Layout Conscious Approach and Bus Architecture Synthesis for Hardware/Software Codesign of Systems on Chip Optimized for Speed

Nattawut Thepayasuwan, *Member, IEEE,* and Alex Doboli, *Member, IEEE*

***Abstract*—This paper presents a layout-conscious approach for hardware/software codesign of systems-on-chip (SoCs) optimized for latency, including an original algorithm for bus architecture synthesis. Compared to similar work, the method addresses layout related issues that affect system optimization, such as the dependency of task communication speed on interconnect parasitic. The codesign flow executes three consecutive steps: 1) combined partitioning and scheduling: besides partitioning and scheduling, this step also identifies the minimum speed constraints for each data link; 2) IP core placement, bus architecture synthesis, and routing: IP cores are placed using a hierarchical cluster growth algorithm; bus architecture synthesis identifies a set of possible building blocks and then assembles them for minimizing bus length and complexity; poor solutions are pruned using a special table structure and select-eliminated method; and 3) rescheduling for the best bus architecture. This paper offers extensive experiments for the proposed codesign method, including bus architecture synthesis for a network processor and a JPEG SoC.**

***Index Terms*—Bus architecture synthesis, hardware/software codesign, systems-on-chip (SoCs).**

I. Introduction

MANY embedded systems must meet stringent cost, timing, and energy consumption constraints [7]. In addition, embedded architectures are very thrifty in employing hardware resources: they include general purpose processors running at low/medium frequencies (like ARM, 801C188EB, Philips 80C552, etc.), have a reduced amount of memory (the memory capacity can be as low as 128 k of RAM and 256 k of flash memory), and incorporate customized coprocessors and I/O peripherals (including radio-frequency and analog circuits). Typical examples include embedded systems for telecommunication and multimedia, like cell phones, digital cameras, and personal communicators. Systems-on-chip (SoCs) are single-chip implementations of embedded systems. Compared to printed circuit board designs, SoCs offer higher performance and reliability at cheaper costs [7]. It is foreseen that advances in device manufacturing technology, including present deep submicron technologies and future nanotechnologies, will continuously reduce the minimum feature size, and thus increase the functional complexity of SoCs.

Manuscript received August 11, 2003; revised February 25, 2004 and June 25, 2004. This work was supported in part by IBM under a Faculty Partnership Award and in part by a DAC Graduate Scholarship Award.

The authors are with the Department of Electrical and Computer Engineering, State University of New York at Stony Brook, Stony Brook, NY 11794-2350 USA (e-mail: nattawut@ece.sunysb.edu; adoboli@ece.sunysb.edu).

Digital Object Identifier 10.1109/TVLSI.2004.842910

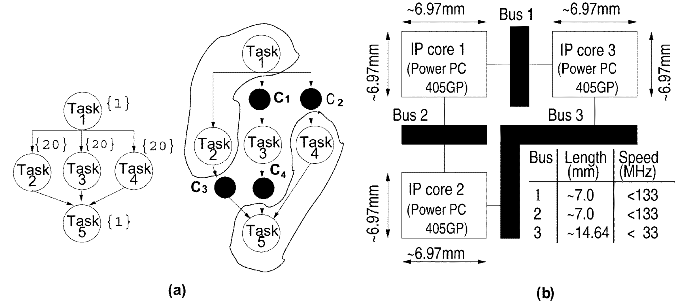


Fig. 1. Impact of layout on data communication speed and system design.

For SoCs realized in deep submicrometer technologies (DSM), physical-level attributes, such as interconnect parasitics, substrate coupling, and substrate noise, significantly influence system performance, e.g., data communication speed, system latency, power consumption, and signal integrity [4], [29]. Fig. 1 illustrates the impact of layout parasitics on data communication speed and system design. Fig. 1(a) presents a task graph with five tasks. Each task is labeled by its execution time on Power PC processor core. Without considering layout information, the codesign step decides to allocate a single 266-MHz system bus for all core communications. This would meet the timing constraints, while keeping the system architecture simple. However, considering the physical distances between cores, shown in Fig. 1(b), it is difficult to implement a bus with the requested speed. The same latency can be obtained with three buses of lower speed, like those in Fig. 1(b), because the system concurrency improves. The bus speeds of 133, 133, and 33 MHz were found based on the physical locations of cores and the RLC parasitic of the routed buses [29]. This example argues that the communication subsystem of an SoC needs to be designed while contemplating layout-related criteria. In general, it is difficult to postulate a unique bus architecture as being optimal for various applications and performance requirements. Instead, bus architectures need to be customized depending on the application specifics and design needs. New synthesis algorithms are needed, such as for bus architecture design, as well as novel modeling methods, like predicting interconnect length at the system level.

System design, including task and communication partitioning and scheduling, must be integrated with relevant knowledge about core placement, bus topology design, and bus routing to guarantee that allocated data communication speeds are realistic. Fig. 1(a) depicts a partitioning solution in which tasks 1 and 2 are mapped to the same core, tasks 4 and 5 are on another core, and task 3 is bound to a third core. To minimize

1063-8210/$20.00 © 2005 IEEE

system latency, the speed of data communications C_1 and C_4 has to be higher than that of communications C_2 and C_3 (assuming that the same amount of data is sent between cores). This speed requirement enforces that the core running task 3 will have to be placed close to the other two cores, whereas the core executing tasks 1 and 2 might be placed further away from the core for tasks 4 and 5. This set of communication speed constraints is feasible, and Fig. 1(b) presents a possible floorplan. However, it is infeasible to impose the additional requirement that the speed of C_2 is much higher than the speed of C_3 because the corresponding floorplan is hard to build. In conclusion, speed constraints for the communication subsystem need to be tackled while contemplating layout-related aspects, e.g., possible core floorplans and achievable bus speeds. This task is obviously challenging and requires new design approaches, in which the top-down codesign process is aware of certain low-level aspects, like core placement and bus routing.

This paper describes a hardware/software codesign method for developing SoC implementations subject to latency minimization. The novelty is in proposing a systematic, layout-conscious approach for tackling the SoC communication subsystem, including an original bus architecture synthesis algorithm. System-level design attempts to minimize latency and maximize the feasibility of constraints imposed to the bus architecture. Applications are task graphs [14] with data dependencies and reduced number of control dependencies. The set of available hardware resources and the SoC area are known. The codesign method includes three subsequent parts:

1) combined partitioning and static nonpreemptive scheduling;
2) bus architecture synthesis;
3) rescheduling for the best bus architecture.

The first step is an exploration process based on simulated annealing algorithm (SA) [25]. The cost function expresses the minimization of system latency and maximization of the feasibility of bus architecture constraints, like required speed, number of links, and amount of resulting connectivity between cores. We propose performance models (PMs), a graph-based description, that symbolically captures the relationships among performance, graph characteristics, and design decisions. PM are general, flexible, and can be easily extended to new design activities without requiring cumbersome validation. The second step synthesizes and routes the bus architecture for an SoC. IP cores are placed using a hierarchical cluster growth algorithm. Using the proposed primary bus structure (PBS) bitwise generation algorithm, bus architecture synthesis first identifies a set of possible building blocks, then assembles them together, such that bus length, bus topology, communication conflicts, and unnecessary core connectivity are minimized. We propose a special table structure (named bus architecture synthesis table) and select-eliminate method to prune poor solutions, such as buses with complex and redundant connectivity. The algorithm was successfully used to automatically synthesize bus architectures for realistic SoC, including a network processor and a JPEG SoC.

This paper is organized as follows. Section II presents related work. Section III discusses system modeling. Section IV introduces the proposed codesign approach. Bus architecture synthesis is presented next. Experimental results are given in Section VI. Finally, conclusions are offered.

II. Related Work

Over the last ten years or so, a variety of hardware/software codesign methodologies were proposed for designing embedded systems optimized for cost, speed, and power consumption [3], [13], [15]. A typical codesign flow includes the following activities: selection of architectures and architectural resources (processors, memories, buses, I/O modules), functionality partitioning, task mapping to resources and scheduling, and communication synthesis. Depending on the targeted applications, codesign approaches can be classified into three groups: for data-dominated systems [3], [6], [18], for control intensive systems [2], and for applications with substantial data processing and reduced amount of control [14], [27]. Balarin *et al.* [2] present POLIS, an approach for control dominated real-time embedded applications. For data-dominated systems, Prakash and Parker [24] formulate the codesign problem as a mixed-integer linear programming (MILP) problem. A linear equation solver finds the optimal implementation. The disadvantage of MILP-based codesign is its limitation to small size applications. The alternative is to employ heuristic algorithms, such as greedy priority-driven clustering [6], list scheduling methods [3], [10], [14], iterative improvement heuristics, like simulated annealing and tabu search [13], and genetic algorithms [5], [9], [27]. Heuristic methods can be used for large task graphs [14]. The disadvantage is that the solution optimality is difficult to characterize. For example, greedy priority-driven algorithms offer good average results, but they might give poor solutions for situations not captured by the priority function [10]. Henkel [19] suggests a hardware/software partitioning method for low-power systems. After scheduling, instruction clusters with a high utilization rate (thus, with less wasted energy) are moved to hardware. Dave *et al.* [6], and Dick and Jha [9] propose cosynthesis methods for the design of heterogeneous systems under a large variety of optimization goals including cost, latency, and average, quiescent, and peak power consumption. The methods perform task allocation, scheduling, and performance estimation while contemplating interprocessor concurrency, preemptive and nonpreemptive scheduling, and memory constraints. Givargis and Vahid [17] describe Platune environment for tuning parameterized uniprocessor SoC architectures to optimize timing and power consumption. Parameters, like processor speed, cache organization, and certain periphery attributes, are decided using the Pareto optimality criterion.

Bus design is critical for SoC. Early work on bus and communication synthesis [8], [16], [23], [34] focuses on multiprocessor embedded systems on a printed circuit board. Research addresses interface design [8], [23], communication packeting [14], and mapping and scheduling [34]. This work does not tackle the hardware and layout details of the SoC communication subsystems. Sgroi *et al.* [28] suggest communication-centric system design motivated by the increasing importance of communication attributes. Communication is layered similar to the OSI reference model. Adapters increase the reusability of components by matching different protocols. Lahiri *et al.* [21] focus on communication protocol selection for a communication architecture template including shared and dedicated buses. Recently, Drinic *et al.* [12] present a method for SoC bus network design to maximize overall processing throughput. The communication architecture includes shared buses connected

through bridges. The design flow includes two steps: one produces the communication topology and the other finds the core floorplanning. Hu *et al.* [20] introduce point-to-point communication synthesis to optimize energy consumption and area. Their work concentrates on bus width synthesis to meet timing constraints on the communication links, and floorplanning to minimize energy consumption and SoC area. Existing approaches use limited layout knowledge to guide system design. In many approaches, bus topology is assumed given [16], [20], [21]. This is reasonable for small SoC for which the designer manually designs the buses. However, it is not effective for SoCs with a large number of cores.

Compared to similar work, this paper proposes a new hardware/software codesign approach that integrates system design with bus architecture synthesis and routing. The suggested bus architecture synthesis method does not require knowing the bus topology, is more sensitive to layout parasitic, and prunes early poor solutions. The codesign algorithm performs combined task partitioning and scheduling using the well-known SA for exploration, but employing a new method for expressing system performance and requirements. The combined method offers shorter system latency, is more flexible toward new design requirements, and scales reasonably well with the application size.

III. System Representation for Codesign

A. Embedded System Modeling

The quadruple $\langle$HDCG, Resources, Floorplan, PM$\rangle$ describes an embedded system: *HDCG* represents the system functionality, *Resources* is the set of IP cores of the implementation, *Floorplan* is the set of all possible floorplans for the IP cores in set *Resources*, and *PM* denotes performance attributes of the implementation, like latency.

1) Hierarchical Data and Control Dependency Graph (HDCG):

Definition: A hierarchical data and control dependency graph is the triplet HDCG$= \langle \mathcal{S}_{\mathrm{CN}}, \mathcal{S}_{\mathrm{CCN}}, \mathcal{S}_A \rangle$, where $\mathcal{S}_{\mathrm{CN}}$ is the set of cluster nodes, $\mathcal{S}_{\mathrm{CCN}}$ is the set of communication cluster nodes, and $\mathcal{S}_A$ is the set of arcs. HDCGs are acyclic polar graphs having one start node and one end node.

Fig. 2 shows an HDCG example.

Cluster nodes (CN) represent tasks, functions, loops, and if-then-else constructs in the system specification. At the fine grain level, each cluster node CN_i is described as the acyclic polar graph

$$\mathrm{CN}_i = \left\langle \mathcal{S}^i_{\mathrm{ON}}, \mathcal{S}^i_{\mathrm{Arcs}} \right\rangle$$

where $\mathcal{S}^i_{\mathrm{ON}}$ is the set of operation nodes forming cluster node i and $\mathcal{S}^i_{\mathrm{Arcs}}$ is the set of arcs connecting the operation nodes. Fig. 2(b) shows the fine grain structure of CN_3. *Operation nodes* (ONs) denote an atomic data processing, such as addition, multiplication, division, comparison, etc. Operation nodes are mapped to small/medium size IP cores, like multipliers and arithmetic and logic units. Each arc $a \in \mathcal{S}^i_{\mathrm{Arcs}}$ is a pair $(\mathrm{ON}_k, \mathrm{ON}_l)$, $\mathrm{ON}_k, \mathrm{ON}_l \in \mathcal{S}^i_{\mathrm{ON}}$. Arcs express data dependencies between ONs: node ON_l can start only after node ON_k was performed. During cosynthesis, ONs are used for exploring hardware resource sharing across tasks.

Each CN and ON has a triplet $\langle T^s, T^{\mathrm{ex}}, T^{\mathrm{end}} \rangle$ representing symbolic variables for the node's start time, execution time, and end time. These variables are used to describe the performance models of the embedded system.

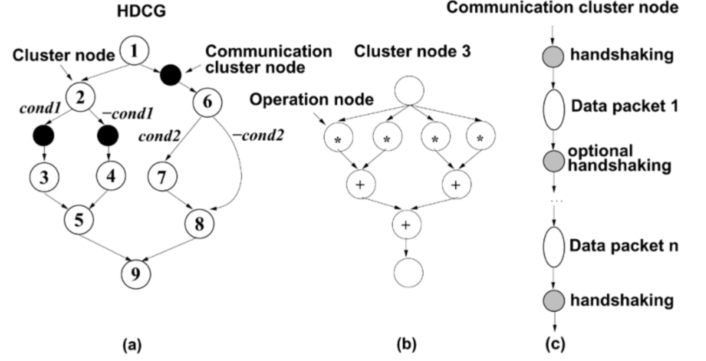


Fig. 2. Hierarchical data and control dependency graph.

Communication cluster nodes (CCNs) represent data communications between CNs mapped to different processing units. CCNs are shown as black bubbles in Fig. 2(a). At the fine grain level, each CCN_j has a linear structure, as shown in Fig. 2(c). CCN_j is an alternating sequence of nodes corresponding to transmissions of data packets of a fixed size and nodes for synchronization. The number of data packet nodes depends on the data quantity specific to a CCN, as well as the fixed size of the data packet. Synchronization nodes express the time overhead for synchronizing two cores through handshaking. The optional synchronization nodes allow packets from different communication links to be interleaved on the same bus. This facilitates the suspension of an ongoing communication in favor of a higher priority data transmission. If successive packets pertain to the same communication link, then the optional synchronization nodes have zero time length.

Arcs describe the data and control dependencies of an HDCG. An arc $a \in \mathcal{S}_A$ is the triplet $\langle n_i, n_j, \mathrm{cond}_k \rangle$, where $n_i, n_j \in \mathcal{S}_{\mathrm{CN}} \bigcup \mathcal{S}_{\mathrm{CCN}}$ and cond_k is a boolean variable or $\emptyset$. For data dependencies, $\mathrm{cond}_k = \emptyset$. Data dependencies impose that the target node n_j starts its execution only after the source node n_i was completed. Similar to conditional process graphs [20], control dependencies are arcs annotated with a boolean variable. For control dependencies, node n_j is executed only if the boolean variable is true. In Fig. 2(a), boolean variables are depicted in italics. Node CN_2 computes variable *cond1*. If variable *cond1* is *true,* then the communication cluster node following CN_2 is performed. CN_4 is executed for a *false* value, indicated as—*cond1* in the figure.

Definition: System latency is the end time of the HDCG end node. For HDCG with conditional dependencies, system latency is the worst case latency for all possible values of the boolean variables. Node execution is nonpreemptive.

Due to the acyclic nature of an HDCG, each CN, ON, and CCN is executed at most once for a traversal of the graph. For an HDCG with p control variables, finding system latency requires the analysis of 2^p cases. This is still feasible for HDCG with reduced number of control dependencies.

HDCGs offer a dual perspective on the system functionality: a task-level description (for partitioning and scheduling) and an operation-level representation (for exploring hardware sharing across tasks). HDCGs are similar to control data flow graphs [15] and conditional process graphs [14]. Even though system functionality could be expressed using operation nodes only,

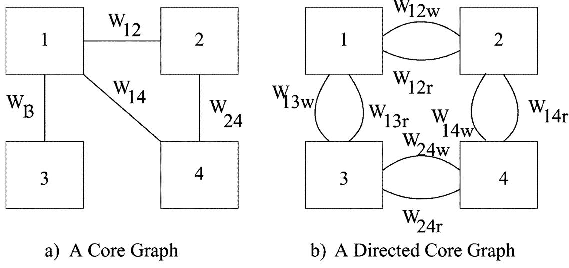


Fig. 3. Core graph and PBS examples.

cluster nodes prevent the unnecessary growth of the design space, and hence, a very lengthy codesign process. If cluster nodes are executed on a general purpose processor as software, then there is no need to explore hardware sharing at the operation level. Besides, for each CN, the execution in software can be accurately estimated using data profiling and performance models for CPU, cache, memory, and communication units [17]. The effect of various compiler optimizations can be tackled more effectively for CNs than for a system expressed using ONs.

2) Resources:

Definition: Resources is the set of IP cores available for the SoC implementation. R_i is the subset of set *Resources* to which node i can be mapped, where $i \in \mathcal{S}_{\rm CN} \bigcup \mathcal{S}_{\rm ON}$. Function $Mappedto : \mathcal{S}_{\rm CN} \bigcup \mathcal{S}_{\rm ON} \rightarrow Resources$ defines the actual hardware resource on which a CN or ON is executed.

As presented in Section IV, the proposed codesign method assumes that the number and type of available hardware resources is known. This set includes GPP cores, FU cores, multiplier cores, and so on. Hence, sets *Resources* and R_i are given. Through exploration, codesign identifies the function *Mappedto* that optimizes the design constraints.

The considered bus model assumes a single transaction phase protocol and no data buffering. The single transaction phase incorporates all activities related to the address and data phases.

Definition: A core graph (CG) is the graph (V,E), where $v_i \in V$ represents core i in the architecture, and $e_{ij} \in E$ is the communication link between cores i and j. The weight w_{ij} is the *communication load* between core i and core j. It expresses the amount of data exchanged between the two cores. The core size $h_i \times w_i$ is described along with node v_i.

This concept has been illustrated in Fig. 3(a). The core graph representation of a system architecture is used for bus architecture synthesis. For simplicity of modeling, CGs do not distinguish between unidirectional and bidirectional dataflow. Communication direction depends on whether an operation is a read or a write, and is not specified directly in a CG. However, the core graph can be modified in order to address the direction of data. Fig. 3(b) presents the core graph for bidirectional communications. In case there is more than one communication channel between two cores, then the communication load is split across the channels.

3) Floorplan:

Definition: Floorplan trees (FTs) are binary tree structures having following two properties: 1) leaf nodes correspond to IP cores and 2) each internal node links the two nodes that exchange the maximum amount of data with each other. By definition, an internal node IN_i exchanges with leaf node LF_k, $LF_k \notin IN_i$, a data quantity equal to the sum of all data communications between node LF_k and all IP cores in the subtree originating at node IN_i ($w_{LF_k\ IN_i} = \sum_{p \in Subtree(IN_i)} w_{LF_k\ LF_p}$). The amount of data communicated between two internal nodes IN_i and IN_j is equal to the sum of all communications between node IN_i and all leaf nodes of the subtree originating at node IN_j.

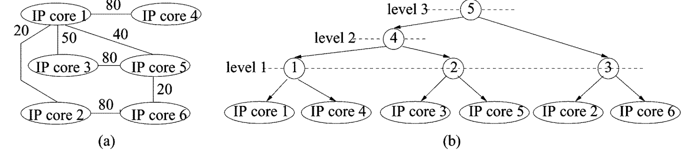


Fig. 4. Floorplan tree.

Fig. 4(a) presents a set of six IP cores, and Fig. 4(b) shows the corresponding FT representation. Arc labels express the amount of data exchanged between cores. Cores 1 and 4, cores 3 and 5, and cores 2 and 6 are heavily communicating. Hence, internal nodes 1, 2, and 3 represent their clustering. The quantity of data communicated between nodes 1 and 2 is 90 (50 for the communication between cores 1 and 3 and 40 for the communication between cores 1 and 5). The bottom-up process continues by considering nodes 1, 2, and 3, until the root node is reached (node 5 in the figure).

An FT models core floorplanning at the system level. It helps to qualitatively approximate the bus delays in an SoC implementation. Section III-B explains that the speed of the link between two cores decreases as the level of their first common internal node increases. For example, it is likely that the link for cores 1 and 4 will be faster than that for cores 1 and 2. The qualitative approximation is needed, because it is too cumbersome to integrate floorplanning and bus routing with the already complex codesign process. Instead, FTs abstract away the horizontal and vertical cutlines in the slicing trees [26] for floorplanning, and replace precise bus speed evaluation with finding a lower bound of the bus speed. This avoids codesign solutions in which links for loosely connected cores are required to operate at high speeds, because after floorplanning those cores will communicate through long buses. Obviously, the actual bus speed after detailed floorplanning and routing might be higher than the lower bound predicted by FTs. However, this gap is not a problem, because FTs were introduced to aid finding constraint satisfying designs.

4) Performance Model (PM): Performance models describe symbolically the semantics of performance attributes, such as latency, with respect to the invariant HDCG characteristics, like CN, CCN, ON, and dependencies, as well as the design decisions contemplated during codesign, such as partitioning and scheduling.

Definition: Performance model (PM) is a graph that contains following elements.

1) The *starting node 0* for setting the modeled performance attributes to their initial value.
2) The *constant part* consists of linked symbolic variables and operational nodes, like *addition* nodes, *multiplication* nodes, $\max$ nodes, and $\min$ nodes.
3) The *variable part* includes additional directed arcs between the operational nodes.

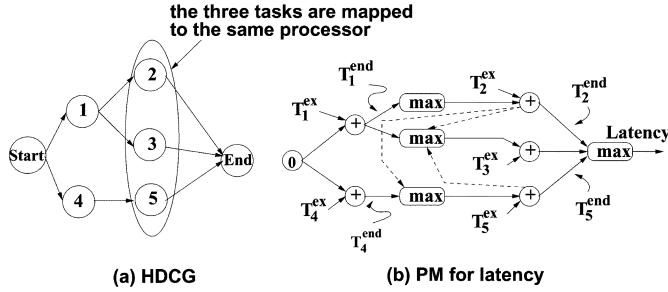


Fig. 5. Performance model for latency.

The numeric values of performance attributes result by evaluating the operational nodes for the operands described by symbolic variables and arcs.

Fig. 5 shows an HDCG, and its corresponding PM for latency. The figure assumes that ON_2, ON_5, and ON_3 are executed in this order on the shared processor. The constant part of the PM includes all nodes and solid edges in Fig. 5(b). $\max$ and *addition* nodes express constraints between start and end times of each cluster node. For example, the outputs of $\max$ nodes define the start time of the corresponding cluster nodes. The start time of a CN has to be larger than the maximum of the end times of all predecessors. Addition nodes express that the end time T_i^{end} of node i is the sum of its start time T_i^s and its execution time T_i^{ex}. The *variable part* presents the relationship between latency and the design decisions taken during codesign, like task partitioning and scheduling. In Fig. 5(b), the variable part includes dashed arcs between the addition nodes for the end times of ON and the $\max$ nodes for the start times. Other ON scheduling orders are easily captured in the PM by changing the orientation of certain arcs.

PM is a general description, which can express different performance attributes and denote various codesign activities. PMs are very flexible, as they allow easy definition of new performance attributes, or description of additional relationships between performance attributes and codesign activities. For example, the attribute of communication speed flexibility, defined in Section III-B, was added without affecting the already existing rules for latency. There is no validation effort for new attributes. Finally, rules can be identified to prune infeasible or dominated solution points. For example, the rules for communication speed flexibility (CSF) calculation avoid generating designs, which are difficult to realize. This helps faster closure by improving the feasibility of system design. Maestro *et al.* [22] suggest timing graphs for symbolically expressing the system execution time. PMs differ from timing graphs by not being limited to timing attributes or coarse-grained descriptions. Timing graphs are employed to avoid overlapped execution of tasks with similar operations. This is not the case for PMs, which are used for characterizing finer grained functionality too. The remaining part of this section presents the rules for building the PMs used in the proposed codesign methodology.

B. Modeling of Codesign Activities

1) Modeling of Data and Control Dependencies: Fig. 6 shows the general rule for expressing data dependencies in a PM. Node n is executed only after all its predecessor nodes 1, $2, \ldots, k$ are performed. A $\max$ node was introduced to express that the start time T_n^s of node n is greater or equal than the end times of all its predecessors. The addition node for node n symbolically relates the node's end times T_n^{end} to its start time T_n^s and its execution times T_n^{ex}. Similar constructs are introduced for all data dependencies. The rightmost addition node of the resulting PM denotes the system latency. In [11], we presented the modeling of control dependencies.

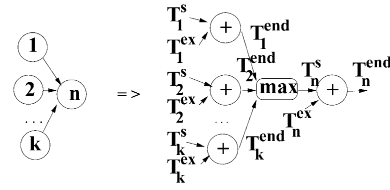


Fig. 6. Modeling of data dependencies.

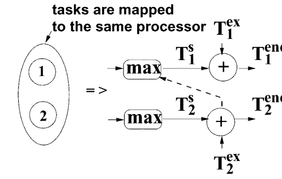


Fig. 7. Modeling of scheduling.

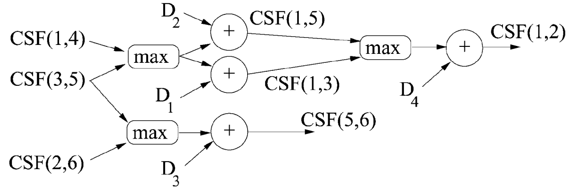
2) Modeling of Cluster Node Partitioning and Operation Binding: From the model point of view, cluster node partitioning and operation binding finds the definition of the function *Mappedto* that optimizes the design performance. Obviously, $Mappedto(i) \in \mathcal{R}_i$ for each node i. The numerical values of the resource-dependent attributes of a node become well defined after partitioning and binding. In our case, the execution time T_i^{ex} of node i changes for each new resource type, and its numerical value is updated in PMs.

3) Modeling of Scheduling: For a given HDCG and a node partitioning/binding to hardware resources, scheduling decides the node execution order on the shared resources. Static nonpreemptive scheduling was used in our approach. Depending on the scheduling decisions, different execution sequences and timing attributes (such as start time and end time) result for the nodes. In the presence of data dependencies only, a certain execution order is modeled by introducing in the PM model a dashed arc from the addition node for the end time of the node to be executed first to the max node for the start time of the node to be executed second. For example, in Fig. 7, Node 2 is executed before Node 1 on the same resource. Accordingly, the PM is updated by introducing a dashed arc that forces Node 1 to start only after Node 2 ends. This arc pertains to the variable part. Different scheduling decisions can be easily captured by changing the orientation of dashed arcs. Scheduling in the presence of control dependencies is described in [11].

4) Modeling of Communication Speed Flexibility: The execution time of CCNs cannot be accurately estimated at the system level. This is because the bus speed depends on the bus length, and thus, on the placement of IP cores, the bus architecture, and bus routing. This information is not available during task partitioning and scheduling.

Definition: For each data link, the *communication speed flexibility* indicates the amount of delay that can be tolerated on that link without violating the required system latency.

To address the unknown communication speed, the codesign methodology first identifies feasible CSF requirements for each




```
input: FT - Floorplan Tree
output: PM for CSFs
for all levels j in FT, starting from level 1 upwards do
  for all nodes p in FT placed on level j do
    identify all communications (m,n), such that
    node p is the first common parent in FT for cores m and n;
    create a max node and an addition node;
    link the output of the max node to the input of the additon node;
    create symbolic variable Dj as the second input to the addition node;
    label the output of the addition node as CSF(m,n);
    for all existing CSF(l,k), k<>n or l<>m do
      if FT level of link (l,k) < FT level of link(m,n) then
        insert an edge from CSF(l,k) to the input of max node for CSF(m,n);
```

Fig. 8. PM modeling of communication speed flexibility.

data link by using a system-level modeling of the bus architecture. CSF requirements are feasible if the bus speed can be achieved in the presence of RLC parasitic. Then, the found CSF values become constraints for the bus architecture synthesis step discussed in Section V.

Lemma: Let (i,j) and (m,n) be the CG edges for the data communications between cores i and j and cores m and n, respectively. In the corresponding FT, let s be the level of the first common parent of cores i and j, and t the level of the first common parent of cores m and n. If $s \leq t$, then the speed of the bus for communication $(i,j) \geq$ the speed of the bus for communication (m,n).

Proof: Considering the construction rules for the binary FT, it results that cores i and j are placed closer to each other than cores m and n. Thus, the bus speed will be higher for link (i,j) than for link (m,n).

In the final SoC layout, it is very likely that cores that are close to each other will use faster buses than cores placed far apart. This observation is summarized by the above lemma. To find feasible delay constraints, a naive solution would assign random values to CSFs, and then check if these values meet the constraints imposed by FT. In reality, this solution does not work, as most of the CSF values will violate the constraints. Instead, PMs for CSF were built to explicitly incorporate all FT constraints. Fig. 8 shows the corresponding algorithm. The algorithm traverses bottom-up the floorplan tree, and for each internal FT node it generates a pair of linked max and addition nodes. The output of the max node is input to the addition node. The output of the addition node represents the CSF constraint for the communication link between cores m and n, such that the internal node is the first parent of both cores in the FT. The CSF constraints for all cores connected through a lower level link are inputs to the newly created max node. This models all requirements expressed by the above lemma.

Fig. 8 shows an example for the PM expressing the constraints between CSF values. CSF values for nodes CSF(1,4), CSF(3,5), and CSF(2,6) (which are all on the first level of the FT) are inputs to the PM. According to the floorplanning, the speed for communications (1,5) and (1,3) will be slower than the slowest of the communications (1,4) and (3,5). The max nodes and the addition nodes in the PM formulate these constraints. Values D_1

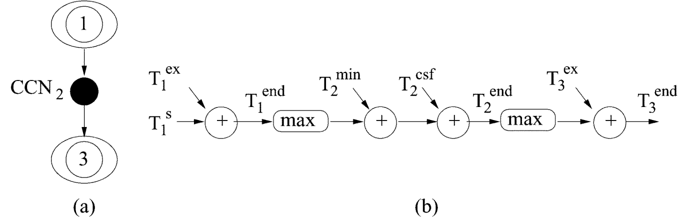


Fig. 9. Communication speed flexibility.

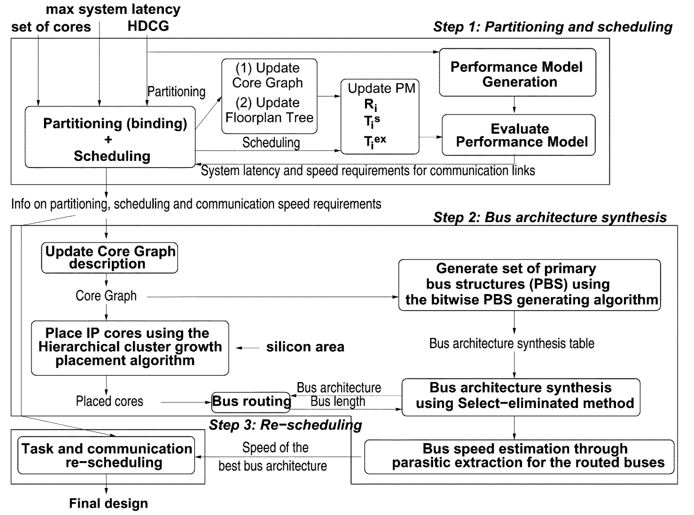


Fig. 10. Hardware–software codesign methodology.

and D_2 express the time amount by which the two communications are slower. Similarly, communication (5,6) will be slower than communications (2,6) and (3,5). Finally, communication (1,2) will be slower than communications (1,5) and (1,3).

For each CCN i, a max node and two addition nodes are introduced into the PM for latency. The max node describes the starting time of data communication. The first addition node has the max node output and variable $T_i^{\min}$ as inputs. The output of the first addition node is input to the second addition node, which has variable T_i^{CSF} as second input. The first addition node models the minimum communication time, which depends on the amount of communicated data, as well as the maximum speed of a given fabrication technology. This value is a lower bound for the CCN execution time. The second addition node expresses the extra bus delay due to floorplanning constraints. Its output is the end time of communication. Variable T_i^{CSF} depends on the CSF value of the communication link used for CCN i and the amount of data. Fig. 9 shows an example. CNs 1 and 3 are mapped to different cores, and CCN 2 is their data communication. Fig. 9(b) presents the PM for latency, including the two components of the communication time.

IV. Codesign Methodology

Fig. 10 presents the proposed hardware/software codesign methodology. Inputs are the HDCG of an application, the maximum system latency, the overall silicon area of the SoC, and the set of available cores, including the number and types of general purpose processors, functional units, etc. The goal is to partition the HDCG nodes to cores, to decide the scheduling of nodes, to synthesize the bus architecture, and to map and schedule data

communications on buses. The overall system latency must be minimized. As a byproduct of bus architecture synthesis, the core floorplanning is found, such that the total area constraint is met.

The codesign methodology includes three consecutive steps. The first step partitions cluster nodes to processor cores, binds operation nodes to functional unit cores, schedules cluster nodes, communication cluster nodes, and operation nodes, and finds the speed requirements for communication cluster nodes. The second step decides the IP core floorplanning, synthesizes the bus architecture, routes the buses, and characterizes the speed achievable on each bus. Finally, the third step reschedules cluster nodes, communication cluster nodes, and operation nodes while keeping the partitioning and the bus architecture unchanged.

Step 1) Partitioning and Scheduling

First, PMs are generated for an HDCG using the rules presented in Section III. Next, a simulated annealing (SA) [25] based exploration loop conducts simultaneous partitioning and scheduling. For each CN (ON) i, attributes $Mappedto(i)$ (the hardware resource that executes the node), T_i^{ex} (the execution time on that resource), and T_i^s (the start time) are the unknowns for codesign. Cluster node partitioning to processors and operation binding to FUs are modeled by the unknowns $Mappedto(i)$ and T_i^{ex}. The scheduling of cluster nodes, communication cluster nodes, and operation nodes is described by the unknowns T_i^s. Possible numerical values for the unknowns R_i and T_i^s are searched during exploration.

SA iteratively selects a new point from the neighborhood of the current solution. The neighborhood was defined as the set of points that differ from 1) the current solution by the execution order of *one* pair of nodes that share a hardware resource or 2) the resource binding of *one* node. PMs, FTs, and CGs are updated for each newly selected solution. For each codesign solution, the system latency and CSF are calculated by evaluating their PMs with all node attributes R_i and T_i^{ex} instantiated to their numerical values. Starting solutions were obtained by uniformly distributing nodes to resources, and then scheduling nodes using list-scheduling with critical path as the priority function [15]. Partitioning, binding, and scheduling steps were executed with different probabilities. The reason is that multiple valid schedules are possible for each partitioning and binding decision. A small probability p_1 was used to select a partitioning step that moves a cluster node from a processor core to another processor core or to hardware. The probability p_2 $(p_2 > p_1)$ binds an operation node to another FU core. The reason for p_2 being greater than p_1 is that multiple hardware designs are possible for each partitioning of clusters to FU cores. Finally, the probability $1-(p_1+p_2)$ decides a scheduling action. This emulates a hierarchical exploration process, in which for each new partition or binding there are $(1-(p_1+p_2)/p_1+p_2)$ analyzed schedules.

The cost function for SA is

$$\mathrm{Cost} = \alpha \times \mathrm{Latency} + \beta \times \prod_{\forall\ \mathrm{links}} \frac{1}{D_i} + \gamma \times \#\ \mathrm{buses} + \delta \times \mathrm{unnecessary\ connectivity}$$

The cost function to be minimized models the system latency and the feasibility of the bus architecture constraints. To maximize feasibility, CSF requirements for each link need to be maximized. Large CSF values relax the constraints for bus architecture synthesis, as slower buses would be acceptable. Section III-B-IV explains that CSF values are maximized if their corresponding D_i values are also maximized. To encourage equal distribution of the tolerable slack time to all links, the product of D_i values was used in the cost function instead of their sum. Using the sum could result in having some very relaxed D values, but very tight values for others. Such a bus architecture would be still difficult to implement. The last two terms in the cost function further express the quality of a bus architecture, as the number of buses and the amount of unnecessary core connections. The number of buses was estimated depending on the likelihood of different communication links to share the same bus. Links are likely to share a bus if they involve the same cores, have the same bus speed requirements, there is little overlapping between communications, and there is little unnecessary core connectivity. A more detailed modeling of these attributes is used for bus architecture synthesis discussed in Section V. α, β, γ, and δ are weights.

Step 2) Bus Architecture Synthesis

CG description is updated based on the information on task partitioning and scheduling. First, the floorplan for the SoC cores is found using the hierarchical cluster growth placement algorithm [26]. Core placement is needed to accurately estimate bus lengths and find the correct rates at which data can be communicated on buses. The introductory section explained that DSM effects are important for characterizing the speed possible on a link. Core placement is communication driven, so that two heavily communicating cores are placed close to each other, the aspect ratio of their rectangular bounding box is close to one, and the total area of the box is minimized. Also from CG, the set of possible primary bus structures (PBSs) is created using the bitwise PBS generating algorithm (presented in Section V-B). PBSs are the building blocks for creating bus architectures. Then, a bus architecture synthesis table is produced to characterize the satisfaction of connectivity requirements by individual PBS structures. The actual bus architecture synthesis algorithm (called select-eliminate method) uses SA. Using BA synthesis tables, the method builds bus architectures, which are PBS sets that meet all the connectivity requirements in a CG. Topological attributes are evaluated for each bus architecture, e.g., number of PBSs in an architecture, bus utilization, communication conflicts, and maximum data loses. The total bus length is estimated using the actual core placement [30]. The best found bus architecture is characterized for speed in the presence of RLC parasitic.

Step 3) Rescheduling

Using SA and PM, the third step binds CCN to buses and reschedules CN, CCN, and ON for the best found bus architecture and the CN (ON) partitioning identified at Step 1). This step may use the fine grain structure of CCN nodes shown in Fig. 2(c).

V. Bus Architecture Synthesis

A. Modeling for Bus Architecture Synthesis

Definition: Primary bus structure is defined as a *potential* cluster of connected cores. A PBS is *valid* if all its node connectivity exist in the original CG. Otherwise, it is *invalid*.

PBSs are the building blocks for bus architecture synthesis. Fig. 3(c) and (d) shows eight PBSs for the CG in Fig. 3(a). PBSs in Fig. 3(c) are valid. PBSs are characterized by the following physical and topological properties.

1) *PBS utilization percentage:* Utilization is defined as the communication spread in a structure. For example, a PBS corresponds to two links in the CG, i.e., l_{12} and l_{13}. This PBS can also contain l_{23}, the connection between core 2 and core 3. There might, however, be no communication between these cores. Therefore the PBS under-uses its structure. We consider the unused element l_{23} as a redundant link of the PBS. The PBS utilization percentage P_u was defined as $P_u = (2N_b/n(n-1)) \times 100\%$, where N_b is the number of links in a PBS, and n is the number of associated cores in a PBS. The maximum PBS utilization occurs when all associated cores communicate between each other, and the PBS corresponds to a clique in the CG.
2) *Communication conflict:* A PBS is implemented as a shared bus in the system architecture. Performance of a bus architecture can be evaluated by its contention. For a static time scheduling of tasks, it is important to evaluate if there is a communication conflict in a PBS. Communication conflict of a PBS C_{conflict} is the amount of time overlaps between communications mapped to the same link.
3) *PBS bus length:* PBS bus length is a vital attribute for evaluating the bus speed in the presence deep submicron effects. Longer buses require more silicon area and additional circuitry like bus drivers [4], [29]. Also, the larger cross-coupling and parasitic capacitances of longer buses increase interconnect latency [29]. Larger power dissipation for interconnect and drivers is caused by longer buses. It is, however, difficult to accurately estimate the PBS bus length without contemplating the SoC layout. As we explained in [30], hierarchical cluster growth placement is used for placing IP cores and estimating PBS bus lengths.

Identifying the set of valid PBS has an exponential complexity, if a brute-force algorithm is applied. The upper bound of the total number of PBS is $P_m = 2^l - 1$, where l and P_m represent the number of links in a CG and the maximum possible number of PBSs, respectively. We suggest a more efficient, bitwise algorithm to generate the set of valid PBSs. The algorithm is presented in Fig. 11. First, using the bitwise decoder algorithm, each link label is translated into binary and stored as a set of *basis elements* (a basis element is a link in the CG). Then, in a loop, the bitwise PBS generating algorithm performs a bitwise OR operation on the basis elements to generate new PBS structures. A produced PBS is valid, if and only if all its basis elements are connected. Otherwise, the PBS includes redundant links. If the PBS is valid, the PBS storage is checked to avoid duplications of the same PBS.

Example: The core graph in Fig. 3(a) has four cores. Binary numbers are used to represent links e_{ij}, i.e., e_{12} is described as "0011." The number of bits is equal to the number of cores (the first core has the rightmost digit, while the last core has the leftmost digit). In this case, there are four basis elements in the PBS set, namely, e_{12}, e_{13}, e_{14}, and e_{24} labeled in order. Therefore the basis set is $B = \{(1,\text{"0011"}), (2,\text{"0101"}), (3,\text{"1001"}), (4,\text{"1010"})\}$, where the first coordinate is the label of the basis element. Bitwise PBS generating algorithm starts with index 0 and empty PBS storage. All basis elements are added as separate PBSs into the PBS storage. Considering index 3, this is decoded into "0011." Therefore, PBS has two basis elements, namely, $(1,\text{"0011"})$ and $(2,\text{"0101"})$. This PBS is valid because all the basis elements are connected. PBS is then validated with the PBS storage. The storage is updated if there is no such PBS.

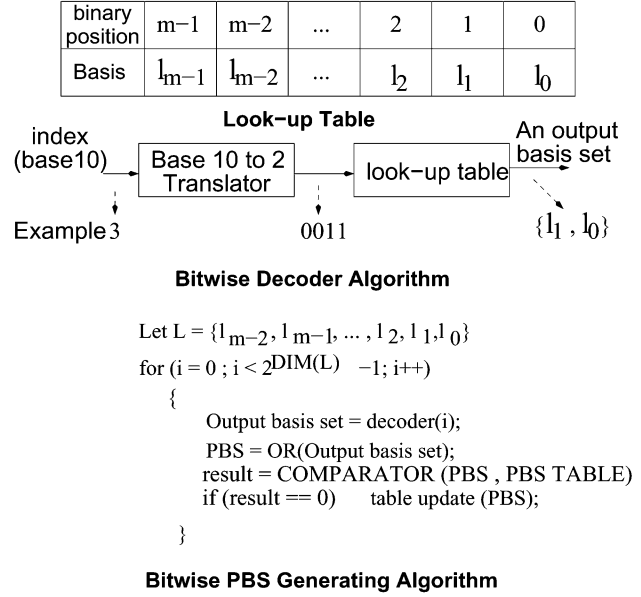


Fig. 11. Bitwise decoder and bitwise PBS generating algorithms.

Definition: Bus architecture synthesis table describes the relationship between a set of PBSs and the connectivity requirements in a CG. The number of rows is similar to the number of basis link elements in the CG. The number of columns is the dimension of the PBS set. An entry in the table has value "X" if the PBS corresponding to the column includes the basis link element specific to the row.

Examples of BA synthesis tables are shown in Fig. 12. The tables are for the CG in Fig. 3(a). Connectivity requirements are expressed as the complete set of basis link elements extracted from a CG. For example, the PBS B_{123} incorporates the basis elements l_{12} and l_{13} (see column B123). Using the BA synthesis table, a bus architecture can be constructed by selecting at least one valid PBS (column) for each basis link element (row). Section V-B explains the synthesis algorithm. Selecting a PBS to satisfy connectivity requirements depends on the PBS properties (utilization percentage, communication conflict, and bus length). For example, if the total utilization percentage has the highest priority then the largest clique must be selected first. The two tables in Fig. 12 correspond to the same CG but have their columns ordered for different performance requirements.

B. Bus Architecture Synthesis Algorithm

Depending on their structure, bus architectures (BAs) can be either *nonredundant* or *redundant* structures, and *flat* or

Primary Bus Structures (Connectivity Requirements)

	B12	B13	B14	B24	B123	B134	B124	B1234
l_{12}	X				X		X	X
l_{13}		X			X	X		X
l_{14}			X			X	X	X
l_{24}				X			X	X

Primary Bus Structures (Connectivity Requirements)

	B12	B123	B124	B14	B13	B24	B1234	B134
l_{13}		X			X		X	X
l_{24}			X			X	X	
l_{14}			X	X			X	X
l_{12}	X	X	X				X	

Fig. 12. Bus architecture synthesis tables.

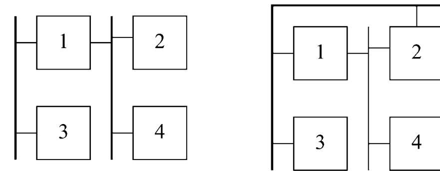


Fig. 13. Nonredundant nonhierarchical and redundant nonhierarchical bus architectures.

Primary Bus Structures (Connectivity Requirements)

		B12	B13	B14	B24	B123	B134	B124	B1234
e1	l_{12}	X				(X)		X	X
e1	l_{13}		X			X	X		X
e2	l_{14}			(X)			X	X	X
e3	l_{24}				(X)			X	X
		e11	e21			e12	e22	e13	e14

```
L = number of basis link elements;
i = 1;
do  selecting a structure to satisfy connectivity i
    eliminate candidate structures;
    pre−eliminate other candidates of other connectivities;
    i = i + 1;
 until  i > L
```

Fig. 14. Select-eliminate algorithm.

hierarchical. The core connectivity offered by a nonredundant bus tightly matches the nature of the communication links in the CG of the application. Also, there is a single connection through a bus for any CG link. There are no core connections, which do not correspond to a link. Nonredundant structures have the benefits of using minimal resources for offering the needed core connectivity, and require no additional control circuitry (like for segmented buses), because a single channel is used to communicate between any two IP cores. The structure is simple (thus simplifies the bus routing step) but lacks the concurrency advantage. In contrast, redundant structures have superior concurrency, and thus decrease communication conflicts. However, expensive control logic is required to intelligently drive the shared bus. In flat (nonhierarchical) bus architectures there are no bus-to-bus communications, as buses link only cores. Hierarchical bus architectures include bus-to-bus communications through bridge circuits [1]. Examples of nonredundant, nonhierarchical (NRNH) and redundant, nonhierarchical (RNH) bus architectures are given in Fig. 13. The NRNH bus architecture in Fig. 13(a) uses the shared bus B_{124} to serve communication links l_{12}, l_{14}, l_{24}, and the point-to-point bus B_{13} for the link l_{13}. Fig. 13(b) shows an RNH bus architecture, in which two buses can be used to implement the link l_{12}. For the NRNH structure, given a destination address, bus selection is statically assigned by the core-bus interface controller. Its routing area is less compared to the redundant structure. The NRNH structure leaves concurrency exploration to task rescheduling at the system level (Step 3 in the methodology in Fig. 10).

We consider only NRNH bus architectures. This is motivated by our goal of designing resource constrained SoC with minimal architectures (thus minimal bus architecture) and software support. However, we showed in [31] and [32] that the modeling for bus architecture synthesis (including CG, PBS, and BA synthesis tables) supports the other bus architecture types. We propose the select-eliminate (SE) algorithm to generate NRNH bus architectures based on the satisfaction of the core connectivity requirements. The SE algorithm is represented in Fig. 14. To illustrate the algorithm, we use the simple BA synthesis table in Fig. 12(a). For example, to satisfy the l_{12} connectivity, one of the four PBS $\{B_{12}, B_{123}, B_{124}, B_{1234}\}$ has to be chosen. Suppose PBS B_{123} is chosen; the rest of the candidates must be eliminated, so that there is no redundancy in the final structure. The horizontal dashed line S_1 represents the eliminated structures. Once a structure is eliminated, it automatically voids the whole column. Vertical dashed lines e_{11}, e_{12}, e_{13}, e_{14} show the eliminated column. PBS B_{123} satisfies only the l_{12} and l_{13} connectivity. Therefore, another horizontal line S_2 is created with vertical lines e_{21} and e_{22}. Connectivity l_{14} is considered next. There is a candidate left, namely, PBS B_{14}. Once PBS B_{14} is chosen, we have only one candidate, PBS B_{24}, left to satisfy l_{24} connectivity. The generated NRNH BA is composed of PBS B_{124}, PBS B_{14}, and PBS B_{24}. Circled structures in Fig. 14 show the final BA.

The size of a synthesis table grows depending on the number of cores and the number of intercore communications. If the number of cores and interconnects between them is small, the SE algorithm contemplates all possible coverings of the CG links using the available PBS structures. However, if a system consists of more than 20 cores intensively tied up together, the exhaustive SE algorithm becomes infeasible. To allow the SE algorithm to explore the PBS candidate space efficiently, we employed simulated annealing algorithm to search different candidate PBSs while satisfying connectivity requirements. The algorithm randomly chooses a PBS from each requirement row, and combines it into a bus architecture. The cost function for simulated annealing is given by the formula

$$\text{Total cost} = w_l L_t + w_n N_b - w_u C_u + w_c C_c + w_{ml} M_l$$

where L_t is the total bus length, N_b is the number of buses, C_u is the total bus utilization $\left(C_u = \left(\sum_{\forall \text{ PBS } i \in BA} P_u^i\right) / N_b\right)$, C_c is the communication conflict, and M_l is the maximum loss. w_l,

TABLE I
SOLUTION QUALITY FOR PROPOSED CODESIGN ALGORITHMS

Example	# of nodes	cond. dep.	List scheduling		Exploration		Rel. impr.	Partitioning and list scheduling			Combined partitioning and scheduling		
			Latency	Time (s)	Latency	Time (s)	(%)	Latency	Iteration	Time (s)	Latency	Iteration	Time (s)
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
Parallel	27	3	120	0.004	120	123	0	135	1,062	2,008	135	1,315	2,229
Tree	39	4	238	0.007	238	207	0	300	5,712	3,599	300	541	2,138
Fork-join	32	2	42	0.004	38	219	9.52	49	6,366	3,502	49	2,067	2,681
Laplace	31	1	62	0.003	56	140	9.67	69	8,656	3,970	68	12,710	5,475
Graph 1	34	3	40	0.005	39	605	2.5	33	3,169	3,719	33	1,751	4,021
Graph 2	15	1	97	0.002	77	69	20.6	93	328	1,713	90	3,973	2,659
Graph 3	23	2	60	0.004	60	340	0	45	1,168	2,869	45	827	1,711

w_n, w_u, w_c, w_{ml} are weight factors. Maximum loss reflects the maximum data loss in a BA, if there is a conflict in a particular PBS. The first three terms describe the complexity of the bus structure, and the last two terms express the timing conflicts between communications sharing the same bus. Term C_u should be as close as possible to one; hence, its negative value was used in the cost function. The objective is to minimize this cost function.

The absence of buffering for data communications would cause data loss, if several cores are accessing a bus at the same time. As this is undesirable, the cost function for bus architecture synthesis will minimize any bus conflict and potential loss. In addition, Step 3 of the codesign methodology in Fig. 10 (rescheduling for the selected bus architecture) performs any necessary communication rescheduling, so that all bus conflicts are solved and no data loss occurs.

VI. EXPERIMENTAL RESULTS

A set of experiments was defined to study the effectiveness of the proposed hardware/software codesign algorithms.

- *Experiment 1* studied the quality of solutions generated using PMs and SA as compared to existing heuristic algorithms, like list scheduling. It also examined combined task partitioning and scheduling.
- *Experiment 2* observed the capability of the algorithms to scale for large task graphs. It also studied the impact of task granularity on synthesis results.
- *Experiment 3* presents results for automatically synthesizing bus architectures.

Experiments were run on a SUN Sparc 80 workstation.

Experiment 1: Quality of Solutions

The first experiment studied the latency of implementations produced by PM and SA for applications with data and reduced amount of control dependencies. SA cost function did not use the terms regarding the feasibility of the bus architecture constraints. Columns 2 and 3 of Table I present the characteristics of the used task graphs: the number of tasks and the number of conditional dependencies of each graph. Examples *Parallel, Tree,* and *Fork-join* describe popular graph structures, such as parallel threads, tree, or sequence of tree and inverted tree. Task *Laplace* calculates the Laplace transform using a tree structure. Example *Graph 1* has a mixed tree and parallel structure [14]. Examples *Graph 2* and *Graph 3* are the motivational examples in [10]. SA was run with a conservative set of parameters, like high initial temperature, slow cooling schedule, and large temperature length. This lengthened the execution time of the algorithm but simplified the tuning of SA parameters for different applications. This was reasonable considering that achieving a high convergence speed for SA was a secondary goal in our experiments.

The quality of solutions produced using SA and PM was initially related to that of list scheduling. Results were compared with solutions obtained by the method suggested in [14] and [10], one of the few scheduling approaches for graphs with data and control dependencies. Column 4 indicates the schedule latencies computed with list scheduling, and Column 6 presents the schedule latencies found with the proposed exploration technique. Column 8 shows the relative improvement over list scheduling. PM and SA offered results that are superior to list scheduling. Improvements can be as high as 20% (for *Graph 3*). This example was indicated as a typical situation for which list scheduling offers poor results [10]. The reason is that list scheduling allocates task priorities for situations that never occur. This is due to the mutual exclusiveness of certain condition values and the controlled tasks. SA-based scheduling does not face this disadvantage. For *Graph 1,* the proposed method offered a slightly better solution, because it left a certain hardware resource idle, even though there were tasks ready for execution on that resource. Then, a higher priority task was scheduled in the "near" future on that resource without having to wait for the smaller priority task to end. This scheduling strategy can not be achieved in nonpreemptive list scheduling, where tasks are greedily scheduled. As shown in columns 5 and 7, list scheduling is significantly faster than SA and PM based scheduling. This suggests that the proposed method is well suited for synthesis, but less applicable for fast performance estimation.

The next experiment analyzed the effectiveness of combined task partitioning and scheduling using SA and PM. Obtained results were compared to the synthesis scenario in which partitioning was based on SA guided by scheduling using list scheduling with critical path as priority function. Columns 9–11 in Table I indicate the resulting system latency, the SA iteration of the best solution, and the corresponding execution time for partitioning guided by list scheduling. Columns 12–14 present the same elements for combined partitioning and scheduling using SA and PM. The combined approach is capable of producing somewhat better results than SA and list scheduling. For the cases in which the two approaches generated solutions with same latency, the combined partitioning and scheduling approach showed faster convergence, and thus shorter execution

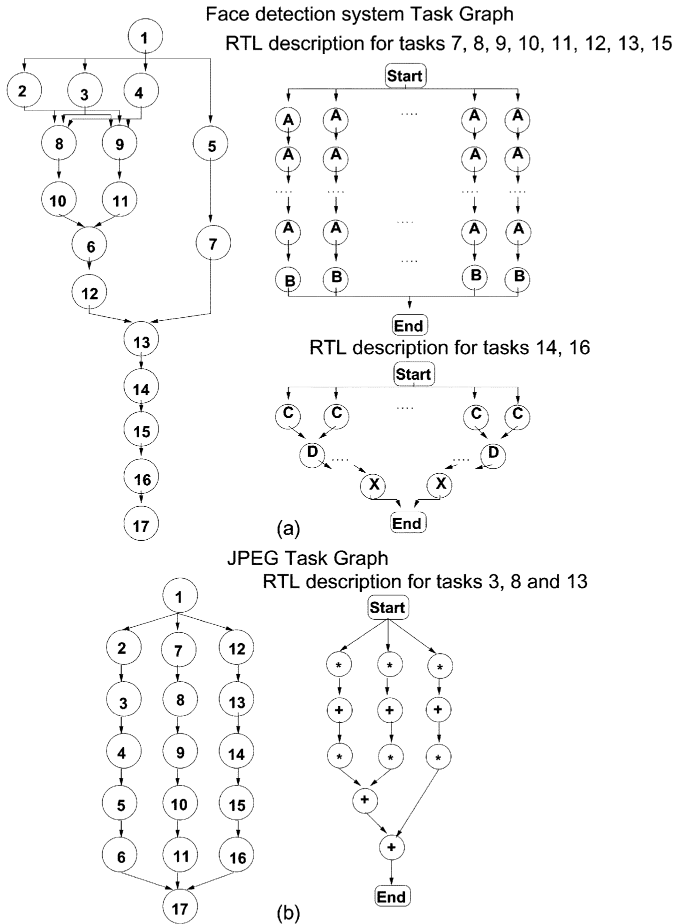


Fig. 15. HDCG for face detection system and JPEG algorithm.

time. The proposed SA and PM based codesign method has a reasonably high computational complexity. Execution time was less than one hour for most of the cases.

Experiment 2: Algorithm Scaling

This set of experiments observed the capability of the codesign algorithms to scale for large task graphs. It also considered the relationship between task granularity and system latency of the implementation. Two examples were used: Face Detection System [33] for wireless sensor networks and JPEG algorithm.

Fig. 15(a) shows the task graph for face detection system. To obtain graphs of different sizes, we specified the system at different levels of granularity. The coarse description included only 17 tasks. Then, tasks 7–16, which include many more operations than the rest of the tasks, were decomposed into smaller tasks, as shown in Fig. 15(a). Smaller task granularities corresponded to situations in which the number of parallel threads (for each of the tasks 7–16) became higher. As granularity went down, a higher number of resources was considered for each example. The assumption was that more simpler hardware blocks can be considered while keeping the total system cost constant. We also assumed that each hardware resource has a single thread of control, thus it cannot execute several tasks simultaneously.

Table II presents the obtained results. The algorithm has a fairly fast convergence for task graphs below 100 tasks. For larger graphs, algorithm convergence is much slower, thus execution times correspondingly increased. Hence, the proposed

TABLE II
EXPERIMENTAL RESULTS FOR FACE DETECTION SYSTEM

Example	Number of nodes	Resources	Latency	Iteration	Time (min)
1	17	3 GPP	1,114129	356	8
2	33	2 GPP/ 1 ALU/ 1 MU	1,114,129	519	14
3	59	2 GPP/ 2 ALU/ 2 MU	860,625	4,063	158
4	135	2 GPP/ 4 ALU/ 4 MU	671,761	10,070	4,500
5	239	2 GPP/ 8 ALU/ 8 MU	456,369	32	2,693

TABLE III
EXPERIMENTAL RESULTS FOR JPEG ALGORITHM

	Example	Time Optimization		
		System level description	System + RTL description	Improvement (%)
1	2GPP + 1A + 1M	184640	182720	1.03
2	2GPP + 2A + 2M	157120	125120	20.3
3	2GPP + 3A + 3M	137920	125120	9.28
4	3GPP + 1A + 1M	182720	182720	0
5	3GPP + 2A + 2M	157120	125120	20.3
6	3GPP + 3A + 3M	137920	99520	27.84

codesign algorithms should not be used for performance estimation, such as when possible architectures are analyzed. Instead, the algorithms are meant to be used for generating an implementation for a given architecture case in which execution time is less important than the quality of the found solutions. Regarding the importance of task granularity, it was noted that smaller granularities help in finding solutions with shorter latency. This is because higher concurrency can be achieved than for graphs expressed at coarser levels. For fine granularities (like Example 5), the exploration algorithm found solutions close to those obtained by greedy heuristics, like list scheduling. This is because the impact of individual partitioning or scheduling decisions became much less than for coarse graphs.

Fig. 15(b) presents the task graph for the JPEG algorithm. The task graph included 17 tasks. The RTL structure of tasks 3, 8, and 13 was shown in the right part of the figure. These tasks represent the IDCT module of JPEG. Six experiments were conducted. Each experiment employed a different number of general purpose processors (GPPs) for software, and a different number of modules [adders (A) and multipliers (M)] for hardware. Column 2 in Table III shows the number of hardware resources for each example. Columns 3, 4, and 5 present the latencies offered by codesign using coarse (system) and fine (system + RTL) descriptions, and the corresponding latency improvements. Note that latency improvement can be as high as 27% (Line 6 in the table), if high concurrency can be secured for the system. For this case, operation concurrency for different hardware tasks resulted in significant latency reductions. In one case (Line 4) there was no improvement. The reason is that only one adder and one multiplier were used for hardware; hence no concurrency improvement resulted by using descriptions of finer granularity.

Experiment 3: Bus Architecture Synthesis

The first example presents bus architecture (BA) synthesis results for a network processor [7]. The processor receives Internet packets, reroutes them, and sends them out. Fig. 16 shows the core graph for the network processor. Node 1 corresponds to core for the Power PC, on-chip SRAM, and SRAM controller. Node 2 is the DDR-SRAM controller. Node 3 is the PCI-X core, node 4 is the MCMAL core, and node 5 describes the DMA

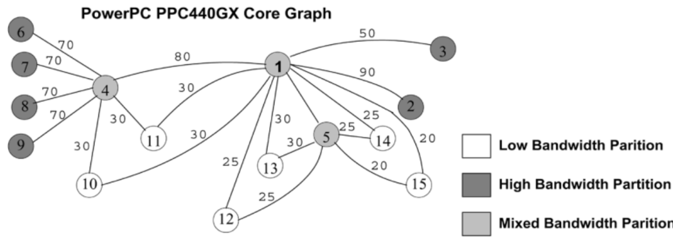


Fig. 16. Core graph for the network processor.

TABLE IV
RESULTS FOR BUS ARCHITECTURE SYNTHESIS FOR THE NETWORK PROCESSOR

(1)	w_l (2)	w_n (3)	w_c (4)	w_r (5)	L_{bus} (6)	N_{bus} (7)	R_{bus} (8)	C_{bus} (9)	l_{max} (10)
1	0.1	0.7	0.1	0.1	0.841	8	0.159	0.125	0.3
2	0.5	0.7	0.1	0.1	0.863	9	0.104	0.111	0.3
3	1.0	0.7	0.1	0.1	0.836	7	0.128	0.428	0.2
4	0.1	0.1	1.0	0.1	0.952	15	0.08	0.0	0.0
5	0.5	0.1	1.0	0.5	0.940	18	0.03	0.0	0.0
6	0.1	0.5	1.0	0.1	0.963	14	0.19	0.0	0.0

core. Nodes 6–9 represent EMAC cores. Nodes 10 and 11 are the high-level data link controller (HDLC) core. Node 12 is the inter-IC (I^2C), node 13 describes the universal asynchronous receiver/transmitter (UART) core, node 14 the general purpose input/output (GPIO) core, and node 15 is the external bus controller core. Edges express the connectivity requirements for cores. Each edge is labeled with the corresponding communication load. Depending on bandwidth requirements, nodes are grouped into the high bandwidth partition, low bandwidth partition, and nodes with mixed bandwidth requirements.

Table IV summarizes the bus architecture synthesis results. Columns 2–5 present the weight factors for bus length, number of buses in the architecture, communication conflicts, and bus redundancies. Different design goals were modeled using the four weight factors. Column 6 shows the resulting bus length. The number of buses in an architecture is indicated in Column 7. Column 8 presents the resulting redundancy. The amount of communication conflicts for a bus architecture is given in Column 9. Finally, the maximum data loss is shown in Column 10.

The first three rows in Table IV correspond to the design scenario in which the bus complexity is minimized, while timing is less important. The weight factor for number of segments has high values. Weights for communication conflicts and redundancies are low. The weight for bus length was varied from small to large values. Bus complexity minimization favored shared buses and discouraged the usage of point-to-point communications. Simple bus architectures resulted, and the number of buses in an architecture is low, between seven and nine. For different weights, however, the bus structure involves different buses. Only three buses were heavily reused in the different architectures. Therefore, it is difficult to postulate that a unique bus will efficiently solve the communication needs for different cases. Complex buses connecting many cores were rarely reused. It is mostly point-to-point links that were reused. The total bus length was high, meaning that individual buses were long. This is reasonable as timing minimization was not a primary concern. If the bus length is important (w_l is high) then the method is able to produce architectures with a small total bus length (see row 3). The average communication conflict was high, around 0.24. Thus, the low communication concurrency resulted in poor timing. Redundancies were also high, meaning that the bus architectures offered core connectivity that was not required. Thus, to obtain simple bus structures with a small total length, all weight factors w_l, w_r, and w_n must have large values.

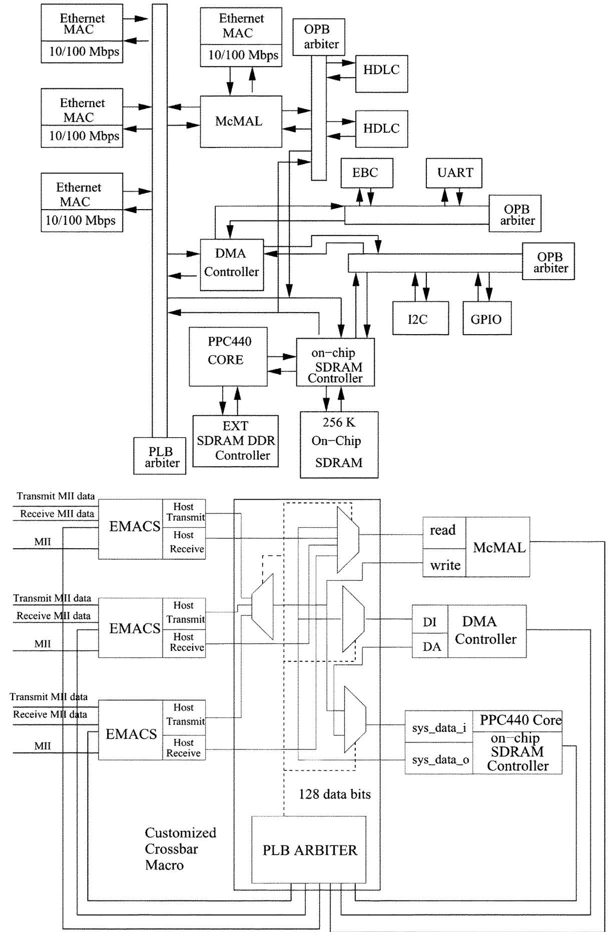


Fig. 17. Synthesized bus architecture for the network processor.

Rows 4–6 correspond to the second design scenario, in which communication overlaps must be avoided, while the other factors are less important. Also, this scenario considers that the required bus speed is low; thus minimizing bus length is secondary. Note that there are no time conflicts and no data losses for the resulting BA. Bus architectures are more complex and include more point-to-point links. Overall bus lengths are larger, which indicates that the individual buses will be slower. Fig. 17 shows the synthesized bus topology for the design scenario in which bus length and complexity are very important ($w_n = w_l = w_r = 1.0$), and communication overlaps ($w_c = 0.1$) are secondary.

The second example consisted of automatically producing optimized bus architectures for the SoC of the JPEG image compression encoder. Fig. 15(b) shows the task graph. After combined hardware/software partitioning, the identified architecture included three processor cores (a distinct core for each parallel sequence), an ASIC for the IDCT tasks, and memory modules for data communication. Each processor has its own local memory. Processors and ASIC communicate through

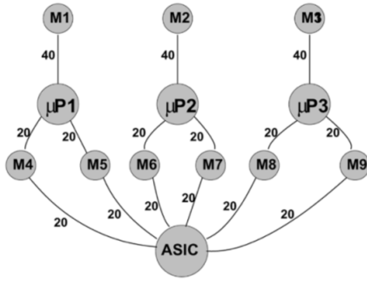


Fig. 18. Core graph for JPEG.

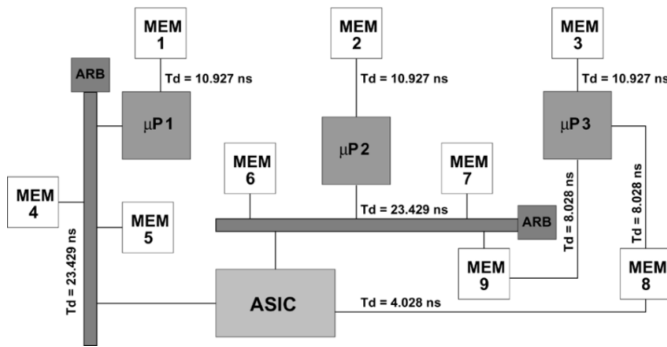


Fig. 19. Bus architecture for JPEG SoC.

shared memory. To improve the processor-ASIC communication speed, interleaved memory blocks were used. This resulted in the core graph shown in Fig. 18. The considered processing technology was an 0.18-μ Taiwan Semiconductor Manufacturing Company (TSMC). The microprocessor cores were of about 5×5 mm^2, memory cores of about 25% of the area of processor cores, and ASICs about 30% of processor core area.

Fig. 19 shows bus architecture synthesis results for the top CG in Fig. 18. The synthesis goal was to generate a fast architecture. Bus architecture complexity was not a major concern, because the number of IP cores was reasonably high. Thus, the goal of BA synthesis was to minimize communication conflicts $(w_c = 1.0)$ and minimize the total bus length $(w_l = 1.0)$ while disregarding the number of buses and redundant structures in a BA $(w_n = w_r = 0.1)$. After bus architecture generation, each of the buses was routed, and the resulting delays are indicated in the figure. Note that the best BA is not perfectly regular, even though the CG is regular. Processor P1, and memory modules M4 and M5, are linked through a shared bus, similar to processor P2 and memory blocks M6 and M7. This happens because the placements of these blocks are similar. However, processor P3 and memories M8 and M9 are linked through a different structure, which improves the speed of the bus for the specific placement of these blocks. This explains that optimized BAs do not depend only on architectural level elements (like the amount of exchanged data between cores) but on layout aspects also.

BA synthesis took less than 5 min on a SUN Blade 100 workstation. This shows that the pruning method of the BA synthesis algorithm allowed to quickly explore the very large solution spaces resulting for SoC with many cores.

VII. Conclusion

This paper presents a layout conscious approach for hardware/software codesign of systems on chip optimized for latency, including an original algorithm for bus architecture synthesis. Compared to similar work, the method addresses layout-related issues that affect system optimization, such as the dependency of task communication speed on interconnect parasitic.

The codesign flow performs three successive steps. 1) Combined partitioning and scheduling uses simulated annealing guided by performance models. PMs symbolically model the relationships between system performance (e.g., latency and communication speed flexibility), system attributes, and design decisions. 2) IP cores are placed using a hierarchical cluster growth algorithm followed by synthesis and routing of the bus architecture. Bus architecture synthesis finds a set of possible building blocks and assembles them together, while pruning the low quality solutions through a novel select-eliminate method. 3) Tasks and communications are rescheduled for the best bus architecture.

The codesign method improves the effectiveness of SoC system-level design. The bus synthesis algorithm creates customized bus architectures in a short time depending on the data communication needs of the application and the required performance. Layout information is important in deciding the bus architecture topology. Experiments showed that it is impractical to postulate a unique bus architecture as being the best, as there is little reuse among bus architectures optimized for different constraints. Experiments also indicated that PMs are more effective than existing metrics, like task priorities. Using PMs, system latency was by 20% shorter than for list scheduling. PMs are able to avoid the modeling limitations of priority functions. PMs are general, flexible, and can be easily extended for new design activities. Their validation is minimal. Combined partitioning and scheduling offers latency improvement and faster convergence compared to explorative partitioning guided by list scheduling.

There are several directions which could extend the presented work. The co-design method could be extended to address new performance attributes, like power consumption. In [11], we discussed PMs for instantaneous and average power consumption of tasks. To include power consumption of interconnect, the concepts of floorplan tree and communication speed flexibility need to be expanded to express interconnect proximity, so that cross-coupling can also be tackled. Also, the co-design method could incorporate new design steps, like finding the time instances at which individual resources can be shut down. This will improve the energy consumption of the system. Other activities, such as functional pipelining or optimization across loops, can be also tackled using PMs. Finally, experiments showed that SA is fairly fast for task graphs with up to 40 tasks. A parallel implementation of SA would allow handling of larger graphs.

Acknowledgment

The authors would like to thank the Associate Editor and the reviewers for their very valuable comments and suggestions.

REFERENCES

[1] IBM CoreConnect bus architecture white paper. [Online]. Available: //http:www-3.ibm.com/chips/products/coreconnect/index.html

[2] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli, "Scheduling for embedded real-time systems," *IEEE Design Test Comput.*, vol. 15, no. 1, pp. 71–82, Jan.–Mar. 1998.

[3] S. Battacharyya, "Hardware/software co-synthesis for DSP systems," in *Programmable Digital Signal Processors: Architecture, Programming, and Application*, Y. Hu, Ed. New York: Marcel Dekker, 2002, pp. 333–378.

[4] T. R. Bednar, P. H. Buffet, R. J. Darden, S. W. Gould, and P. S. Zuchowski, "Issues and strategies for the physical design of system-on-chip ASICs," *IBM J. Res. Develop.*, vol. 46, no. 6, pp. 661–673, Nov. 2002.

[5] T. Blickle, J. Teich, and L. Thiele, "System-level synthesis using evolutionary algorithms," *J. Des. Automation Embedded Syst.*, vol. 3, no. 1, pp. 23–58, 1998.

[6] B. Dave, G. Lakshminarayana, and N. Jha, "COSYN: Hardware-software co-synthesis of heterogeneous distributed embedded systems," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 7, no. 1, pp. 92–104, 1999.

[7] J. Darringer, R. Bergamaschi, S. Battacharyya, D. Brand, A. Herkersdorf, J. Morell, I. Nair, P. Sagmeister, and Y. Shin, "Early analysis tools for system-on-a-chip design," *IBM J. Res. Develop.*, vol. 46, no. 6, pp. 691–707, 2002.

[8] J. M. Daveau, G. F. Marchioro, T. B. Ismail, and A. A. Jerraya, "Protocol selection and interface generation for HW-SW codesign," *IEEE Trans. Very Large Scale (VLSI) Syst.*, vol. 5, no. 3, pp. 136–144, Mar. 1997.

[9] R. Dick and N. Jha, "MOGAC: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, 1997, pp. 522–529.

[10] A. Doboli and P. Eles, "Scheduling under control dependencies for heterogeneous architectures," in *Proc. Int. Conf. Computer-Aided Design*, 1998, pp. 602–608.

[11] A. Doboli, "Integrated hardware/software co-design and high-level synthesis under time, area and energy consumption constraints," in *Proc. Design, Automation and Test Eur. Conf.*, 2001, pp. 612–619.

[12] M. Drinic, D. Kirovski, S. Meguerdichian, and M. Potkonjak, "Latency-guided on-chip bus network design," in *Proc. Int. Conf. Computer-Aided Design*, 2000, pp. 420–423.

[13] R. Ernst, "Codesign of embedded systems: Status and trends," *IEEE Des. Test Comput.*, vol. 13, no. 2, pp. 45–54, Apr.–Jun. 1998.

[14] P. Eles, A. Doboli, P. Pop, and Z. Peng, "Scheduling with bus access optimization for distributed embedded systems," *IEEE Trans. Very Large Scale (VLSI) Syst.*, vol. 8, no. 10, pp. 472–491, Oct. 2000.

[15] D. Gajski and F. Vahid, "Specification and design of embedded hardware/software systems," *IEEE Des. Test Comput.*, pp. 53–67, Spring 1995.

[16] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system level," *ACM Trans. Design Automation Electron. Syst.*, vol. 4, no. 1, pp. 1–11, Jan. 1999.

[17] T. Givargis and F. Vahid, "Platune: A tuning framework for systems-on-a-chip platforms," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 21, no. 11, pp. 1–11, Nov. 2002.

[18] R. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Norwell, MA: Kluwer, 1995.

[19] J. Henkel, "A low power hardware/software partitioning approach for core-based embedded systems," in *Proc. Design Automation Conf.*, 1999, pp. 122–127.

[20] J. Hu, Y. Deng, and R. Marculescu, "System-level point-to-point communication synthesis using floorplanning information," in *Proc. Int. Conf. VLSI Design*, 2002, pp. 573–579.

[21] K. Lahiri, A. Raghunathan, and S. Dey, "Efficient exploration of the SoC communication architecture design space," in *Proc. Int. Conf. Computer-Aided Design*, 2000, pp. 424–430.

[22] J. A. Maestro, D. Mozos, and H. Mecha, "A macroscopic time and cost estimation model allowing task parallelism and hardware sharing for the codesign partitioning process," in *Proc. Design, Automation and Test Eur. Conf.*, 1998, pp. 218–225.

[23] R. Ortega and G. Boriello, "Communication synthesis for distributed embedded systems," in *Proc. Int. Conf. Computer-Aided Design*, 1998, pp. 437–444.

[24] S. Prakash and A. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel Distrib. Comput.*, vol. 16, pp. 338–351, 1992.

[25] C. Reeves *et al.*, *Modern Heuristic Techniques for Combinatorial Problems*. New York: Wiley, 1993.

[26] N. Sherwani, *Algorithms for VLSI Physical Design Automation*. Norwell, MA: Kluwer, 1999.

[27] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich, "Scheduling hardware/software systems using symbolic techniques," in *Proc. Int. Workshop Hardware/Software Co-Design*, 1999, pp. 173–177.

[28] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the system-on-a-chip Woes through communication-based design," in *Proc. Design Automation Conf.*, 2001, pp. 667–672.

[29] D. Sylvester and K. Keutzer, "A global wiring paradigm for deep submicron design," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 19, no. 2, pp. 242–252, Feb. 2000.

[30] N. Thepayasuwan, V. Damle, and A. Doboli, "Bus architecture synthesis for hardware–software co-design of deep submicron systems on chip," in *Proc. ICCD*, 2003, pp. 126–133.

[31] N. Thepayasuwan and A. Doboli, "Layout conscious bus architecture synthesis for deep submicron systems on chip," in *Proc. Design, Automation and Test Eur. Conf.*, 2004, pp. 108–113.

[32] ——, "OSIRIS: Automated synthesis of flat and hierarchical bus architectures for deep submicron systems on chip," in *Proc. ISVLSI*, 2004, pp. 264–265.

[33] Y. Weng and A. Doboli, "Smart sensor architecture customized for image processing applications," in *Proc. 10th IEEE Real-Time and Embedded Technology and Applications Symp.*, 2004, pp. 396–403.

[34] T. Y. Yen and W. Wolf, *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. Norwell, MA: Kluwer, 1997.

Nattawut Thepayasuwan (M'01) received the B.Eng. degree from Khon Kaen University, Khon Kaen, Thailand, in 1993, and the M.S.E.E. degree from Rochester Institute of Technology, Rochester, NY, in 1996. He is currently pursuing the Ph.D. degree in the Department of Electrical and Computer Engineering, State University of New York (SUNY) at Stony Brook.

His research interests are in the area of hardware/software codesign and system-level synthesis.

Mr. Thepayasuwan is a member of Eta Kappa Nu.

Alex Doboli (S'99–M'01) received the M.S. and Ph.D. degrees in computer science from Politehnica University, Timisoara, Romania, in 1990 and 1997, respectively, and the Ph.D. degree in computer engineering from the University of Cincinnati, Cincinnati, OH, in 2000.

He is currently an Assistant Professor in the Department of Electrical and Computer Engineering, State University of New York (SUNY) at Stony Brook. His research is in VLSI system design automation, with a special interest in mixed-signal CAD and hardware/software codesign.

Dr. Doboli is a member of Sigma Xi.