IEICE TRANS. FUNDAMENTALS, VOL.E84–A, NO.2 FEBRUARY 2001

PAPER

Hardware-Software Multi-Level Partitioning for Distributed Embedded Multiprocessor Systems

Trong-Yen LEE[†], ***Regular Member***, **Pao-Ann HSIUNG**[††], ***and*** **Sao-Jie CHEN**[†], ***Nonmembers***

SUMMARY A novel *Multi-Level Partitioning* (MLP) technique taking into account *real-world constraints* for hardware-software partitioning in *Distributed Embedded Multiprocessor Systems* (DEMS) is proposed. This MLP algorithm uses a gradient metric based on hardware-software cost and performance as the core metric for selection of optimal partitions and consists of three nested *levels*. The innermost level is a simple binary search that allows quick evaluations of a large number of possible partitions. The middle level iterates over different possible allocations of processors (that execute software) to subsystems. The outermost level iterates over the number of processors and the hardware cost range. Heuristics are applied to each level to avoid the expensive exhaustive search. The application of MLP as a recently purposed *Distributed Embedded System Codesign* (DESC) methodology shows its feasibility. Comparisons between real-world examples partitioned using MLP and using other existing techniques demonstrate contrasting strengths of MLP. Sharing, clustering, and hierarchical system model are some important features of MLP, which contribute towards producing more optimal partition results.

key words: *distributed embedded multiprocessor system, multi-level partitioning, codesign, clustering, sharing*

1. Introduction

Most of today's computerized embedded systems such as coal mine monitoring and control systems [1], automatic parking management systems, flexible manufacturing systems, and security systems are all *Distributed Embedded Multiprocessor Systems* (DEMS). They are distributed in more than one physical location, have multiple processors and ASICs performing individual functions at different locations, are embedded within environment systems, and contain both hardware and software parts. Traditional hardware-software copartitioning techniques are based on task-graphs with edge weights denoting communication costs, which is a flat representation of system functionalities, and thus results in loss of information that could otherwise be used for obtaining better copartitioning results. Traditional techniques thus fail to obtain good partitions when DEMS are targeted.

A novel technique called *Multi-Level Partitioning* (MLP) is proposed for DEMS here, which is based on objects, thus maintaining *system hierarchy*. Due to its distributed characteristics, a DEMS has an inherent hierarchy that allows a very large number of possible hardware-software partition choices. The large number of partitions is a result of the multiplication of manifold choices to be made at each level: total number of system processors, different allocation schemes of processors to subsystems, hardware-software copartitioning within a subsystem, sharing of hardware among subsystems, sharing of software among subsystems, various clustering possibilities for hardware, and various grouping possibilities for software on multiple processors. To cover all these possibilities in a DEMS, MLP takes into account *real-world system constraints* such as physical inter-distance between subsystems, sharing of CPUs and ASICs among subsystems, modularized hierarchical structure in distributed systems, clustering of hardware components, and grouping of software modules.

MLP consists of mainly three nested levels: the innermost *binary search copartitioning* (BSC) level, the middle *system structural partitioning* (SSP) level, and the outermost *codesign space exploration* (CSE) level. The inner BSC level is the core part of hardware-software copartitioning, where the metric used for comparison between two or more partitions is based on hardware cost, software cost, hardware performance, and software performance. Hardware clustering and software grouping are performed at this level. The middle SSP level iterates over possible allocations of processors (that execute software) to the subsystems in a DEMS. Heuristics are applied here to avoid exhaustive iteration over all possible allocations. Hardware sharing and software sharing are also considered at this level. The outer CSE level iterates over the number of processors feasible under user-given cost constraints. Heuristics are also applied at this level so that iteration is limited to near-optimal design points.

This article is organized as follows. Section 2 gives some previous work related to DEMS codesign methodologies, existing partitioning techniques that target distributed systems, and how MLP is related to existing techniques. Section 3 discusses *Object Modeling Technique* (OMT) upon which MLP is based. Section 4 describes MLP in detail. Section 5 demonstrates the usefulness of MLP through several examples. Section 6

Manuscript received December 20, 1999.
Manuscript revised July 17, 2000.

[†]The authors are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, R.O.C.
[††]The author is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Taipei, Taiwan, R.O.C.

concludes the article with some future work.

2. Previous Work

Hardware-software partitioning techniques for the 1-CPU-1-ASIC topology have been studied by many researchers. Gupta and De Micheli [2], [3] at Stanford University developed the VULCAN co-synthesis system, based on the Olympus high-level synthesis [4]. COSYMA (COSYnthesis of eMbedded Architectures), developed by Ernst et al. [5], [6] at Technical University of Braunschweig, is an experimental co-synthesis system for embedded controllers. Vahid et al. [7] proposed a binary-constraint search algorithm for hardware-software partition. The algorithm is based on an iterative improvement partition algorithm such as simulated annealing. Barros et al. [8] used a multi-stage clustering technique to partition a system specified by the UNITY language. Jantsch et al. [9] added a separate phase into GNU C compiler to partition the implementation of a C program on a Sparc CPU and an FPGA chip. Kalavade and Lee [10] proposed a GCLP algorithm for hardware-software partition. The application is represented by a directed acyclic graph, where nodes are processes and arcs are data or control precedences between nodes. The GCLP algorithm has been integrated into Design Assistant [11], a system-level codesign framework.

As far as the n-CPU/m-ASIC topology is concerned, related works are fewer. Yen and Wolf [12] proposed a sensitivity-driven method for the co-synthesis of real-time distributed embedded systems. The cosynthesis algorithm selects the number of processing elements (PEs), the type of each PE, allocates functions to PEs, and schedules their executions. Dick and Jha [13] proposed an algorithm for hardware-software cosynthesis of distributed embedded systems, namely MOGAC, which partitions and schedules embedded system specifications consisting of multiple periodic task graphs. MOGAC synthesizes real-time heterogeneous distributed architectures using an adaptive multiobjective genetic algorithm that can escape local minima. Recently, Dave, Lakshminarayana, and Jha [14] proposed a heuristic-based constructive cosynthesis technique, called COSYN, which includes allocation, scheduling, and performance estimation steps. COSYN takes periodic acyclic task graphs as input and generates a low-cost heterogeneous distributed embedded-system architecture meeting real-time constraints.

For hardware-software partitioning of distributed embedded multiprocessor systems, existing techniques include the following. Kalavade and Subrahmanyam [15] proposed two methods for multifunction partitioning, namely *hardware-oriented partitioning* and *consistency and hardware-oriented partitioning*, which assume that each application is specified by a directed acyclic graph. Recently, some research works have started considering system hierarchy for hardware-software partitioning problems in a variety of ways [16]–[18].

In comparison to previous researches on hardware-software partitioning, which were based on task graphs as system model, our partitioning method uses object-oriented (OO) hierarchy as system model. The task graph model basically assumes an arbitrary system architecture, without considering any restrictions on subsystem locations. In contrast, our OO model takes realistic physical restrictions into consideration for the target system architecture. This is more appropriate for the inherent architectural restrictions of distributed systems. Many cosynthesis algorithms do not have an explicit partitioning step. But, we are inclined towards an explicit partitioning step, especially for distributed systems. Task graph based partitioning flattens out system architecture, thus information on subsystem boundaries is lost, which results in no sharing possible among subsystems. Our MLP emphasizes on sharing during partitioning for a more optimal partition result.

3. Object Modeling Technique

Multi-Level Partitioning is based on *objects*. MLP takes objects as input and the popular *Object Modeling Technique* (OMT) developed by Rumbaugh et al. [19] is used for specifying input to MLP. A distributed embedded multiprocessor system can be described using the three models of OMT, namely *Object Model*, *Dynamic Model*, and *Functional Model*.

Relationships between objects determine the possibility of object sharing among subsystems. Among all the different kinds of object oriented relationships that may exist between two objects, such as whole/part, a-kind-of, and others, only those relationships that represent *mutually exclusive temporal behavior* or *schedulably implementable dynamic behaviors* can be used as indicators of possible sharing.

Events in state diagrams determine the dynamic relationships among objects, such as method invocation, object creation, object destruction, etc. These dynamic relationships indicate how closely coupled are two objects in an object oriented system model. Hardware clustered into individual ASICs and software grouped into scheduled programs on processing elements require the knowledge of how closely two or more objects interact dynamically.

Functional model can help estimate the execution steps required for a particular process, which might eventually be used for estimating the performance of a hardware component or a software program.

3.1 Illustration Example

In the following, we will describe a distributed embedded system which will be a running example for illus-

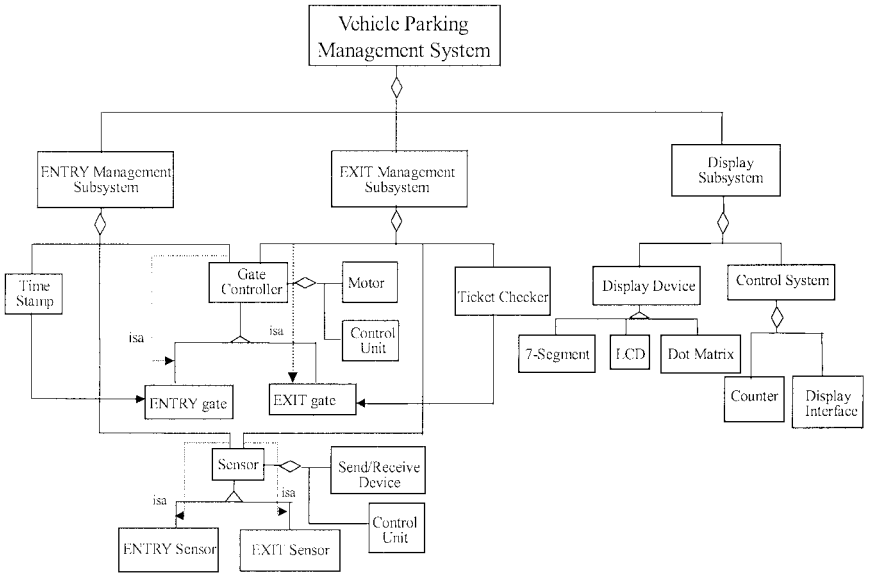


Fig. 1 Object model of VPMS.

trating our partitioning methodology. This example consists of a project for designing a vehicle parking system, we call this system *Vehicle Parking Management System* (VPMS). A preliminary case study on VPMS illustrating hardware-software codesign of distributed systems can be found in [20]. VPMS consists of three subsystems: ENTRY management, EXIT management, and DISPLAY. As both of the ENTRY and EXIT subsystems allow vehicles to pass through them one by one, they are similar in most respects. DISPLAY subsystem shows the current number of vacant parking space available in a parking lot or garage.

An ENTRY (or an EXIT) subsystem consists of three parts: a ticket facility, a gate controlled by a gate-motor, and a pair of sensors. The ticket facility at the entry stamps the current date and time and gives a new ticket to an in-coming vehicle. The ticket facility at the exit checks whether the ticket (parking) fees have been paid and the current time is within 15 minutes of the ticket fee payment. After a positive response is received from the ticket facility, a gate controller opens the ENTRY (EXIT) gate to allow a vehicle to drive in (out). A pair of sensors are located after the gate (in the direction of the vehicle, that is, further in for the entry and further out for the exit). The sensors then send a signal to the gate controller to close the gate after a vehicle has passed by. At the same time, the sensors also send a signal to the display for updating the displayed number of parking vacancies. The DISPLAY system consists of a control system (counter and display interface) and a display device such as 7-segment display, LCD, or dot matrix LED display. The counter value (*count*) indicates the number of available parking vacancies.

Constraints for the VPMS system include: a maximum cost of \$1,300, a maximum *display response time* (DRT) of 14,000 μs, and a maximum ENTRY (EXIT) *gate response time* (GRT) of 250 μs. An n-CPU/m-ASIC system cost includes the total cost for m ASICs in the system and the total cost for n PEs used for executing software. Display response time (DRT) includes a signal sensing period, a signal processing time, the time for a signal transmission from sensor to display, the time for updating (incrementing or decrementing) the count value, and the time for updating the display. Gate response time (GRT) includes a signal sensing period, a signal processing time, the time for a signal transmission from sensor to gate, the time for a motor driver to respond to a signal, and the time for a signal to be transmitted from a motor driver to a gate.

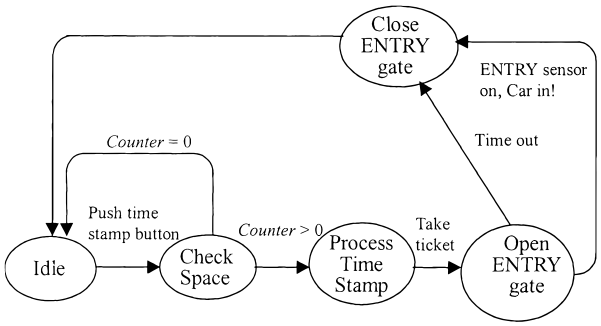


Fig. 2 Dynamic model of ENTRY management subsystem.

The three models of VPMS are illustrated in Fig. 1, Fig. 2, and Fig. 3, respectively. Some dynamic and functional models are not shown here.

4. Multi-Level Partitioning

Since our target systems are distributed with n-

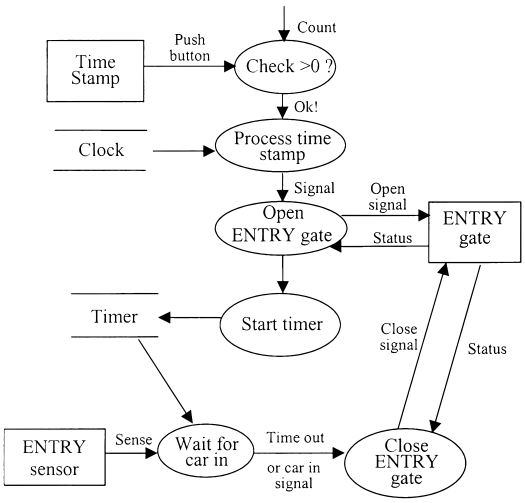


Fig. 3 Functional model of ENTRY management subsystem.

CPUs/m-ASICs, the inherent hierarchy in system structure necessitates a *multiple level* partitioning scheme. We must explore not only how many CPUs and ASICs to use, but also where they must be located or distributed in the system. For example, if we have a system consisting of three subsystems, such as VPMS, and if we decide on using two CPUs for software implementation and execution, we must also decide on where the two CPUs are located among the three subsystems. We propose a *Multi-Level Partitioning* (MLP) algorithm which consists of three nested levels called *codesign space exploration* (CSE), *system structural partitioning* (SSP), and *binary search copartitioning* (BSC) as described in the following three subsections. The flow diagram for MLP is shown in Fig. 4.

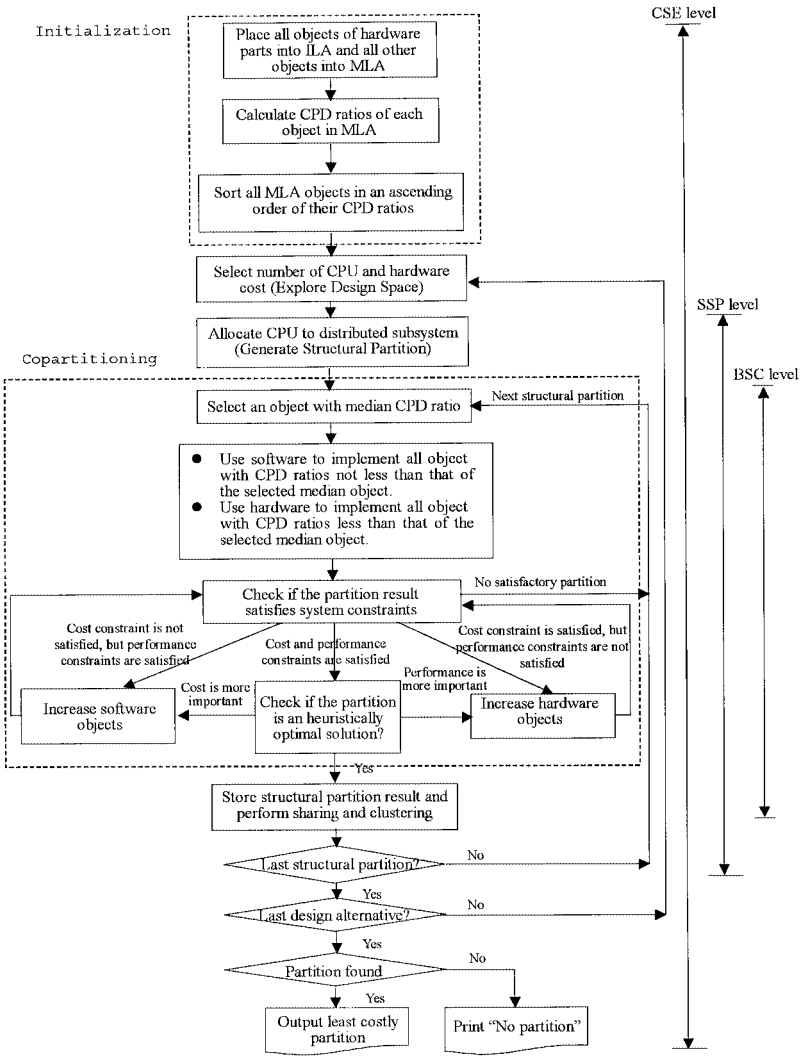


Fig. 4 Flow diagram of multi-level partitioning.

4.1 Codesign Space Exploration Level

At the CSE level, the outermost level of partitioning, we iterate on the possible number of CPUs to use for software implementation and execution and the possible hardware costs after the cost of CPUs is deducted from the total maximum cost bound. In general, each subsystem can have either zero or some positive number of processors, depending on the system cost bound. In *Distributed Embedded System Codesign* (DESC), by default the maximum number of CPUs in a system is constrained by the total number of different parts that could be implemented as hardware or software. Such parts are called *codesign parts*. System designers can easily override this default restriction, but doing so lengthens the period of design space exploration due to a much larger design space size. Suppose a distributed embedded system under design has n codesign parts. Currently, exhaustive search is carried out under the default of at most n processors. This level (loop) produces a multiplicative factor of n in the overall partitioning complexity. For the VPMS example, there are three codesign parts: Counter, Sensor Driver, and Motor Driver. Hence, by default in DESC at most three CPUs are considered. We begin by no CPU for the system, then one, two, and finally three. Hence, this design space exploration step incurs a factor of 4 in the partitioning complexity.

4.2 System Structural Partitioning Level

At the second level, we must decide where the selected CPUs are distributed among the different codesign parts. This leads to a combination and permutation problem. For example, for one CPU and k parts, there are only k different ways (structural partitions). For two CPUs and k parts, there are $k + C_2^k$ different structural partitions, where C_2^k is the total number of ways (combinations) of selecting 2 objects out of k. For three CPUs and k parts, there are $k + k \times (k-1) + C_3^k$ different structural partitions. With an increase in the number of CPUs, the total number of ways (structural partitions) increases at an exponential rate. Exhaustive evaluation of all possible structural partitions would require an unacceptable amount of time and space. Hence, heuristics are applied at this level in DESC partitioning.

Codesign parts that have one or more hard real-time constraints have a higher probability of being implemented in hardware, while the other parts with relatively soft constraints will be implemented in software. This is because dedicated ASIC hardware usually can be optimized to meet hard constraints, while software optimization is limited by the capabilities of the processor executing the software. A useful heuristic discovered here is that we can dedicate processors (that execute software) to only those parts that do not have hard constraints.

Furthermore, codesign parts that have a greater number of associated soft constraints must usually strive to meet its deadlines or specifications. Thus, another heuristic that comes handy is that parts with a greater number of constraints are assigned more number of processors. In the structural partitioning of DESC, we use approximated ratios of processors depending on the degrees of constraint satisfaction requirements for each part. Parts that are to be implemented as hardware are called *hardware parts*, while the parts that are to be implemented as hardware-software are called *hardware-software parts*.

4.3 Binary Search Copartitioning Level

The first two levels concentrated on the distributed characteristics of a system. This level will form the core part of hardware-software copartitioning. Here, we do not start from either of the two extreme solutions: all-hardware and all-software. Rather we start somewhere in-between, we start moving towards the optimal feasible solution based on two simple *assumptions* as follows. First assumption is that hardware implementations always cost more than software solutions. This is true in general when costs are amortized over several components of a system. Second assumption is that hardware implementations always perform better than software solutions. This is true in general because hardware ASICs can be optimized to a greater extent than software. Software optimizations are often restricted by compiler technology and microprocessor architecture that is hosting and executing it. Though these two assumptions may sound not so realistic at first, yet they have been validated by most design experiences [21], [22].

The copartitioning flow diagram and algorithm are given in Fig. 4 and Algorithm 1 (Fig. 5), respectively. In Algorithm 1 (Fig. 5), Steps (2)–(5) correspond to Initialization in Fig. 4. The first for-loop in Step (6) corresponds to the CSE level. The second for-loop in Step (9) corresponds to the SSP level. Steps (10)–(40) correspond to the BSC level.

Two linear arrays are used to store system objects during copartitioning, namely, *Immovable Linear Array* (ILA) and *Movable Linear Array* (MLA). ILA is used to store all objects that belong to hardware parts, while MLA is used to store the objects of hardware-software parts. Objects in ILA are not movable as they are already selected for hardware implementations. ILA is not used in this phase of copartitioning. Copartitioning will be performed only on the objects in MLA. Each component in a system under partition is associated with a *Cost-Performance Difference* (CPD) ratio as defined below:

```
HW/SW_PARTITION(N_1, N_2, N_3, ..., N_i, ..., N_r)
/* N_1, N_2, N_3, ..., N_i, ..., N_r, where N_i is a system object */ 1
Generate Immovable Linear Array (ILA) and
Movable Linear Array (MLA).                                          2
Calculate Cost-Performance Difference (CPD) ratio for
each MLA object.                                                     3
Sort all MLA objects in an ascending order of their CPD
ratios such that MLA = ⟨M_1, M_2, M_3, ..., M_m⟩                     4
u:= Number of PE;                                                    5
for (p = 0, p ≤ u, p++)                                              6
{
 HW_Cost := Max_Cost − Cost(PE) × p;                                 7
 sp(p):= {all ways of distributing p CPU
          among the subsystems};                                     8
 for each structural partition s ∈ sp(p) do                          9
 {
  k := 1, j := m; /* where k is called the lower bound
  object index, j is called the upper bound object index */          10
  i := ⌈m/2⌉ /* where i represents the divider object index */      11
  Use software to implement objects from M_i to M_j and
  use hardware to implement objects from M_k to M_{i−1}              12
  while true do                                                      13
  {
   switch(cost and performance estimations(s))                       14
   {
    Case 1: Cost constraint not satisfied,
            but performance constraints satisfied                    15
        j := i;                                                      16
        i := i − ⌈(i−k)/2⌉;                                          17
        break;                                                       18
    Case 2: Cost and performance constraints satisfied               19
        if the partition result is heuristically optimal{            20
            s' = Store structural partition result for s;            21
            Share_Components(s');
            /* Refer to Algorithm 2
    (Fig. 6)*/                                                       22
            Cluster_Components(s');
            /* Refer to Algorithm 5
    (Fig. 9)*/                                                       23
        }                                                            24
        else { if performance is more important{                     25
                k := i;                                              26
                i := i + ⌈(j−i)/2⌉; }                                27
            else { j := i; /* cost is more important */               28
                 i := i − ⌈(i−k)/2⌉;}                                29
        }                                                            30
        break;                                                       31
    Case 3: Cost constraint satisfied, but
            performance constraints not satisfied                    32
        k := i;                                                      33
        i := i + ⌈(j−i)/2⌉;                                          34
        break;                                                       35
    Case 4: No satisfactory partition                                36
            break;                                                   37
   } /* end of switch */                                           38
   if (heuristically optimal partition found or
       no satisfactory partition) break;                           39
  } /* end of while */                                             40
 }                                                                   41
}                                                                    42
if partition found then output least costly partition                43
else print "No partition"                                            44
```

Fig. 5 Multi-level partitioning algorithm (Algorithm 1).

$$CPD(x) = [HW\_Cost(x) - SW\_Cost(x)] \Big/ \frac{|HW\_Perf(x) - SW\_Perf(x)|}{Perf\_Bound(x)} \quad (1)$$

where x is an object in MLA; $HW\_Cost(x)$ is either the actual cost or the VLSI area of x; $SW\_Cost(x)$ is either the cost of the main memory spent or the cost of the CPU used for executing x as a software program code; $HW\_Perf(x)$ is the hardware response time; $SW\_Perf(x)$ is the program execution time as implemented in a processor; and $Perf\_Bound(x)$ is the value of the performance bound associated with part x. Here, it is assumed that each part is associated with only one performance bound. The denominator in CPD(x) is a normalization of the performance difference, which is required for a fair comparison between different parts and performance bounds. The CPD metric is a useful hardware-software partitioning criterion. All objects are sorted in an ascending order of their CPD ratios and placed in a horizontal one-dimensional array called *Movable Linear Array* (MLA) from left to right.

The copartitioning method begins somewhere around the middle of the sorted sequence of objects in MLA. A median object is selected as an initial *divider or separator*. The role played by a divider is that all objects to the right of the divider, including the divider, are implemented in software and the rest (at the left of the divider) are implemented in hardware. The reason, that such an implementation is correct, is two-folds. Firstly, the objects in the left part of the sequence have a greater gain in performance if implemented as hardware (i.e., a larger difference between hardware and software performance) and at a smaller expense (i.e., a smaller difference between hardware and software costs). Secondly, the objects in the right part of the sequence have a greater gain in saving costs if implemented as software (i.e., a larger difference between hardware and software costs) and at a smaller loss in performance (i.e., a smaller difference between hardware and software performance). Thus, the intuition for the CPD definition is clear from the role of the divider in copartitioning.

The initial partition obtained is then tested for feasibility under the given system constraints on cost and performance. Four cases are encountered here. First, if the cost specifications are not satisfied but performance specifications are satisfied, then we must increase the software part by selecting a new divider towards the right of the current divider along the linear array of sorted objects. Second, if both cost and performance specifications are satisfied, then depending on whether preference is given to minimizing cost or to maximizing performance we move towards the left or right, respectively. This will lead to a more cost-oriented or performance-oriented heuristically optimal solution. Third, if the cost specifications are satisfied but performance specifications are not, then we must increase

```
Share_Components(s) {
 /* s = ⟨s1, s2, ..., sΨ⟩, si = (si1, si2) where si1 is the
 number of PE in subsystem Si and si2 is the number
 of ASIC in subsystem Si. si1, si2 ∈ {0, 1, ...}*/
 for (i = 1, i ≤ Ψ, i++) {
  for (j = i, j ≤ Ψ, j++) {
   if SLI(Si, Sj) ≤ STD {
    if (si1 > 0 ∧ sj1 > 0)
      Share_PE(Si, Sj);
    /* Refer to Algorithm 3 (Fig. 7)*/
    if (si2 > 0 ∧ sj2 > 0)
      Share_ASIC(Si, Sj);
    /* Refer to Algorithm 4 (Fig. 8)*/
   }
  }
 }
}
```

Fig. 6 Share components algorithm. (Algorithm 2)

the hardware part by selecting a new divider towards the left of the current divider along the linear array of sorted objects. Lastly, if both cost and performance specifications are not satisfied, then the algorithm declares that no feasible partition can be found for the given system under the given constraints.

4.4 CPU/ASIC Sharing

Often two or more subsystems of a DEMS are quite close to each other physically. For example, the ENTRY management subsystem and the EXIT management subsystem of VPMS can be located adjacent to each other and yet cannot be considered as one subsystem due to functional differences. Under light workloads, these two subsystems might share a single CPU for executing software programs. This reduces overall cost without affecting performance. Such overlapping of a subsystem with other subsystems in its vicinity is often not taken into consideration while partitioning by conventional techniques. This results in redundant hardware components and under-utilized processors in budget-limited embedded systems. MLP thus considers sharing of CPUs and ASICs among subsystems while partitioning. A sharing algorithm is given in Algorithm 2 (Fig. 6), which calls two functions given in Algorithms 3 (Fig. 7) and 4 (Fig. 8).

In contrast to hardware clustering and software grouping which are confined to a single subsystem, CPU and ASIC sharing require the consideration of two subsystems simultaneously. Examples of CPU sharing include time quantum-based multitasking on a single CPU (using multi-threading) and multiprocessor-scheduled programs. Examples of ASIC sharing include history-less counters (based on time-division), timers, and data compressor.

Sharing is not as simple as it appears to be. Several factors affect whether sharing can be implemented.

```
Share_PE(S1, S2) {
 /* S1 = {P11, P12, P13, ..., P1k} */
 /* S2 = {P21, P22, P23, ..., P2l} */
 for (i = 1, i ≤ k, i++) {
  MinP := 0; MinIC := −1; MoveDir := NULL;
  for (j = 1, j ≤ l, j++) {
   if PE_shareable (P1i, P2j) {
    /* PE_shareable(P1i, P2j) = ∧r=4...8 Cr(P1i, P2j)*/
    /* For Cr() see Eq. (3), P1i, P2j are PE. */
    if IC(P1i, P2j) < IC(P2j, P1i) {
     if (MinP = 0) or (IC(P1i, P2j) ≤ MinIC) {
      MinP := j; MinIC := IC(P1i, P2j);
      MoveDir := S2 to S1;
     }
    }
    else {
     if (MinP = 0) or (IC(P2j, P1i) < MinIC) {
      MinP := j;
      MinIC := IC(P2j, P1i);
      MoveDir := S1 to S2;
     }
    }
    if (MoveDir = S2 to S1) {
     P1i := schedule (P1i, P2MinP);
     remove (P2MinP);
     interconnect(P1i, S2);
    }
    else if MoveDir = S1 to S2 {
     P2j := schedule(P2MinP, P1i);
     remove(P1i);
     interconnect(P2j, S1);
    }
   }
  }
 }
}
```

Fig. 7 Share PE algorithm. (Algorithm 3)

Among them, one main factor is the physical distance. Sharing among subsystems should be allowed only among those that are near each other and not among faraway ones. A distance metric called *Sharing Threshold Distance* (STD) is proposed for deciding on whether two subsystems can share some hardware or software.

Definition 1: Sharing Threshold Distance (STD). STD is defined as a minimum physical distance between any two subsystems in a *Distributed Embedded Multiprocessor System*, which is required for these subsystems to share any component such as CPU or ASIC.

The actual distance between any two subsystems is called the *Subsystem Location Inter-distance* (SLI). When SLI(S_1, S_2) is not greater than a given value of STD, subsystems S_1 and S_2 are qualified for sharing. But, sharing is not that simple or straightforward, several conditions need to be satisfied for two subsystems

```
Share_ASIC(S1, S2) {
 /* S1 = Q11Q12Q13...Q1p */
 /* S2 = Q21Q22Q23...Q2q */ for (i = 1, i ≤ p, i++) {
  MinQ := 0; MinIC := −1; MoveDir := NULL;
  for (j = 1, j ≤ q, j++) {
   if ASIC_shareable(Q1i, Q2j) {
    /* ASIC_shareable(Q1i, Q2j) = ∧r=1...5 Cr(Q1i, Q2j)*/
    /* For Cr() see Eq. (3), Q1i, Q2j are ASIC. */
    if IC(Q1i, Q2j) ≤ IC(Q2j, Q1i) {
     if (MinQ = 0 or (IC(Q1i, Q2j) < MinIC) {
      MinQ := j; MinIC := IC(Q1i, Q2j);
      MoveDir := S2 to S1;
     }
    }
    else {
     if (MinQ = 0 or IC(Q2j, Q1i) < MinIC) {
      MinQ := j;
      MinIC := IC(Q2j, Q1i);
      MoveDir := S1 to S2;
     }
    }
    if (MoveDir = S2 to S1) {
     D = enhance(Q1i);
     remove(Q2j);
     interconnect(D, S2);
    }
    else if (MoveDir = S1 to S2) {
     D = enhance(Q2i);
     remove(Q1i);
     interconnect(D, S1);
    }
   }
  }
 }
}
```

Fig. 8 Share ASIC algorithm. (Algorithm 4)

to actually share anything.

Sharing can be easily turned on or off by a system designer. The threshold distance STD can be given specific values. When STD is zero, it implies that no sharing of any sort is allowed. STD can be given a special value denoting infinity, which implies all subsystems can take part in sharing. This is called *global sharing*. When STD is given a small positive value such as one meter, it is called *local sharing*.

Before going into the conditions that must be satisfied for sharing, we define an *Interconnect Cost Model* for interconnecting two subsystems via the shared CPU or ASIC.

Definition 2: Interconnect Cost (IC) Model. Suppose two subsystems S_1 and S_2 are qualified for sharing, that is, SLI(S_1, S_2) is not greater than a given STD. Let X_1 and X_2 be a component (PE or ASIC) in S_1 and S_2, respectively. The cost for using X_1 as the *shared* component between S_1 and S_2, such that all the functions of X_2 are performed by the shared X_1, is defined as follows:

$$\begin{aligned} \mathrm{IC}(X_1, X_2) = \; & \alpha \times SLI(S_1, S_2) \\ & \times \#Link(X_1, S_2) \\ & \times BW(X_1, S_2) + EC(X_1) \end{aligned} \tag{2}$$

where α is a parameter that depends on the interconnection technology, $SLI(S_1, S_2)$ is the *Subsystem Location Inter-distance* between S_1 and S_2, $\#Link(X_1, S_2)$ represents the number of links between X_1 and S_2, and $BW(X_1, S_2)$ represents the communication bandwidth between X_1 and S_2, and $EC(X_1)$ is the cost for enhancing X_1 such that both S_1 and S_2 can use X_1.

Here, a processing element (PE) is considered instead of a CPU. A PE consists of a CPU and associated memory or caches. PEs allow CPU-sharing between two subsystems because only with memory installed can we consider program scheduling. Henceforth, PE and CPU are used interchangeably. Given the IC model, we can now list the constraints that two subsystems and their components must satisfy for sharing. The constraints are listed as conditions $\mathbf{C_1}$ to $\mathbf{C_8}$, which must be satisfied by the components to be shared.

- $\mathbf{C_1}$**:** same functional specifications or superset/subset relationship,
- $\mathbf{C_2}$**:** mutually exclusive temporal behavior,
- $\mathbf{C_3}$**:** history-less (for example, combinational circuits and sequential circuits with resets),
- $\mathbf{C_4}$**:** the interconnect cost of two sharing subsystems should be less than the total cost of all shared components,
- $\mathbf{C_5}$**:** all real-time constraint specifications of the system should be met when sharing,
- $\mathbf{C_6}$**:** the total size of all the programs sharing a single PE should not be larger than the memory capacity of the shared PE,
- $\mathbf{C_7}$**:** the programs should be schedulable on the shared PE, including priority-based interrupt handling, and
- $\mathbf{C_8}$**:** there should be sufficient number of I/O ports for all the programs sharing the PE.

For any two components X_1 and X_2, we define the following function to represent satisfaction of conditions:

$$C_r(X_1, X_2) = \begin{cases} TRUE & \text{if condition } C_r \\ & \text{holds for } X_1,\ X_2, \\ FALSE & \text{otherwise.} \end{cases} \tag{3}$$

Sharing Algorithms: Some of the above constraints are suitable for PE sharing and some for ASIC sharing. Corresponding to the different architectural characteristics between software and hardware, we have two different algorithms: Algorithm 3 (Fig. 7) and Algorithm 4 (Fig. 8) for **Share_PE()** and **Share_ASIC()**, respectively. In the **Share_PE**() algorithm, suppose we have

two subsystems S_1 and S_2 such that they have k and l PEs, respectively. For each PE, P_{1i} in S_1, the algorithm finds the best possible sharing candidate in S_2 in terms of the PE shareable conditions ($\mathbf{C_4}$ to $\mathbf{C_8}$) and the interconnect cost (Definition 2) using three variables: *MinP*, *MinIC*, and *MoveDir*. *MinP* records the index of a PE in S_2 which is currently the best candidate for sharing. *MinIC* records the current minimum IC value. *MoveDir* records the sharing direction for the current candidate sharing. Sharing direction can be from S_1 to S_2 or from S_2 to S_1. Sharing is performed by scheduling all the programs from the redundant PE to the shared PE (schedule()), removing the redundant PE (remove()), and finally interconnecting the shared PE with the subsystem that has a redundant PE removed (interconnect()). The EC cost in IC (Definition 2) for PE is incurred in the interconnect() function, where I/O interface has to be constructed for connection to the shared PE. The other **Share_ASIC**() algorithm works in a similar strategy except it is for sharing ASICs that have different shareable conditions ($\mathbf{C_1}$ to $\mathbf{C_5}$). The enhance() function in **Share_ASIC**() algorithm is for increasing the connectivity of the shared ASIC. The EC cost in IC (Definition 2) for ASIC is incurred in enhance().

4.5 Hardware Clustering and Software Grouping

Previous works on hardware-software copartitioning mostly produced results about which components (tasks) are to be implemented in hardware and which in software. This information is often not adequate or complete for a system to be fabricated due to the lack of knowledge on exactly which components are to be implemented together as one ASIC and exactly which components are to be grouped into one program on a single PE. We call this information as *hardware clustering and software grouping*. Although system-level design methodologies might have techniques for clustering/grouping, yet we think that pre-design clustering/grouping would save design time and efforts.

Hardware clustering is performed by utilizing conventional hardware partitioning techniques such as the group migration method of Kernighan and Lin [23] and its extensions, simulated annealing [24].

Software grouping is performed by scheduling object functions on a set of processors. The number of processors and processor allocation scheme were selected in the codesign space exploration level and the system structural partitioning level, respectively. Several multiprocessor task scheduling algorithms such as *Largest Scheduled Parallelism First* (LSPF) [25], *Largest Width with Largest Processing Time first* (LWLPT) [26], and a heuristic algorithm [27] for multiprocessor task scheduling can be used here. In MLP, clustering is performed independently for each subsystem in a DESC as illustrated in Algorithm 5 (Fig. 9).

```
Cluster_Components(s) {
  /* s = ⟨s_1, s_2, ..., s_Ψ⟩, s_i = (s_i1, s_i2) where s_i1 is the number of PE in subsystem S_i and s_i2 is the number of ASIC in subsystem S_i. s_i1, s_i2 ∈ 0, 1, .....*/
  for (i = 1 to Ψ) do {
    Cluster_HW(S_i); /* Refer to [23], [24] */
    Group_SW(S_i); /* Refer to [25], [26], [27] */
  }
}
```

Fig. 9 Cluster components algorithm. (Algorithm 5)

4.6 Analysis and Validation of MLP

The MLP algorithm is analyzed in this section in terms of its execution complexity and important features. A formal validation of MLP is given in three theorems in Sect. 4.6.2.

4.6.1 Complexity Analysis

In analyzing the Algorithm 1 (Fig. 5) of MLP, if the number of objects is r and the number of subsystems is Ψ, then its complexity can be given as follows:

$$\begin{aligned}\tau_{MLP} &= Init\_time + BSC \\ &\quad \times \sum_{p=0,\ldots,\Psi} [sp(p) \\ &\quad \times (Share\_time + Cluster\_time)]\end{aligned}$$

$$\begin{aligned}\tau_{MLP} &= r + r\log r + \log r \\ &\quad \times \sum_{p=0,\ldots,\Psi} [sp(p) \times C_2^{\Psi} \\ &\quad \times (2 \times max_{1\le k<p}(p-k) \times k + r^2)] \quad (4)\end{aligned}$$

where $sp(p)$ is the number of ways in which p processors can be assigned to r objects, BSC is the time required for binary search copartitioning, *Init_time* is the time required for MLP initialization as illustrated in Fig. 6, *Share_time* and *Cluster_time* are the times required for **Share_Components()** and **Cluster_Components()**, respectively. From Eq. (4), it can be observed that MLP spends a certain amount of time in sharing and clustering. Thus, to find a better solution (i.e., one with a lower cost) MLP takes longer time than that required for a straightforward solution.

4.6.2 MLP Features and Validation

There are several qualities of our algorithm that deserve further investigations. Firstly, whenever there exists a feasible solution for a system, our algorithm will definitely output heuristically optimal solution. Secondly, if a feasible solution is found by the algorithm, then the final result will also be feasible. Finally, if a completely

Table 1 Partitioning results for three VPMS specifications with and without sharing.

Specification								
		VPMS-1			VPMS-2		VPMS-3	
STD(m)		1.0			1.0		1.0	
SLI(ENTRY, EXIT)(m)		6.0			**0.5**		**0.8**	
SLI(DISPLAY, EXIT)(m)		7.0			3.0		**0.5**	
SLI(DISPLAY, ENTRY)(m)		2.0			3.0		**0.5**	
Partitioning Results								
Number and Locations of PE in Subsystems (parts implemented)		3	(1)ENTRY (M_{ENTRY}) (2)EXIT (M_{EXIT}) (3)DISPLAY	2	(1)ENTRY/EXIT (M) (2)DISPLAY	1	(1)ENTRY/EXIT/ DISPLAY	
Number and Locations of ASIC in Subsystems (parts implemented)		2	(1)ENTRY (S_{ENTRY}) (2)EXIT (S_{EXIT})	1	(1)ENTRY/EXIT (S)	1	(1)ENTRY/EXIT/ DISPLAY	
System Cost($)			1,430		1,250		1,180	
Performance	DRT (μs)		13,200		13,200		14,020	
	GRT (μs)		210		210		1,030	
MLP Execution Time (sec)			0.602		3.857		14.789	

M=Motor Driver, M_{ENTRY}=ENTRY Motor Driver, M_{EXIT}=EXIT Motor Driver, S=Sensor Driver, S_{ENTRY}=ENTRY Sensor Driver, S_{EXIT}=EXIT Sensor Driver

non-feasible solution (both cost and performance specifications are not satisfiable) is found, then there does not exist a feasible solution for the given system and thus our algorithm stops all further searching. The validity of the above three qualities requires the two basic assumptions as mentioned in Sect. 4.3. With these two assumptions and some useful features of binary search method adapted in MLP, the following three theorems can then be formalized, where divider(Z) is the object that acts as a divider of the MLA array into hardware and software parts corresponding to some partition Z. Further, cost(Z) and performance(Z) are the cost estimate and performance estimate of partition Z, respectively. These concepts are formalized into the following three theorems.

Theorem 1: If a feasible partition exists, then the BSC copartitioning algorithm will find one.

Theorem 2: Once a feasible partition is found, the final heuristically optimal partition found by the copartitioning algorithm is always feasible.

Theorem 3: If a completely infeasible partition (both cost and performance constraints are not satisfied) is ever found during BSC, then there exists no feasible partition for the system. Hence, the partitioning algorithm can stop searching.

When all objects in a system satisfy the two assumptions on hardware-software cost and performance, MLP will find a heuristically optimal solution. But, if there are one or more objects whose hardware-software cost and performance do not satisfy the two assumptions, then MLP will not be able to find a feasible solution as MLP is not an exhaustive approach.

All performance readings for each of the examples were taken for the worst case. Although the dynamic behavior may differ between two executions of a system, worst-case readings usually suffice for accounting dynamic behavior. This is because worst-case readings correspond to a critical path within the behavior space or reachability tree. Thus, although our MLP algorithm does not explicitly consider the dynamic behavior of a system, it is handled by performance estimation.

5. Experiment Results

Through the following examples, we present the advantages of the use of the sharing and clustering techniques in MLP based on several variants of the VPMS case study (see Sect. 3.4). Finally, we give the partition results for VPMS.

5.1 Advantage of Sharing in MLP

In this subsection, we illustrate how the use of sharing techniques in MLP produces partitions with a lower cost, without or with a slight performance degradation. For the VPMS example described in Sect. 3.4, Let *Sharing Threshold Distance* (STD) be 1 meter (m). MLP was applied to three versions of the VPMS specification: VPMS-1, VPMS-2, and VPMS-3, as shown in Table 1.

From experiment results, VPMS-1 (without sharing) needs 3 PEs and 2 ASICs for basic system feasibility which results in the cost being higher than all the other versions. VPMS-3 shows maximum sharing due to all subsystem functions being implemented using only one PE and one ASIC. In VPMS-2, ENTRY and EXIT gate controls share the same PE and DISPLAY subsystem uses one PE. Here, both the performance and cost constraints are satisfied (see Sect. 3.4 for constraints). Comparing the three results, we observe that sharing to a certain extent (as in VPMS-2) results in a significant cost reduction (12.6%) without affecting overall system performance. Whereas, when sharing is carried out to a larger extreme, the further slight reduction in cost has an adverse effect on performance such that constraints might not be satisfied.

Table 2 Partitioning results for five VPMS specifications with and without clustering.

Specification											
		VPMS-A		VPMS-B		VPMS-C		VPMS-D		VPMS-E	
Num. of Subsystems		1		2		2		2		3	
Subsystems		(1)ENTRY/ EXIT/ DISPLAY		(1)ENTRY/ EXIT (2)DISPLAY		(1)ENTRY/ DISPLAY (2)EXIT		(1)ENTRY (2)EXIT/ DISPLAY		(1)ENTRY (2)EXIT (3)DISPLAY	
Partitioning Results											
Num./locations of PE		1	(1)M/C	2	(1)M (2)C	2	(1)M_{ENTRY}/C (2)M_{EXIT}	2	(1)M_{ENTRY} (2)M_{EXIT}/C	3	(1)M_{ENTRY} (2)M_{EXIT} (3)C
Num./locations of ASIC		1	(1)S	1	(1)S	2	(1)S_{ENTRY} (2)S_{EXIT}	2	(1)S_{ENTRY} (2)S_{EXIT}	2	(1)S_{ENTRY} (2)S_{EXIT}
System Cost ($)		1,180		1,250		1,340		1,340		1,430	
Performance	DRT (μs)	14,020		13,200		13,100		13,100		13,200	
	GRT (μs)	1,030		210		110		110		110	

M=Motor Driver, M_{ENTRY}=ENTRY Motor Driver, M_{EXIT}=EXIT Motor Driver, C=Counter, S=Sensor Driver, S_{ENTRY}=ENTRY Sensor Driver, S_{EXIT}=EXIT Sensor Driver

Table 3 Eight critical partitions for VPMS.

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
H_C, H_S, H_M	S_C, H_S, H_M	H_C, S_S, H_M	H_C, H_S, S_M	S_C, S_S, H_M	S_C, H_S, S_M	H_C, S_S, S_M	S_C, S_S, S_M

H:hardware, S:software, Suffixes: C=Counter, S=Sensor Driver, M=Motor Driver

Table 4 Partitioning results of VPMS.

Partitions	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
System Cost ($)	1450	1420	1425	1280	1395	1250	1255	1225
Display response time (DRT) (μs)	190	13,100	290	190	13,200	13,100	290	13,200
Gate response time (GRT) (μs)	0.2	0.2	820	210	820	210	1030	1030

5.2 Advantage of Clustering in MLP

The clustering process is used to select a set of components from the same subsystem such that they can be scheduled into a single PE or implemented into an ASIC. In order to demonstrate the benefits of clustering during partitioning, MLP was applied to five variants of the original VPMS specification (see Sect. 3.4): VPMS-A, VPMS-B, VPMS-C, VPMS-D, VPMS-E, as shown in Table 2.

On one hand, only one PE and one ASIC are required for a feasible partitioning of VPMS-A. While, on the other hand three PEs and two ASICs are required for feasibly partitioning VPMS-E. The costs and performance also vary among the different variants of VPMS. Although VPMS-A has the lowest implementation cost, yet its performance shows a degradation compared to the others. This degraded performance might not be acceptable if the original performance constraints (as given in Sect. 3.4) are considered. Further, we observe that the performances of the other four versions (VPMS-B, VPMS-C, VPMS-D, and VPMS-E) are almost the same, while there is a difference in cost. In summary, we can conclude that clustering also allows a decrease in cost without affecting performance, provided that some subsystems could be grouped into new ones to allow clustering. Here, VPMS-B is the best partitioning result as it has the lowest cost while satisfying all performance constraints. The total time taken by running MLP for this example is 30.54 seconds.

5.3 VPMS Partition Results

Coming back to the original VPMS specification, experimental results of applying MLP show that we need consider only 19 hardware-software partitions of the system, instead of a much larger number that depends on the cost constraints. For illustration purpose, 8 critical partitions of VPMS are shown in Table 3.

The VPMS partitioning results are as shown in Table 4. Out of the eight partitions only two of them (D and F) satisfy the cost ($1,300), sensor to display response time (14,000 μs), and sensor to gate response time (250 μs) constraints. Hence, there are two satisfactory partition results for VPMS, namely the D and F partitions in Table 4.

6. Conclusions

A partitioning technique, called *Multi-Level Partitioning* (MLP) and targeted at *Distributed Embedded Multiprocessor Systems* (DEMS), was proposed. MLP covered several characteristics of DEMS that existing techniques have not touched: inter-distance between physical locations (subsystems), sharing of hardware, sharing of software, clustering of hardware, grouping of

software, and maintaining the hierarchical structure of DEMS. MLP is based on objects, which allows a high-level hierarchical view of DEMS as well as modularized partitioning appropriate for distributed systems. The feasibility of applying MLP as a *Distributed Embedded System Codesign* (DESC) methodology showed that it is a working technique. Several examples partitioned using MLP showed improvements in lower cost and higher performance over the results produced by existing techniques.

For the future, MLP will try to incorporate more sophisticated heuristics to improve upon its performance and results. Other features of DEMS not yet considered would also be taken into consideration such as priority-based scheduling, degree of communication among subsystems, and nested hierarchies.

Acknowledgment

This work was supported by the National Science Council, R.O.C. under grant NSC89-2213-E002-097.

References

[1] R.S. Raman, R.S. Nutter, and Y.V. Reddy, "A production system for intelligent monitoring systems," IEEE Trans. Ind. Appl., vol.24, no.5, pp.862–865, Sept./Oct. 1988.

[2] R.K. Gupta, Co-synthesis of Hardware and Software for Digital Embedded Systems, Ph.D. Thesis, Stanford University, Dec. 1993.

[3] R.K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," IEEE Design and Test of Computers, vol.10, no.3, pp.29–41, Sept. 1993.

[4] G.D. Micheli, D.C. Ku, F. Mailhot, and T. Truong, "The Olympus synthesis system for digital design," IEEE Design and Test of Computers, vol.7, no.5, pp.37–53, Oct. 1990.

[5] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," IEEE Design and Test of Computers, vol.10, no.4, pp.64–75, Dec. 1993.

[6] D. Herrmann, J. Henkel, and R. Ernst, "An approach to the adaptation of estimated cost parameters in the COSYMA system," Proc. International Workshop on Hardware-Software Co-Design, pp.100–107, 1994.

[7] F. Vahid, J. Gong, and D.D. Gajski, "A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning," Proc. European Design Automation Conference, pp.214–219, 1994.

[8] E. Barros, W. Rosenstiel, and X. Xiong, "A method for partitioning UNITY language in hardware and software," Proc. European Design Automation Conference, pp.220–225, 1994.

[9] A. Jantsch, P. Ellervee, J. Oberg, A. Hermani, and H. Tenhumen, "Hardware/software partition and minimizing memory interface traffic," Proc. European Design Automation Conference, pp.226–231, 1994.

[10] A.P. Kalavade and E.A. Lee, "A global criticality/local phase driven algorithm for the constrained hardware/software partition problem," Proc. International Workshop on Hardware-Software Co-Design, pp.42–48, 1994.

[11] A.P. Kalavade, System-Level Codesign of Mixed Hardware-Software Systems, Ph.D. Thesis, University of California, Berkeley, Sept. 1995.

[12] T.Y. Yen and W. Wolf, Hardware-Software Co-synthesis of Distributed Embedded Systems, Kluwer Academic Publishers, 1996.

[13] R.P. Dick and N.K. Jha, "MOGAC: A multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems," IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., vol.17, no.10, pp.920–935, Oct. 1998.

[14] B.P. Dave, G. Lakshminarayana, and N.K. Jha, "COSYN: Hardware-software co-synthesis of heterogeneous distributed embedded systems," IEEE Trans. Very Large Scale Integration (VLSI) Systems, vol.7, no.1, pp.92–104, March 1999.

[15] A. Kalavade and P.A. Subrahmanyam, "Hardware/software for multifunction systems," IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., vol.17, no.9, pp.819–837, Sept. 1998.

[16] B. Dave and N. Jha, "COHRA: Hardware-software cosynthesis of hierarchical distributed embedded system architectures," Proc. 1998 International Conference on VLSI Design, pp.347–354, 1998.

[17] J. Kenkel and R. Ernst, "A hardware/software partitioner using a dynamically determined granularity," Proc. 1997 International Conference on Design Automation, pp.691–696, 1997.

[18] G. Quan, X. Hu, and G. Greenwood, "Preference-driven hierarchical hardware/software partitioning," Proc. 1999 International Conference on Computer Design, pp.652–658, 1999.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[20] T.-Y. Lee, P.-A. Hsiung, and S.-J. Chen, "A case study in hardware-software codesign of distributed systems—Vehicle parking management system," Proc. 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), vol.6, pp.2982–2987, June 1999.

[21] P.-A. Hsiung, "CMAPS: A cosynthesis methodology for application-oriented parallel systems," ACM Trans. Design Automation of Electronic Systems, vol.5, no.1, pp.51–81, Jan. 2000.

[22] P.-A. Hsiung, C.-H. Chen, T.-Y. Lee, and S.-J. Chen, "ICOS: An intelligent concurrent object-oriented synthesis methodology for multiprocessor systems," ACM Trans. Design Automation of Electronic Systems, vol.3, no.2, pp.109–135, April 1998.

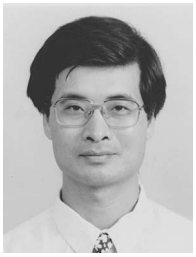
[23] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," The Bell System Technical Journal, vol.49, pp.291–307, 1970.

[24] S. Kirkpatrick, "Optimization by simulated annealing: Quantitative studies," J. Statistical Physics, vol.34, pp.975–986, 1984.

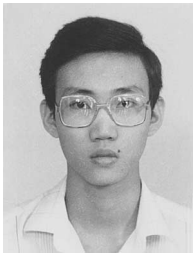
[25] J.-F. Lin, W.-B. See, and S.-J. Chen, "Performance bounds on scheduling parallel tasks with communication cost," IEICE Trans. Inf. & Syst., vol.E78-D, no.3, pp.263–268, March 1995.

[26] J.-F. Lin and S.-J. Chen, "An analysis of multiprocessor tasks scheduling," Computer Systems Science and Engineering, vol.11, no.2, pp.117–120, March 1996.

[27] J.-F. Lin and S.-J. Chen, "Scheduling algorithm for non-preemptive multiprocessor tasks," Computers and Mathematics with Applications, vol.28, no.4, pp.85–92, 1994.



Trong-Yen Lee received the B.S. and M.S. degrees in Industrial Education (major in Electronic Engineering), National Taiwan Normal University, Taiwan, Republic of China, in 1981 and 1988, respectively. Currently he is a Ph.D. candidate in the Department of Electrical Engineering, National Taiwan University. He has been teaching electronic engineering in Nankang Vocational High School, Taipei, Taiwan since 1980. His current research interests include hardware-software codesign, parallel architectures, simulation, and design automation systems.



Pao-Ann Hsiung received the B.S. degree in mathematics and the Ph.D. degree in electrical engineering from the National Taiwan University (NTU), Taipei, Taiwan, ROC, in 1991 and 1996, respectively. From 1993 to 1996, he was a Teaching Assistant and System Administrator in the Department of Mathematics, NTU. Currently, he is a post-doctoral researcher at the Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC. He will be taking up a faculty position at the Department of Computer Science and Information Engineering, National Chung Cheng University, Taiwan starting February 2001. His main research interests include: hardware-software codesign, real-time systems, formal verification, system-level design automation of multiprocessor systems, and object-oriented technology transfer.

Sao-Jie Chen received the B.S. and M.S. degrees in electrical engineering from the National Taiwan University, Taipei, Taiwan, ROC, in 1977 and 1982 respectively, and the Ph.D. degree in electrical engineering from the Southern Methodist University, Dallas, USA, in 1988. Since 1982, he has been a member of the faculty in the Department of Electrical Engineering, National Taiwan University, where he is currently a full professor. From 1985 to 1988, he was on leave from National Taiwan University and working toward his Ph.D. at Southern Methodist University. During the fall of 1999, he was a visiting scholar in the Department of Computer Science and Engineering, University of California, San Diego. His current research interests include: VLSI circuits design, VLSI physical design automation, object-oriented software engineering, and multiprocessor architecture design and simulation. Dr. Chen is a member of the Chinese Institute of Engineers, the Association for Computing Machinery, the IEEE, and the IEEE Computer Society.