# Sharing and Binding for resource dominated circuits

Kashif Raza
*Hochschule Hamm-Lippstadt*
*Bachelor of Engineering - Electronic Engineering*
Lippstadt, Germany
kashif.raza@stud.hshl.de

***Abstract*—This paper explores the crucial role of resource sharing and binding in optimizing resource-dominated circuits. Resource sharing is a process in which each of the multiple non-concurrent operations is assigned a single resource. Resource binding allows the explicit assignment of resources to operations. These methods lead to a minimum area of the circuit, enhancing performance through a decrease in the number of required resources, such as functional units and registers. It includes both hierarchical and non-hierarchical sequencing graphs; such strategies reduce silicon area, interconnection complexity, and the number of multiplexers, hence helping efficient designs of high-performance circuits. It also brings in certain challenges: dealing with resource conflicts and scaling the solution to large designs. The paper also highlights challenges like managing resource conflicts and scaling solutions for complex designs. Future research should focus on developing efficient algorithms and integrating machine learning to better manage these conflicts. Continuous advancements in resource optimization are essential to meet the demands of next-generation circuit designs.**

***Index Terms*—Resource sharing, resource binding, resource-dominated circuits,TDM, SDM, CDM ,resource compatibility graph, resource conflict graph, sequencing graphs, register sharing, silicon area reduction, interconnection complexity, multiplexers, resource conflicts.**

## I. INTRODUCTION

**Resource sharing** maps a single resource to several operations in a circuit. The primary objective of resource sharing is to reduce the area of a circuit by allowing multiple non-concurrent operations to utilize the same hardware operator, thereby maximizing the usage of the resource efficiently. This approach is especially important in designs that have specified upper bounds on circuit areas or resource usage [1].

The **Resource binding** process explicitly defines how operations are mapped to resources [1]. A binding may imply that some resources are shared, but it does not necessarily imply resource sharing. Instead, resource binding, or partial binding, could be part of the original circuit specifications, in which some operations are forced to share resources because of predefined constraints. This implies that even if they do not specify any particular binding, constraints on the usage of the resource may implicitly indicate some degree of resource sharing [1].It can be applied to both scheduled and unscheduled sequencing graphs. In the case of a scheduled sequencing graph, the area has already been decided by resource usage. The role of binding and sharing here is to refine structural details for efficient connectivity synthesis [1]. For an unscheduled graph, it is not only resource usage that matters for the overall area and performance but also configuration aspects such as registers, wiring, steering logic, and control circuits [1]. Thus, the binding and sharing strategies have a high potential of affecting the area and performance of the circuit [1]. When considering **Resource-dominated circuits**, which are designed with stringent resource constraints, the area is often predetermined by the resources employed [1]. In such cases, the objective of binding and sharing is more to tighten the structural information since it again leads to efficient synthesis [1]. For more general circuits, however, the impact of binding and sharing extends beyond area optimization. . This has implications for performance metrics, latency, and throughput since it changes resource usage for things like registers, wiring, and control circuits [1]. There could be different scenarios with respect to the variety of operations and resources involved in binding and sharing. The simplest scenario is when operations of the same type share a resource, which is straightforward to implement [1]. Another complex scenario arises when several types of operations should be performed by one resource, able to realize the functions of several types. For example, the same Arithmetic Logic Unit could be used to perform subtraction and comparison. Imagine a mapping function characterizing each operation with the resource type that it requires, such as an ALU [1]. Further generalization occurs when an operation can be implemented using a variety of resource types, where the different resource types exhibit very different area and performance characteristics. For example, an addition operation might be performed by a number of different adder types, each offering different area-performance trade offs. In this flexible strategy, a designer will be able to pick the type of resource that is appropriate for the needs of a circuit .

In the following sections, this paper explores different aspects of sharing and managing resources. It starts by looking at how resource compatibility and conflict graphs are created and used to solve allocation problems. Then, it discusses resource sharing in both simple and complex sequencing graphs, noting the unique methods and challenges of each. The paper also covers how to share registers and reduce their use through

compatibility graphs and conflict resolution methods. Additionally, it examines multi-port memory binding, highlighting its benefits and ways to optimize memory port use in complex circuit designs. Finally, the paper looks at the challenges and future directions in this field, suggesting advanced algorithms and the use of machine learning to improve resource management in future circuits. [1].

## II. SHARING AND BINDING FOR RESOURCE DOMINATED CIRCUITS

Efficient sharing and binding methods have to be used to really optimize resource usage and system performance. Time-division multiplexing TDM provides non-overlapping time slots for the tasks, so there is little idle time and high throughput [3]. On the other hand, space-division multiplexing SDM allows parts of a resource to be used by different tasks executed concurrently, hence increasing parallelism [4]. Code-division multiplexing CDM uses a unique code sequence to run tasks on the same resource simultaneously, which is very common in communication systems. Hybrid approaches to TDM, SDM, and CDM adapt to dynamic resource demand to ensure flexible and robust resource utilization across changing workload conditions. Efficient resource binding, classified as static and dynamic, is essential for allocating tasks to resources. If performed at design time, static binding provides simple predictability with reduced overhead during operation; however, due to a lack of flexibility in the face of changing workload conditions, it often results in inefficient resource usage. In contrast, dynamic binding allocates resources at runtime pursuant to the current demand and status of a system. In light of this, it becomes highly adaptive and quite appropriate for real-time systems and cloud computing environments where tasks are variable and unpredictable. Dynamic binding, on the other hand, adds complexity to resource management and coordination, introducing overhead and latency in mismanagement cases [6].

In resource-dominated circuits, sharing and binding problems are approached using scheduled sequencing graphs $G_s(V, E)$ [1]. These graphs represent operations and their dependencies, excluding source and sink vertices treated as No-Operations. Therefore, the focus is on the set of operations $\{v_i \mid i = 1, 2, \ldots, n\}$ [1].

Operations can share the same resource if they are not concurrent and can be implemented by resources of the same type [1]. Operations meeting these criteria are considered compatible. Specifically, operations are non-concurrent if one begins only after the other completes, or if they are mutually exclusive alternatives in a decision branch [1]. Analyzing the sequencing graph helps determine the compatibility of operations for resource sharing [1].

A pivotal tool in this analysis is the **resource compatibility graph** $G^+(V, E)$, where the vertex set $V = \{v_i \mid i = 1, 2, \ldots, n_{ops}\}$ corresponds to operations, and the edge set $E = \{(v_i, v_j) \mid i, j = 1, 2, \ldots, n_{ops}\}$ denotes pairs of compatible operations [1]. This graph ensures that each pair of operations linked by an edge can share a resource without conflict [1].

The resource compatibility graph typically consists of disjoint components equal to the number of resource types. A maximal clique in this graph represents a set of operations that are all mutually compatible and can thus share a single resource . Minimizing the number of required resource instances involves partitioning the graph into the fewest cliques possible, known as the clique cover number $\kappa(G^+(V, E))$ . For example, in a scheduled sequencing graph in Fig.1 where operations may require either a multiplier or an Arithmetic Logic Unit (ALU), each with a single unit execution delay , the compatibility graph in Fig.2 might include cliques such as $\{v_1, v_3, v_7\}$ and $\{v_2, v_6, v_8\}$, representing sets of operations that can share the respective resource types. The clique cover number derived from these cliques determines the minimal number of resource instances needed. [1]

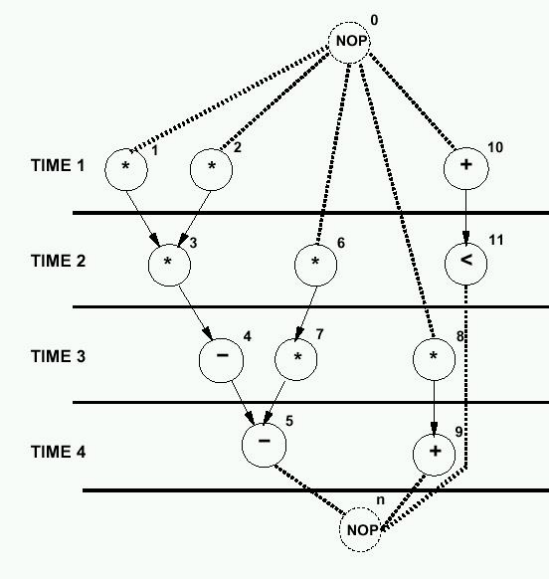


Fig. 1. Scheduled sequencing Graph [1]

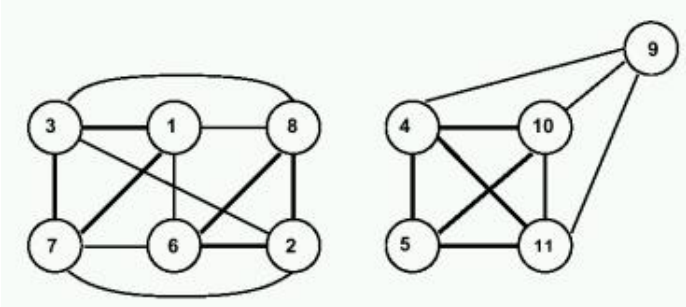


Fig. 2. Compatibility Graph [1]

Another dimension of the problem includes **resource conflict graphs**, underscoring the pairs of operations that cannot

coexist because of resource constraints [1]. Knowledge of the conflicts enables making resource allocation decisions and, at the same time, has the potential to point out the need for extra resources in order to resolve the conflict [1].

In resource-dominated circuits, high resource utilization requires efficient treatment of resource conflicts. In this respect, a resource conflict graph $G_{-}(V, E)$ is a helpful tool. Every vertex in the graph corresponds to an operation. An edge between two vertices means the two corresponding operations end up being allocated the same resource, which means they are not compatible. Hence, the conflict graph is the inverse of a resource compatibility graph.While the compatibility graph shows which pairs of operations may share a resource, the conflict graph shows which ones may not share a resource. Indeed, this duality often makes it easier to determine resource allocation by making the conflicts explicit. Any set of operations that may be assigned the same resource will constitute an independent set in the conflict graph; no two vertices have an edge between them. The goal will thus be to color the conflict graph with a minimum number of colors, since each of the colors corresponds to a different resource. This minimum coloring is called the chromatic number $\chi(G_{-}(V, E))$ [1], which will mean exactly the minimum number of different resources required. In practice, various kinds of operations (e.g., multipliers and ALUs) typically clash with each other; hence, they usually belong to different conflict sub graphs. Ultimately, all these graphs can be combined into a single global conflict graph containing all operation types, augmented by additional edges between operations of different kinds, modeling their real conflicts.

Consider any scheduled sequencing graph, with operations requiring either a multiplier or an ALU as shown in Fig. 3, and each with a unit execution delay of a single unit. The conflict graph corresponding to such a situation will have sets of operations that are not executable concurrently on the same resource. For instance, the independent sets $\{v_1, v_3, v_7\}$ and $\{v_4, v_5, v_{10}, v_{11}\}$ denote subsets of operations that possibly share resources and nevertheless ensure no conflicts. Coloring these sets appropriately will demonstrate that four different resources are enough to perform all operations without conflict. The solution to the resource sharing problem using conflict graphs resorts to the vertex coloring problem [1], which has been comprehensively surveyed. Even though heuristic and exact methods provide a practical solution for special classes of circuits, finding the exact solution can be computationally very expensive for big graphs. Such attention to compatibility and conflict issues can help designers optimize resource sharing and binding strategies for resource-dominated circuits as effectively as possible, improving both efficiency and performance [1].

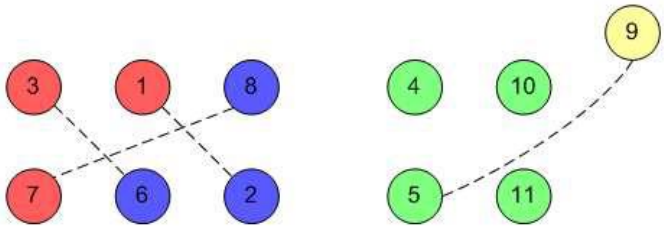


Fig. 3. Conflict graph for multiplier and ALU [2]

## III. RESOURCE SHARING IN NON-HIERARCHICAL SEQUENCING GRAPHS

In non-hierarchical sequencing graphs, each path from source to sink represents a parallel stream of operations without alternative paths. These graphs help in visualizing and managing the execution order and resource allocation for various operations in a circuit design. Each operation in these graphs is denoted by a resource type $T(v_i)$, where $i$ ranges from 1 to the total number of operations, $n_{ops}$ [1]. Additionally, each operation has a start time $t_i$ and an execution delay $d_i$, meaning the operation $v_i$ begins at $t_i$ and finishes at $t_i+d_i-1$ [1]. In this context, data-dependent delays are not considered as the sequencing graph assumes scheduled operations [1].

Two operations are considered compatible if they can be implemented by the same resource type and do not overlap in time [1]. Thus, the compatibility graph $G_{+}(V, E)$ can be defined with edges,

$$E = \{(v_i, v_j) \mid T(v_i) = T(v_j) \text{ and } (t_i+d_i \leq t_j) \text{ or } (t_j+d_j \leq t_i)\}$$

ensuring that only non-concurrent operations with the same resource type are connected [1]. Constructing this graph involves traversing the sequencing graph and checking each operation's compatibility based on start times and execution intervals, which can be done in $O(|V|^2)$ time [1].

For example, consider a scheduled sequencing graph like the one in Fig. 1, where each operation has a unit execution delay. Suppose we are considering an operation $v_1$ which has a start time of $t_1 = 1$; any operation of the same resource type but with a start time of $t_j \geq 2$ will be compatible with $v_1$ [1]. The operations starting time intervals less than or equal to 1 cannot be in compatibility with $v_1$, thus forming edges in the compatibility graph with vertices corresponding to compatible operations like $v_3$, $v_6$, $v_7$, and $v_8$ [1]. A solution for the sharing problem can be derived by constructing a directed graph in which orientations by start times are assigned and edges have the following form: $(v_1, v_3)$, $(v_1, v_6)$, $(v_1, v_7)$, and $(v_1, v_8)$ as shown in [1]. This graph will help in clearly defining the operation order and the possible conflicts. It follows from the transitive orientation of these relations that the compatibility graph is a comparability graph, as shown in Figure 3 [1].

We can also have detailed conflict graphs for each type of resource. Such graphs form when execution intervals for every operation are represented in the form $[t_i, t_i + d_i - 1]$ [1]. Any

edges within the conflict graph take the meaning of overlaps in such intervals and denote conflicts. The process of finding the minimum coloring in an interval graph can be done in polynomial time by algorithms like LEFT_EDGE [1]. This is normally done since it is assumed that the resources will be of a single type only, considering conflict graphs for each type separately [1]. For instance, in the sequencing graph with unit execution delay, as shown in Fig. 4 [1], the conflict graphs for every type of resource would expose intersections among the intervals, such as $\{v_1, v_2\}$ with multipliers and $\{v_5, v_9\}$ with ALUs. This will reduce the overall conflict graph to only one type of resource, thus making it easier to analyze and optimize.

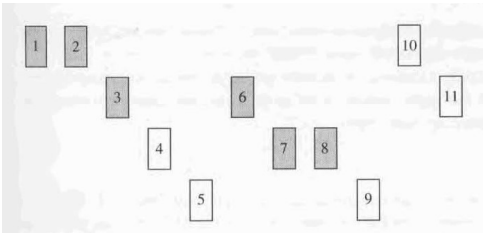


Fig. 4. Interval corresponding to conflict graph [1]

The provided Fig.5 illustrates a bound sequencing graph. Here, operations are assigned specific resources and scheduled across different time slots. Each resource is denoted by a pair of indices indicating the resource type and its instance number. For instance, $\beta(v_1) = (1, 1)$ means that operation $v_1$ uses the first instance of resource type 1 (a multiplier). In the example depicted in the Fig.5 , the bound sequencing graph shows the scheduling of operations across four time slots. Each operation is color-coded based on its assigned resource, with time slots clearly marked:

- At Time 1, the execution includes $v_1$ and $v_2$ operations on the first instance of respective resources.
- At Time 2, the operations $v_3$, $v_6$, and $v_7$ are executed using different resources.
- At Time 3, operations $v_4$ and $v_8$ are scheduled.
- At Time 4, operations $v_5$, $v_9$, $v_{10}$, and $v_{11}$ are executed.

This scheduling ensures that no two operations that share the same resource overlap, thus adhering to the constraints imposed by the compatibility graph. The table provides detailed binding information for each operation:

$$\beta(v_1) = (1, 1)$$
$$\beta(v_2) = (1, 2)$$
$$\beta(v_3) = (1, 1)$$
$$\beta(v_4) = (2, 1)$$
$$\beta(v_5) = (2, 1)$$
$$\beta(v_6) = (1, 2)$$
$$\beta(v_7) = (1, 1)$$
$$\beta(v_8) = (1, 2)$$
$$\beta(v_9) = (2, 2)$$
$$\beta(v_{10}) = (2, 1)$$
$$\beta(v_{11}) = (2, 1)$$

This binding ensures that operations are mapped to available resources efficiently, minimizing resource usage while meeting all operational constraints. By employing these techniques, designers can optimize resource sharing in non-hierarchical sequencing graphs, ensuring efficient and conflict-free execution of operations in resource-dominated circuits.

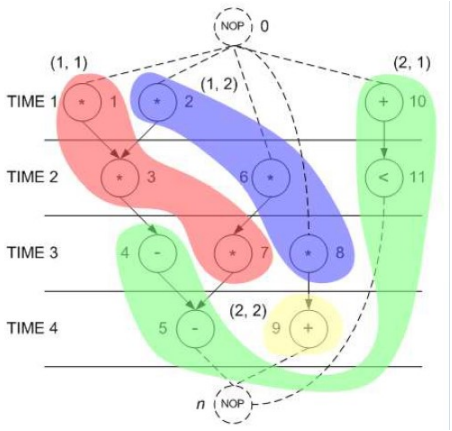


Fig. 5. Bound sequencing graph [2]

## IV. RESOURCE SHARING IN HIERARCHICAL SEQUENCING GRAPHS

Resource sharing in hierarchical sequencing graphs is more realistic because of their multi-level structure. In non-hierarchical graphs, the sharing of resources within any sequencing entity is independent, whereas in hierarchical graphs, the sharing must be among different hierarchy levels. This enables efficient usage of resources that are otherwise confined to one level. In hierarchical sequencing graphs, there can be a variety of operational modules or functions at different levels. Therefore, resource sharing needs to take into account such different levels. The naive approach is to share resources independently within each entity. However, this turns out to be too restrictive because it does not allow sharing among different hierarchy levels. Instead, we would provide support for resource sharing across these hierarchy levels to achieve whole-graph optimal resource usage [1]. For instance, let us consider a model consisting of two operations: addition followed by multiplication. Here, while calling this model from a higher-level model, checking the compatibility is not only searched within the same level but also between different levels. For example, in the case where a higher-level model contains another set of operations that may overlap in execution time with lower-level model operations, resource sharing has to take this overlap into consideration in order to avoid possible conflicts [1]. This means that, in practical terms, two compatible operations from different models share the same resource. For example, given that model m1 calls model a at time 1 and model b at time 4, all these operations need to be checked for compatibility in order that resource sharing

can be allowed. A more detailed example in Fig.6 involves a scenario where multiple calls to the same model occur. If model m2 calls model a and model b in overlapping intervals, we cannot assume that operations in a and b are conflict-free. This necessitates a thorough analysis of execution intervals to identify potential conflicts and ensure efficient resource allocation [1].

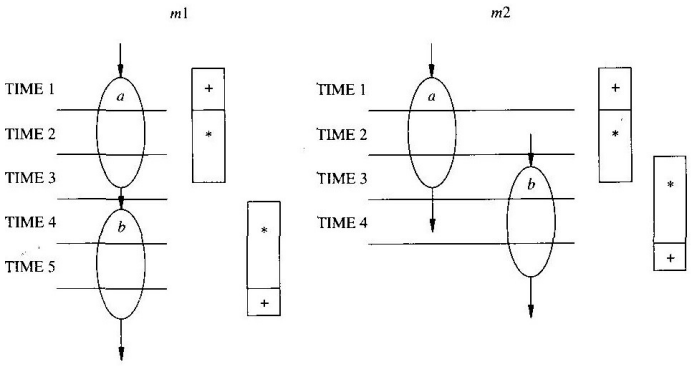


Fig. 6. Hierarchical conflicts and compatibility [1]

One method of determining the compatibility of operations across the various levels is to "flatten" the hierarchy. This may be done explicitly, with link-vertices replaced by corresponding models. The same results may also be obtained implicitly by computing execution intervals for each operation relative to the root model. In this way, both interaction and resource sharing are more easily handled since all possible conflicts, and hence all compatibilities, are exposed [1]. The graphs were thus generated. can have different properties depending on whether models are called once or multiple times. In multi-call scenarios, Particular care has to be taken to handle resources efficiently. For instance, iterative constructs inside loops need resources, that need to be distributed over multiple incarnations of the same operation, which can be difficult to handle without appropriate hierarchical flattening [1].

## V. REGISTER SHARING

One of the main techniques that is targeted at optimizing is register sharing that is the most effective method of resource usage. It can be described as a procedure for allocating registers to variables, capturing their values, and reusing them efficiently throughout their entire lifetime.. Every variable may be said to have a lifetime, which starts when the variable is generated as output from an operation and ends when the variable is last used as input for another operation [2]. Variables with multiple assignments in the same model are treated as aliases for processing variables with multiple assignments in the same model. It assumes that each variable has a single, fairly general, continuous lifetime with respect to any sequencing graph in which it appears [2]. Data dependencies and other flow-of-control dependencies, resulting from branching and iterative constructs in the circuit, often determine its lifetime, which varies in duration [2]. The simplest method, that of assigning a different register to each variable, is very inefficient. Clearly, variables active at different times or under different conditions may share the same register, since their lifetimes do not overlap. Conversely, such definitions are considered compatible [2]. It follows from the above that sharing would hence be based on register compatibility and conflict graphs, as it is used for other resources [2]. This can be performed either by partitioning the compatibility graph into the minimum number of cliques or by obtaining the minimum coloring for the conflict graph, each aiming to minimize the number of registers. This approach will ensure that the registers are optimally shared among variables that are not going to interfere with each other's variables' lifetime.

Consider the graph of sequencing shown in Fig. 7 (a), which contains several intermediate variables $z_i$ (where $i = 1, 2, \ldots, 6$) that must be stored in registers as depicted in Fig. 7 (b). Graphically, the lifetimes of variables are shown, indicating the periods for which each register is required, as illustrated in Fig. 7 (c). Some of these variables have conflicting lifetimes, as depicted in a conflict graph. It turns out that a conflict graph will need only two registers to store all the variables, as in Fig. 7 (d), corresponding to its chromatic number by determining its minimum coloring. [2]

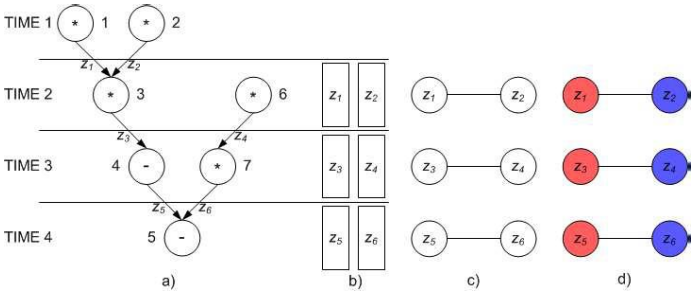


Fig. 7. Non-Hierarchical Scheduling Graphs [2]

**Register Sharing in Iterative Bodies** When dealing with sequencing models that include iterative bodies, certain variables, such as loop-counter variables, live across the iteration boundaries. The lifetime of such a variable will then exhibit a cyclic nature that can be captured rather precisely by a circular-arc conflict graph. This approach assists in modeling the cyclic nature of variable lifetimes. The problem is how to minimize, as much as possible, the number of registers that will be required. This can be formulated as a minimum coloring problem for the circular-arc conflict graph. Unfortunately, this problem is inherently complex and thus intractable for large graphs, making efficient solutions hardly achievable [2].

## VI. MULTI-PORT MEMORY BINDING

In the circuit design, multi-port memory binding is used to optimize memory resource utilization. Several registers are combined into one memory unit here, reducing the total number of design spaces and chip areas. Since interconnections, multiplexers, and the input required to multiplexers in the

data path are reduced, the latter becomes more streamlined [2]. Also, the number of individual design modules decreases, which normally makes the final design easier to test [2]. However, it is important to ensure that such grouping of registers does not negatively impact the operating frequency of the design [2].

Consider a memory with multiple read ($r$) and write ($w$) ports, where each port takes one cycle per access. This type of memory serves as a large general-purpose register file that can hold all the required data [2]. When every variable accesses this memory repeatedly through the exact equivalent port, then the memory binding problem gets reduced to the problem of assigning variables to ports [2]. In this way, we can use functional resource binding techniques to such memory ports and consider them interface resources [2]. By definition, the minimum required number of memory ports can be computed as

$$r + w = \max_{1 \le l \le \lambda} \sum_{i=1}^{n_{var}} x_{il}$$

In this formula, $n_{var}$ represents the total number of variables, and $X$ is a set of binary constants determined by the scheduling process [1]. The variable $x_{il}$ is 1 if the variable $i$ is accessed at step $l$, indicating the maximum number of concurrent accesses. This value determines the minimum number of ports required [2]. We build conflict graphs for variables that are to be read or written, and the coloring for these graphs gives us the minimum number of ports necessary to bind variables to the memory read and write ports [2]. In fact, a minimum coloring for the read variable conflict graph gives us the number of read ports needed, and a minimum coloring for the write variable conflict graph shows us the minimal number of write ports needed [2].

## VII. BENEFITS, CHALLENGES AND APPLICATIONS

Resource-dominated circuits benefit significantly from optimized resource sharing and binding strategies. These optimizations have the effect of reducing the total area and power consumption by lowering the number of needed resources, for instance, functional units and registers. For instance, resource sharing of non-overlapping operations and efficient variable binding to registers can decrease silicon area for setting up, interconnect complexities, and accordingly the number of multiplexers required. Inherent in such optimizations are simpler, more modular designs with easier testability and reliability. This ability to share resources across different hierarchical levels further optimizes resource utilization and allows high-performance designs to meet the most stringent design constraints and functional requirements [2].

Even with advantages in resource-dominated circuits, a number of challenges still remain. One of the major challenges is the complexity of accurately modeling and managing resource conflicts in circuits that have iterative constructs or hierarchical designs. This means that optimal resource sharing does not affect the operating frequency of the circuit; sophisticated algorithms and tools are required. Another problem is scaling, because as designs become more complex, the computational effort required to find the best solution for resource sharing and binding increases significantly. Future research directions for this work include developing more heuristic and formal algorithms with higher efficiency for handling increasing design size and complexity. Another area is improving current automated design tools to better integrate resource optimization techniques, as well as applying machine learning approaches for dynamic prediction and management of resource disputes. Moreover, technology improvements in new circuit designs—such as designing flexible and more efficient memory structures—can potentially realize resource sharing and binding fully [2]. The practical application of sharing and binding techniques in resource-dominated circuits is realized in several advanced systems, such as FPGA-based systems, multi-core processors, and NoC architectures.

This is where the flexibility of programmable logic blocks in **FPGA-based systems** comes in: resources can be variably changed dynamically according to the functionalities desired by an application. In these systems, static binding was applied at design time for any fixed bindings of logic blocks and interconnections that needed to be in place for predictable tasks. Instead, dynamic binding will allow reconfiguration at runtime, matching the various workloads in place for optimal performance benefits. For example, in digital signal processing applications, the FPGAs will act as dynamically reconfigured processing elements by using hardware resources to handle different signal processing requirements.[7]

These techniques of sharing and binding are widely exploited in optimizing both performance and energy in **Multi-core processors**. Static binding, on the other hand, will bind certain tasks to bound cores, therefore reducing scheduling overhead and context switching in a multi-core processor. With dynamic binding, it is therefore possible for workload demands to be scheduled by the OS on different cores, therefore improving load balancing and corresponding reductions in power consumption. Therefore, sophisticated algorithms for task allocation and strategies in resource mapping form an integral function to ensure that multi-core system management works effectively through core utilization by allocating higher-priority tasks to the cores.[9]

**NoC architectures**, which are intended to solve communication issues in multicore systems and SoC designs, enforce newly advanced sharing and binding methods to control data flows between the cores and other resources available on the chip. Static binding will instance fixed paths of communication for periodic data flows, as in NoC, whereas dynamic binding will enable the same packets to go through different paths depending on the state of the network traffic and priority levels. We believe that dynamic resource binding would improve data throughput by reducing bottlenecks. Further, time division multiplexing, or TDM, is also one of the common techniques used in NoC to share the communication channel among several data streams, whereby network bandwidth is being used up fully and efficiently.[8]

## VIII. CONCLUSION

The paper demonstrates the importance of efficient resource sharing and binding in resource-dominated circuits. Such optimized strategies can dramatically reduce the circuit's total area and power dissipation, resulting in cost-effective circuit design solutions with improved performance metrics. Resource sharing and binding reduce not only the silicon area needed for implementation, but also the interconnection complexity and the number of multiplexers required. This leads to easier, more modular designs that are simpler to test and more reliable. Experimental results demonstrate that the ability to manipulate resource sharing across different hierarchical levels allows for maximum resource utilization, resulting in high-performance designs that meet the most stringent design constraints and functional requirements. Despite all of the valuable benefits it offers, challenges still exist when it comes to accurately modeling resource conflicts and correctly handling them in circuits with iterative constructs or hierarchical designs. Compared to the operating frequency increase, it may involve complex algorithms and tools to ensure optimal resource sharing with no performance impact on the circuit. Furthermore, the increasing complexity of circuit designs significantly increases the required computer effort to find an optimal solution for resource sharing and binding. These include further developments of heuristic and exact algorithms that will be more effective for larger and more complex designs, better integration within the automated design tools of resource optimization techniques, and exploiting machine learning approaches for dynamic resource conflict prediction and handling. Advancing technology in the form of more versatile and efficient memory structures, for example, is another prime requirement for further harvesting the resource sharing and binding benefits in next-generation circuit designs.

## IX. REFERENCES

### REFERENCES


[1] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994, pp. 229-225, Chapter 6. [Accessed: 20-Jun-2024].

[2] PresentationPoint, "High-Level Synthesis: Introduction, Scheduling, Resource Sharing and Binding," *Projektovanje složenih digitalnih sistema, Lecture IX-X*, 2018. [Accessed: 22-Jun-2024].

[3] J. Smith, "Resource Sharing in Computer Networks," Heavy Coding, 2023. [Online]. Available: https://heavycoding.com/resource-sharing-in-computer-networks/. [Accessed: 29-Jun-2024].

[4] A. Brown, "Sharing resources — Computer Networking: Principles, Protocols and Practice," *Computer Networking*, 2023. [Online]. Available: https://www.computer-networking.info/principles/sharing.html. [Accessed: 29-Jun-2024].

[5] Wikipedia, "Shared resource," [Online]. Available: https://en.wikipedia.org/wiki/Shared resource. [Accessed: 29-Jun-2024].

[6] A. Brown, "Resource Management in Distributed Systems," GeeksforGeeks, 2023. [Online]. Available: https://www.geeksforgeeks.org/resource-management-in-distributed-system/. [Accessed: 29-Jun-2024].

[7] V. Betz and J. Rose, "FPGA Architecture: Survey and Challenges," *IEEE Transactions on Very Large-Scale Integration (VLSI) Systems*, vol. 6, no. 2, pp. 159-172, 1998.

[8] L. Benini and G. De Micheli, "Networks on Chips: A New SoC Paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70-78, 2002.

[9] Y. Wang, D. Zhu, and P. Mishra, "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-Time Multi-Core Systems," *IEEE Transactions on Embedded Computing Systems*, vol. 14, no. 3, pp. 1-21, 2015.


## X. DECLARATION OF ORIGINALITY

I hereby declare that I myself have written this paper, and that I have not used any external sources other than those mentioned. Anything borrowed from a different source, whether phraseology or idea, is duly acknowledged. I further declare that this paper has not previously been submitted for any course or examination, in this form or in a similar version.

Kashif Raza
Lippstadt, 01.07.2024