated as in Figure 7.(b). For allocation-independent partitioning, each task graph of a system is partitioned independently in our algorithm. Figure 7.(c) gives the partitioned task graphs with each vertex representing a process, which clusters several modules in the original graph. The structure of the partitioned graph is flexible, depending on the structure of the original graph and the time constraints.

We implemented our partitioning algorithm in C++. The most time-consuming procedure is hierarchical clustering, which has the computation complexity of O(n^2), where n is the number of nodes and edges in the original data flow graph.

We constructed three embedded systems as examples in Table 1. Each one is described in two tasks presented in Figure 7. We feed the partitioned task graphs in Figure 7(c) and the unpartitioned(original) task graphs in Figure 7.(a) to Yen and Wolf's co-synthesis algorithm [1]. The co-synthesis results for partitioned and unpartitioned graphs of the three examples are presented in Table 1. Our algorithm considerably simplifies the data flow graphs of the embedded system. The partitioned graph, when given to the co-synthesis algorithm, always gives lower or equal cost solutions and allows the co-synthesis program to run much faster. The CPU time of our partitioning program running on a Sun Sparcstation SS20 is less than 0.4 second for every example in Figure 7.(a).

7 Conclusions

Process partitioning is an effective method to manage the complexity of distributed embedded system design. Since the system often consists of several threads of tasks, the real time system overhead of context switch must be considered. The definition of a process being completely executable makes the modeling of process partitioning consistent to that of distributed system scheduling and allocation problem. With the three phase co-synthesis strategy, complex embedded systems described with hundreds of modules can be synthesized efficiently. While data flow graphs are basic for the description of embedded systems, we are working on the extension of the partitioning modeling and algorithms to deal with a distributed embedded system described in control and data flow(CDF), or synchronous data flow(SDF) modeling.

Acknowledgements

Hou's work was supported by a scholarship from the State Education Commission of China. Wolf's work was supported by a grant from the NSF under grant 9424410.

References

Figure 7: Four examples for partitioning, (a) described in task graphs of modules, with communication data volumes, periods and deadlines; (b) the cost on each type of PE and the computation time of each module on each PE; (c) the partitioned graphs, with each process clustering several modules.

<table>
<thead>
<tr>
<th>Embedded systems</th>
<th>tasks</th>
<th>unpartitioned graphs</th>
<th>partitioned graphs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#PE</td>
<td>#Link</td>
</tr>
<tr>
<td>ex1</td>
<td>Task1, Task2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ex2</td>
<td>Task1, Task3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>ex3</td>
<td>Task3, Task4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Three examples for co-synthesis