Sensitivity-Driven Co-Synthesis of Distributed Embedded Systems*

Ti-Yen Yen and Wayne Wolf
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544

Abstract

This paper describes a new, sensitivity-driven algorithm for the co-synthesis of real-time distributed embedded systems. Many embedded computing systems are distributed systems: communicating periodic processes executing on several CPUs/ASICs connected by communication links. We use performance estimates to compute a local sensitivity of the design to process allocation. We propose a priority prediction method to schedule processes. Based on these techniques, we develop a gradient-search algorithm which co-synthesizes a heterogeneous distributed systems of arbitrary topology and the associated application software architecture. Experimental results show that our algorithm can find good implementation architectures in small amounts of CPU time.

1 Introduction

This paper describes new techniques for the co-synthesis of distributed embedded systems. We present an iterative improvement strategy which uses the sensitivity of the implementation to incremental modification. Our algorithm can simultaneously design the **hardware engine** which consists of a network of heterogeneous **processing elements** (PEs), either CPUs or ASICs, and the **application software architecture** which consists of allocating functions to PEs in the hardware engine and scheduling their execution. We refer to the hardware engine and application software architectures together as the **embedded system architecture**, since both the hardware and software architectures contribute to the system design.

Distributed co-synthesis is important because many embedded systems are heterogeneous distributed machines: Rosebrugh and Kwang [15] described a pen-based system built from four processors, all of different types; modern automobiles include up to 60 microcontrollers ranging in size from 4-bit to 32-bit; many 35mm cameras include several microprocessors.

Embedded system synthesis is **co-synthesis** because the hardware and software must be designed together to meet performance and cost goals. In contrast to traditional distributed system design or hardware-software partitioning, we cannot assume that the topology of the distributed system is given. Our co-synthesis algorithm selects the number of PEs, the type of each PE, as well as configuring the software architecture.

The synthesis of communication is important for distributed embedded system design. We do not discuss communication delay and cost in this paper. However, our methods for the allocation and scheduling of processes can be extended to handle communication.

2 Previous Work

Performance analysis techniques are essential to any co-synthesis algorithm. Our co-synthesis algorithm in this paper is based on an analytic delay estimation algorithm [20] which extends rate-monotonic analysis (RMA) [7, 8, 16] to derive tight delay bounds on periodic tasks executing on a distributed system. Our algorithm can handle problems which include sets of processes with data dependencies; each set has its own period and deadline; the computation times of processes and the periods are bounded but not necessarily constant.

A great deal of recent work has studied hardware-software partitioning, which targets a one-CPU-one-ASIC topology. The algorithm of Gupta and De Micheli [5] moves operations from hardware to software to reduce system cost. The algorithm of Ernst et al. [4] moves operations from software to hardware to meet performance goals. Vahid et al. [18] use a binary-constraint search algorithm to minimize hardware for performance constraints. Barros et al. [3] use a clustering algorithm to partition a program into hardware and software.

Shin et al. [17], Ramamritham et al. [13], and Sha et al. [16] survey real-time scheduling algorithms. There is a large amount of literature on scheduling or allocation of real-time distributed systems. The architectures of the systems are assumed to be given. Many of this work focus only on a single nonperiodic task [1, 14] and cannot handle the RMA model. In most real-

*This work was supported in part by a grant from the National Science Foundation.

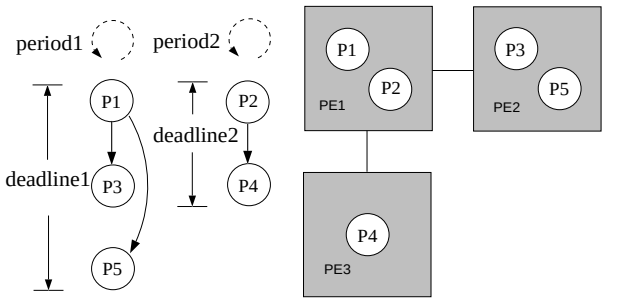


Figure 1: An embedded system specification and implementation.

time embedded system, different tasks run in different rates, and one process can be interrupted by another. Many algorithms [6, 12, 10] for periodic tasks in distributed systems form a big task with length of the least common multiple (LCM) of all the periods. The LCM method is not efficient when the periods are large and coprime; it is inaccurate to handle non-constant periods or computation times; it discourages static allocation and scheduling because it treats different instances of a task as different nodes for the length of LCM. These methods on distributed systems are not suitable for co-synthesis.

Prakash and Parker [11] formulated distributed system co-synthesis as an integer linear program (ILP). They could simultaneously allocate and schedule processes while designing the underlying distributed engine. However, their ILP formulation cannot handle periodic and preemptive scheduling of processes in the RMA model. Their ILP algorithm sometimes required hours to execute. Wolf [19] developed a heuristic algorithm for distributed system co-synthesis which gives results comparable to ILP in many cases, but this algorithm uses scheduling bounds similar to those of Prakash and Parker. D'Ambrosio and Hu [2] use simulation to judge the feasibility of a schedule during co-synthesis: they first enumerate a set of pareto-optimal solutions, then screen those candidates for feasibility by simulation. Enumeration of all possible configurations restricted them to explore only one-CPU architectures. Simulation is both time-consuming and not guaranteed to prove feasibility.

3 Problem Formulation

Our problem formulation is similar to those used in distributed system scheduling and allocation problems. A **process** is a single thread of execution, characterized by a **computation time**, which is a function of PE type to which it is allocated. We often use a table to show the computation time of a process on each type of PE which can implement the process. A **task** is a partially-ordered set of processes, which may be represented as an acyclic directed graph known as a **task graph**, in which a directed edge represents a data dependency. A problem specification may contain several concurrently running tasks. Processes and tasks are illustrated on the left-hand side of Figure 1. Each task is given a **period** (sometimes referred to as a **rate constraint**), which defines the time between two consecutive initiations, a **hard deadline**, which defines the maximum time allowed from initiation to termination of the task and must be satisfied, and a **soft deadline**, which describes the optimization goal of the task delay but does not have to be satisfied. The computation time of a process or the period of a task can be a constant or an interval specified by a lower bound and an upper bound. Release times (i.e., delayed initiation of a process) and multiple deadlines can be modeled by inserting dummy processes—processes with delay but not allocated on any physical PE—in the task graph.

Co-synthesis produces an embedded system architecture. As illustrated in the right-hand side of Figure 1, the hardware engine architecture is a labeled graph whose nodes represent PEs and whose edges represent communication links. The **allocation** is given by a mapping of processes onto PEs. Some processes may be implementable by either a CPU or an ASIC; we assume that the processes have been partitioned so that they do not cross CPU-ASIC or CPU-CPU boundaries. The **schedule** of processes is an assignment of priorities to processes. The CPU always executes the highest-priority ready process to completion. There is a cost associated with each PE type. Cost can be the price, area, or power consumption of the system. The designer can specify a hard constraint and a soft constraint on the total system cost.

4 Sensitivity Analysis

Embedded system performance is not characterized by a single number, since each task can have its own deadline to meet. In non-real-time systems, we can evaluate the system by the magnitude of a single number—the total execution time [17]. In real-time systems, the satisfaction of the deadlines is more important than shorter delays. Unlike design problems on a fixed architecture, we need also take into consideration the change in other criteria such as price, area, or power consumption, since the underlying hardware may change.

As in most gradient-search methods, we compute a local **sensitivity**: given the current design, we estimate how much the system performance and cost will change when a single process is reallocated. Given a design goal attribute—a task delay, cost—let the corresponding hard constraint or deadline be h_i and the soft constraint or deadline be s_i. Suppose the value of the attribute in the current solution is u_i and the

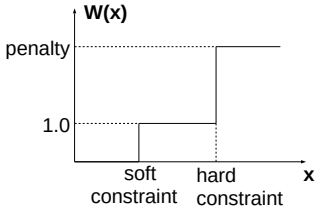


Figure 2: The weight function which emphasizes the change above the hard constraint and ignores the change below the soft constraint.

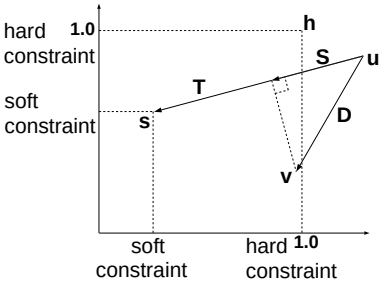


Figure 3: The illustration for the sensitivity vector **S**, which is the orthogonal projection of the displacement vector on the target vector. The current solution is at **u**, and the next solution is at **v**. The sensitivity is the magnitude of **S**.

value will become v_i after a reallocation. The values of u_i and v_i for a task delay are estimated by a performance analysis algorithm [20] with the scheduling method discussed in Section 5. Define the i^{th} component of the *displacement vector* **D** to be

$$D_i = \frac{1}{h_i} \int_{u_i}^{v_i} W(x)dx$$

where $W(x)$ is a weight function given in Figure 2. In other words, D_i represents the amount of the change $v_i - u_i$, but we give higher weight (penalty) for the portion of change above the hard constraint, and no credit (zero) for the portion below the soft constraint. It is divided by h_i for normalization—the tighter the hard constraint is, the higher the weight on the change. The displacement vector characterizes the *nonlinear* design goal under the hard and soft constraints. Let the i^{th} component of the *target vector* **T** be

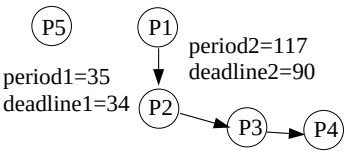
$$T_i = (s_i - u_i)/h_i$$

As shown in Figure 3(b), because the final goal is the soft constraint, the vector from the current position to the soft constraint is the direction in which we want to move. We also normalize it by h_i. The sensitivity S is the magnitude of the projection of **D** on **T**:

$$S = \frac{\mathbf{D} \cdot \mathbf{T}}{||\mathbf{T}||}$$

This tells how much closer we move the design towards the target, as illustrated in Figure 3.

The target vector automatically balance weights among different design goals, according to the current system architecture. The attributes with tighter hard constraints and the attributes whose current values are farther from soft constraints are weighted more heavily. A positive sensitivity implies an improvement, while a negative value means the result may become worse.



Two task graphs

PE type	Computation time				
	P_1	P_2	P_3	P_4	P_5
X	–	20	20	–	10
Y	15	–	–	26	–
Z	–	–	–	–	17

Process characteristics

Process	P_1	P_2	P_3	P_4	P_5
latest initiation time	0	15	35	55	0
latest termination time	15	35	55	81	17
latest required time	50	64	64	90	34
fractional deadline	50	49	29	35	34

Calculation of fractional deadlines

Figure 4: An example for priority prediction.

5 Priority Prediction

In the computation of sensitivities, we need to estimate the delay values u_i and v_i for an attribute corresponding to task delays. Determining reallocation is not enough to estimate the delays; we also need to reschedule these processes on each PE. We assume the deadline of each process does not exceed the period, in which case the inverse-deadline priority assignment [7] is optimal for one processor. However, in our model the deadline is specified end-to-end for a whole task, not for individual processes. We develop a heuristic to use the inverse-deadline priority assignment.

We define the **fractional deadline** of a process—the portion of the task deadline which a particular process must meet—as follows. Our performance analysis algorithm for the worst-case task delay [20] calculates the **latest request time** and **latest finish time** relative to the start of a task for each process. Assign each process a weight equal to its latest finish time minus its latest request time. For each CPU R, temporarily assign weight zero for the set of processes $\mathcal{J}^R$ allocated on R in the task. Then apply the longest-path algorithm backwards from the end of the task. The **latest required time** of each process in $\mathcal{J}^R$ is the hard deadline the task minus the longest

path weight of the process. The calculation of latest required times is similar to the technique in as-late-as-possible (ALAP) scheduling of high-level synthesis [9]. The fractional deadline d_i of each process $P_i \in \mathcal{J}^R$ is its latest required time minus its latest request time. We can then order the priority by d_i—the shorter the fractional deadline is, the higher the priority is.

Example 1 For the example in Figure 4, suppose in the current design, P_2 and P_3 are allocated to a PE of type X, P_1 and P_4 are allocated to a PE of type Y, and P_5 is allocated to a PE of type Z. If we move P_5 from Z to X, the three processes P_2, P_3, and P_5 will share the same PE. According to the fractional deadlines, their priorities should be ordered as $P_3 > P_5 > P_2$. The task deadlines are satisfied under this schedule. Among six possible schedules for three processes, this is the only feasible schedule for the example. ■

Unfortunately, when we reallocate a process, the delay information about latest initiation times and latest termination times will change. We should use the new delay information to decide the process deadlines for priority assignment, but before we schedule the priority we cannot know the new delay values. We solve this problem by using the delay information in the current solution before the reallocation to *predict* the fractional deadlines of processes and assign priority, and use the new schedule to compute the new delay information. The new delay information is used to decide priority assignment in the next step. Since we only move one process at a time, the change in the delay is usually not large, making this an acceptable approximation.

6 Idle-PE Elimination

In the computation of sensitivities, we need to know the total system cost. The system cost is usually the sum of the cost of each individual component. Consider the situation where two processes are in a PE R, and it is feasible to move these two processes to another PE, remove R, and reduce the cost. Since we are allowed to move only one process at a time, when the first process on R is reallocated, cost is not reduced immediately and such a movement may induce additional delay caused by a higher load on another PE. On the other hand, if R cannot be removed for a feasible solution, it might be desirable to move some processes from a highly-utilized PE to R to increase the performance. To maximize the satisfaction of goals during both the early and late stages of optimization, we use different criteria at different stages of synthesis:

1. **idle-PE-elimination** If R is the least-utilized PE, add the product of the cost and the processor utilization of R to the total system cost. When we move one of the processes from the least-utilized PE to another PE, we can immediately determine that the system cost is reduced, which increases the possibility to accept the first move and then take the other processes away in the next steps and remove R.
2. **load-balancing** After we have removed as many PEs as we can, we calculate the cost function as usual without considering the least PE utilization, concentrating on balancing PE utilization to increase the performance whenever it is possible.

The idle-PE-elimination criterion helps the solution jump off local minima; the load-balancing criterion brings the solution back to a minimum if a better solution cannot be found.

7 The Co-synthesis Algorithm

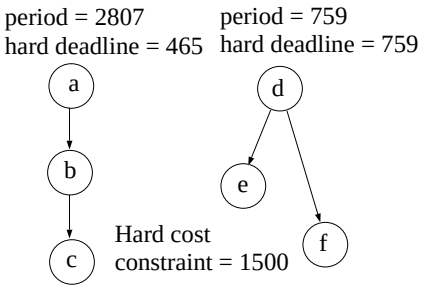
Our synthesis algorithm uses an optimization procedure, `findbest()`, which is parameterized by the optimization criterion to be used. The procedure is called twice: once with the idle-PE-elimination criterion and again with the load-balancing criterion. The `findbest()` procedure operates as follows:

1. Compute the sensitivity for each possible movement of a process from a PE to another, as described in Section 4. If there are $|P|$ processes and $|R|$ PEs in the current solution, there are at most $|P| \times (|R| - 1)$ possible movements. Exclude those movements whose sensitivities are negative.
2. When no movement remains feasible in step 1, create a new PE of a certain type and move a process to the new PE. Compute the sensitivities for each possible PE type and each process. If there are $|R_t|$ different PE types, there are at most $|P| \times |R_t|$ possible creations. Exclude the choices whose sensitivities are negative. If there is still no feasible movement, return.
3. Among the remaining movements or creations, choose the one with highest sensitivity. Make such a movement and reschedule each PE using priority prediction mentioned in Section 5.
4. Repeat steps 1–3 until no movement is feasible.

Our complete synthesis algorithm consists of the following steps:

1. Create an initial solution by assigning only one process to each PE. The PE with highest performance-to-cost ratio is chosen for each process.
2. Call `findbest(idle-PE-elimination)`.
3. Call `findbest(load-balancing)`.
4. For each PE R_i, replace R_i with a lower-cost PE R_j $(i \neq j)$ if R_j can implement all the processes on P_i and no hard deadlines are violated.

The priority prediction technique described in Section 5 cannot be applied to the initial solution, because



Task graphs

PE type	Cost	Computation time					
		a	b	c	d	e	f
X	\$800	179	100	95	213	367	75
Y	\$500	204	173	124	372	394	84
Z	\$400	210	193	130	399	494	91

The computation time

Figure 5: A small example *ex1*.

there is no previous solution providing delay information for prediction. However, since there is only one process on each CPU in our initial solution, there is no need for scheduling. All the following solutions can depend on the delay information in the previous one for scheduling. The final step in the algorithm tries to replace PEs which are poorly utilized.

We disallow a processor to be moved back to the same PE or the same type of PE it is moved away from in a previous iteration. Under this condition, the number of iterations in each call to `findbest()` is bounded by $|P|^2 \times |R_t|$, because each process can be allocated to at most $|R_t|$ different PE types, and we can have at most $|P|$ PEs for each PE type. In practice, the algorithm converges on a solution in many fewer steps.

8 Experimental Results

We implemented our algorithm in C++ and performed experiments on several examples. All experiments were performed on a Sun Sparcstation SS20. The results of all our experiments are summarized in Table 8.

The first example, *ex1* is a small example shown in Figure 8. We assume the designer wants to reduce the delay and the cost as much as possible, so the soft constraints and soft deadlines are set to zero to encourage optimization whenever it is possible. The embedded system architecture, total cost and the satisfaction of real-time deadlines after each iterative step are given in Table 8.

The second example is created by D'Ambrosio and Hu [2]. Our result is the second best shown in [2]. However, by exhaustive simulation, we found their best design with the minimum cost of 3.25 is actually infeasible. Our result is the best feasible design achievable. They used simulation to verify whether a system configuration is feasible but simulation does not guarantee the deadlines are satisfied.

Our third example is the second example of Prakash and Parker [11]; their algorithm works on a single task graph. We use their assumption about communication cost and delay, and put a hard cost constraint of 10. Our result for the task delay is 7 which is worse than, but similar to 6 in their result. Their integer linear programming approach is optimal but takes hours on this example, while ours takes only seconds. The results are shown in the row labeled *prakash-parker-1*.

We also combine Prakash and Parker's two examples together and assign the periods as well as the deadlines of 7 and 15 to the two tasks. The results are given under *prakash-parker-2*. This example demonstrates how our algorithm can co-synthesize from multiple disjoint task graphs.

9 Conclusions

Distributed computers are often the most cost-effective means of meeting the performance requirements of an embedded computing application. Embedded distributed computing is a particularly challenging design problem because the hardware and software architectures must be simultaneously optimized. We have presented a new co-synthesis algorithm for heterogeneous distributed systems of arbitrary topology. We plan to extend this work to synthesize communication, handle control flow, and consider fault tolerance. We believe that algorithms such as this are an important tool for the practicing embedded system designer.

References

[1] W. W. Chu and L. M.-T. Lan. Task allocation and precedence relations for distributed real-time systems. *IEEE Transactions on Computers*, C-36(6), June 1987.

[2] J. G. DAmbrosio and X. Hu. Configuration-level hardware/software partitioning for real-time embedded systems. In *Proceedings, International Workshop on Hardware-Software Co-Design*, 1994.

[3] E.Barros, W. Rosenstiel, and X. Xiong. A method for partitioning UNITY Language in hardware and software. In *Proceedings, European Design Automation Conference*, 1994.

[4] R. Ernst, J. Henkel, and T. Benner. Hardware-software co-synthesis for microcontrollers. *IEEE Design & Test of Computers*, 10(4), December 1993.

[5] R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10(3), September 1993.

Step	embedded system architecture	hard deadlines	cost
1	(Y: e) (Z: a) (Z: b) (Z: c) (Z: d) (Z: f)	unsatisfied	$2500
Stage 1: idle-PE elimination			
2	(X: b) (Y: e) (Z: a) (Z: c) (Z: d) (Z: f)	unsatisfied	$2900
3	(X: b d) (Y: e) (Z: a) (Z: c) (Z: f)	satisfied	$2500
4	(X: b e d) (Z: a) (Z: c) (Z: f)	satisfied	$2000
5	(X: b e d) (Z: c a) (Z: f)	satisfied	$1600
6	(X: b e d) (Z: c a f)	satisfied	$1200
7	(X: b e d f) (Z: c a)	satisfied	$1200
Stage 2: load balancing			
8	(X: b e d) (Z: c a f)	satisfied	$1200

Table 1: The iterative optimization for *ex1*. Each processor in the system is shown by its type, followed by a colon, and then the processes allocated on it are listed from the highest priority to the lowest.

Example	Problem size			The result		CPU
	#task	#process	#PE type	#PE	cost	time
ex1	2	6	3	2	$1200	0.72s
dambrosio-hu	9	9	11	1	$3.50	31.2s
prakash-parker-1	1	9	3	3	$10	30.03s
prakash-parker-2	2	13	3	3	$12	119.78s

Table 2: The problem size, the final result, and the CPU time of running our algorithm for each example.

[6] C. J. Hou and K. G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems. In *Proceedings, Real-Time Systems Symposium*, 1982.

[7] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2, 1982.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), Jan. 1973.

[9] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2), February 1990.

[10] D.-T. Peng and K. G. Shin. Static allocation of periodic tasks with precedence constraints. In *Proceedings, International Conference on Distributed Computing Systems*, 1989.

[11] S. Prakash and A. C. Parker. SOS: synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16, 1992.

[12] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *Proceedings, International Conference on Distributed Computing Systems*, 1990.

[13] K. Ramamritham and J. A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1), January 1994.

[14] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[15] C. Rosebrugh and E.-K. Kwang. Multiple microcontrollers in an embedded system. *Dr. Dobbs Journal*, January 1992.

[16] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1), January 1994.

[17] K. G. Shin and P. Ramanathan. Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1), January 1994.

[18] F. Vahid, J. Gong, and D. D. Gajski. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In *Proceedings, European Design Automation Conference*, 1994.

[19] W. Wolf. Architectural co-synthesis of distributed embedded computing systems. Submitted to IEEE Transactions on VLSI Systems, 1994.

[20] T.-Y. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. In *Proceedings, IEEE International Conference on Computer Design*, 1995.