Hardware-Software Co-Design of Resource Constrained Systems on Chip in a Deep Submicron Technology

Nattawut Thepayasuwan and Alex Doboli
Department of Electrical and Computer Engineering
State University of New York at Stony Brook
Stony Brook, NY, 11794-2350
Email: {nattawut, adoboli}@ece.sunysb.edu

June 18, 2003

Abstract

This paper presents a hardware-software co-design methodology for resource constrained SoC fabricated in a deep submicron process. The novelty of the methodology consists in contemplating critical hardware and layout aspects during system level design for latency optimization. The effect of interconnect parasitic and delays is considered for characterizing bus speed and data communication times. The methodology permits coarse and medium grained resource sharing across tasks for execution speed-up through superior usage of hardware. The hardware-software co-design methodology executes three consecutive steps: (1) It performs combined task partitioning to processor cores, operation binding to functional unit cores, and task and communication scheduling. It also identifies minimum speed constraints for each data communication. (2) The bus architecture is synthesized, and buses are routed. IP cores are placed using a hierarchical cluster growth algorithm. Bus architecture synthesis identifies a set of possible building blocks (using the proposed PBS bitwise generation algorithm), and then assembles them together using simulated annealing algorithm. For early elimination of poor solutions, the paper suggests a special table structure and select-eliminated method. Each bus architecture is routed, and after parasitic extraction, bus speeds are characterized. (3) For the best bus architecture, the methodology re-schedules tasks, operations, and communications to minimize system latency. At this step, bus speed accounts for layout parasitic. The paper offers extensive experiments for the proposed co-design methodology, including a network processor and a JPEG SoC.

1 Introduction

There is a growing demand for embedded systems targeting multimedia applications [12] [20] [37]. The success of systems like cell phones, digital cameras, and personal communicators critically depends on meeting stringent cost, timing and energy consumption constraints. As compared to desktop computers, embedded system performance is at least one order of magnitude higher, while their cost is significantly lower [37]. Also, hardware resources are scarce for most embedded systems: 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 various I/O peripherals (including analog and mixed-signal circuits). Systems-on-Chip (SoC) are single-chip implementations of embedded systems. They offer higher performance and reliability at cheaper costs as compared to printed circuit board designs [12]. SoC include multiple IP cores connected through complex data, address and control buses. The variety of IP cores is large. For example, the H.263 encoder includes digital DSPs and microprocessor cores, memories, RF and mixed-signal blocks (analog-to-digital and digital-to-analog converters, filters, and PLL circuits) [29]. It is foreseen that the number of SoC cores will steadily increase

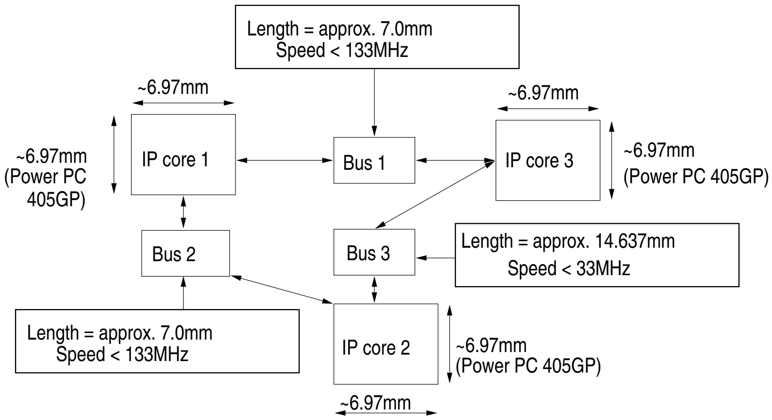


Figure 1: **Impact of layout on data communication speed**

over the next 4-7 years, while clock frequencies will range around 10-15 GHz [4]. Effectively designing SoC necessitates the development of new design automation tools at various levels of abstraction, including system, logic and layout level [9] [12] [18] [31].

SoC design must be short, require a reduced designer effort, and offer the confidence of getting correct implementations [12]. For very deep submicron designs (VDSM), physical level attributes such as interconnect parasitics, substrate coupling, and substrate noise significantly influence system performance, e.g. bus communication speed, system latency, power consumption, and signal integrity [7] [14] [44]. The usual paradigm of abstracting physical details during system level design is of limited use for SoC implemented in VDSM technologies. Research must focus on incorporating relevant physical design attributes into system level synthesis, including hardware/software co-design. This task is challenging, and requires new design approaches, in which the top-down co-design process is aware of lower level aspects, like core placement and bus routing. Besides, new synthesis algorithms are needed, such as for bus architecture design, as well as novel modeling methods, like describing interconnect length at the system level [41] [42] [48]. For resource constrained systems, maximizing the usage of hardware resources is important in meeting performance requirements, and avoiding system overdesign. Hardware abstraction at a high-level, as in many traditional co-design methods [21] [23] [26] [47], does not allow resource sharing across tasks. Each task mapped to hardware posses its dedicated set of modules. However, hardware sharing across tasks introduces a new optimization dimension, and offers the potential of improving performance without using more hardware. In [37], the authors explain that customizing the medium grained application specific cores improves performance and cost of multimedia systems by important amounts. They indicate, however, that none of the existing CAD tools exploits this opportunity. This paper presents a novel hardware-software co-design methodology, in which task and communication partitioning and scheduling are performed under detailed observation of available hardware resources, and dependency between data communication speed and physical design (like core placement and bus routing).

The first motivating example illustrates the impact of layout design on data communication speed. Figure 1 shows a set of three Power PC 405GP cores [2], which exchange data as part of their functionality. The physical dimensions of the cores are presented in the figure. Without considering layout information, the co-design step decides to allocate a single system bus of speed 266MHz for all core communications. All timing constraints were met in this case. However, given the physical dimensions of the cores, it is difficult to implement a bus with the requested speed. The same latency performance can be obtained with three buses of lower speed, as shown in the figure. The bus speeds of 133MHz, 133MHz and 33 MHz were found based on the physical locations of cores, and the RLC effects of the routed buses [44]. This example arguments that the communication sub-system of an SoC needs to be designed while contemplating design feasibility criteria (e.g., achievable bus speeds). Task and communication scheduling must be

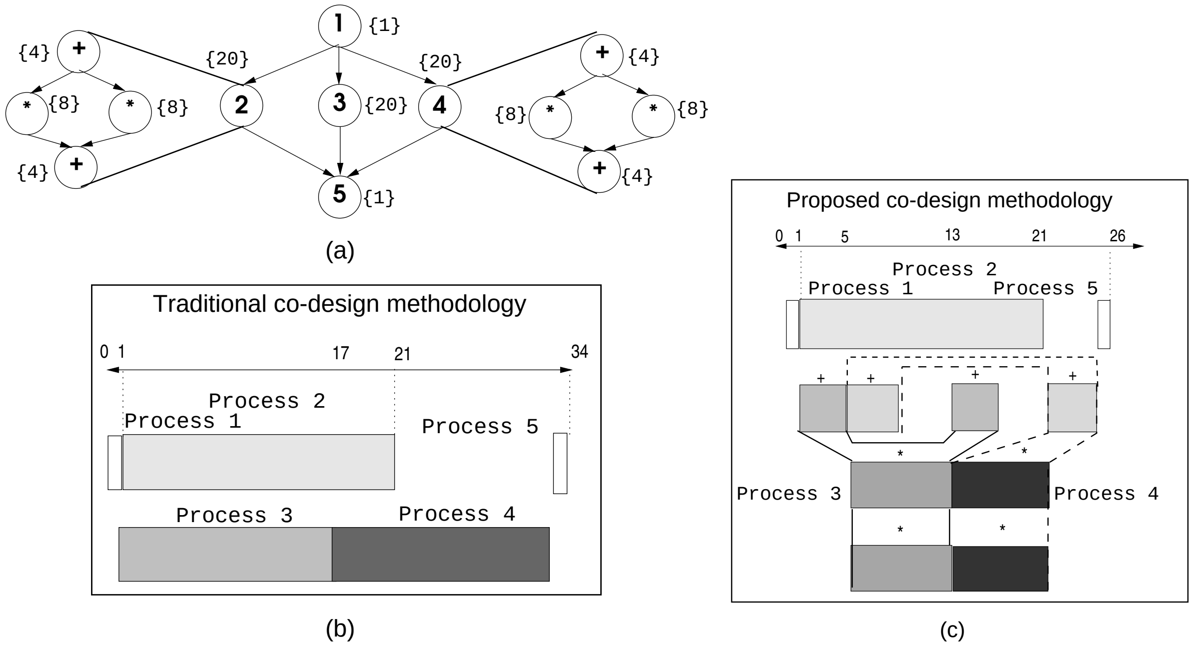


Figure 2: **Importance of hardware sharing across tasks for improving latency**

integrated with core placement, bus speed allocation, bus topology design, and routing to guarantee that identified bus speeds are possible for a real SoC.

The second motivating example explains the importance of hardware sharing across tasks. This improves the overall latency. Figure 2 presents an example, in which the detailed perspective on hardware offers a solution with shorter latency. Figure 2(a) presents a task graph with 4 tasks. Each task has associated the execution time on a general-purpose processor (GPP). Additionally, it is assumed that tasks 2 and 4 can be also realized as hardware ASIC. The figure presents the operation level structure of the two tasks, and the execution time for each operation. Considering that resource sharing is not possible between tasks in hardware, the resulting hardware/software design has a latency of 34 time units. The schedule is shown in Figure 2(b). In the second case, all possible hardware sharings are contemplated between the operations of Task 2 and Task 4 (additions and multiplications). The resulting design has a latency of 26 units. Figure 2(b) depicts the schedule. The second case offers a design with a higher operation concurrency, thus a shorter latency, and less GPP idle time.

This paper presents a hardware-software co-design methodology to address (a) the dependency of task communication speed and time on layout attributes like interconnect parasitic, as well as (b) the possibility of contemplating coarse grained (like processors and memories) and medium grained (i.e. multipliers, decoders etc) hardware resource sharing to improve system latency without increasing cost. The methodology incorporates an original algorithm for bus architecture synthesis. All hardware resources are assumed to be given in this methodology. The co-design process performs *combined* task and communication partitioning and scheduling, and finds *feasible* requirements for the minimum bus speed of each data communication link. The bus architecture of an SoC is found as part of the process. The paper offers extensive experiments, including bus architecture synthesis for a network processor using the CoreConnect bus architecture [3], and a JPEG SoC design. The co-design methodology improves the practicality of system-level design for VDSM technologies. It is also useful for implementing resource constrained SoC needed for multimedia applications.

The co-design methodology includes three main parts: (1) the step of combined partitioning and scheduling followed by (2) the step of bus architecture synthesis, and (3) re-scheduling of tasks, operations, and communications for the best found bus architecture. The first step is an exploration process based on simulated annealing algorithm (SA) [35]. We propose Performance Models (PM), a graph-based description to capture the relationships between performance (i.e. latency and communication speed flexibility), graph characteristics (like data and control dependencies), and design decisions (such as binding and scheduling). PM are general, flexible, and can be easily extended for new design activities without requiring cumbersome validation. Experiments showed that (in conjunction with SA) PM are more efficient than well known synthesis metrics, like task priorities, forces, degree of concurrency, utilization rate, idle times [15] [23]. The paper presents rules for automatically generating and managing PM. The first step ends with the creation of a Core Graph structure (CG) that expresses the data volume and timing for each communications link. The second step uses CG to synthesize and route bus architectures for an SoC. IP cores are placed using a hierarchical cluster growth algorithm, which places highly communicating cores close to each other. Bus architecture synthesis first identifies a set of possible building blocks (using the PBS bitwise generation algorithm), and then assembles them together using SA. The cost function models bus length, bus topology complexity, communication conflicts over time, and amount of unnecessary core connectivity (which needlessly decrease bus speed). We propose a special table structure (named bus architecture synthesis table) and select-eliminate method to prune solutions dominated by other (for example, buses with complex and redundant connectivity). This reduces the exponential number of possible bus architectures.

The paper is organized as follows. Section 2 presents related work on hardware/software co-design and bus architecture synthesis. Section 3 discusses the proposed system representation. Section 4 introduces the co-design approach. Bus architecture synthesis is presented next. Experimental results are given in Section 6. Finally, conclusions are offered.

2 Related Work

Over the last ten years or so, a variety of hardware/software co-design methodologies were proposed for optimizing cost, speed, and power consumption [6] [21] [23]. A typical co-design flow includes 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 [6] [11] [26] [30], for control intensive systems [5], and for applications with substantial data processing and reduced amount of control [22] [40]. Balarin *et al* [5] present the POLIS approach for control dominated real-time embedded applications. For data dominated systems, Prakash and Parker [34] and Bender [8] formulate the co-design problem as a mixed-integer linear programming (MILP) problem. A linear equation solver produces the optimal implementation. The disadvantage of MILP-based co-design is the very large time required to solve the model. Therefore, these methods are limited to small size applications. The alternative is to employ heuristic algorithms for co-design, such as greedy priority-driven clustering [11], list scheduling methods [6] [17] [22] [30], iterative improvement heuristics, like simulated annealing and tabu search [22], and genetic algorithms [10] [16] [40]. The principal advantage is that heuristic methods can be used for large task graphs [22]. The disadvantage is that the optimality of the solution is difficult to characterize. For example, greedy priority-driven algorithms offer good average results, but they might give poor solutions for situations, which are not captured by the priority function [17]. Recent hardware-software co-synthesis work tackles minimization of the energy consumption. This is important for mobile and wireless systems. Henkel [27] suggests a hardware-software partitioning method for low-power systems. Partitioning is at the level of operation (instruction) clusters. After cluster scheduling, clusters with a high utilization rate (thus, with less wasted energy) are moved to hardware. Dave, Lakshminarayana and Jha [11], and Dick and Jha [16] 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 level allocation, scheduling and performance estimation while contemplating interprocessor concurrency, preemptive and non-preemptive scheduling, task duplication on processors, and memory constraints. An initial clustering step uses critical-path analysis to identify the tasks that will share a processor. Then, clusters are allocated and scheduled using a priority function. Givargis and Vahid [25] describe Platune environment for tuning parameterized uni-processor SoC architectures to optimize timing and power consumption. The emphasis is set on high-level simulation of timing and power consumption. Architectural parameters like processor speed, cache organization, and certain periphery attributes are decided using a method based on the Paretto optimality criterion. Yang *et al* [46] present an energy aware scheduling method for multiprocessor SoC using genetic algorithms. Sgroi *et al* [43] suggest a communication centric approach motivated by the increasing importance of communication attributes. Communication is layered similar to the OSI Reference Model. Adapters increase reusability of components by matching different protocols.

Bus design is critical for SoC. Early work on bus and communication synthesis [13] [24] [33] [47] focuses on multiprocessor embedded systems on a printed board. Research addresses interface design [13] [33], communication packeting [22], mapping and scheduling [47]. This work does not tackle the hardware and layout aspects of SoC communication sub-systems. Lahiri *et al* [32] focus on communication protocol selection for a communication architecture template including shared and dedicated buses. Recently, Drinic *et al* [19] 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. Seceleanu *et al* [38] discuss segmented bus architectures for improving speed and power consumption. Bus segments are dynamically linked through inter-segment bridges (glue logic + tri state buffers) under the control of a Central Arbiter (CA). Hu *et al* [29] introduce point-to-point (P2P) 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 [24] [29] [32]. This is reasonable for small SoC (and for which the designer manually designs the buses). However, it is not effective for SoC with large number of cores.

This paper proposes a new hardware-software co-design approach that integrates system design with bus architecture synthesis and routing, and contemplates hardware sharing more aggressively. Original algorithms are suggested for partitioning and scheduling, so that relevant layout attributes are addressed at the system-level. This work presents new bus architecture synthesis methods, which do not require the bus topology to be given, and are more sensitive to layout parasitic.

3 System Representation for Co-Design

An embedded system is expressed for co-design as the quadruple $< HDCG, Resources, Floorplan, PM >$. $HDCG$ describes system functionality as a hierarchical data and control graph. $Resources$ is the set of IP cores used in the implementation. $Floorplan$ is the set of all possible floorplans for the IP cores in set $Resources$. PM is a graph-based representation that denotes performance attributes, like latency and communication speed flexibility.

A *Hierarchical Data and Control Dependency Graph* (HDCG) offers a dual perspective on system functionality: a task-level description (for co-design) and an operation-level representation (for exploring hardware sharing across tasks). HDCG are based on well known system descriptions, like control data flow graphs [23] and conditional process graphs [22]. Figure 3 presents an HDCG example. HDCG nodes are of three types:

- *Cluster nodes* (CN) represent loops, if-then-else constructs, functions, and tasks. CN are mapped to the software domain, and are executed on coarse grained cores, like general purpose processors (GPP). Each CN is a polar sub-graph built from operation nodes. Figure 3(a) shows the detailed operation structure of CN 3.
- *Operation nodes* (ON) denote an atomic data processing such as addition, multiplication, etc. These are mapped to small/medium grained IP cores, like multipliers and arithmetic and logic units (ALU). During co-synthesis, ONs are employed for analyzing the effect of hardware resource sharing across tasks.

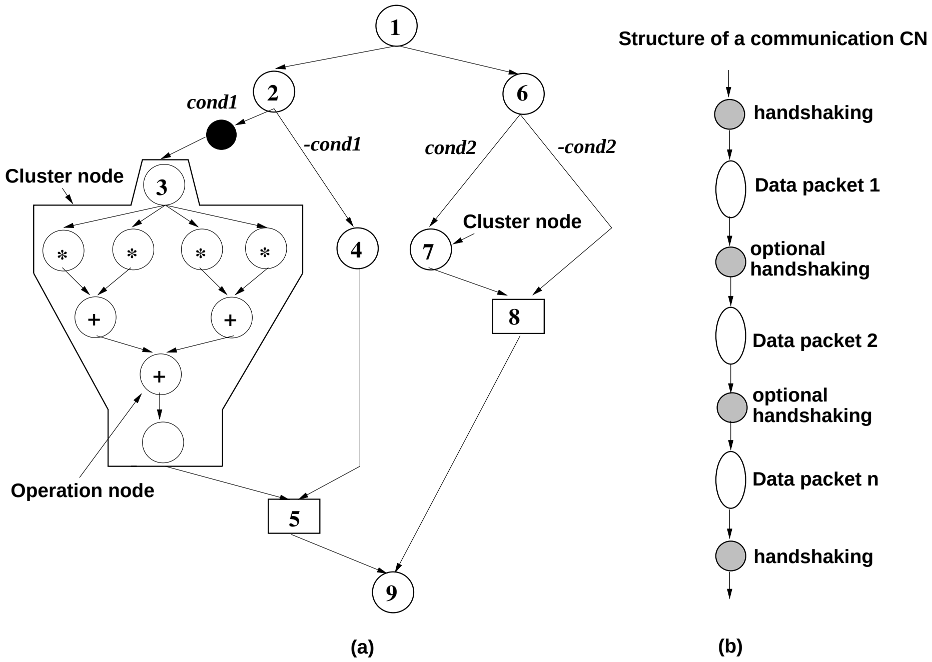


Figure 3: **Hierarchical Data and Control Dependency Graph**

- *Communication cluster nodes* (CCN): Data communications are modeled using CN with a special structure (see Figure 3(b)). CCNs are shown as black bubbles in Figure 3(a). A CCN includes an alternating sequence of ON nodes representing transmissions of data packets of a fixed size, and synchronization nodes. 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. Optional synchronization points represent the time overhead for synchronizing the cores. If successive packets pertain to the same communication link, then the optional synchronization points have zero time length.

CN, CCN, and ON are linked through data and control dependencies. Data dependencies, shown as directed arcs, impose a certain execution order: the target node starts its execution only after the source node was completed. Similar to conditional process graphs [22], some arcs are annotated with condition values for expressing control dependencies. In Figure 3(a), conditions are depicted in italics. Node 2 computes condition *cond1*, and depending on its value, *one* of its out-branches is selected. For a *true* value, the graph execution activates the communication between nodes 2 and 3, followed by starting the operations pertaining to node 3. Node 4 is executed for a *false* condition value (*false* is indicated by - *cond1* in the figure). Nodes 5 and 8 are fictitious *join nodes* for outlining the control structure of a HDCG. CN and ON are annotated with their execution times for all hardware resources that could implement the node. Handling of CCN execution times is presented at the end of this section. The execution semantics of the HDCG model assumes that each CN, ON, and CCN is executed at most once for each traversal of the graph.

Even though system functionality can be expressed at the lowest abstraction level using ON and CCN only, CN prevent the unnecessary growth of the design solution space, and hence, a very cumbersome co-design process. There is no hardware resource sharing between tasks executed in software, thus there is no need for a detailed, operation level description in this case. The execution time of CN in software can be accurately estimated using data profiling and performance models for CPU, cache, memory, and communication units [25]. Compiler optimizations, like loop unrolling and tilling, can be also considered using CNs. This is more difficult for a description using ONs.

Performance Model (PM) representation is a graph-based description that relates system performance attributes to

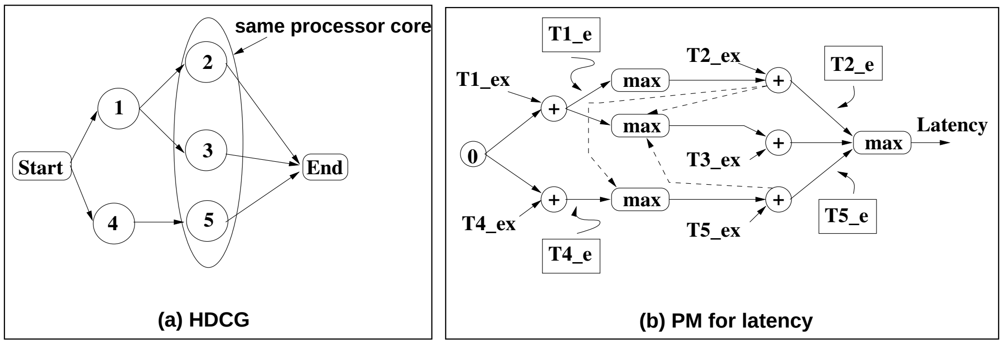


Figure 4: **Performance Model for latency**

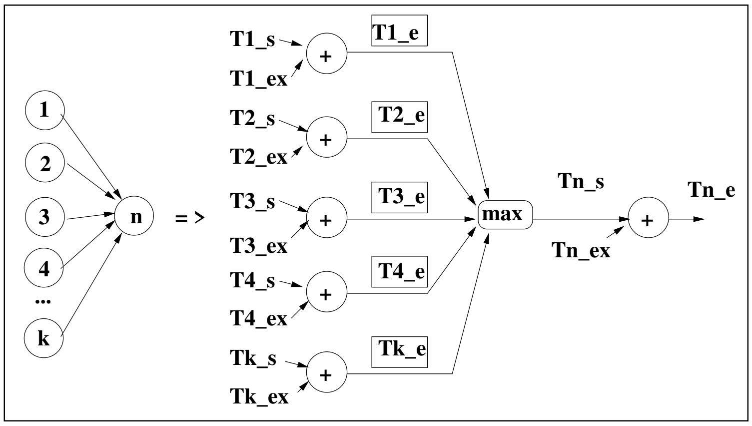
HDCG characteristics, and to the design decisions executed during co-design. Figure 4(a) shows an HDCG, and Figure 4(b) depicts the corresponding PM for latency, assuming that ON 2, ON 5, and ON 3 are executed in this order on the shared resource. PM include following elements:

- Starting node *0* indicates that all observed performance attributes (latency in our case) are set to value 0.
- The *constant part* describes the semantics of performance attributes with respect to the invariant HDCG characteristics (like data and control dependencies). It is represented in the figure as nodes and solid edges. *max* and *addition* nodes are used in Figure 4(b) to express ON (CN) start and end times. *Max* nodes describe that an ON (CN) can not start earlier than the moment when all its predecessors are finished (thus, the starting time has to be larger than the maximum of the end times). Outputs of *max* nodes indicate the starting time of their corresponding ON (CN). Addition nodes describe that the end time Ti_e of ON op_i is the sum between its start time and its execution time Ti_ex.
- The *variable part* presents the relationship between performance attributes and design decisions taken during co-design, like partitioning and scheduling. For instance, the execution order ON 2, ON 5, ON 3 is represented in the figure as dashed arcs between the addition nodes that calculate the end times for ON (CN), and *max* nodes characterizing the starting times. Other ON scheduling orders are easily captured in the PM by accordingly changing the orientation of the corresponding arcs.
- Performance attribute values (like latency in the figure) for a certain co-design solution results by numerically evaluating its PM.

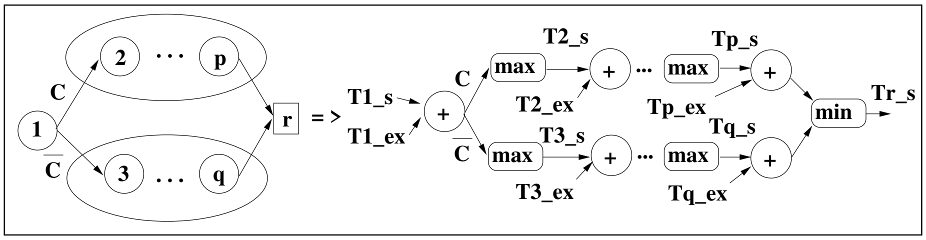
PM are the principal data representation for the exploration loop of the hardware/software co-design process. PM is a general description, which can denote various design activities, like scheduling, partitioning, and pipelining. Design activities can be expressed as individual tasks, or as an integrated flow. The co-design process presented in this paper uses PM for combined scheduling and partitioning based on a simulated annealing algorithm. PM are flexible, as they allow easy definition of new performance attributes, or description of additional relationships between performance attributes and co-design decisions. For example, the communication speed flexibility (CSF) metric was added without affecting the already existing PM rules for latency. Finally, rules for PM handling can be set-up to avoid exploration of infeasible or dominated solution points. For example, the rules for CSF calculation avoid generating co-design solutions, which are difficult to realize. This helps the exploration process, as it eliminates additional steps of verifying the feasibility of a design. The remaining part of this section presents the rules for automatically building PM.

Modeling of Data and Control Dependencies

Data dependencies introduce a required sequencing of HDCG node executions. For example, in Figure 5(a) node n can be executed only after all its predecessors 1, 2, ..., k are performed. The right part of Figure 5(a) depicts the PM



(a) Modeling of data dependencies



(b) Modeling of control dependencies

Figure 5: **Modeling of data and control dependencies**

that expresses these execution order requirements. For each operation, addition nodes indicate how node end times $Ti\_e, (i = 1, k)$ are computed depending on node starting times Ti_s and node execution times Ti_ex. The *max* node models the constraint that node n starts only after all its predecessors are terminated.

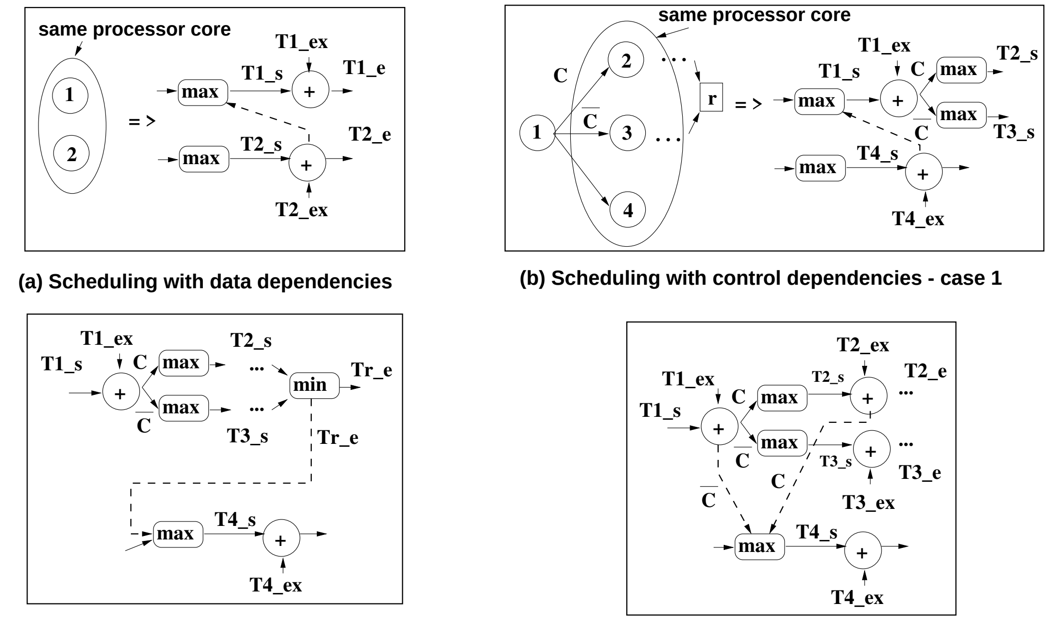
Control dependencies describe conditional node executions in an HDCG. In Figure 5(b), for example, if the value of condition C is true then nodes 2, ..., p are performed. Nodes 3, ..., q are executed for a false condition $\overline{C}$. Join node r is introduced in the HDCG to point the end of the conditional construct. The right part of Figure 5(b) introduces the PM generated for the HDCG. Similar to data dependencies, control dependencies define an execution ordering among nodes. For example, nodes 2, 3, etc can not start their executions until node 1 is completed. Analogous to data dependencies, the built PM reflects this requirement through *max* nodes.

Conditional execution of nodes is presented in the PM by annotating edges with condition values. The following semantics is applied when numerically evaluating this PM: edges annotated with a true condition will propagate the numerical values that result from the PM evaluation (and corresponds to the sink node of the edge). Edges annotated with a false condition will propagate the value ∞. The *min* node in a PM corresponds to node r in the HDCG. It eliminates the infinite values propagated on the non-selected branches. This permits calculating the correct time values for nodes that succeed node *r*.

Modeling of Cluster Partitioning and Operation Binding

The set $Resources$ of available IP cores (including GPP cores, FU cores and buses) is given in the presented co-design methodology. Thus, the number and type of hardware resources that can realize each CN, CCN, and ON in the HDCG is known. $\mathcal{R}\_op_i$ is the subset of set $Resources$ to which node op_i can be mapped. Also, we define the function

$$\mathcal{R}esource : \mathcal{N}odes\ in\ HDCG \rightarrow Resources$$



(a) Scheduling with data dependencies

(b) Scheduling with control dependencies - case 1

(c) Scheduling with control dependencies - case 2

(d) Scheduling with control dependencies - case 3

Figure 6: **Representation of scheduling decisions under data and control dependencies**

that presents the resource to which each node is bound. The goal of cluster partitioning and operation binding is finding the definition of function $\mathcal{R}esource$ (thus, its values for all nodes op_i). Obviously, binding has to be done such that $\mathcal{R}esource(op_i) \in \mathcal{R}\_op_i$. A consequence of cluster partitioning and operation binding is that any attribute value (in our case, execution time) of node op_i that depends on the resource type becomes well defined. For example, lets assume that node execution time Ti_ex varies with the resource type. After binding, the execution time becomes known, and its value is updated in the PM.

Modeling of Scheduling

Given a HDCG and a node binding to hardware resources, scheduling decides the execution order of nodes on the shared resources. Depending on the scheduling decisions, different node sequences and different timing attributes (such as starting time and end time) result for the nodes. These correlations between scheduling and performance metrics have to be captured by a PM. If only data dependencies occur then the imposed ordering can be modeled by introducing an arc from the PM addition node (for the end time) of the HDCG node to be executed first to the PM max node (for the starting time) of the HDCG node to be executed second. For example, in Figure 6(a) Node 1 and Node 2 share the same resource. The scheduler decides to execute Node 2 before Node 1. 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 of the PM. Different scheduling decisions can be captured in a PM by simply changing the orientation of arcs.

Scheduling in the presence of control dependencies is more difficult due to the uncertain character of control dependencies [22]. The difficulty increases if scheduling is performed across control constructs. The main challenge is to generate node schedules that maintain the HDCG semantics with respect to three criteria: (1) respecting the execution order defined by data and control dependencies, (2) maintaining the conditional node execution as defined by a HDCG (i.e. if a condition value is true then only nodes from the true branch must be executed), and (3) executing at most once each CN in the HDCG. If the three criteria are met then tasks can be correctly scheduled across control constructs. The first two requirements are already captured by the PM modeling of data and control dependencies. To exemplify the third requirement, lets assume a schedule for the graph in Figure 6(b), so that Node 4 is 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

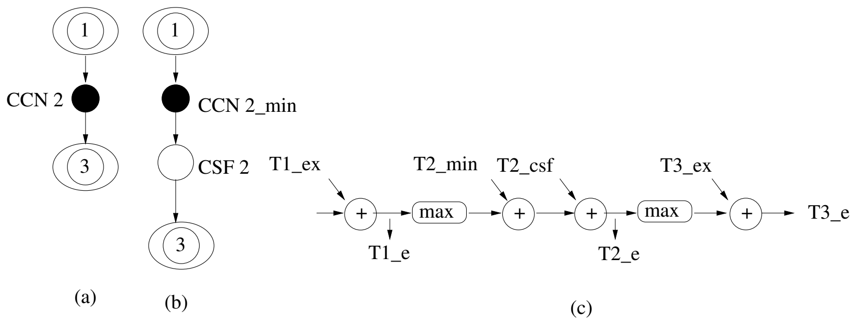


Figure 7: **Communication speed flexibility**

occurs if for condition C true the branch is long, and the branch for condition C false is short compared to the path, which Node 4 pertains to. This schedule is incorrect because if condition C is false then Node 4 is executed twice (before and after Node 1).

For modeling the third scheduling correctness requirement, three different cases are possible in a PM:

- **Case 1**: Node 4 is executed before Node 1. Its scheduling does not depend on the value of condition C. Thus, there is no risk that Node 4 is executed multiple times for a single execution of the HDCG. Figure 6(b) depicts this situation, and a dashed arc enforces that Node 1 starts only after Node 4 terminates.
- **Case 2**: Node 4 executes after Node r. Its scheduling time depends on the value of condition C. However, the value of condition C is already known by the time Node 4 starts. Thus, there is no danger of executing Node 4 multiple times. This situation is reflected in Figure 6(c) by the dashed arc between the *min* node for Node *r* and the *max* node for Node 4.
- **Case 3**: For a given condition value (i.e. condition C is true), Node 4 is scheduled to execute after Node 1 but before Node r starts. 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 6(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.

Modeling of Communication Speed Flexibility

The speed 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 and the bus architecture and routing. Defining the bus architecture includes finding the number of different buses present in a design, and the set of cores sharing a bus. This information is not available during task partitioning and scheduling (see Figure 10). Section 5 details the suggested bus architecture synthesis algorithm. The co-design methodology in Figure 10 identifies communication speed requirements for each data link, while relying on a system-level modeling of bus architecture synthesis. Communication speed requirements are feasible, if needed bus speeds can be achieved in the presence of delays caused by the RLC parasitic of bus routings.

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. CSF values are found as part of the co-design methodology, and become constraints for the bus architecture synthesis step discussed in Section 5. Figure 7 shows the PM modeling for finding the CSF values. CN 1 and CN3 in Figure 7(a) are allocated to different processing cores, and connected through CCN 2. The execution time of CCN 2 is unknown, as the bus architecture is synthesized in a subsequent step. Figure 7(b) shows the HDCG description for finding communication speed requirements. For each CCN, two

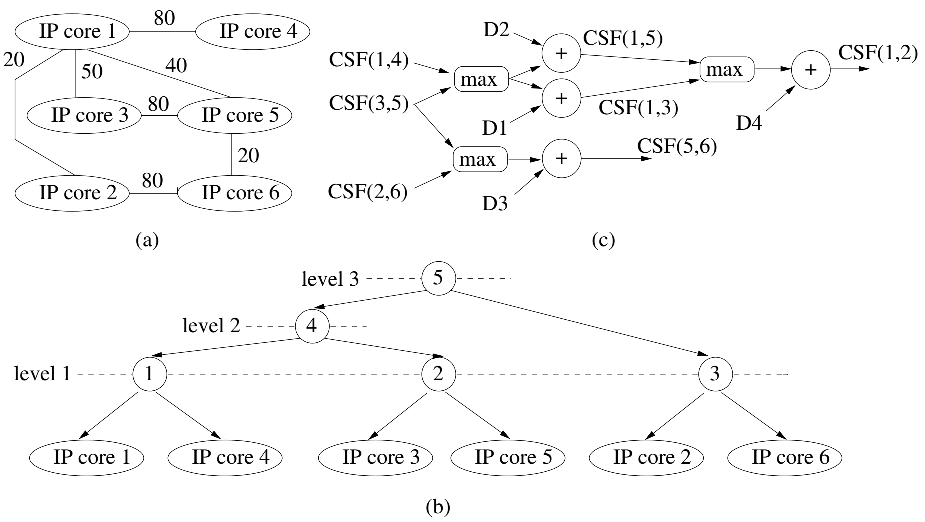


Figure 8: **PM modeling of communication speed flexibility**

nodes are introduced: node CCN_min describes the minimum latency for data communications. Minimum latency depends on the amount of communicated data, and the maximum speed achievable for a given fabrication process and minimum interconnect length. This describes the lower bound for the communication time between two tasks. Node CSF models the unknown communication speed flexibility, for which a numerical value is found during the co-design process. Figure 7(c) presents the PM including CSF. PM was build using the rule for modeling data dependencies.

The feasibility of a set of CSF values depends on the capability of the bus synthesis algorithm to meet the constraints formulated by the CSF set. Section 5 explains that the quality of a bus architecture depends on (1) the speed of individual buses, (2) the amount of time overlapping between communications mapped to the same link, (3) the complexity of the bus architecture expressed as the number of buses in the architecture, and (4) the amount of core connections not required in a bus architecture.

At the system-level, criteria 2-4 are modeled by the likelihood of two communication channels sharing the same bus. Two communication links are likely to share a bus, if following conditions are met: (i) same bus speed requirements (expressed at the system level through the corresponding CSF values) are acceptable for both links, (ii) there are no (very few) time overlapping between the communication schedulings of the links, (iii) there is little amount of additional unnecessary core connectivity, if the two links share the same bus, and (iv) the estimated bus length does not conflict with the required CSF values. Criteria (ii)-(iv) can be estimated at the system level, when mapping two CCN to the same bus. We present next the system-level modeling of bus speed to address criteria (1) and (i).

Floorplan Trees (FT), a tree structure, is used to model the IP core floorplanning at the system-level. Figure 8(a) presents a set of six IP cores and the data communication between them. This representation is called *Core Graph*, and Section 5 offers more details on it. *Communication Load* (CL) expresses the amount of data exchanged between cores, and labels each edge in the graph. To favor tightly interacting IP cores, the placement algorithm places close to each other those IP cores, which exchange large amount of data. The hierarchical cluster growth placement algorithm (HCGP), described in Section 5, proceeds in a bottom-up fashion, and creates clusters of cores depending on the CL of edges between cores. Figure 8(b) shows the FT modeling. Cores 1 and 4, cores 3 and 5, and cores 2 and 6 are heavily communicating. Nodes 1, 2, and 3 represent their clustering. Resulting clusters are interconnected by edges describing the amount of data communications between all cores in the cluster. The clustering process continues by considering nodes 1, 2, and 3, and so on, until the root node is reached (node 5 in the figure).

```
input: BT - Floorplan tree
output: CSF PM
for all leaf nodes i in BT do
  generate a PM input node, and label it as CSFi;
end for
for all levels j in BT, starting from level 1 do
  for all nodes p in BT on level j do
    identify all communications (m,n), such that
    node p is the first parent in BT for
    cores m and n;
    create max and addition nodes and variable
    D_mn and label the output as CSF(m,n);
    for all existing CSF(l,k), k!=n or  l!=m do
      insert an edge from CSF(l,k) to the max
      node for CSF(m,n);
    end for
  end for
end for
```

Figure 9: **Algorithm for building CSF Performance Models**

Lemma: Let (i, j) and (m, n) be the CG edges for data communications between core i and core j, and core m and core n. 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 $(i, j)\, ge$ the speed of the bus for communication (m, n).

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

The bus speed values searched during co-design must accommodate the bus speed constraints imposed by the IP core placement. Otherwise, bus speed requirements are *unreasonable* (though some of them might be possible to implement). For example, it is unreasonable to request a high communication speed for cores placed far apart. Hence, CFS values fixed for CCNs must meet the constrained expressed by the above lemma. A naive solution would assign random values to CFS, and then check if these values meet the constraints imposed by FT. In reality, this solution does not offer good results, as most of the analyzed CFS values would violate the constraints. Instead, PM for CFS implicitly incorporate all speed constraints due to the floorplanning model. For example, Figure 8(c) shows the corresponding CSF PM. CSF values for the leaf nodes (CSF(1,4), CSF(3,5), and CSF(2,6)) are input values to the PM. According to the floorplanning, the speed for communications (1,5) and (1,3) has to 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 $D1$ and $D2$ express the time amount by which the two communications are slower. Similarly, communication (5,6) must be slower than communications (2,6) and (3,5). Finally, communication (1,2) must be slower than communications (1,5) and (1,3). Figure 9 shows the algorithm for building CSF PM meeting the constraints fixed by FT.

4 Co-Design Methodology

Figure 10 presents the proposed hardware-software co-design methodology. Inputs are the HDCG of an application, latency constraints, and the set of available IP cores (including number and types of GPP, FU etc). The co-design flow partitions HDCG nodes to cores, decides the scheduling order of nodes, synthesizes the bus architecture, and maps and schedules data communications on buses. Goal is to minimize the overall system latency.

The method includes three steps. The first step partitions CN nodes to GPP cores, binds ON nodes to FU cores, schedules CN, CCN and ON, and finds minimum speed requirements for CCN. First, Performance Models (PM) are generated for an HDCG using the rules in Section 3. Next, the simulated annealing exploration loop simultaneously conducts

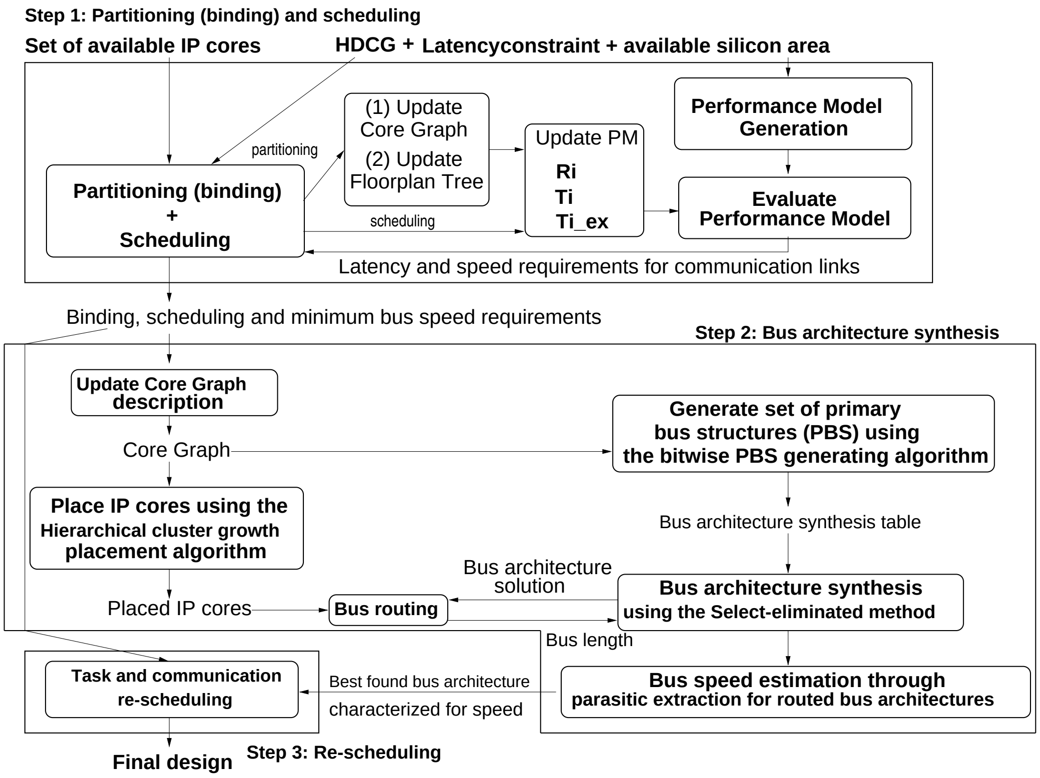


Figure 10: **Hardware-software co-design methodology**

partitioning and scheduling. For each CN (ON), attributes R_i (the hardware resource that executes the node), T_{i_ex} (the execution time on the resource), and T_i (the starting time of node execution) are unknowns for co-design. CN partitioning to GPPs and ON binding to FUs is modeled by unknowns R_i and T_{i_ex}. CN, CCN, and ON scheduling is described by unknowns T_i. Possible values for unknowns R_i and T_i are searched during exploration. Latency is computed by numerically instantiating all node characteristics R_i, T_i and T_{i_ex}, and then evaluating their PMs. Co-design optimization was realized using simulated annealing algorithm (SA) [35]. The algorithm examines the quality of numerous partitioning (binding), and scheduling solutions by numerically evaluating PMs for latency and communication speed flexibility (CSF).

SA iteratively selects a new point from the neighborhood of the current solution. 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. PM, Floorplan Tree (FT), and Core Graph (CG) are updated for each newly selected solution, and used for numerically evaluating the resulting performance. If the resulting solution has a better quality than the current solution then it is unconditionally accepted. A worse solution is accepted with a higher probability at the beginning of exploration. Acceptance probability decreases as exploration progresses. The starting solution is obtained by uniformly distributing nodes to resources, and then scheduling nodes using list-scheduling [15]. Critical path [15] was the priority function for list-scheduling. An initial FT and CG are found for the starting exploration point.

Partitioning (binding) and scheduling steps are executed with different probabilities. The reason is that multiple valid schedules are possible for each resource partitioning (binding) decision. A small probability p_1 is used to select a partitioning step that moves a cluster from a GPP core to another GPP core or to hardware. A probability p_2 ($p_2 > p_1$) binds an ON 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, a probability 1 - $(p_1 + p_2)$ decides a scheduling action. This strategy emulates a hierarchical exploration process because for each new partition (binding) there are $\frac{1-(p_1+p_2)}{p_1+p_2}$ analyzed schedules. For example, if p_1 = 0.01 and p_2 = 0.1 then on the average, 8 schedules are examined for each partition

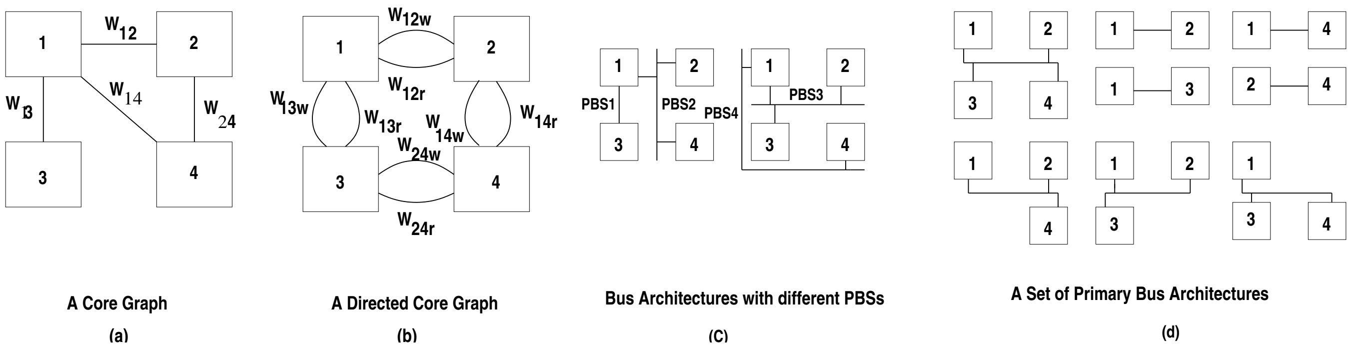


Figure 11: **Core Graph and PBS examples**

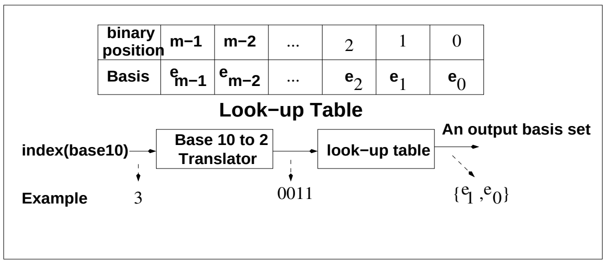
(binding). If the execution order of a node pair is modified then the algorithm also verifies that the new ordering is feasible. This means that no cycles can occur in the updated PM. The cost function for SA is

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

The cost function models system latency, communication speed flexibility, bus complexity, and unnecessary bus connectivity. Communication speed flexibility forces D_i values for CCN to be maximized. Larger D_i values denote more feasible bus speed constraints. Bus complexity is described by the number of buses in an architecture. Number of buses is estimated by the number of links having similar D_i values and reduced time overlapping of their data communications. Estimating the number of buses, thus cores which share a bus, also permits finding the number of unnecessary connectivity in an architecture.

The second step is bus architecture synthesis. The co-synthesis flow continues by updating the Core Graph description (see Subsection 5.1) based on information on task partitioning and scheduling. Then, the detailed floorplan for the IP cores in the design 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 critical for characterizing the speed possible for 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 the core graph, 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) is based on simulated annealing. Using BA synthesis tables, the method builds bus architectures, which are PBS sets that meet all the connectivity requirements in the core graph. Topological attributes are evaluated for each bus architecture, i.e., 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 to account for DSM effects. The best found bus architecture is characterized for speed under RLC effects.

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 CN (ON) partitioning identified at Step 1.



Bitwise Decoder Algorithm

```
Let L = {e_m-2, e_m-1, ..., e_2, e_1, e_0 }
for (i = 0 ; i < 2^DIM(L) -1; i++)
    {
     Output basis set = decoder(i);
     PBS = OR(Output basis set);
     if (validate(PBS)==1){
       result = COMPARATOR (PBS , PBS Storage)
       if (result == 0) table update (PBS);
       }
    }
```

Bitwise PBS Generating Algorithm

Figure 12: **Bitwise decoder and bitwise PBS generating algorithms**

5 Bus Architecture Synthesis

5.1 Modeling for Bus Architecture Synthesis

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

This concept has been illustrated in Figure 11(a). The core graph representation of a system architecture is used for bus architecture synthesis. For simplicity of modeling, CG do not distinguish between unidirectional and bi-directional dataflow. Communication direction depends on whether an operation is a read or a write, and it isn't specified directly in a CG. However, the core graph can be modified in order to address the direction of data. Figure 11(b) presents the core graph for bi-directional communications. In case there is more than one communication channel between two cores, then the communication load is split across the channels. This type of CG will have multiple channels between cores, and these channels might get synthesized as separate buses in an architecture.

Industrial bus standards can be expressed using the CG formalism. The superposition principle can be applied, if an industrial bus standard is used at the transaction level. This can be done by classifying links according to their bandwidth (low, medium, and high), and generating core graphs for each category. This idea is well defined because most of the industry standards for SoC, i.e., AMBA [1] and IBM CoreConnect [3], have different buses to support data communication at different bandwidths. After bus architecture synthesis, bus sub-architectures associated with its core graph are combined together to get a finalized optimal solution.

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 11(c) and 11(d) show eight PBS for the CG in Figure 11(a). PBS on Figure 11(c) are valid. PBS are characterized by following physical and topological properties:

- *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 $P_u = \frac{2N_b}{n(n-1)} \times 100\%$, 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.
- *Communication conflict:* A PBS is implemented as a shared bus in the system architecture. Performance of a bus

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	

Figure 13: **Bus architecture synthesis tables**

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.

- *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 [7] [44]. Also, larger cross coupling and parasitic capacitances of longer buses increase interconnect latency [44]. Larger power dissipation for interconnect and drivers can be caused by longer buses. It is, however, difficult at the architecture level to estimate PBS bus length with high accuracy 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 12. 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 11(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 basis 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 connectivity requirements in a CG. Number of rows is similar to number of basis link elements in the CG. 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 13. The tables are for the CG in Figure 11(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). Subsection 5.2 explains the synthesis algorithm. Selecting a PBS to satisfy connectivity requirements depends on the PBS properties (utilization

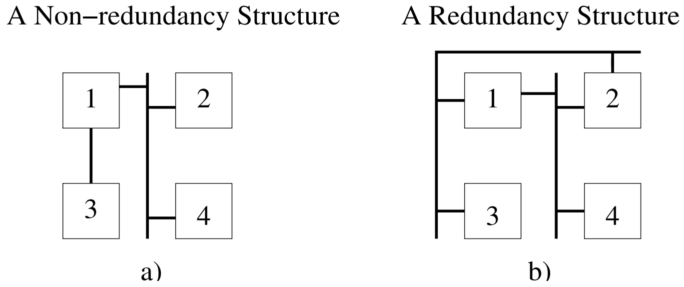
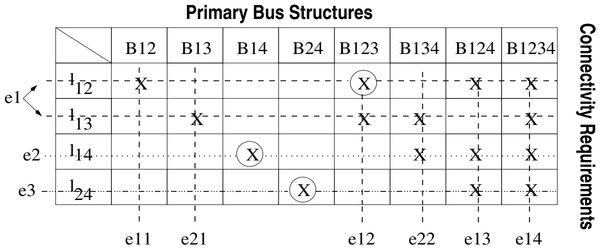


Figure 14: **Non-redundant non-hierarchical and redundant non-hierarchical bus architectures**




```
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
```

Figure 15: **Select-eliminate algorithm**

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 13 correspond to the same CG, but have their columns ordered for different performance requirements.

5.2 Bus Architecture Synthesis Algorithm

Depending on their structure, bus architectures (BA) can be either *non-redundant* or *redundant* structures, and *flat* 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 [3]. Examples of non-redundant, non-hierarchical (NRNH) and redundant, non-hierarchical (RNH) bus architectures are given in Figure 14. The NRNH bus architecture in Figure 14(a) uses the shared bus B_{124} to serve communications links l_{12},l_{14},l_{24}, and the point-to-point bus B_{13} for the link l_{13}. Figure 14(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. The modeling for bus architecture synthesis (including CG, PBS and BA synthesis tables) supports the other three bus architecture classes. We propose the select-eliminate (SE) algorithm to generate NRNH bus architectures based on the

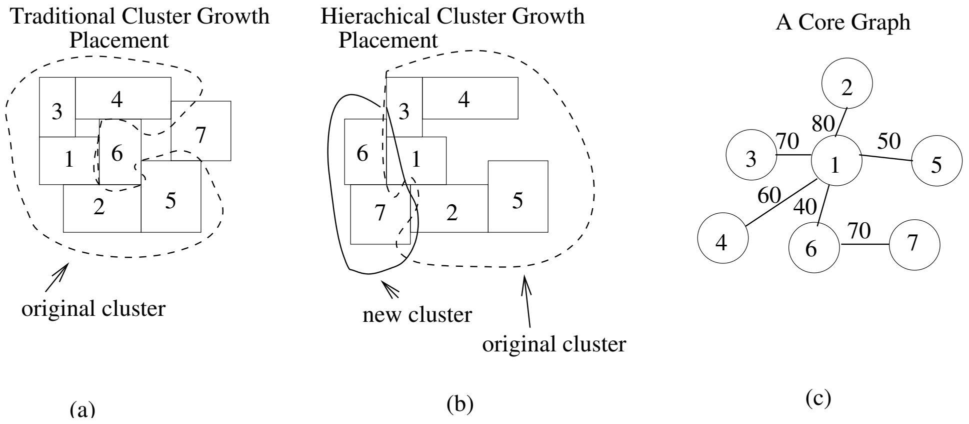


Figure 16: **Hierarchical cluster growth placement**

satisfaction of the core connectivity requirements. SE algorithm is represented in Figure 15. To illustrate the algorithm, we use the simple BA synthesis table in Figure 13(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 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. It is noticed that no more horizontal lines or vertical lines are created because there is no structure existing in a table. The generated NRNH BA is composed of PBS B_{124}, PBS B_{14}, and PBS B_{24}. Circled structures in Figure 15 show the final BA.

The size of a synthesis table grows depending on the number of cores and the number of inter-core communication. If 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 SE algorithm explore the PBS candidate space efficiently, we employed a 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 total cost function that guides simulated annealing is given by the formula

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

where L_t is the total bus length, N_b is the number of buses, C_c is the communication conflict, M_l is the maximum loss, and C_u is the total bus utilization. w_l, w_n, w_c, w_{ml}, w_u are weight factors. Maximum loss reflects the maximum data loss in a BA, if there is a conflict in a particular PBS. The algorithm objective is to minimize this cost function.

5.3 Hierarchical Cluster Growth Placement Algorithm

IP cores have to be placed for accurate estimation of bus lengths and bus speed. Goal is to place highly connected cores closely together. We decided to use a fast constructive placement algorithm based on cluster growth, so that it can be used in combination with the slower simulated annealing method. There are several cluster growth placement algorithms for ASIC [39]. However, the nature of placement problem for ASIC and SoC is different. In case of ASIC, most of the nodes are at gate/RTL level, thus are of a small and comparable size. Each connection between ASIC nodes uses a single wire, and the connectivity degree is relatively low. In contrast, SoC are composed of IP macros of irregular sizes. For example, the IBM PPC405 processor core has approximately 5 million gates [2]. Furthermore,

```
L={e1 ,e2 ,...,en }
SL= SORT (L);
cluster_list = {all macros in a core graph};
while (dim(cluster_list)!=1)
{
          link = choose a link(SL);
          [core1,core2] = extract_core(link)
          cluster1=getcluster(core1,cluster_list);
          cluster2=GetCluster(core2,cluster_list);
          if (cluster1 is different than cluster 2)
            {  cluster3= PlaceCluster(cluster1,cluster2);
               DeleteCluster(cluster_list,cluster1,cluster2);
               AddCluster(cluster_list,cluster3);
              }
     delete(SL,link);
}
```

(a)

```
basis_element set = extract(PBS);
sorted_element set = SORT_ASCENDING(basis_element set);
sum = 0;
while (count_number_of_element(sorted_element set)!=0)
    {
     get the link with the lowest communication load from sorted_element set;
    extract the link to get two connected clusters;
    get two sets of interconnect nodes from intersection layout;
    sum=sum+interconnect of two clusters(cluster1,cluster2);
    cluster3= update_cluster(cluster1,cluster2);
    delete the link from sorted_basis set;
    }
```

(b)

Figure 17: **Hierarchical cluster growth placement algorithms**

the degree of an SoC node is defined not only by the link to other nodes but also the bus width. Traditional flat cluster growth placement methods are thus not appropriate for SoC.

We propose a hierarchy based cluster growth placement (HCGP) for IP core placement. Because of its constructive approach, the algorithm is fast while preserving a good placement solution for SoC. The algorithm is therefore well fit for bus length estimation. The algorithm is illustrated in Figure 17(a). 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 16(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 [39]. Depending on the IP vendor, a number of metal layers is available to route cores. To easy the analysis of deep submicron 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 proposed algorithm for inter-core routing is presented in Figure 17(b). The algorithm starts by finding the shortest path of the link with the lowest communication load. This strategy is motivated by HCGP always placing two highly communicating cores close together. This implies that the longest path between two cores is at the link with the lowest communication load. Therefore, this link influences the total bus length of a PBS. We thus give it the highest priority to find the shortest path.

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 18(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. As seen in the bus routing diagram in Figure 18(b), the bus length is approximately half of the core perimeter. Therefore, if there is also another intersection point at point B, which is the longest distance form point A, all the wires have to route against the original direction. That is the total bus length is approximately a perimeter.

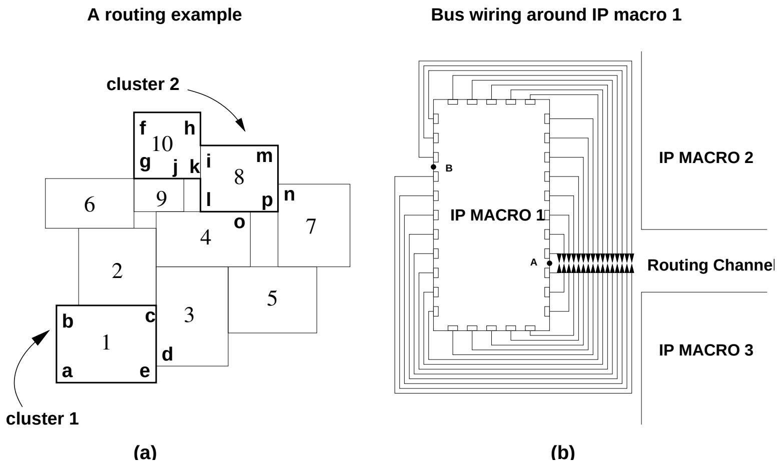


Figure 18: **Routing example**

This is considered to be the worst case, however, it is a good approximation if a macro has several intersection points required for inter-core routing. We use this worse case approximation in our bus length calculation. Lets 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_{macros} be the length of the bus-wiring around the cores associated with i_{th} PBS, the PBS bus length is defined as $l_{PBSi} = l_{inter} + l_{macros}$.

6 Experimental Results

A set of experiments was set up to study the effectiveness of the proposed hardware/software co-design algorithms. Due to the difficulty to relate our co-design objectives to other work, experiments addressed subsequent aspects, starting from aspects that are more similar to existing methods, and continuing with those that are specific to this work. We discussed three consecutive experiments:

- *Experiment 1* studied the latency of solutions generated using Performance Models and SA exploration as compared to existing heuristic algorithms. It experimented with applications having data and control dependencies. It also analyzed the effectiveness of combined task partitioning and scheduling.
- *Experiment 2* examined the usefulness of hardware resource sharing across tasks as opposed to co-design methodologies, which do not allow sharing. Two examples were designed using the two methodologies, and obtained results were contrasted.
- *Experiment 3* presents results for automatically synthesizing bus architectures. The paper discussed bus synthesis results for the IBM network processor system [12] and for a JPEG SoC.

The co-synthesis method was implemented in about 3,000 lines of C code. Bus synthesis algorithm is about 1,500 lines of C code. Experiments were run on a SUN Sparc 80 workstation. SA was run with the following parameters: initial temperature was 100,000, temperature length was set to 900, and cooling schedule was 0.7. The SA exploration finished when 100,000 solutions were analyzed, or when no improvement was observed for the last 7,000 moves.

Table 1 presents the characteristics of the task graphs used for experiments. Examples correspond either to task graphs with different structures, or to graphs used as benchmarks in related work. Examples *Parallel*, *Tree*, *Inverted Tree*, and *Fork-join* describe popular graph structures, such as parallel threads, tree, inverted tree, or sequence of tree and inverted tree. Task *Laplace* calculates the Laplace transform using a tree structure. Example *Linear solver* corresponds to parallel implementation of a linear equation solver. Examples *Graph 1*, *Graph 2*, *Graph 3*, and *Graph 4* are presented as benchmarks for partitioning and scheduling in [28]. Task *Graph 5* has a mixed tree + parallel structure, and is used in [22] as a motivational example. Tasks *Graph 6* and *Graph 7* are described in [17] as motivational examples for list scheduling. Columns 2 and 3 in the table show the number of tasks in the graph, and Column 4 indicates the number of conditional dependencies present in the task graph.

Experiment 1

The first experiment studied the latency of scheduling solutions produced by PM and SA exploration for applications with data and reduced amount of control dependencies. Results were compared with solutions obtained by the method suggested in [22] [17], one of the few approaches for scheduling of tasks with many data dependencies and reduced number of control dependencies. For a fair comparison, we restricted the co-synthesis methodology to task scheduling by setting the partitioning (binding) probability to 0. Table 2 shows the obtained results. Column 2 indicates schedule latencies computed through list scheduling, and Column 3 presents schedule latencies found by the proposed exploration technique. Column 4 shows the relative improvement of SA exploration algorithm over list scheduling.

Table 2 motivates that PM and SA offer improved results over list scheduling algorithm. Improvements can be as high as 20% (for example *Graph 7*). This example was indicated as a typical situation in which list scheduling offers poor results. Reason is that list scheduling might allocate task priorities depending on situations that never occur. This is due to the mutual exclusiveness of condition values and their controlled tasks. Hence, "infeasible" priorities determine poor scheduling results. SA exploration-based scheduling does not face this disadvantage. A second benefit of PM and SA exploration is that it does not require fine-tuning of the method for a new application. In [17] [22], the authors motivate that extensive experimenting with thousand of graphs was conducted for validating a new cost function for list scheduling. The proposed co-design method offered for example *Graph 5* an improved solution. Reason is that the SA-based method is able to produce solutions, where a particular resource was idle, even though there were tasks ready for execution on that resource. Then, a higher-priority task could be scheduled in the "near" future for execution on that resource, without having to wait for the small priority task to end. This scheduling strategy can not be achieved in list scheduling, where tasks are scheduled greedily depending on their priorities.

The next experiment analyzed the effectiveness of combined task partitioning and scheduling. A set of experiments studied the influence of partitioning probability on the quality of results. These experiments studied both the complexity of the search process, expressed as the number of iterations until finding the best solution, and the total execution time of the exploration method. Partitioning probability was changed from a low value (such as 0.05) to a high partitioning probability, such as 0.4 (in this situation, the number of scheduling and partitioning steps are very similar). Table 3 presents the obtained results. For each probability, the table shows the best obtained solutions, the iteration numbers when best solutions were produced, and the total execution time of the search. Experiments show that the binding

Example (1)	Number of tasks (2)	Number of edges (3)	Number of control dependencies (4)	Graph structure (5)
Parallel	27	31	3	parallel
Tree	39	50	4	tree
Inverted tree	29	37	-	inverted tree
Fork-join	32	40	2	sequence of tree + inverted tree
Laplace	31	42	1	tree
Linear solver	17	19	-	parallel
Graph 1 [28]	23	26	-	sequence of tree + inverted tree
Graph 2 [28]	25	32	-	acyclic graph
Graph 3 [28]	25	31	-	acyclic graph
Graph 4 [28]	26	33	-	acyclic graph
Graph 5 [22]	34	41	3	tree + parallel
Graph 6 [17]	15	19	1	tree + inverted tree
Graph 7 [17]	23	27	2	tree + inverted tree

Table 1: **Characteristics of the benchmark task graphs**

Example (1)	List scheduling (2)	Exploration (3)	Relative improvement (%) (4)	Combined partitioning and scheduling (5)	Relative improvement (%) (6)
Parallel	120	120	0	70	41.6
Tree	238	238	0	226	5.0
Fork-join	42	38	9.52	36	5.2
Laplace	62	56	9.67	51	8.9
Graph 5	40	39	2.5	31	20.5
Graph 6	97	77	20.6	73	5.1
Graph 7	60	60	0	46	23.3

Table 2: **Scheduling under data and control dependencies**

Example	Probability = 0.05			Probability = 0.15			Probability = 0.2			Probability = 0.4		
	Best	Iteration	Time (sec)	Best	Iteration	Time (sec)	Best	Iteration	Time (sec)	Best	Iteration	Time (sec)
Parallel	90	1118	157	75	2519	188	70	4060	123	80	235	148
Tree	246	4686	285	232	1154	263	238	5130	207	226	8223	278
Inverted tree	114	4489	217	113	3766	229	115	1450	211	115	759	113
Fork-join	37	1907	274	36	6808	294	37	4871	280	37	2484	247
Laplace	55	5114	235	55	817	157	53	120	140	51	4848	189
Linear solver	176	80	119	180	41	95	176	1169	160	176	2726	156
Graph 1	183	3420	381	183	3117	308	168	10069	423	142	4605	364
Graph 2	147	1172	450	119	8040	398	119	8040	588	119	6646	511
Graph 3	145	4115	419	137	8303	584	112	3347	294	107	5324	344
Graph 4	282	4660	522	282	7232	249	282	5324	453	282	459	254
Graph 5	32	5944	789	32	14198	895	33	6648	742	31	6891	605
Graph 6	73	328	69	73	92	129	73	126	66	73	392	91
Graph 7	46	7031	410	47	822	340	46	5690	440	46	6567	415

Table 3: **Influence of partitioning probability on system latency**

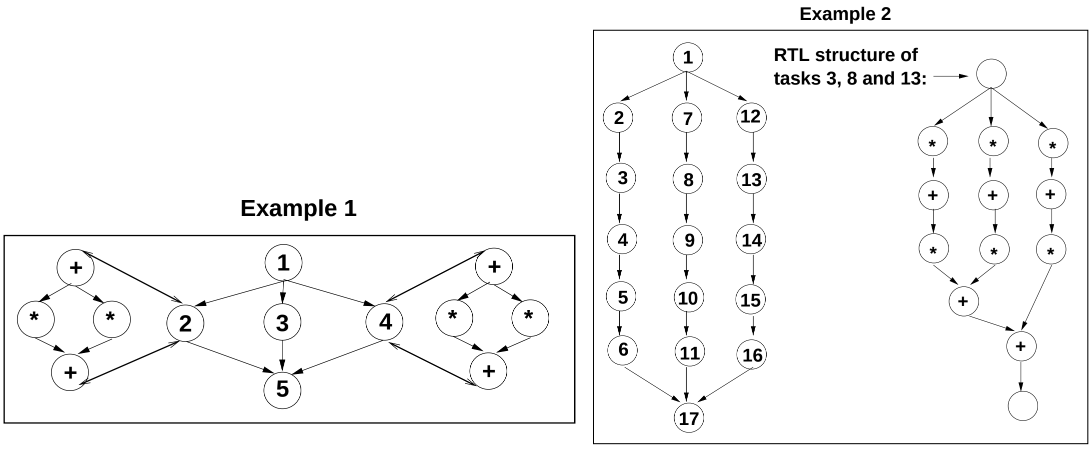
probability offering the best solution depends on the application kind. For example, for graph *Inverted tree* smaller binding probabilities, such a 0.05 and 0.15, produced better solutions than larger binding probabilities, such as 0.2 and 0.4. However, in most of the situations a binding probability of 0.2 provides a good trade-off between the number of partitioning steps and scheduling steps. Hence, the hardware/software co-design exploration has to be conducted for various partitioning probabilities to find the best solution for a new application. Results also indicated a reasonably high computational complexity, as execution time was in the range of maximum several hundreds of seconds.

The solutions obtained by combined partitioning and scheduling were compared to the results when tasks are pre-placed to resources. Column 5 in Table 2 indicates the best latency obtained for combined partitioning and scheduling. Column 7 shows the relative improvement of the schedule lengths as compared to the results for pre-partitioned situations, which are indicated in Column 2 of the table. Relative improvements of the schedule lengths can be as high as 40%, if task pre-partitioning is less efficient. This illustrates the difficulty of identifying good task partitions before scheduling, thus explaining the need for a combined partitioning-scheduling co-design approach.

Experiment 2

Two examples were used to study the effect of hardware sharing across tasks on reducing system latency. The first example studied the aspects that influence the latency of a hardware/software solution. A simple example was used in this case. The second experiment validated the conclusions of the first example for a bigger real-life inspired example.

Figure 19(a) presents the task structure of the first example. Tasks 2 and 4 are also presented at the operation level, as shown in the figure. Table 4 presents the time characteristics of each task for the five experiments that were designed. Time (T-S) of the software implementation for each task remained unchanged in each experiment. The five experiments differ by the time (T-H) characteristics of the adder and multiplier blocks used for the hardware implementation. Rows 5 and 6 contain this information. Each hardware implementation used 2 adder and 2 multiplier blocks. The execution



(a) (b)

Figure 19: **Task graph examples for co-synthesis with hardware sharing across tasks**

Task	Expmnt. 1				Expmnt. 2				Expmnt. 3				Expmnt. 4				Expmnt. 5			
	T-SW	T-HW	Sep/Int	Imp (%)	T-SW	T-HW	Sep/Int	Imp (%)	T-SW	T-HW	Sep/Int	Imp (%)	T-SW	T-HW	Sep/Int	Imp (%)	T-SW	T-HW	Sep/Int	Imp (%)
1	1	-	-/-	-	1	-	-/-	-	1	-	-/-	-	1	-	-/-	-	1	-	-/-	-
2	130	66	-/-	-	130	10	-/-	-	130	24	-/-	-	130	136	-/-	-	130	192	-/-	-
3	130	-	-/-	-	10	-	-/-	-	24	-	-/-	-	136	-	-/-	-	192	-	-/-	-
4	130	66	-/-	-	130	10	-/-	-	130	24	-/-	-	130	136	-/-	-	130	192	-/-	-
+	-	1	-/-	-	-	1	-/-	-	-	8	-/-	-	-	64	-/-	-	-	164	-/-	-
*	-	64	-/-	-	-	8	-/-	-	-	8	-/-	-	-	8	-/-	-	-	164	-/-	-
	-	-	134/132	1.5	-	-	22/20	9.0	-	-	50/34	32	-	-	268/258	3.7	-	-	324/258	20.4

Table 4: **Example 1: Task graph characteristics and experimental results**

time (T-H) of a task realized as hardware was determined after operation binding and scheduling for the five distinct experiments. The resulting values are shown in the table. The last row indicates the latency for the traditional co-synthesis approach (no hardware sharing across tasks), for the proposed method (if sharing across tasks is allowed), and the percentage improvement. If hardware sharing was contemplated, latency improved between 1.9% (for Experiment 1) up to 32% (for Experiment 3). The highest improvement corresponds to Experiment 3, in which the resource sharing between operations of tasks 2 and 4 is maximized. For this case, both tasks are realized in hardware. The addition operation of task 4 is performed concurrently with the two multiplications of task 2. Similar operation concurrencies exists for the other 4 experiments. Speed-ups, however, are smaller because the relative execution time of an addition as compared to a multiplication operation are smaller than for Experiment 3. This suggests that hardware sharing of medium grained IP cores (like multipliers) can significantly improve latency. Hardware sharing of low grained cores does not offer relevant improvements.

Example	Time Optimization		
	Without HW sharing	With HW sharing	Improvement (%)
(1)	(2)	(3)	(4)
2GPP + 1A + 1M	184640	182720	1.03
2GPP + 2A + 2M	157120	125120	20.3
2GPP + 3A + 3M	137920	125120	9.28
3GPP + 1A + 1M	182720	182720	0
3GPP + 2A + 2M	157120	125120	20.3
3GPP + 3A + 3M	137920	99520	27.84

Table 5: **Example 2: experimental results**

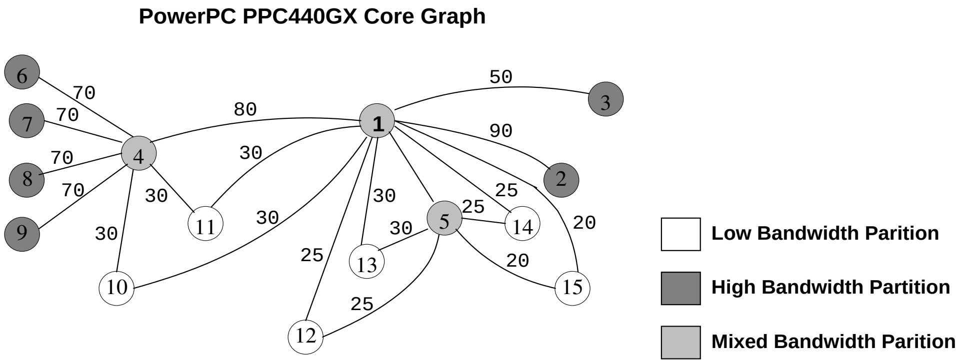


Figure 20: **Core graph for the network processor**

Figure 19(b) presents a second example inspired from the JPEG algorithm. The task graph included 17 tasks. The RTL structure of tasks 3, 8, and 13 is shown in the right part of the figure. Six experiments were conducted. Each experiment employs a different number of general purpose processors (GPPs) for the software part, and a different number of modules (adders (A) and multipliers (M)) for the hardware part. Column 1 in Table 5 shows the number of hardware resources for each example. Columns 2, 3, and 4 present the latencies offered by co-design without and with hardware sharing across tasks, and the corresponding latency improvements. Note that latency improvement can be as high as 27% (Line 6 in the table), if a high task and operation concurrency can be secured for the system. For this case, concurrency of operations of different hardware tasks results in significant latency improvements. In one case (Line 4 in the table) latency improvement was 0%. Reason is that only 1 adder and 1 multiplier were used for the hardware part. The ability of co-design to perform resource sharing across hardware tasks could not be used in this case.

Experiment 3

The first experiment presents the bus architecture synthesis results for a network processor [12]. The processors 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 20 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 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 (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 (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 6 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.

First twelve rows in Table 6 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

	w_l	w_n	w_c	w_r	Bus length	# of buses	Redundancy	Conflicts	Max loss
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(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	1.0	0.1	0.1	0.838	7	0.104	0.222	0.3
5	0.5	1.0	0.1	0.1	0.850	8	0.140	0.25	0.45
6	1.0	1.0	0.1	0.1	0.836	9	0.138	0.142	0.31
7	0.1	0.7	0.1	0.5	0.833	9	0.114	0.285	0.28
8	0.5	0.7	0.1	0.5	0.855	8	0.183	0.375	0.32
9	1.0	0.7	0.1	0.5	0.838	7	0.132	0.111	0.26
10	0.1	1.0	0.1	0.5	0.872	8	0.154	0.125	0.28
11	0.5	1.0	0.1	0.5	0.841	7	0.166	0.428	0.37
12	1.0	1.0	0.1	0.5	0.863	7	0.154	0.285	0.37
13	0.1	0.1	0.7	0.1	0.941	16	0.08	0.0	0.0
14	0.5	0.1	0.7	0.1	0.941	18	0.03	0.0	0.0
15	0.1	0.1	1.0	0.1	0.952	15	0.08	0.0	0.0
16	0.5	0.1	1.0	0.1	0.908	16	0.07	0.0	0.0
17	0.1	0.1	1.0	0.5	0.950	19	0.01	0.0	0.0
18	0.5	0.1	1.0	0.5	0.940	18	0.03	0.0	0.0
19	0.1	0.5	1.0	0.1	0.963	14	0.19	0.0	0.0
20	0.5	0.5	1.0	0.1	0.988	14	0.17	0.0	0.0
21	0.1	0.5	0.7	0.5	0.961	15	0.07	0.0	0.0
22	0.5	0.5	0.7	0.5	0.933	15	0.12	0.0	0.0
23	0.1	0.5	1.0	0.5	0.933	15	0.11	0.0	0.0
24	0.5	0.5	1.0	0.5	0.952	14	0.16	0.0	0.0

Table 6: **Results for bus architecture synthesis for the network processor**

redundancies are low. The weight for bus length is varied from small to large values. Bus complexity minimization will favor shared buses, and discourage usage of point-to-point communications. Simple bus architectures result, 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. 30 buses were used in the six cases. Only three buses are heavily re-used in 5 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 are rarely re-used. It is mostly point-to-point links that are re-used. The total bus length is high, meaning that individual buses are 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 rows 3, 6, 9 and 12). The average communication conflict is high, around 0.24. Thus, the low communication concurrency will result in poor timing. Redundancies are also high, meaning that the bus architectures offers some core connectivity that is 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.

Rows 13-24 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 21 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.

BA synthesis was used to automatically produce optimized bus architectures for the SoC of the JPEG image compression encoder. The task graph for the JPEG encoder included three identical and parallel sequences of tasks. Each sequence processes a different color of an image (RGB), and includes five consecutive tasks: preprocessing, FDCT, quantization, zig-zag, and RLE & Huffman coding. The hardware-software co-design methodology in Figure 10 was performed on the JPEG task graph. For each sequence, tasks preprocessing, quantization, zig-zag, and RLE & Huffman coding were partitioned into software, and task FDCT into hardware. The architecture included three processor

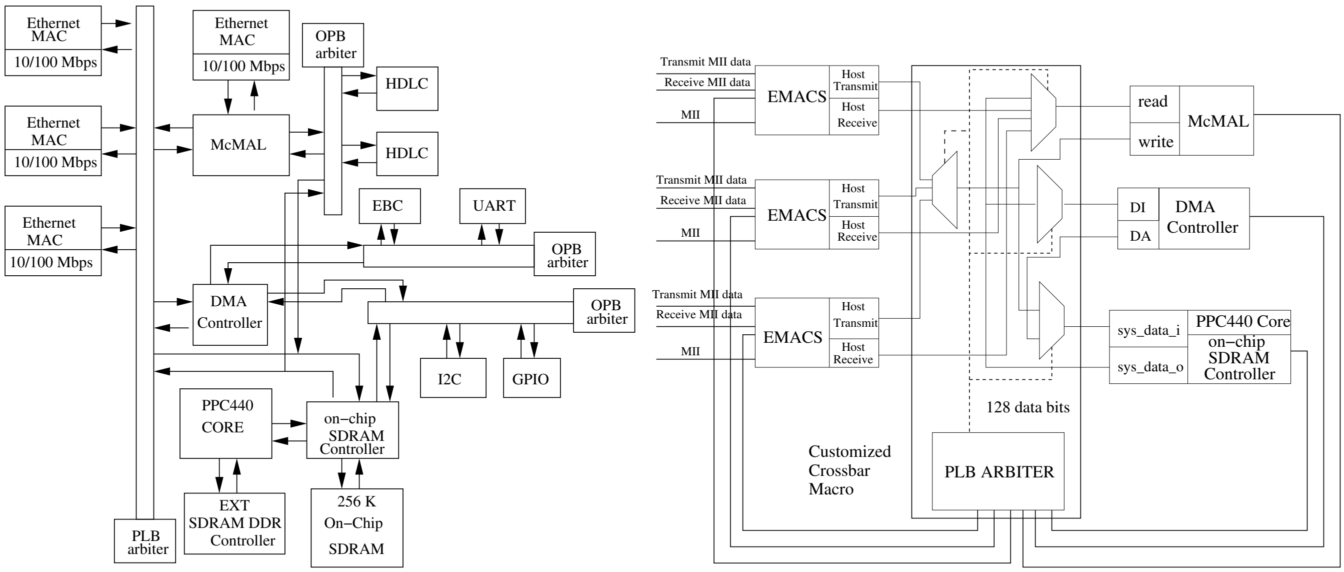


Figure 21: **Synthesized bus architecture for the network processor**

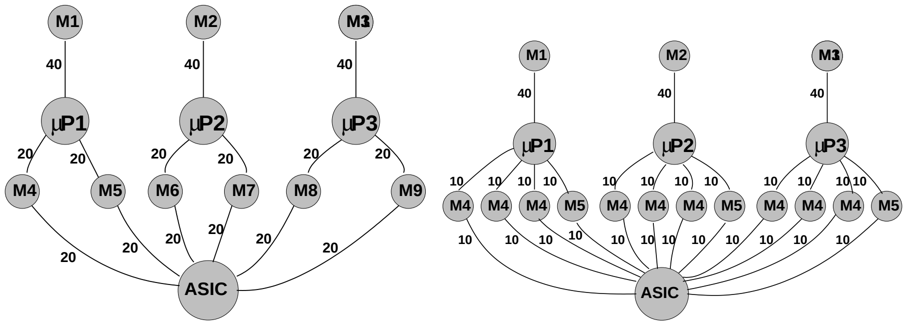


Figure 22: **Core Graph for JPEG**

cores (a distinct core for each parallel sequence), an ASIC for the FDCT tasks, and memory modules for data communication. Each processor has its own local memory. Processors and ASIC communicate through shared memory. To decrease memory access times, the first architecture considered interleaved memory blocks M4-M9. The top part of Figure 23 shows the resulting Core Graph. To further improve processor-ASIC communication speed, additional interleaved memory blocks were used. This resulted in the Core Graph shown at the bottom of Figure 22. The considered processing technology was 0.18μ TSMC. Microprocessor cores were of about $5 \times 5\ mm^2$, memory cores of about 25% of the area of processor cores, and ASIC were about 30% of processor core area.

Figure 23 shows bus architecture synthesis results for the top CG in Figure 22. The hierarchical cluster growth algorithm generated the core placement shown at the top, and the bus architecture is presented at the bottom of Figure 23. The synthesis goal was to generate a fast architecture. Bus architecture complexity was not a major concern, because the number of IP cores is 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.

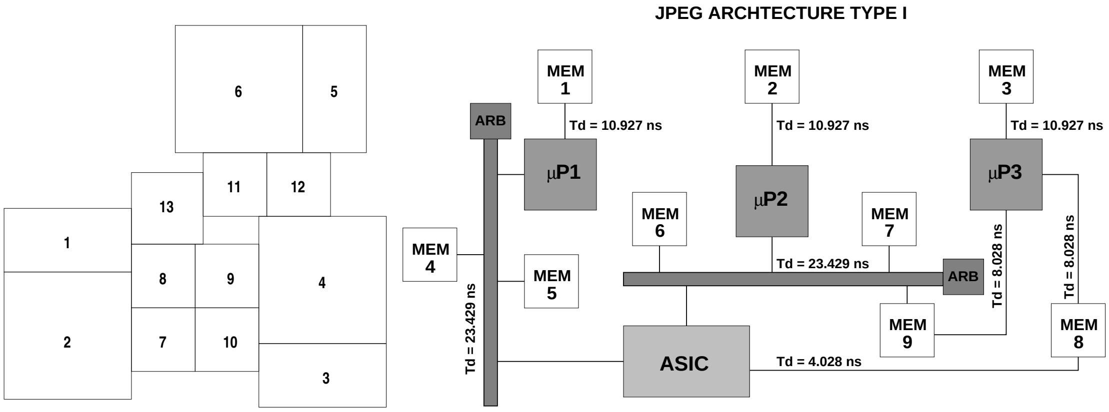


Figure 23: **Bus architecture for JPEG SoC**

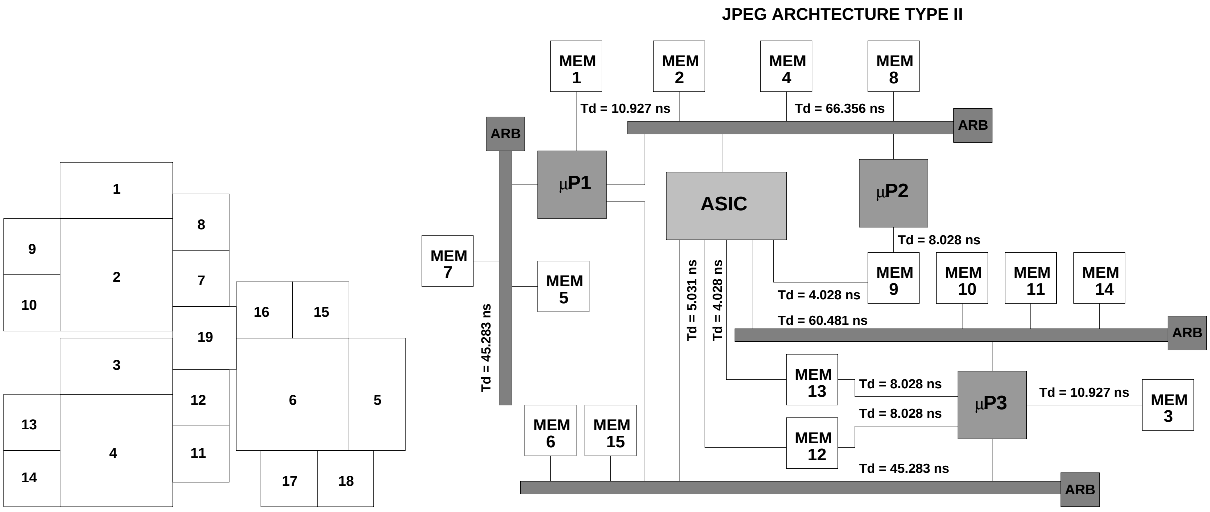


Figure 24: **Bus architecture for JPEG SoC**

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. Same design constraint were used for the CG at the bottom of Figure 22. Experiments used the core placement in Figure 24, and the synthesized BA (including bus delays) is shown at the bottom of Figure 24. BA synthesis took less than 5 minutes on a SUN Blade 100 workstation. This motivates that the pruning method of the BA synthesis algorithm allowed to quickly explore the very large solution spaces resulting for SoC with many cores.

7 Conclusion

This paper presents a new hardware-software co-design methodology for resource constrained SoC realized in a deep submicron process (DSM). The methodology is useful for multimedia applications optimized for latency. The approach addresses layout and hardware aspects relevant to system design: it considers the dependency of task communication speed on interconnect parasitic, as well as the possibility of sharing coarse and medium grained IP cores across tasks. The methodology includes an original algorithm for bus architecture synthesis. The co-design process performs three successive steps: Using simulated annealing, co-design explores for combined task and communication partitioning and scheduling, and finding feasible requirements for the minimum speed of communication links. Exploration employs Performance Models (PM) to express relationships between system performance (latency and communication speed flexibility), specification attributes, and design decisions. The second step synthesizes and routes the bus architecture. IP cores are placed using a hierarchical cluster growth algorithm. Next, bus architecture synthesis finds a set of possible building blocks (using the PBS bitwise generation algorithm), and assembles them together using simulated annealing. The paper presents a special table structure, named bus architecture synthesis table, and select-eliminate method to prune low quality bus architectures. The third step re-schedules tasks and communications using precise information on bus speed for the best found bus architecture.

The co-design methodology improves the practicality of system-level design for DSM technologies. 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 the best, as there is little re-using among bus architectures optimized for different constraints. Experiments also indicated that PM are more effective than well known synthesis metrics, like task priorities. For SA-based exploration using PM, system latency was up to 20% shorter than for list scheduling. Reason was that PM avoid the modeling limitations of priority functions. PM are general, flexible, and can be easily extended for new design activities without requiring cumbersome validation. The combined partitioning and scheduling technique offers latency improvements of up to 40% as compared to a method, which separates partitioning and scheduling. For highly parallel applications, sharing of medium and coarse grained resources improves latency by up to 30% as compared to sharing of coarse grained cores only.

References

[1] "DesignWare AMBA On-Chip Bus Solution", *www.convergencepromotions.com/ARM/catalog/156.html.*

[2] IBM Blue Logic Technology, *//http:www-3.ibm.com/chips/bluelogic/.*

[3] IBM CoreConnect Bus Architecture White Paper, *//http:www-3.ibm.com/chips/products/coreconnect/index.html.*

[4] SRC Research needs in Logic and Physical Level Design and Analysis, August 2002.

[5] F. Balarin, L. Lavagno, P. Murthy, A. Sangiovanni-Vincentelli, "Scheduling for Embedded Real-Time Systems", *IEEE Design & Test of Computers*, Jan-March 1998, pp. 71-82.

[6] S. Battacharyya, "Hardware/Software Co-synthesis for DSP Systems", in Y. Hu, editor, *Programmable Digital Signal Processors: Architecture, Programming, and Application*, pp. 333-378, Marcel Dekker, 2002.

[7] T. R. Bednar, P. H. Buffet, R. J. Darden, S. W. Gould, P. S. Zuchowski, "Issues and Strategies for the Physical Design of System-on-Chip ASICs", *IBM Journal of Research & Development*, Vol. 46, No. 6, Nov. 2002, pp. 661-673.

[8] A. Bender, "MILP Based Task Mapping for Heterogeneous Multiprocessor Systems", *Proc. of the European Design Automation Conference*, 1996, pp.283-288.

[9] R. Bergamaschi, W. Lee, "Designing Systems-on-Chip using Cores", *Proc. of the Design Automation Conference*, 2000, pp. 420-425.

[10] T. Blickle, J. Teich, L. Thiele, "System-level Synthesis using Evolutionary Algorithms", *Journal of Design Automation for Embedded Systems*, pp. 23-58, 1998.

[11] B. Dave, G. Lakshminarayana, N. Jha, "COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems", *IEEE Transactions on Computer-Aided Design*, Vol. 7, No. 1, 1999, pp.92-104.

[12] J. Darringer, R. Bergamaschi, S. Battacharyya, D. Brand, A. Herkersdorf, J. Morell, I. Nair, P. Sagmeister, Y. Shin, "Early Analysis Tools for System-on-a-Chip Design", *IBM Journal of Research & Development*, Vol. 46, No. 6, Nov. 2002, pp. 691-707.

[13] J. M. Daveau, G. F. Marchioro, T. Ben Ismail, A. A. Jerraya, "Protocol Selection and Interface Generation for HW-SW Codesign", *IEEE Transactions on VLSI Systems*, Vol. 5, No. 1, pp. 136-144, March 1997.

[14] R. Davis, N. Zhang, K. Camera, F. Chen, D. Markovic, N. Chan, B. Nikolic, R. Brodersen, "A Design Environment for High Throughput, Low Power Dedicated Signal Processing Systems", *Proc. of the IEEE Custom Integrated Circuits Conference*, 2000.

[15] G. De Micheli, "Synthesis and Optimization of Digital Circuits", *McGraw-Hill*, 1994.

[16] R. Dick, N. Jha, "MOGAC: A Multiobjective Genetic Algorithm for the Co-Synthesis of Hardware-Software Embedded Systems", *Proc. of the International Conference on Computer-Aided Design*, 1997.

[17] A. Doboli, P. Eles, "Scheduling under Control Dependencies for Heterogeneous Architectures", *Proc. of the International Conference on Computer Design*, 1998, pp. 602-608.

[18] G. W. Doerre, D. E. Lackey, "The IBM ASIC/SoC Methodology - A Recipe for first-time Success", *IBM Journal of Research & Development*, Vol. 46, No. 6, Nov. 2002, pp. 649-660.

[19] M. Drinic, D. Kirovski, S. Meguerdichian, M. Potkonjak, "Latency-Guided On-Chip Bus Network Design", *Proc. of the International Conference on Computer-Aided Design*, 2000, pp. 420-423.

[20] S. Dutta, R. Jensen, A. Rieckmann, "Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems", *IEEE Design & Test of Computers*, Vol. 16, No. 5, September-October 2001, pp. 21-31.

[21] R. Ernst, "Codesign of Embedded Systems: Status and Trends", *IEEE Design & Test of Computers*, April-June, 1998, pp. 45-54.

[22] P. Eles, A. Doboli, P. Pop, Z. Peng, "Scheduling with Bus Access Optimization for Distributed Embedded Systems", *IEEE Transaction on VLSI*, Vol. 8, No. 5, pp. 472-491, October 2000.

[23] D. Gajski, F. Vahid, "Specification and Design of Embedded Hardware/Software Systems", *IEEE Design & Test of Computers*, Spring 1995, pp. 53-67.

[24] M. Gasteier, M. Glesner, "Bus-Based Communication Synthesis on Aystem Level", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 4, No. 1, January 1999, pp. 1-11.

[25] T. Givargis, F. Vahid, "Platune: A Tuning Framework for Systems-on-a-Chip Platforms", IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 21, No. 11, November 2002, pp. 1- 11.

[26] R. Gupta, "Co-Synthesis of Hardware and Software for Digital Embedded Systems", *Kluwer*, 1995.

[27] J. Henkel, "A Low Power Hardware/Software Partitioning Approach for Core-based Embedded Systems", *Proc. of the Design Automation Conference*, 1999, pp.122-127.

[28] J. Hou, W. Wolf, "Process Partitioning for Distributed Embedded Systems", *Proc. of the International Workshop on Hardware/Software Co-Design*, 1996, pp. 70-76.

[29] J. Hu, Y. Deng, R. Marculescu, "System-level Point-to-Point Communication Synthesis Using Floorplanning Information", *Proc. of the International Conference on VLSI Design*, 2002.

[30] A. Kalavade, E. Lee, "A Globaly Critically/Local Phase Driven Algorithm for Constrained Hardware/Software Partitioning Problem", *Proc. of the International Workshop on Hardware/Software Co-Design*, 1994, pp. 42-48.

[31] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey and A. Sangiovanni-Vincentelli, "System Level Design: Orthogonolization of Concerns and Platform-Based Design", *IEEE Transactions on CADICS*, Vol. 19, No. 12, December 2000.

[32] K. Lahiri, A. Raghunathan, S. Dey, "Efficient Exploration of the SoC Communication Architecture Design Space", *Proc. of the International Conference on Computer-Aided Design*, 2000, pp. 424-430.

[33] R. Ortega, G. Boriello, "Communication Synthesis for Distributed Embedded Systems", *Proc. of the International Conference on Computer-Aided Design*, 1998, pp. 437-444.

[34] S. Prakash, A. Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems", *Journal of Parallel and Distributed Computing*, 16, 1992, pp. 338-351.

[35] C. Reeves *et al*, "Modern Heuristic Techniques for Combinatorial Problems", *J. Wiley*, 1993.

[36] K. Van Rompaey, D. Verkest, I. Bolsens, H. De Man, "Co-Ware - a Design Environment for Heterogeneous Hardware/Software Systems", *Proc. of the Design Automation Conference*, 1996, pp. 252-257.

[37] M. Rutten, J. van Eijndhoven, E. Jaspers, P. Wolf, O. Gangwal, A. Timmer, E. Pol, "A Heterogeneous Multiprocessors Architecture for Flexible Media Processing", *IEEE Design & Test of Computers*, July-August 2002, pp. 39-50.

[38] T. Seceleanu et al, "On-Chip Segmented Bus: A Self-Timed Approach", *Proc. of the IEEE ASIC/SOC Conference*, 2002, pp. 216-220.

[39] N. Sherwani, "Algorithms for VLSI Physical Design Automation", *Kluwer*, 1999.

[40] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, J. Teich, "Scheduling Hardware/Software Systems Using Symbolic Techniques", *Proc. of the International Workshop on Hardware/Software Co-Design*, 1999, pp. 173-177.

[41] D. Stroobandt, "A Priori Wire Length Estimates for Digital Design", Kluwer, 2001.

[42] D. Stroobandt, "A Priori Wire Length Distribution Models With Multiterminal Nets", *IEEE Transactions on VLSI*, Vol. 11, No. 1, February 2003, pp. 35-43.

[43] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli, "Adressing the System-on-a-Chip Woes Through Communication-Based Design", *Proceedings of the Design Automation Conference*, 2001, pp. 667-672.

[44] D. Sylvester, K. Keutzer, "A Global Wiring Paradigm for Deep Submicron Design", *IEEE Transactions on CADICS*, Vol. 19, No. 2, pp. 242-252, February 2000.

[45] Y. Xie, W. Wolf, "Allocation and Scheduling of Conditional Task Graphs in Hardware/Software Co-synthesis", *Proc. of Design, Automation and Test in Europe Conference*, 2001, pp. 620-625.

[46] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, R. Lauwereins, "Energy-Aware Runtime Scheudling for Embedded-Multiprocessor SOCs", *IEEE Design & Test of Computers*, September-October 2001, pp. 46-58.

[47] T. Y. Yen, W. Wolf, "Hardware-Software Co-synthesis of Distributed Embedded Systems", *Kluwer*, 1997.

[48] P. Zarkesh-Ha, J. Davis, J. Meindl, "Prediction of Net-Length Distribution for Global Interconnects in a Heterogeneous System-on-a-Chip", *IEEE Transactions on VLSI*, Vol. 8, No. 6, December 2000, pp. 649-659.