

Latency-Driven Design Of Multi-Purpose Systems-On-Chip

Seapahn Meguerdichian
UCLA Computer Science Dep.
4732 Boelter Hall
Los Angeles, CA 90095-1596
seapahn@cs.ucla.edu

Milenko Drinic
UCLA Computer Science Dep.
4732 Boelter Hall
Los Angeles, CA 90095-1596
milenko@cs.ucla.edu

Darko Kirovski
Microsoft Research
One Microsoft Way
Redmond, WA 98052
darkok@microsoft.com

ABSTRACT

Deep submicron technology has two major ramifications on the design process: (i) critical paths are being dominated by global interconnect rather than gate delays and (ii) ultra high levels of integration mandate designs that encompass numerous intra-synchronous blocks with decreased functional granularity and increased communication demands. These factors emphasize the importance of the on-chip bus network as the crucial high-performance enabler for future systems-on-chip. By using independent functional blocks with programmable connectivity, designers are able to build systems-on-chip capable of supporting different applications with exceptional levels of resource sharing. To address challenges in this design paradigm, we have developed a methodology that enables efficient bus network design with approximate timing verification and floorplanning of multi-purpose systems-on-chip in early design stages. The design platform iterates system synthesis and floorplanning to build min-area floorplans that satisfy statistical time constraints of applications. We demonstrate the effectiveness of our bus network design approach using examples from a multimedia benchmark suite.

1. INTRODUCTION

Due to high design complexities and time-to-market pressure, it is expected that future SOC's are designed as networks of virtual components. A virtual component is a core wrapped with logic that enables it to communicate data to the attached bus with an arbitrary bus protocol. Due to high deep-sub-micron (DSM) integration (100M transistors), a typical SOC is estimated to contain several hundreds of cores with each core having less than 100K gates [Sy99]. Since decreased levels of module granularity in computation results in higher communication costs, inter-core communication has great impact on the performance of SOC designs. The communication among cores poses several design issues that can be classified as: (i) synchronization and (ii) performance optimization problems. While latency insensitive synchronization between cores can be resolved using relay stations and appropriate communication protocols, bus network design techniques have been proposed only for single-application-multi-module SOC's [Dri00]. However, design tools for multi-application SOC's have tremendous importance as the nature of many applications such as multimedia collaboration, ubiquitous Web browsing, and mobile VPN networks, encompass synchronous co-operation of a number of applications. Furthermore, within the mass of computationally demanding applications, data communication, compression, encryption, and DSP, modules are commonly shared. Hence, trends in the design of multi-purpose SOC's are bound to converge towards using independent application-specific and programmable blocks integrated using a highly programmable and flexible net-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA
© 2001 ACM 1-58113-297-2/01/0006..\$5.00.

work of buses that enables high levels of resource sharing.

We present a novel vertical design framework that targets early-stage design of a crucial component of multi-purpose SOC, namely the bus network. In particular, we focus on approximate verification of the inter-module communication latencies and floorplanning. The developed platform iterates between system synthesis and floorplanning to build min-area floorplans with an aim to satisfy the real-time constraints of specified applications.

2. RELATED WORK

Reference [Mad95] presents a discussion on the challenges of designing single- and multi-processor application-specific programmable systems (ASPS). The trade-offs in building performance evaluation frameworks for ASPSS have been studied in [Deg90]. Inter-core communication issues such as collusion minimization for such SOC's have been elaborated in [Sur96]. A detailed discussion of the related work regarding bus network design and SOC technologies can be found in [Dri00].

3. GLOBAL OVERVIEW

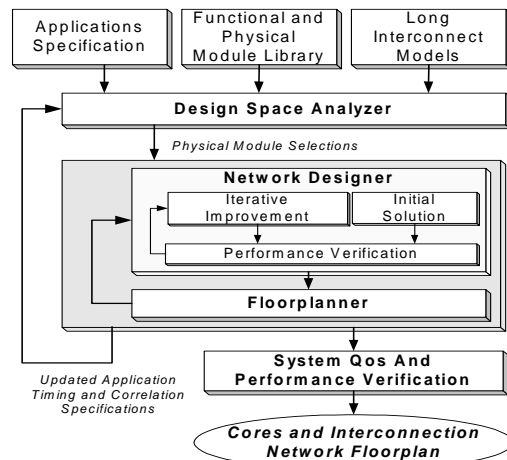


Figure 1 - Global Design Flow

The global framework shown in Figure 1 depicts the interaction of the three main components: (i) **Design Space Analyzer (DSA)**, (ii) **Network Designer**, and (iii) the latency-guided **Floorplanner**. Each application is specified using functional modules from the module library with statistical information about timing and communication patterns obtained using the *Communication Profiler*. In addition to the functional (algorithmic) specifications, the module library contains a set of Pareto-optimal physical implementation instances of each functional module. The DSA selects several sets of physical modules that meet the initial timing and QoS requirements of all applications. The selected module set is passed to the Network Designer that creates a suitable bus network. The Floorplanner verifies the area requirement and the feasibility of the interconnection network based on the provided long-interconnect models. The Floorplanner outputs several statistics

about the most-constraint and least-constraining buses to the Network Designer as part of an iterative improvement loop. After this optimization process, the newly acquired application timing and communication statistics are passed back to the DSA to improve the module selection and repeat the optimization process. Using this approach, a designer is able to create designs that satisfy estimations of DSM effects in the lower levels quite efficiently. In addition, the framework provides the necessary means to build systems based on complex clusters of application built on top of the same set of reusable modules.

4. LATENCY-DRIVEN DESIGN

The generic hardware model that we have adopted assumes the following set of constraints: Cores have the property of being hard, i.e. with constant layout; They are connected with buses of limited length which is governed by the maximum bus transaction clock cycle; The bus network is switched using a topology of bus bridges where a bus bridge is a crossover of at most four buses; The area of a bus bridge is approximated at 5% of an average core size per bridge port; Bus arbitration is performed with static priority assignments; Bus latency is modeled using a second order polynomial of the wire length; Data routing across the network is assumed to be deadlock free.

The communication profiler summarizes the essential statistics of communication patterns of the implemented application. The input to the communication profiler is an instance of the communication model of our target system. The communication model has been adopted from [Dri00]. In our experiments, we have used traffic patterns extracted and extrapolated from the MediaBench benchmarks suite [Lee97].

Definition 1: A functional module library M is a set of modules $\{M_1, M_2, \dots, M_n\}$. Each functional module $M_i \in M$ has at least one physical implementation (core) that is denoted using the notation C_{iA}, C_{iB}, C_{iC} , etc, with known layout and timing characteristics.

Definition 2: A set of applications A_i from a set $A = \{A_1, A_2, \dots, A_k\}$, is modeled using a Functional Module Dependency Graph ($FMDG_i$) and a communication correlation table (CCT_i).

Definition 3: A Functional Module Dependency Graph, $FMDG$ is a directed graph with vertex set V and g -weighted edge set E . Each vertex $v \in V$ corresponds to a functional module in M . Each directed edge $e \in E$ indicates the communication dependency among the modules connected by e . The weight of each edge $e \in E$ is a Gaussian random variable with mean μ and variance σ^2 and is used to stochastically model the wide range of typically bursty traffic patterns an application may generate. The communication profiler generates the Gaussian weights $e(\mu, \sigma^2)$ by collecting run-time information for each individual application.

Definition 4: For each $FMDG$ with the g -weighted edge set E , we define the CCT as an $|E| \times |E|$ real valued matrix. For any pair of edges $\{e_i, e_j\} \in E$, element (i, j) of the corresponding CCT represents the temporal correlation of e_i and e_j , i.e. the percentage of time e_i and e_j are simultaneously active.

4.1 Design Space Analyzer

We use dynamic programming to select several initial sets of modules to satisfy the timing requirements of all applications subject to area constraints. Since at this stage no actual interconnection network exists, we use the function $exec_time$ to estimate and verify the timing (QoS) requirement for each application ($FMDG$). Given an $FMDG$ G and a set of physical module assignments S ,

$$exec_time(G, S) = \sum_{\forall m \in Critical_Path(G)} [map(m, S)].worst_run_time$$

where $map(m, S)$ returns the selected physical implementation of a module m . The $worst_run_time$ attribute is obtained from the module library. As shown above, $exec_time(G, S)$ approximates the total execution time for a given application, by adding the individual worst case module execution times along the critical path of the corresponding $FMDG$. Thus, the DSA task can be stated as:

Problem: Physical Module Selection

Input: $FMDG$ set; timing constraint set T .

Question: Is there a set of physical modules S with minimal area in M such that $\forall fmdg_i \in FMDG, exec_time(fmdg_i, S) \leq T_i$?

Figure 2 shows the pseudo-code for the key algorithms used in DSA , namely the $FMDG$ merge and the module selection algorithms. In addition to the edges being combined, the parameters ϕ_1 and ϕ_2 are used in the $Combine_Gaussian$ function to calculate a weighted average of the edge weights of e_1 and e_2 . If for instance, the application represented by G_1 has more total communication requirements than G_2 ($\phi_1 > \phi_2$), then e_1 is given a higher weight in the averaging than e_2 to satisfy the more constraining communication requirements of G_1 . The main module selection algorithm begins by combining pairs of $FMDGs$ using the $Combine$ process. The algorithm then replaces the original set by the newly combined $cFMDG$ and repeats until all application $FMDGs$ have been grouped together into one $cFMDG$. Since the QoS optimality and physical feasibility of $cFMDG$ depends on the bus network and the nature of the $Combine$ function, the initial permutation of the $FMDGs$ is not relevant here. To select the set of modules that satisfy the requirements of the $cFMDG$ with respect to the area constraints, we calculate $total_communication$ for each module m_i in the $cFMDG$ by summing the Gaussian weights of all the communication edges connected to m_i . We *Select* the physical instance of m_i from the module library M that can satisfy the computed statistical communication requirement using the smallest area.

```

Procedure Combine ( $G_1(V_1, E_1), G_2(V_2, E_2)$ )
   $\phi_1 = \sum e.weight \forall e \in E_1$  and  $\phi_2 = \sum e.weight \forall e \in E_2$ 
  Create new  $FMDG$   $G(V, E)$ 
   $V = V_1 \cup V_2$ 
  For Each  $u \in V$ 
    For Each  $v \in V$ 
       $e_1 = edge\ u \rightarrow v$  in  $E_1$  and  $e_2 = edge\ u \rightarrow v$  in  $E_2$ 
      If ( $e_1 \neq \emptyset$  or  $e_2 \neq \emptyset$ )
         $e = Create\ new\ g\text{-weighted}\ edge\ u \rightarrow v$  in  $E$ 
         $e.weight = Combine\_Gaussian(e_1, e_2, \phi_1, \phi_2)$ 

Procedure Select_Physical_Module_Sets
   $S = \emptyset, cFMDG = \emptyset$ 
  While ( $|FMDG| > 1$ )
    For  $i = 1, 3, \dots, |FMDG|$ 
       $cFMDG = cFMDG \cup Combine(FMDG_i, FMDG_{i+1})$ 
     $FMDG = cFMDG$ 
  For Each modules  $m_i \in cFMDG$ 
     $total\_communication = \sum e.weight \forall e$  connected to  $m_i$ 
     $s = Select(m_i, total\_communication)$ 
   $S = S \cup s$ 

```

Figure 2 – Design Space Analyzer Algorithms

4.2 Network Designer

To formally state the task of the network designer, we first define the CCG and the $hCCG$:

Definition 5: A Communication Connectivity Graph (CCG) is a undirected weighted graph with a set of nodes C representing cores and a set of edges E modeling the communication between two cores. The weight of an edge in a CCG represents the overlap that the communication between C_i and C_j has with the communication among all other cores.

Definition 6: A *Hyperedge Communication-Connectivity Graph* (*hCCG*) is an undirected hypergraph where a *Hypernode* hN is a collection of nodes in *CCG* and a *Bus Hyperedge* BhE is a hyperedge that connects at most M hNs . An *hCCG* formally defines a bus network, an hN represents a bus segment, and a BhE represents a bridge and its relation to adjacent buses [Dri00].

Problem: Balanced hCCG Partitioning.

Input: An *hCCG*; set of applications A ; timing constraint set T .

Question: Is there a balanced partitioning of *hCCG* into a set of hNs that results in a valid *hCCG* such that *QoS* for each application is satisfied?

To address this problem, we use an iterative improvement algorithm that relies on a constructed initial solution by: (i) assigning cores to hNs , and (ii) connecting hNs into $BhEs$ to form a valid *hCCG*. The maximum bus length and the sum of core areas are sufficient to evaluate β_r , the total number of hNs , and θ_r , the average number of cores per hN . Initially, we form an array Γ_r of empty hNs of size β_r . We sort $E_{ij} \in aFMDG$ by weights $\mu(E_{ij})$ in order to assign cores to elements of Γ_r . Starting from edges that have the highest weight $\mu(E_{ij})$, we assign cores C_i and C_j to the same element $hN_a \in \Gamma_r$. The next pair of nodes C_p and C_q ($\mu(E_{ij}) \geq \mu(E_{pq})$) are assigned to hN_b where $hN_a \neq hN_b$ in case when $p \neq i \neq q$ and $p \neq j \neq q$. In cases where $C_p \in hN_a$, C_q is assigned to hN_a except when $|hN_a| > \theta_r$. In that situation we consider the next edge from *cFMDG* with a lower weight. When all edges from *cFMDG* are processed, C_q is assigned to the hN_a with the smallest number of elements in the array Γ_r . When the percentage of assigned cores exceed the user specified parameter π , we shift our attention to E_{ij} where $\mu(E_{ij}) = \min_{p,q \in aFMDG_r} (w(E_{pq}))$. Corresponding cores C_i and C_j

are assigned to hN_a and hN_b , where $hN_a \neq hN_b$, so that in the initial bus network there will be one or more bridges on the data path between these two cores. hN_a and hN_b are selected according to the largest weights $\mu(E_{ik})$ and $\mu(E_{mj})$ where $k \neq j$ and $m \neq i$, if such edges exist in *cFMDG* and their cardinality is less than θ_r . Otherwise C_i and C_j are placed into hNs with the smallest cardinality from Γ_r . We continue to process edges in increasing weight order until all edges from *cFMDG* are processed. If there are cores that are left unassigned, they are placed in hNs with the smallest cardinality.

After assigning cores to elements of Γ_r , we form a $\beta_r \times \beta_r$, symmetric matrix F . Element $f_{pq} \in F$ represents attractive forces between hN_p and hN_q and is calculated using the formula $f_{pq} = \sum_{C_i \in hN_p} \sum_{C_j \in hN_q} E_{ij}$. Parameter δ_r represents the longest path (number of bridges) and is determined by the area constraint. Elements of F are distributed into categories $[1 \dots \delta_r]$ where the strongest forces are placed in category 1 and the weakest forces are placed in category δ_r . Starting from the strongest attractive force f_{pq} , we consider the following cases: (i) Both hN_p and hN_q are not assigned to any BhE : form a new BhE and assign hN_p and hN_q to this BhE ; (ii) hN_p is assigned to BhE while hN_q is not: assign hN_q to a BhE_j on distance from hN_p proportional to category of f_{pq} with the largest sum of forces between hNs from BhE_j and hN_q ; (iii) hN_p and hN_q are both assigned to a different $BhEs$: if the shortest path between hN_p and hN_q is larger than the category of f_{pq} (including the case of the path of length ∞) assign hN_q to a BhE_j on distance from hN_p proportional to category of f_{pq} with the largest sum of forces between hNs from BhE_j and hN_q . If hN_q is already assigned to two $BhEs$, repeat the search considering hN_p . If hN_p is also committed to two $BhEs$, go to next f_{pq} . During this procedure we maintain the validity of *hCCG* with a relaxation that allows the *hCCG* to be unconnected until the final step.

Procedure *Construct CCG*

For Each $r \in S$

Construct *initial_hCCG*

curPartition = *initial_hCCG*

While (*solutions unsatisfying*)

For Each $A_i \in A$

Verify timing for the current *hCCG*

List $BhEs$ and hNs violating constraints

$a = \text{random}()$

Case ($a < c_1$) *curPartition.SwapNodes*(*random*())

Case ($(a > c_1)$ and ($a < c_2$)) *curPartition.MoveNode*(*random*())

Case ($a > c_2$) *curPartition.MoveBhE*(*pseudo-random*())

If ($OF_{\text{after}} > OF_{\text{before}}$) accept move

Else reject move

Output the resulting *hCCG* to the floorplanning tool

Figure 3 – Network Designer Algorithm

4.2.1 Network Designer: Iterative improvement

The pseudo code of the iterative improvement algorithm used in the Network Designer is presented in Figure 3. The first step of the algorithm is to build the underlying *CCG*. The union of functional modules of all applications represents the nodes of *CCG*. The weight of the edge e_{ij} between modules M_i and M_j is calculated as $e_{ij} = \frac{1}{k'} \sum_{i=1}^k \alpha_i e_{ij}^{(i)}$ where $e_{ij}^{(i)}$ is a sum of elements from the row that corresponds to the edge between modules M_i and M_j from *CCT*. α_i is an empirical smoothening parameter for each *FMDG* and k' is calculated as the sum of α_i over all applications where $e_{ij}^{<i>}$ is non-zero.

A cause of congestion on a bus is an attempt by two or more cores to deliver data at the same time. One core will get access to the bus and the rest of those who attempt to access the bus have to stall, possibly causing undesirable QoS degradation. We use the objective function *OF* to formalize the trade-offs involved in the generation of efficient bus networks guided by: (i) cores that communicate frequently should be placed on the same bus, (ii) cores communicating through bridges should have data paths that are as short as possible, and (iii) congestion of bus segments should be minimized. Thus, *OF* can be stated as:

$$OF = \sum_{i=1}^{|CCG|} \left(\sum_{C_j, C_k \in hN_i} w(E_{i,j}) - \sum_{\forall C_j \in hN_i, C_k \in hCCG-hN_i} (w(E_{i,j}) \cdot \pi(C_j, C_k) \cdot \Phi_{j,k}) \right)$$

where $\pi(C_j, C_k)$ represents the length of the routing path between cores C_j and C_k (latency is modeled proportional to the number of bridges between C_j and C_k) and $\Phi_{j,k}$ is defined as the sum of bus segment workloads on $\pi(C_j, C_k)$. A workload $\phi(hN)$ of a bus segment is defined as the percentage of time when a communication exists on the bus segment, hN .

For each potential solution r from S the following algorithm is applied. After building of initial *hCCG*, timing and QoS violations are checked for every $A_i \in A$, by calculating workload of hNs for every A_i using the data from *FMDGs* and *CCTs*. For each violation of constraints, likelihood of selection of one of the moves of hN_i and the corresponding $BhE(s)$ increases. Alternation of *hCCG* is done with two categories of moves. In the first case, we distinguish two different move actions: (i) *SwapNodes*(C_i, hN_b, C_j, hN_j) node swapping between hypernodes, and (ii) *MoveNode*(C_i, hN_p, hN_j) node transfer from one hypernode hN_i to another hN_j . The second category of moves, *MoveBhE*(\cdot), modifies hyperedges and consequently the bus network structure. A valid *hCCG* must result in every move or it is canceled.

```

Procedure Latency_Guided_Floorplanner
   $T = T_0; M = C \cup BhE; F = \text{Initila\_Floorplan}(M)$ 
  While ( $T > T_{Final}$ )
    While (cumulative improvement  $\geq \sigma$ )
       $a = \text{random}(\text{LongestBus}, \text{ShortestBus})$ 
      Case( $a > \text{Const}$ ) :  $F.\text{Greedy\_Move}()$ 
      Case( $a \leq \text{Const}$ ) :  $F.\text{Enabling\_Move}()$ 
       $\text{Current\_Area} = \text{Area}(F)$ 
      For Each  $hN_i \in hCCG$ 
        If ( $\text{LengthOfBounding\_Box}(hN_i) > \lambda_{max}$ )
          Add  $BB_i$  to violated constraints list
       $\text{Current\_Cost} = \text{Cost}(\text{Current\_Area}, BB)$ 
      SA Decision: Accept or reject  $F$ 
      If  $\text{Current\_Cost} \geq \min(\text{Cost}(\Pi_i \in \Pi(K)))$ 
        Add  $F$  to  $\Pi(K)$ 
      Decrease temperature  $T$ 

```

Figure 4 – Pseudo code for Latency Guided Floorplanner

4.3 Latency Guided Floorplanner

The main task of the Floorplanner is to estimate the area requirements of the cores and check the implementation feasibility of the interconnection network. Formally, we state the problem as:

Problem: Latency-Guided Floorplanner

Input: Cores C ; Bridges BhE ; an $hCCG$; Bus constraint λ_{max} .

Question: Is there a floorplan F for the set of physical modules $C \cup BhE$ such that $\forall hN_i \in hCCG, \text{Length}(hN_i) \leq \lambda_{max}$?

At each iteration of the simulated annealing process (Figure 4), we calculate the minimum area required by each instance using the method presented in [Mur95]. During the search, the algorithm keeps track of the K best solutions encountered ($\Pi(K)$). For each solution $\Pi_i \in \Pi(K)$, we report the area and the violated bus constraints. In addition to the best solution instances found, we also report the percentage of instances that each bus violates the bus constraint. The standard simulated annealing process is augmented with solution transformation actions called moves. We define two types of moves: (i) *greedy* and (ii) *enabling*. The probability of performing each depends on the lengths of the shortest and the longest buses as shown in Figure 4. In the *greedy* move, we select the longest bus and try to improve the placement of its modules. We calculate the center-of-mass C_m of the bus by averaging the x and y coordinates of each external connection of the bus. We then calculate a force vector \mathbf{V} for the module that has the longest Manhattan distance from C_m . \mathbf{V} is used to update the sequence pair strings such that the selected module is moved in the direction of \mathbf{V} in proportion to $|\mathbf{V}|$. Similarly, in the *enabling* move, we select the shortest bus and try to relax the placement of its modules. We calculate the center-of-mass C_m and force vector \mathbf{V} as described above with the exception of selecting the module closest to C_m . We update the sequence pair strings such that the selected module moves in the opposite direction of \mathbf{V} in proportion to $|\mathbf{V}|$.

5. EXPERIMENTAL RESULTS

In this section we compare the performance of each application on the design developed using the *MMSOC* optimization approach to the performance of each application on an individually optimized design [Dri00]. The first column of the results in Table 1 lists the name of the application being considered. The next four columns list the number of cores, gates, buses, and bridges in the final design respectively. The next two columns list the number of iterations and the total run-time of the optimization task. Columns 8-11 list the total communication throughput and the median bus idle time ratio of each application for the single application system and the multi-application system respectively. As the data in Table 1 shows, the gate count of the *MMSOC* design is 40% larger than the worst case single application requirement. The bus count is 14.5% larger for the *MMSOC* design in comparison to the worst case single application system. Overall, after only 4 iterations, the *MMSOC* design can satisfy all the application requirements with minimal area and bus interconnection overheads.

6. CONCLUSION

To address the challenges arising in multi-application SOC integration, we presented a vertical system-design methodology that enables the design of SOC bus networks with floorplanning and approximate timing verification in early design stages. The design platform iterates system synthesis and floorplanning to build min-area floorplans that satisfy statistical time constraints of applications. We demonstrated the effectiveness of the bus network design approach using a multi-core designs selected from a multimedia benchmark suite. The results indicate that with minimal area and bus network overheads, a single SOC can be optimized to meet the demands of multiple applications effectively.

REFERENCES

- [Deg90] A. De Gloria, P. Faraboschi, "An evaluation system for application specific architectures," Microprogramming and Microarchitecture. p.80-9. x+299 pp. 1990.
- [Dri00] M. Drinic, D. Kirovski, S. Meguerdichian, M. Potkonjak, "Latency Guided On-Chip Bus Network Design". ICCAD'00, pp. 240, 2000.
- [Lee97] C. Lee, M. Potkonjak, W.H. Mangione-Smith, "Mediabench: A Tool For Evaluating And Synthesizing Multimedia And Communications Systems," Int. Symp. on Microarchitecture, pp.330-5, 1997.
- [Mad95] V.K. Madiseti, "Rapid Prototyping Of Application-Specific Signal Processors: Current Practice, Challenges, And Roadmap," Proc ICPP Workshop on Challenges for Parallel Processing. p. 96-103. 1995.
- [Mur95] H. Murata, K. Fujiyoshi, S. Nakatake, Y. Kajitani, "Rectangle-Packing-Based Module Placement", ICCAD'95, pp 472-479, 1995.
- [Sur96] D.R. Surma, E.H. Sha, "Static Communication Scheduling For Minimizing Collisions In Application Specific Parallel Systems," Int. Conf. on Application-Specific Systems, p. 240-9. 1996.
- [Syl99] D. Sylvester, and K. Keutzer. Rethinking deep-submicron circuit design. Computer, vol. 32, (no. 11), pp. 25-33, 1999.

Table 1 - Experimental Results: Single Application Optimization and Multi-Application Single SOC Design (T_p -Throughput)

Application Specification	Optimized Single Application Properties and Run-Time Statistics							MMSOC		
	Cores	Gates	Buses	Bridges	Synthesis loop		Solution properties		MMSOC Properties	
					Iterations	Time	T_p	Bus Idle	T_p	Bus Idle
DSP +crypto	13	1M	4	2	3	15min	1.46	0.45	1.39	0.55
GPP + communication	30	1.8M	11	4	4	1h	1.57	0.71	1.47	0.70
GPP+DSP	75	4.2M	21	7	5	5h	2.03	0.43	1.95	0.52
Communication + speech	100	5.4M	31	12	5	11h	3.11	0.79	2.92	0.74
DSP + speech	125	7.5M	41	15	5	17h	2.17	0.40	2.00	0.56
GPP + peripherals + comm	150	9.6M	47	18	7	21h	2.42	0.43	2.12	0.49
GPP + crypto +peripherals	200	15M	62	23	9	35h	3.16	0.67	2.97	0.76
Single SOC Multi-App Sys	240	21M	71	26	4+1	23h				