Layout Conscious Approach and Bus Architecture Synthesis for Hardware-Software Co-Design of Systems on Chip Optimized for Speed

Nattawut Thepayasuwon, Member, IEEE and Alex Doboli, Member, IEEE

Abstract—This paper presents a layout conscious approach for hardware-software co-design 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 co-design flow executes three consecutive steps. (i) Combined partitioning and scheduling: Besides partitioning and scheduling, this step also identifies the minimum speed constraints for each data link. (ii) 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 together for minimizing bus length and complexity. Poor solutions are pruned using a special table structure and select-eliminated method. (iii) Re-scheduling for the best bus architecture. The paper offers extensive experiments for the proposed co-design method, including bus architecture synthesis for a network processor and a JPEG SoC.

Index Terms—Bus Architecture Synthesis, Hardware/Software Co-design, Systems-on-Chip.

I. INTRODUCTION

Many embedded systems must meet stringent cost, timing, and energy consumption constraints [10] [18] [35]. 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 128k of RAM and 256k of flash memory), and incorporate customized co-processors and I/O peripherals (including RF and analog circuits). Typical examples include embedded systems for telecommunication and multimedia, like cell phones, digital cameras, and personal communicators. Systems-on-Chip (SoC) are single-chip implementations of embedded systems. Compared to printed circuit board designs, SoC offer higher performance and reliability at cheaper costs [10]. 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 [3].

For SoC realized in deep submicron 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 [6] [12] [40]. Figure 1 illustrates the impact of layout parasitics on data communication speed and system design. Figure 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 co-design step decides to allocate a single 266MHz 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 Figure 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 Figure 1(b), because the system concurrency improves. The bus speeds of 133MHz, 133MHz, and 33 MHz were found based on the physical locations of cores, and the RLC parasitic of the routed buses [40]. This example arguments that the communication sub-system 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 [38] [47].

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

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. Figure 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 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 Figure 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 sub-system 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 co-design process is aware of certain low level aspects, like core placement and bus routing.

This paper describes a hardware-software co-design method for developing SoC implementations subject to latency minimization. The novelty is in proposing a systematic, layout-conscious approach for tackling the SoC communication sub-system, 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 [20] with data dependencies and reduced number of control dependencies. The set of available hardware resources and the SoC area are known. The co-design method includes three subsequent parts: (1) combined partitioning and static non-preemptive scheduling, (2) bus architecture synthesis, and (3) re-scheduling for the best bus architecture. The first step is an exploration process based on simulated annealing algorithm (SA) [34]. 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 (PM), a graph-based description, that symbolically captures the relationships between 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 PBS bitwise generation algorithm, bus architecture synthesis first identifies a set of possible building blocks, and 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.

The paper is organized as follows. Section 2 presents related work. Section 3 discusses system modeling. Section 4 introduces the proposed co-design approach. Bus architecture synthesis is presented next. Experimental results are given in Section 6. Section 7 enumerates plans for future work. Finally, conclusions are offered.

II. RELATED WORK

Over the last ten years or so, a variety of hardware/software co-design methodologies were proposed for designing embedded systems optimized for cost, speed, and power consumption [5] [19] [21]. A typical co-design 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, co-design approaches can be classified into three groups: for data dominated systems [5] [9] [24] [28], for control intensive systems [4], and for applications with substantial data processing and reduced amount of control [20] [37]. Balarin et al [4] present POLIS, an approach for control dominated real-time embedded applications. For data dominated systems, Prakash and Parker [33] and Bender [7] formulate the co-design problem as a mixed-integer linear programming (MILP) problem. A linear equation solver finds the optimal implementation. The disadvantage of MILP-based co-design is its limitation to small size applications. The alternative is to employ heuristic algorithms, such as greedy priority-driven clustering [9], list scheduling methods [5] [15] [20] [28], iterative improvement heuristics, like simulated annealing and tabu search [20], and genetic algorithms [8] [14] [37].

Heuristic methods can be used for large task graphs [20]. The disadvantage is that the solution optimality is difficult to be characterized. For example, greedy priority-driven algorithms offer good average results, but they might give poor solutions for situations not captured by the priority function [15]. Henkel [25] 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, Lakshminarayana and Jha [9], and Dick and Jha [14] propose co-synthesis 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 inter-processor concurrency, preemptive and non-preemptive scheduling, and memory constraints. Givargis and Vahid [23] 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 [11] [22] [32] [46] focuses on multiprocessor embedded systems on a printed circuit board. Research addresses interface design [11] [32], communication packeting [20], mapping and scheduling [46]. This work does not tackle the hardware and layout details of the SoC communication sub-systems. Sgroi et al [39] 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 [30] focus on communication protocol selection for
a communication architecture template including shared and dedicated buses. Recently, Drinic et al. [17] 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. [26] 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 [22] [26] [30]. This is reasonable for small SoC for which the designer manually designs the buses. However, it is not effective for SoC with large number of cores.

Compared to similar work, this paper proposes a new hardware-software co-design 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 co-design 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 towards new design requirements, and scales reasonably well with the application size.

III. SYSTEM REPRESENTATION FOR CO-DESIGN

A. Embedded System Modeling

The quadruple < HDCG, Resources, Floorplan, PM > 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.

A. HDCG (Hierarchical Data and Control Dependency Graph)

Definition: A Hierarchical Data and Control Dependency Graph is the triplet $HDCG = \langle S_{CN}, S_{CGCN}, S_{A} \rangle$, where $S_{CN}$ is the set of cluster nodes, $S_{CGCN}$ is the set of communication cluster nodes, and $S_{A}$ is the set of arcs. HDCGs are acyclic polar graphs having one start node and one end node.

Figure 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 $CN_i = \langle S_{ON}, S_{Arca} \rangle$,

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

Each CN and ON has a triplet $< T^s, T^{cz}, T^{end} >$ 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.

Communication cluster nodes (CCN) represent data communications between CNs mapped to different processing units. CCNs are shown as black bubbles in Figure 2(a). At the fine grain level, each $CCN_j$ has a linear structure, as shown in Figure 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 $a \in S_A$ is the triplet $< n_i, n_j, cond_k >$, where $n_i, n_j \in S_{CN} \cup S_{CGCN}$, and $cond_k$ is a boolean variable or $\emptyset$. For data dependencies, $cond_k = \emptyset$. 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 Figure 2(a), boolean variables are depicted in italics. Node $CN_2$ computes variable $cond_1$. If variable $cond_1$ is true then the communication cluster node following $CN_2$ is performed. $CN_2$ is executed for a false value, indicated as $-cond_1$ 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 non-preemptive.

Due to the acyclic nature of an HDCG, each CN, ON, and CCN is executed at most once for a traversal of the graph.
HDCG with reduced number of control dependencies.

actual hardware resource on which a CN or ON is executed.

the communication link between cores given. Through exploration, co-design identifies the function

assumes that the number and type of available hardware and an operation-level representation (for exploring hardware sharing across tasks). HDCG are similar to control data flow graphs [21] and conditional process graphs [20].

Even though system functionality could be expressed using operation nodes only, cluster nodes prevent the unnecessary growth of the design space, and hence, a very lengthy co-design process. This is consistent as most bus standards for SoC, i.e., AMBA [1] and IBM CoreConnect [2], have different buses to support data communication at different bandwidths.

C. Floorplan

Definition: Floorplan Trees (FT) are binary tree structures having following two properties: (1) Leaf nodes correspond to IP cores. (2) Each internal node links the two nodes that exchange the maximum amount of data with each other. By definition, an internal node \( I_{i} \) exchanges with leaf node \( LF_{k} \) if \( LF_{k} \notin I_{i} \), 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 \( I_{i} \) (\( w_{LF,k} I_{i} = \sum_{p \in \text{Subtree}(I_{i})} w_{LF,k} F_{p} \)).

The amount of data communicated between two internal nodes \( I_{i} \) and \( I_{j} \) is equal to the sum of all communications between node \( I_{i} \) and all leaf nodes of the subtree originating at node \( I_{j} \).

Figure 4(a) presents a set of six IP cores, and Figure 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. Subsection 3.2 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 co-design process. Instead, FTs abstract away the horizontal and vertical cutlines in the slicing trees [36] for floorplanning, and replace precise bus speed evaluation with finding a lower bound of the bus speed.

This avoids co-design 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
of its start time
presents the relationship between latency and the design
decisions taken during co-design, like task partitioning and
part of the PM includes all nodes and solid edges in Figure
max
5(b).
max
The maximum of the end times of all predecessors. Addition
outputs of cluster nodes. The start time of a CN has to be larger than
nodes express that the end time
are easily captured in the PM by changing the orientation of
the operational nodes for the operands described by symbolic
variables and arcs.

Figure 5 shows an HDCG, and its corresponding PM for latency. The figure assumes that ON2, ON5, and ON3 are executed in this order on the shared processor. The constant part of the PM includes all nodes and solid edges in Figure 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
end
of node i is the sum of its start time
start
and its execution time
exec.
The variable part presents the relationship between latency and the design decisions taken during co-design, like task partitioning and scheduling. In Figure 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 co-design activities. PMs are very flexible, as they allow easy definition of new performance attributes, or description of additional relationships between performance attributes and co-design activities. For example, the attribute of communication speed flexibility, defined in Subsection 3.2, 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 CSF calculation avoid generating designs, which are difficult to realize. This helps faster closure by improving the feasibility of system design. Maestro et al [31] 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 co-design methodology.

B. Modeling of Co-Design Activities

A. Modeling of Data and Control Dependencies

Figure 6(a) shows the general rule for expressing data dependencies in a PM. Node n is executed only after all its predecessor nodes 1, 2, ..., k are performed. A max node was introduced to express that the start time
start
of node n is greater than or equal than the end times of all its predecessors. The addition node for node n symbolically relates the node’s end times
end
and its execution time
exec.
Similar constructs are introduced for all data dependencies. The right most addition node of the resulting PM denotes the system latency.

Figure 6(b) presents the general rule for representing control dependencies in a PM. According to the HDCG definition, if condition C is true then nodes 2, ..., p are performed, otherwise nodes 3, ..., q are executed. Max and addition nodes are introduced using the same rules as for data dependencies. The conditional execution of nodes 2 and 3 was represented in the PM by annotating the input arcs to their max nodes with the corresponding condition value (true condition C for node 2 and false condition C for node 3). Node r, which unifies the two branches, has a min node instead of the max node.

The following rule is applied for numerically evaluating PMs with conditional dependencies: for a certain condition value, the arcs labeled with that condition will propagate the numerical values that result from the PM evaluation. Arcs annotated with the opposite condition value will propagate the value ∞. For example in Figure 6(b), for a true condition C, the input arc to the max node for node 2 propagates the output
Fig. 7. Scheduling with data dependencies of the addition node $T_{c}^{+} + T_{e}^{+}$. The input arc to the max node for node 3 transmits $\infty$. The min node of node $r$ eliminates the infinite values propagated through the non-selected branch.

B. Modeling of Cluster Node Partitioning and Operation Binding

From model point of view, cluster node partitioning and operation binding finds the definition of the function $M_{p} \text{mappedto}(i) \in \mathcal{T}_{s}$ 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 $T_{c}^{+}$ of node $i$ changes for each new resource type, and its numerical value is updated in PMs.

C. 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 non-preemptive 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 Figure 7(a) 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.

In the presence of control dependencies, scheduling is more difficult due to the uncertainty of control dependencies [20]. If scheduling is performed across control constructs, node schedules must satisfy following three requirements [20]: (1) respecting the execution order defined by data and control dependencies, (2) maintaining the conditional node execution as defined by a HDCG (e.g., if a condition value is true then only the nodes from the true branch will be executed), and (3) executing at most once each CN and CCN in the HDCG.

The first two requirements are already captured by the PM modeling of data and control dependencies. To illustrate the third constraint, lets assume a schedule for the graph in Figure 7(b), so that Node 4 would be executed before Node 1 if condition $C$ is true, and after Node 1 if condition $C$ is false. Nodes 2, 3, and 4 share the same resource. This situation could occur if the branch for true condition $C$ is long, and the branch for false condition $C$ is short as compared to the path that Node 4 pertains to. This schedule is incorrect because for a false condition $C$, Node 4 is executed twice, both before and after Node 1.

Three cases are possible for satisfying the third correctness requirement:

1) Before the control structure: Node 4 is executed before Node 1 that calculates condition $C$. In this case, the scheduling of Node 4 does not depend on condition $C$, and Node 4 is executed only once. Figure 7(b) depicts this situation, and the dashed arc enforces that Node 1 starts after Node 4 terminates.

2) During the control structure: For a given condition value (e.g., true condition $C$), Node 4 is scheduled to execute after Node 1 but before Node $r$. To maintain the scheduling correctness, for the opposite value of condition $C$, it is required that Node 4 executes only after Node 1 ends. Figure 7(d) depicts this case. Two new dashed arcs are introduced, so that Node 4 starts after Node 2 if condition $C$ is true, and after Node 1 if condition $C$ is false.

3) After the control structure: Node 4 executes after Node $r$. Its scheduling time depends on the value of condition $C$. However, as the value of condition $C$ is already known by the time Node 4 starts, it is trivial to guarantee single execution for Node 4. In Figure 7(c), this case is reflected by having the dashed arc between the min node of Node $r$ and the max node of Node 4.

D. Modeling of Communication Speed Flexibility

The execution time of communication cluster nodes (CCN) can not be accurately estimated at the system level. This is because the bus speed depends on the bus length, thus, on the placement of IP cores, the bus architecture and bus routing. As shown in Figure 10, this information is not available during
task partitioning and scheduling.

Definition: For each data link, the communication speed flexibility (CSF) 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 co-design methodology in Figure 10 first identifies feasible CSF requirements for each 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 5.

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. Figure 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.

Figure 8 shows an example for the PM expressing the constraints between CSF values. CSF values for nodes \(\text{CSF}(1, 4), \text{CSF}(3, 5), \) and \(\text{CSF}(2, 6)\) (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 \(D_1\) 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 \(T_i^{\text{min}}\) as inputs. The output of the first addition node is input to the second addition node, which has variable \(T_i^{\text{CSF}}\) 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 \(T_i^{\text{CSF}}\) depends on the CSF value of the communication link used for CCN \(i\) and the amount of data. Figure 9 shows an example. CNs 1 and 3 are mapped to different cores, and CCN 2 is their data communication. Figure 9(b) presents the PM for latency, including the two components of the communication time.

IV. CO-DESIGN METHODOLOGY

Figure 10 presents the proposed hardware-software co-design 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 co-design 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 re-schedules cluster nodes, communication cluster nodes, and operation nodes while keeping the partitioning and the bus architecture unchanged.

The proposed co-design methodology is sub-optimal for the given co-design problem. The optimal solution requires simultaneous partitioning, scheduling, and bus architecture synthesis. The experimental section shows that this is difficult, because the three activities are computationally fairly complex. Hence, the proposed methodology sequences these activities while accommodating the circular reasoning [45] inherent to the co-design problem: partitioning and scheduling are solved for a certain data communication speed, even though the bus architecture and speed are known only at the later
design steps. For handling circular reasoning, Wolf suggests a methodology that serializes the co-design activities depending on their importance [45]. We adopted a similar strategy with the modification that critical information about later steps is incorporated into the earlier co-design activities. For partitioning and scheduling, we used floorplan trees to qualitatively predict the structure and lengths of buses, thus their achievable speed.

**Step 1. Partitioning and scheduling:** First, Performance Models (PM) are generated for an HDCG using the rules presented in Section 3. Next, a simulated annealing (SA) [34] based exploration loop conducts simultaneous partitioning and scheduling. For each CN (ON) i, attributes `Mappedto(i)` (the hardware resource that executes the node), `T^se_i` (the execution time on that resource), and `T^s_i` (the start time) are the unknowns for co-design. Cluster node partitioning to processors and operation binding to FUs are modeled by the unknowns `Mappedto(i)` and `T^s_i`. The scheduling of cluster nodes, communication cluster nodes, and operation nodes is described by the unknowns `R_i` and `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 (1) differ from 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, Floorplan Trees (FT), and Core Graphs (CG) are updated for each newly selected solution. For each co-design solution, the system latency and communication speed flexibility (CSF) are calculated by evaluating their PMs with all node attributes `R_i` and `T^se_i` instantiated to their numerical values. Starting solutions were obtained by uniformly distributing nodes to processors and operation binding to FUs are modeled by the unknowns `Mappedto(i)` and `T^s_i`. The scheduling of cluster nodes, communication cluster nodes, and operation nodes is described by the unknowns `R_i` and `T_i^s`. Possible numerical values for the unknowns `R_i` and `T_i^s` are searched during exploration.

The cost function for SA is

$$\text{Cost} = \alpha \times \text{Latency} + \beta \times \frac{1}{\sum \text{links}} + \gamma \times \# \text{ buses} + \delta \times \text{unnecessary connectivity},$$

The function to be minimized models the system latency and the feasibility of the bus architecture constraints. To maximize feasibility, communication speed flexibility (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. Subsection 3.2D 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 bus values, but very tight values for other. Such a bus architectures 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 few overlapping between communications, and there is little unnecessary core connectivity. A more detailed modeling of these attributes are used for bus architecture synthesis discussed in Section 5. α, β, γ and δ are weights.

**Step 2. Bus architecture synthesis:** Core Graph (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 described in Subsection 5.3. 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 (PBS) is created using the bitwise PBS generating algorithm (presented in Subsection 5.2). PBS 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 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 $1 - (p_1 + p_2)$ analyzed schedules. For example, if $p_1 = 0.01$ and $p_2 = 0.1$ then on the average, eight schedules are examined for each partition. If the execution order of a node pair is modified then the algorithm verifies that the new ordering does not create cycles in the PMs.

Fig. 10. Hardware-software co-design methodology
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. The best found bus architecture is characterized for speed in the presence of RLC parasitic.

Step 3. Re-scheduling: Using SA and PM, the third step binds CCN to buses and re-schedules 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 Figure 2(c).

V. BUS ARCHITECTURE SYNTHESIS

A. Modeling for Bus Architecture Synthesis

Definition: Primary bus structure (PBS) 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.

PBS are the building blocks for bus architecture synthesis. Figures 3(c) and 3(d) show eight PBS for the CG in Figure 3(a). PBS on Figure 3(c) are valid. PBS are characterized by 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 \( P_u = \frac{N_u}{N} \times 100\% \), where \( N_u \) 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 [6] [40]. Also, the larger cross coupling and parasitic capacitances of longer buses increase interconnect latency [40]. 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 explained in Subsection 5.3, 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 Figure 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 Figure 3(a) has 4 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 number of cores (the first core has the right most digit, while the last core has the left most digit). In this case, there are 4 basis elements in the PBS set, namely, \( e_{12}, e_{13}, e_{14}, \) and \( e_{24} \) labeled in order. Therefore the basis set is \( B = \{(1,0011),(2,0101),(3,1001),(4,1010)\} \), where the first coordinate is the label of the basis element. Bitwise PBS generating algorithm starts with index 0 and empty PBS storage. All basic elements are added as separate PBS into the PBS storage. Considering index 3, this is decoded into "0011". Therefore, PBS has two basis elements, namely, \((1,0011)\) and \((2,0101)\). 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.

Definition: Bus architecture synthesis table describes the relationship between a set of PBS 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 Figure 12. The tables are for the CG in Figure 3(a). Connectivity requirements are expressed as the complete set of basis link elements.

![Bus Architecture Synthesis Table](image-url)
A synthesis table, a bus architecture can be constructed by selecting at least one valid PBS (column) for each basis link element (row). Subsection 5.2 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 Figure 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 (BA) can be either non-redundant or redundant, non-hierarchical or hierarchical. The core connectivity offered by a non-redundant 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. Non-redundant 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 (non-hierarchical) 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 [2]. Examples of non-redundant, non-hierarchical (NRNH) and redundant, non-hierarchical (RNH) bus architectures are given in Figure 13. The NRNH bus architecture in Figure 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 \( l_{13} \). Figure 13(b) shows a 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 re-scheduling at the system level (Step 3 in the methodology in Figure 10).

We consider only non-redundant, non-hierarchical (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 [41] and [42] 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. SE algorithm is represented in Figure 14. To illustrate the algorithm, we use the simple BA synthesis table in Figure 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 dash line \( S_1 \) represents the eliminated structures. Once a structure is eliminated, it automatically voids the whole column. Vertical dash 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 \( B_{124}, B_{14}, \) and PBS \( B_{24} \). Circled structures in Figure 14 show the final BA.

The size of a synthesis table grows depending on the number of cores and the number of inter-core communications. If number of cores and interconnects between them is small, the
The object is to minimize the exhaust SE algorithm becomes infeasible. To allow the SE algorithm 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

\[
\text{Total cost} = w_1 L_1 + w_2 N_b - w_3 C_u + w_4 C_c + w_m M_l,
\]

where \( L_1 \) is the total bus length, \( N_b \) is the number of buses, \( C_u \) is the total bus utilization \( C_u = \sum_{PBS \in L_b} \frac{Pu}{NH} \), \( C_c \) is the communication conflict, and \( M_l \) is the maximum loss. \( w_1, w_2, w_3, w_4, w_m \) 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 conflict, and the potential loss. In addition, Step 3 of the co-design methodology performs an necessary communication rescheduling, so that all bus conflicts are solved and no data loss occurs.

C. Hierarchical Cluster Growth Placement Algorithm

IP cores have to be placed for accurately estimating bus lengths and speed. There are many fast placement methods for ASIC design [36]. However, the nature of placement for ASIC and SoC is quite different. In case of ASIC, most of the nodes are at gate/RTL level, thus are of a small and comparable size. In contrast, SoC are composed of IP macros of irregular sizes. Each connection between ASIC nodes uses a single wire, and the connectivity degree is relatively low. The degree of an SoC node is defined not only by the link to other nodes but also the bus width. Finally, most important, the handling of critical paths is very different. For ASIC, the critical paths in gate netlists originate the critical paths of the circuits of library cells, which timing driven placement algorithms [36] will transform into critical paths at the layout level. Because of resource sharing, critical paths in a task graph do not necessarily convert into critical paths at the SoC layout level. The same core might execute both critical and non-critical tasks. Hence, to mitigate the influence on system latency, the primary goal of placement was to place highly connected cores closely together. The secondary goal was minimizing the total area, and having an aspect ratio close to one.

We selected a fast constructive placement algorithm that could be used in combination with the slower simulated annealing. We modified the well-known cluster growth placement method [36] for IP core placement. The algorithm is illustrated in Figure 15. The hierarchical cluster growth placement (HCGP) algorithm starts with sorting in descending order, by their communication load, all links in a CG. The two cores associated with the link having the maximum communication load are first selected for placement. The same approach with traditional cluster growth is then used to place them such that the rectangular bounding area of the two macros is minimized. If the bounding area is equal for several positions, the Manhattan distance between the centers of the two macros is used to find the best position. The first cluster is thus formed, and its aspect ratio is calculated. The next iteration will place two associated cores, such that the aspect ratio is close to one, and total area is minimized. The iteration stops when all the cores are placed.

A difference between traditional cluster growth placement and HCGP is that HCGP guarantees to have the minimum Manhattan distance between cores with highest communication loads. For example, consider the CG in Figure 15(c). Traditional cluster growth will place Core 6 close to the cores in cluster \{1, 2, 3, 4, 5\} before considering placement of Core 7. The Manhattan distance between Core 6 and Core 7 may become larger because of blockings due to cores previously placed. In contrast, HCGP will consider placement of cores 6 and 7 before placing the cluster, thus shortening the interconnection distance between the two cores.

Buses are routed based on the IP core placement. The layout is described as an intersection graph [36]. To easy the analysis of RLC effects, inter-macro communication uses only routing channels along the macro borders. To calculate the bus length of a PBS, inter-core routing path and bus wiring on a macro are taken into account. The algorithm starts by finding

![Hierarchical cluster growth placement](image)
the length of the bus-wiring around the cores associated with
be the inter-core bus length of the
Fig. 16.
two highly communicating cores close together. Hence, the
lengths have to be smaller than the CSF constraints found

The shortest path between two cores is defined as the
shortest path between any pair of intersection nodes around
the border of two clusters. An intersection graph is illustrated
in Figure 16(a). Cluster 1 contains core 1, with the interconnect
set \{a,b,c,d,e\}, and cluster 2 contains cores 8 and 10 with
the interconnect set \{f,g,h,i,j,k,l,m,n,o,p\}. The shortest path
between clusters 1 and 2 is the shortest path from any two
nodes of the two interconnect sets. If an intersection point
of a core is chosen, all the wires which connect to every pin have
to route around the core from the pin to a chosen intersection
point. The bus length in Figure 16(b) is approximately half
the length of the bus-wiring of the link with the lowest
communication load. This strategy is motivated by HCGP always placing
two highly communicating cores close together. Hence, the
longest path between two cores is at the link with the lowest
communication load. This link influences the total bus length
a lot, and we give it the highest priority.

The shortest path between two cores is defined as the
shortest path between any pair of intersection nodes around
the border of two clusters. An intersection graph is illustrated
in Figure 16(a). Cluster 1 contains core 1, with the interconnect
set \{a,b,c,d,e\}, and cluster 2 contains cores 8 and 10 with
the interconnect set \{f,g,h,i,j,k,l,m,n,o,p\}. The shortest path
between clusters 1 and 2 is the shortest path from any two
nodes of the two interconnect sets. If an intersection point
of a core is chosen, all the wires which connect to every pin have
to route around the core from the pin to a chosen intersection
point. The bus length in Figure 16(b) is approximately half
the core perimeter. Therefore, if there is also another
intersection point at point B, which is the longest distance form
point A, all wires have to route against the original direction.
That is the total bus length is approximately a perimeter. This
is the worst case, however, it is a good approximation if a
macro has several intersection points required for inter-core
routing. Let \(l_{PBSi}\) be the bus length of the \(i^{th}\) PBS, \(l_{inter}\)
be the inter-core bus length of the \(i^{th}\) PBS, and \(l_{macro}\) be
the length of the bus-routing around the cores associated with

\(i^{th}\) PBS, then the PBS bus length is defined as \(l_{PBSi} = l_{inter} + l_{macro}\). Finally, the bus delays for the estimated

lengths have to be smaller than the CSF constraints found
by the first step of the co-design flow (Figure 10).

VI. EXPERIMENTAL RESULTS
A set of experiments was defined to study the effectiveness
of the proposed hardware/software co-design 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 synthe-
sizing 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
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 [20]. Examples Graph 2 and Graph
3 are the motivational examples in [15]. 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 ini-
tially related to that of list scheduling. Results were compared
with solutions obtained by the method suggested in [20] [15],
one of the few scheduling approaches for graphs with data and
control dependencies. Column 4 indicates the schedule laten-
cies 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 [15]. 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 non-preemptive
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,
10 and 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, 13 and 14 present the same elements for com-
Table I
Solution quality for proposed co-design algorithms

<table>
<thead>
<tr>
<th>Example</th>
<th># of nodes</th>
<th>cond. dep.</th>
<th>List scheduling</th>
<th>Exploration</th>
<th>Rel. impr. (%)</th>
<th>Partitioning and list scheduling</th>
<th>Combined partitioning and scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Latency</td>
<td>Time</td>
<td>Latency</td>
<td>Time</td>
<td>Latency</td>
</tr>
<tr>
<td>Parallel</td>
<td>27</td>
<td>3</td>
<td>120</td>
<td>0.004</td>
<td>120</td>
<td>123</td>
<td>0</td>
</tr>
<tr>
<td>Tree</td>
<td>39</td>
<td>4</td>
<td>238</td>
<td>0.007</td>
<td>238</td>
<td>207</td>
<td>0</td>
</tr>
<tr>
<td>Fork-join</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>0.004</td>
<td>85</td>
<td>219</td>
<td>0</td>
</tr>
<tr>
<td>Laplace</td>
<td>31</td>
<td>1</td>
<td>62</td>
<td>0.003</td>
<td>56</td>
<td>140</td>
<td>0</td>
</tr>
<tr>
<td>Graph 1</td>
<td>34</td>
<td>3</td>
<td>40</td>
<td>0.005</td>
<td>39</td>
<td>605</td>
<td>0</td>
</tr>
<tr>
<td>Graph 2</td>
<td>15</td>
<td>2</td>
<td>97</td>
<td>0.002</td>
<td>97</td>
<td>69</td>
<td>0</td>
</tr>
<tr>
<td>Graph 3</td>
<td>23</td>
<td>2</td>
<td>60</td>
<td>0.004</td>
<td>60</td>
<td>340</td>
<td>0</td>
</tr>
</tbody>
</table>

Experiment 2: Algorithm Scaling
This set of experiments observed the capability of the co-design 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 [44] for wireless sensor networks and JPEG algorithm.

Face detection system Task Graph
RTL description for tasks 7, 8, 9, 10, 11, 12, 13, 15

JPEG Task Graph
RTL description for tasks 3, 8 and 13

Figure 17(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 more operations than the rest of the tasks, were decomposed into smaller tasks, as shown in Figure 17(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
Experimental results for Face Detection System

Example | Number of nodes | Resources | Latency | Iteration | Time (min) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>3 GPP</td>
<td>1,114</td>
<td>356</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>2 GPP/1 ALU/1 MU</td>
<td>1,141</td>
<td>519</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>59</td>
<td>2 GPP/2 ALU/2 MU</td>
<td>860,625</td>
<td>4,063</td>
<td>158</td>
</tr>
<tr>
<td>4</td>
<td>135</td>
<td>2 GPP/4 ALU/4 MU</td>
<td>671,761</td>
<td>10,070</td>
<td>4,500</td>
</tr>
<tr>
<td>5</td>
<td>239</td>
<td>2 GPP/8 ALU/8 MU</td>
<td>400,369</td>
<td>32</td>
<td>2,693</td>
</tr>
</tbody>
</table>

Figure 17(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 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 time. The proposed SA and PM based co-design method has a reasonably high computational complexity. Execution time was less than one hour for most of the cases.

Figure 17. HDCG for Face Detection System and JPEG algorithm
Experiment 3: Bus Architecture Synthesis

The first example presents bus architecture (BA) synthesis results for a network processor [10]. The processor receives Internet packets, re-routes them, and sends them out. A packet arrives through an Ethernet media access controller interface core (EMAC), and is sent to the multi-channel memory access layer core (MCMAL), a specialized DMA controller. MCMAL stores the packet in a buffer, and then transmits the buffer descriptor to the processor. The processors calculates the new destination address for the packet. MCMAL and one of the EMAC will send the packet out. Figure 18 shows the core graph for the network processor. Node 1 corresponds to core for the PowerPC, 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 direct memory access (DMA) 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 (I²C) core, 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 (EBC) 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. 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 7 and 9 buses. For different weights, however, the bus structure involves different buses. Only three buses were heavily re-used 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 re-used. It is mostly point-to-point links that were re-used. 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 \( w_L, w_C, \) and \( w_R \) must have large values.

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. Figure 19 shows the synthesized bus topology for the design scenario in which bus length and complexity are very important (\( w_L = w_C = w_R = 1.0 \)), and communication

**Table III**

Experimental results for JPEG algorithm

**Table IV**

Results for bus architecture synthesis for the network processor
The hierarchical cluster growth algorithm of about 25% of the area of processor cores, and ASIC were used architecture is presented at the bottom of Figure 21.

The considered processing technology was 0.18 \( \mu \) TSMC. Microprocessor cores were of about 25% of the area of processor cores, and ASIC were about 30% of processor core area.

Fig. 19. Synthesized bus architecture for the network processor

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. Figure 17(b) shows the task graph. After combined hard-ware-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 shared memory. To improve the processor-ASIC communication speed, interleaved memory blocks were used. This resulted in the Core Graph shown in Figure 20. The considered processing technology was 0.18\( \mu \) TSMC. Microprocessor cores were of about 5 \times 5 \( \text{mm}^2 \), memory cores of about 25% of the area of processor cores, and ASIC were about 30% of processor core area.

Figure 21 shows bus architecture synthesis results for the top CG in Figure 20. The hierarchical cluster growth algorithm generated the core placement shown at the top, and the bus architecture is presented at the bottom of Figure 21. 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 \((w_c = 1.0)\), 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 is 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 BA 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 minutes 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. Future Work

There are several directions, which could extend the presented work:

- **Additional performance metrics:** The co-design method could be extended to address new performance attributes, like power consumption. In [16], 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 be also tackled.

- **New design tasks:** 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. The challenge is to identify an algorithm that can explore the increased solution space.

- **Different placement methods:** New placement techniques can be studied to address more aggressively SoC area minimization. As shown in Figure 21, HCGP might result in some wasted area because area minimization is not its primary target. Partitioning, scheduling, and bus architecture will remain unchanged for the new technique.

- **Improving the exploration algorithm:** 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. However, if new performance criteria or additional design activities were to be included then SA could be enhanced by incorporating pruning techniques. We feel that PMs can be employed to avoid
Fig. 21. Bus architecture for JPEG SoC

poor designs, similar to using PMs to eliminate infeasible CSF values. Also, the co-design method requires a number of parameters, such as the weights for the cost function and the partitioning/scheduling probabilities. In our experiments, these values were manually selected, thus required several trials until the best parameter set was identified. To aid their re-use, these values can be stored into a database together with the attributes of the corresponding application.

VIII. CONCLUSION

This paper presents a layout conscious approach for hardware-software co-design 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 co-design 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 re-using among bus architectures optimized for different constraints. Experiments also indicated that PM are more effective than existing metrics, like task priorities. Using PM, system latency was by 20% shorter than for list scheduling. PMs are able to avoid the modeling limitations of priority functions. PM 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.

ACKNOWLEDGMENT

The authors would like to thank the Associate Editor and the reviewers for their very valuable comments and suggestions. This work was partially supported by IBM Faculty Partnership Award and DAC Graduate Scholarship Award.

REFERENCES


Nattawut Thepayasuwon (M’01) received B.Eng. from Khon Kaen University in 1993, MS.EE. from Rochester Institute of Technology in 1996. He is currently a Ph.D. candidate at the Department of Electrical and Computer Engineering, State University of New York (SUNY) at Stony Brook, NY. His research interest are in the area of hardware-software codesign and system-level synthesis. He is a member ofEta 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 a Ph.D. degree in Computer Engineering from University of Cincinnati, Cincinnati, OH in 2000. He is currently an Assistant Professor at the Department of Electrical and Computer Engineering, State University of New York (SUNY) at Stony Brook, Stony Brook, NY. His research is in VLSI system design automation, with special interest in mixed-signal CAD and hardware-software codesign. Dr. Doboli is a member of Sigma Xi.