# On-Chip Resource Allocation Algorithm for Reconfigurable Computing Machines

Kagan Agun
Illinois Institute of Technology
Computer Science Department
agunsal@charlie.iit.edu

Morris Chang
Illinois Institute of Technology
Computer Science Department
chang@charlie.iit.edu

## Abstract

*Recent advances in Field-Programmable Gate Arrays (FPGAs) allow more designs to fit into a single device. Partial configuration at run time would allow multiple independent applications to share chip resources such as logic blocks. Configuration sequence of applications is unpredictable, therefore, resource allocation has to be done on-the-fly. This paper introduces chip level resource allocation algorithm that is implemented in the hardware. This approach provides fast allocation over the software approaches assuming there can be sufficient Operating System (OS) support. Our approach also makes feasible self-configuration through on-chip resource manager.*

## 1 Introduction

Since the emergence of configurable devices, configurable computing has demonstrated significant success in mapping computational intensive applications to hardware. Networking applications, portable products, embedded controller including general purpose processors are some of the examples that have benefited from reconfigurable computing systems. Advanced submicron technologies enable a complete design on a single chip. Increasing density and speed of FPGAs lead to the adoption of IP core based System-on-Chip (SoC) designs as a new paradigm. Embedding microprocessors into the SoCs is the result of these technological advances. Moreover, OS Support and new futures to make FPGAs more efficient and flexible are on the horizon. Now, it is time to explore new paradigms to determine the emerging trends in reconfigurable computing.

As FPGAs become larger and faster, large number of components can be fitted into them. Using software algorithms, design tools can map and place a design into a target device. Configuration of the targetted device can be done at run-time (known as Run-Time reconfiguration, RTR). The ability to configure parts of the device without disturbing the rest (partial configuration) would allow greater flexibility. Sometimes this process may require swapping of components in and out due to size and location restrictions. These partial configurations have to be done with pre-scheduled orders and into the pre-allocated areas.

Configurable computing is also dedicated to a specific application. The whole device can not be used by other applications during the runtime, because current design trend doesn't support multiple applications for a single configurable device. On the other hand, applications often do not use all the resource on the chip (depends on the size of the chip). The unused partition of FPGA can not be configured for other applications.

It is desirable for related applications to share the hardware platforms and components. Sharing whole configurable resource or part of it can be accomplished through operating system task scheduling algorithms as long as the designs do not overlap. Again all designs have to be pre-allocated and pre-scheduled. Once a design is mapped, it can be scheduled dynamically for the same pre-allocated location during the run time. However, making placement decision on-the-fly requires careful resource management. Applications need to know the available resources that can be used to map by their designs.

Chip resources can be allocated independently among the applications through software approaches. However, on-chip resource allocation is the most efficient way to accomplish this task. Giving the resource allocation functions to configurable computing device may accomplish two major advantages over the software approaches.First, allocation is much faster than its software counterparts. Second, it makes feasible to apply chip level placement and routing on-the-fly. In this paper, we describe an on-chip resource allocation approach, that may lead to efficient component configuration and high component reuse through on-chip-resource manager. Our proposed resource allocation scheme achieves fast resource sharing and truly on-the-fly configurations.

The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 details the resource allocation algorithm. Section 4 describes the resource allocation management and shows the results of simulation. The last section presents the conclusion of this paper.

## 2 Related Work

Compile-Time Configuration (CTC) is the simplest and the most commonly used approach for configurable computing machine implementation. All of the chip resources are dedicated to design tools which arrange the logic cells on FPGAs with various cell placement techniques[10] e.g. simulated annealing, placement by partitioning, etc. Once the configuration is loaded, it will remain constant in the FPGA through out the application lifetime. First attempt to overcome this limited device reuse problem is Run-Time-Configuration (RTR). After completing the placement of several designs for the same targetted device, RTR applications uses dynamic allocation scheme that re-allocates configurable hardware through downloading configuration data of these designs at run-time[1,3]. Sequential configuration of FPGA has been implemented successfully by several systems. A more flexible approach than sequential configuration is configuring subsets of the logic in the reconfigurable device e.g RRANN-2[2]. This approach is known as dynamic partial configuration. Placement algorithms used for compile and run time configurations are optimal but slow due to software implementation. In addition, they don't support resource sharing for multitasking.

The Dynamic Instruction Set Computer (DISC) [4] was developed to support demand-driven modifications of its instruction set takes advantage of partial FPGA configuration to implement dynamic instruction paging. DISC instructions are designed once for multiple locations on the FPGA to overcome instruction overlapping. DISC is the only application that schedules these precompiled instructions through configuration libraries. While DISC is executing on reconfigurable machine, Programmable Reduce Instruction Set Computers, (PRISC) [5] introduces Programmable Functional Units (PFU) in the context of RISC datapath in addition to RISC hard core. Applications can add some custom instructions into the PFUs before starting execution. Design of an instruction is pretty much the same with compile time configuration. PRISC controller chooses which FPU is going to implement these instructions.

Self-modifying, On-the-fly Alterable Logic (SONAL) [6] as a coprocessor, has a similar approach for dynamic configuration but it requires hardware enhancement to enable internal configuration loading. An extra memory, small SRAM, added to SONAL keeps as many configurations to load as needed. Configuration data can be downloaded into the extra memory without disturbing operation on the FPGA. A controller manages the configuration based on order of coming internal or external requests.

Recently on-line placement algorithms have been introduced to make resource sharing of configurable computing machines among the applications and provide fast placement. These algorithms have been implemented in software. First fit, best fit, bottom-left and two dimensional versions of these algorithms are proposed and tested for variety of the applications[7]. Furthermore, local repacking, order compaction, and genetic algorithm approaches are represented to find free resource on partially reconfigurable FPGAs[8]. In the local repacking, first a quadtree decomposition of the free space is used to find a subarray which contains sufficient cells, then two-dimensional strip packing method is used for repacking. In the ordered compaction method, a favourable location is allocated for a task. if there are tasks in this location, these are slided in one direction. A genetic algorithm is a probabilistic search method. Possible solutions are listed in a data structure for evolution. These software approaches are used by a controller application to dynamically schedule tasks.

## 3 Allocation - Deallocation Algorithms

Dynamic reconfiguration is usually performed by software systems. These systems simply provide static scheduling through dynamic downloading of configuration of preallocated cells of an FPGA. To configure different cells, hardware design has to recompiled for these locations to be re-routed and re-placed. Resource sharing is a way of allocating FPGA logic resource so that each application executes independently. In that case of on-line placement decision, resource allocation must be efficient. All of the published allocation algorithms, we know of, are based on software implementation.

The problem of logic-block allocation is similar to that of memory allocation which has been studied for years. Memory allocation deals with the memory cells that are arranged in one dimensional array fashion. Logic block arrangement in most FPGAs are arranged in two dimensional matrix architecture. Cell allocation in such systems can be done in both ways, one dimensional or two dimensional array. Hardware implementation of cell allocation algorithm based on optimized binary buddy system[11] takes advantage of the speed of a pure combinational-logic system. This approach also helps the development of on-chip resource management systems which may support self configuration and increase flexible component reuse..

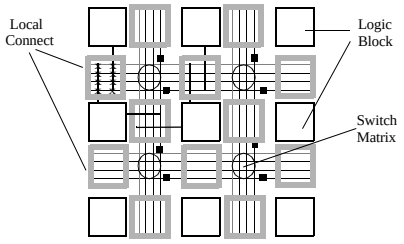
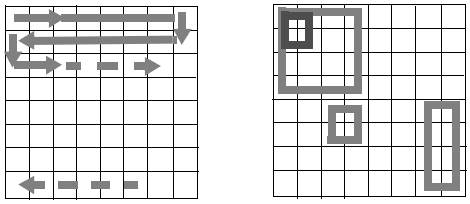



**Figure 1: The most common FPGA architecture (Sea of Gates)**

FPGAs consist of arrays of logic modules that provide gate level functionality and interconnect resources. The logic blocks are cable to implement 4-input 2-output simple

boolean logic. Two type of logic blocks have been favored in commercial FPGAs: Lookup Tables (LUTs) and multiplexers. Interconnect resources provide routing paths to connect the inputs and outputs of the logic blocks and I/O pins of device onto each other. Versatility of the FPGAs depend on the interconnection structure in the device. Figure 1. shows the most common structure of the FPGAs

In this paper, we introduce cell allocation algorithm which is drived from Binary Buddy System. Operation consists of finding free locations and marking free locations of requested size. A bit map is used to represent the status of free or used logic blocks (1 for used, 0 for free). The operation of the proposed allocation algorithm will be implemented in the hardware. The one dimensional array case will be addressed first. Then, implementation of one dimensional array approach will be demonstrated. Figure 2 represent three basic allocation schemes. These are *array allocation, matrix allocation*. These two algorithms can be implemented in hardware. In the next subsection, we will examine these algorithms.



(a) Array (b) Matrix

**Figure 2: Logic block allocation schemes**

### 3.1 One cell or continues array of cells allocation

Because of the similarities between the memory and logic blocks, memory allocation algorithms can be applied to FPGAs. Among these algorithms, a binary buddy system is feasible and fast because of its hardware applicability. Allocation is done using hardware maintained binary tree. In this algorithm, an array of connected cells (bit array) represents the status of cells such as 1 is occupied, 0 is free. This bit-array forms the binary tree. Allocation can be done by simply setting bits in the bit array, similarly deallocation can be done by resetting the bits in the bit-array. Therefore same logic can be applied to both allocation and deallocation schemes in the marking phase. This algorithm is implemented in pure combinational logic.

Binary buddy system allocates cells in sizes of powers of 2. In order to allocate free logic blocks requested by the application, allocation system must do three things: a) determine the availability of size in power of 2, if so, b) find the beginning address of that free space, c) mark the corresponded bits to complete allocation. Figure 3 illustrates these three operations in bit-array form. In this example, each bit represents the smallest blocks of cells that can be allocated at any given time.

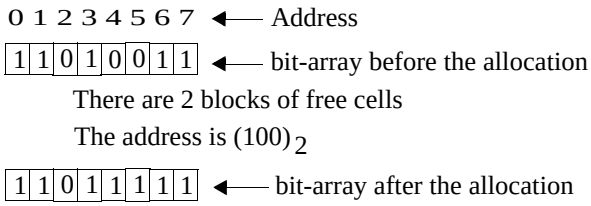



**Figure 3: Example for allocation of 2 blocks.**

In this approach, the OR-gate tree determines whether there are enough free cells of requested size to allocate. Anding output of OR gates at the same level gives the availability of requested size of blocks. If one of the nodes in the same level has zero value which shows the availability of requested size, "and" results of this level will be zero. For example, at level 2, each node, b and c, represent $2^2$ cells and anding b and c provides availability of free cells of this size. Figure 4a demonstrates free cell availability.

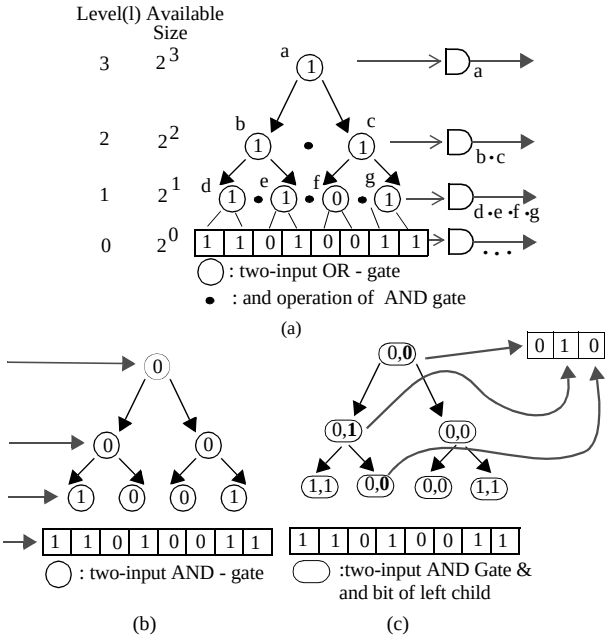



**Figure 4: Cell allocation algorithm (a) OR-gate tree, (b) AND-gate tree, (c) Propagated AND-gate tree**

Then to find first address of the free cell(s), OR-gate results of this level transfers to the AND-gate tree which passes the availability of cell(s) at the required level upward toward the root. Suppose the AND-gate tree determines the address of the first zero at level l of the OR-gate tree. However, in the search of free cells, it can be returned to the parent node to check the next leaf node, therefore AND-gate tree needs a backtracking algorithm to calculate the exact address of free cell(s). By propagating left child of each AND-node to the parent node, this need is avoided and target address is formed via these propagated bits (Figure

4b,c). But this address is a relative address which is the beginning of $2^{level}$ size cells. Therefore this address should be multiplied with $2^{level}$ or simple shifted to right with *level*. Figure 4 demonstrates the finding address of first free cell at the zero level.

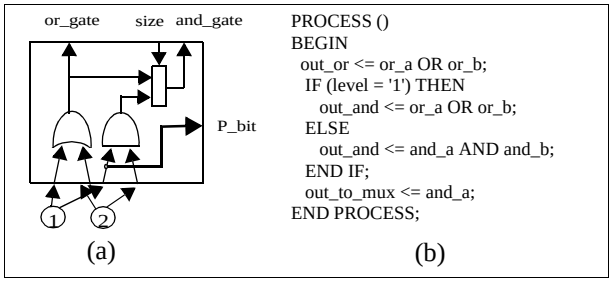



**Figure 5: One bit free cell locator**

In the Figure 5, one node slice cell locator is shown. Combining the AND-gate tree and the OR-gate tree reduces the design size and complexity and increases reuse. *Size* signal represents the level and allows the AND-gate tree to use the OR-gate results of the same level. Figure 6 details the complete implementation of an 8 cell locator.

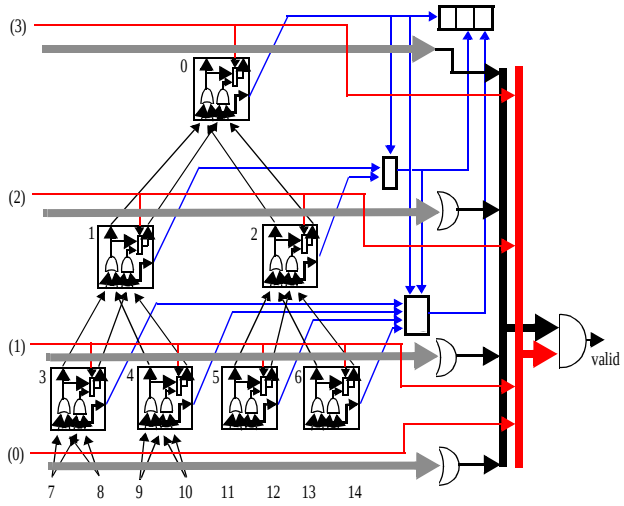



**Figure 6: OR-AND-gate Tree**

When the logic blocks are located by the OR-gate tree and the AND-gate tree, the number of bits requested for allocation need to be marked as occupied. Thus we need an algorithm which is feasible to implement in hardware to mark the corresponded bits. Free cell locator provides the starting address of these cells. What we need is to mark requested size of logic blocks from starting address. Basic implementation of marking bit array is described in Figure 7. Size value is transferred into consequent bit array, then shifted with starting address. This shifted bit array is "or"ed with the actual bit array to mark allocated bits.

"**11010011**" actual bit array

$(100)_2$ is the first free address of $2^1$ block size

"**11**" = block size

"**000011xx**"

or

"11011111" result bit array

**Figure 7: Marking free bits**

Previous marking algorithm is very simple and easy to implement. On the other hand, it is not efficient due to undeterministic design size. This approach takes more space than available cell in configurable computing (Table 1). Therefore much smaller version of marking algorithm is needed. Considering OR-gate tree and AND-gate tree, marking can be achieved through signal propagation. Marking signals will be directed by size and starting address inputs, so only corresponded bits will receive the marking signal. Unlike the AND-gate tree, bit-marker propagates signals from the root of the tree to the leaves.

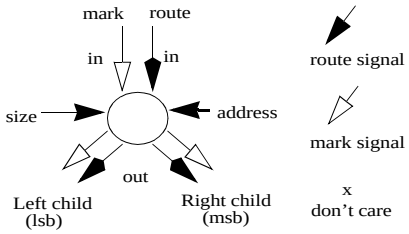



**Figure 8: A marking node**

Here is our first attempt to design marking nodes on the tree (Figure 8). This approach not only takes less space but also reduces the complexity through bit-slice design methodology (12). Initially we need the route signal which is set to 1 by default. This signal travels from top of the tree to down and routed starting address bits in the each level. For example the highest bit, 0, of starting address (010)2 = 2 routes the signal to left child node 1. Node 2 receives no route signal. Requesting size $(10)_2 = 2$ tells that one of the $2^1$ blocks will be allocated at level 1. In this case node 1 and 2 receive size signal, '1' which indicates one of the subnodes will be marked. A node which receives the route signal marks one of its subnodes. Starting address bits at this level determines which nodes will receive mark signal. other node will be received route signal in case of partial allocation in this subtree. In this example node 4 receives mark signal and propagates it to the left and the right sub nodes and so on. Once mark signal is activated to the one direction, route signal is routed to other direction. Figure 8 shows the marking operations. The routing signals of the marking nodes in the last row aren't shown to make diagram simple enough. Complete marking algorithm is shown in Figure 8b.

The advantage of using the buddy system is fast cell allocation. This approach is using significantly more cells

than other strategies at the cell locating stage. However the final marking can mark the exact size. The modified buddy system does not always yield better cell utilization. Due to external fragmentation and blind spot, there would be cell wastes among the allocated cells.

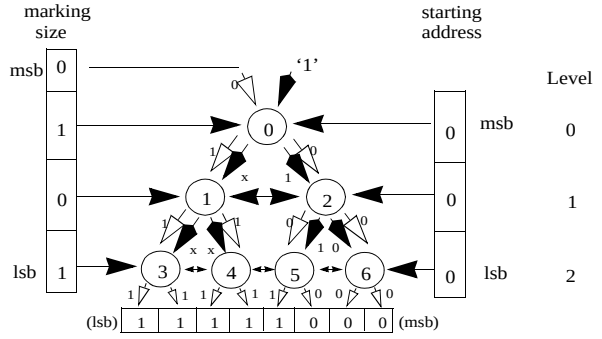



**Figure 1.** Marking four bits in and 8-bit array

Implementation of the marking tree is based one one-bit marking node. Management of the signal distribution among the nodes is the most curial design challenge. This is achieved through careful node and signal labelling. In each nodes sending signals are labelled with the target node label. For example, node 1 sends signals to node 3 and 4, respectively outgoing signals are labelled 3 and 4. *Out* signals, mark and route consist of two bits, most significant bit is for right node, least significant bit for left node. *In* signals are labelled with receiving node label, so node 1 receives mark signal [1] and route signal[1]. Labelling nodes starting from 0 gives advantage to determine starting and ending node number in each row.

starting node number = $2^{level} - 1$
ending node number = 2 * starting node number

By using above formula, marking-tree is formed very efficiently. Next figure demonstrate the construction of marking-tree.

```
FOR i IN 0 TO level-1 GENERATE
    FOR j IN 2**i - 1 TO 2*(2**i - 1) GENERATE
        bitalloc PORT MAP(in_flip => temp_flip(j),
            in_route => temp_route(j),
            s_addr => m_addr(level-1-i),
            size => m_size(level-1-i),
            o_route => temp_route(2*j+2 DOWNTO 2*j+1),
            o_flip => temp_flip(2*j+2 DOWNTO 2*j+1)
            );
    END GENERATE;
END GENERATE;
```

**Figure 9: Generation of marking-tree**

One locator design and behavioral and structural designs of marking tree are formed to implement the cell allocation algorithm on reconfigurable computing machine. Foundation series 1.5i tools was used in the design process for target device, XC4000E. Table 1 shows the result of three designs in terms of number of CLBs, delays and Line of Code (LOC) of VHDL source codes. Hardware implementation of optimized buddy system demonstrates high-performance for allocation of free cells.

**Table 1: Locator and Marker implementation results on XC4000E***

| Component | LOC | Size (bits) | # of Logic Cell | Gate Count | Max Net Delay(ns) | Max Delay(ns) |
|---|---|---|---|---|---|---|
| Locator (Structural) | 160 | 16 | 28 (%07) | 310 | 6.5 | 41.2 |
| | | 32 | 58 (%14) | 654 | 7.5 | 56.5 |
| | | 64 | 118 (%29) | 1336 | 14.1 | 83.1 |
| | | 128 | 244 (%61) | 2715 | 24.6 | 155.7 |
| Marking (Structural) | 64 | 16 | 22 (%05) | 264 | 5.8 | 22.5 |
| | | 32 | 46 (%11) | 552 | 11.5 | 26.6 |
| | | 64 | 94 (%23) | 1128 | 14.3 | 29.3 |
| | | 128 | 190 (%49) | 2280 | 11.9 | 41.8 |
| Marking (Behavioral) | 30 | 16 | 88 (%22) | 1301 | 13.6 | 34.6 |
| | | 32 | 194 (%48) | 3136 | 22.9 | 44.1 |
| | | 64 | 417 (%104) | 7238 | -- | -- |
| | | 128 | -- | -- | -- | -- |

* Xilinx 4000E-1BG225 (400 CLBs, 10K max logic gates)

## 3.2 Matrix Algorithm

As noted above, the OR-gate tree and AND-gate tree are used to locate and mark the available cells. In a two dimensional case, the address is expressed as a pair of addresses which consist of column and row information. To determine the availability and the first address of the cells, two dimensional matrix is arranged in one dimensional array. Once the bits can be viewed as one dimensional, bit array approach is easily applied. An one-dimensional address can be transferred back to a two dimensional address through formula: (x div m) and (y div n) in a m x n matrix. Figure 10 shows the representation of two-dimensional matrix in one dimensional array.

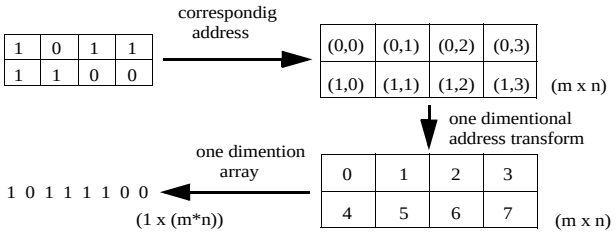



**Figure 10: Transformation of two dimensional matrix to one dimensional array**

The only difference is the address of the OR-gate tree and the AND-gate tree which is now two dimensional. A partial OR-gate tree represents the status of a bit map of (*n* x *n*) mesh. (column,row) shows the size of the submesh. Column and row must be a power of two. Free submesh can be easily found in the outputs of the OR-gate in Figure 12. There are three different mesh structure in the scheme: hor-

izontal rectangular, vertical rectangular, and overlapping of these two rectangles. The address calculations of these tree approaches are based on different base addresses.

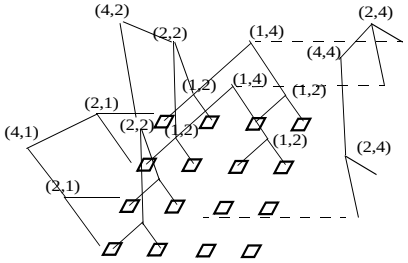



**Figure 11: Partial or-gate tree and 4 x 4 mesh structure.**

The one-dimensional cell-marker is used to form a two dimensional cell marking system. While each row is representing one dimensional array, we can use one dimensional array marker per row for x axis. There is only one cell marker for y axis. This y-axis cell marker enables the proper x-axis markers which mark x number of cells from the starting x address. The device can mark the exact number of cells needed for the submesh through pure combinational logic. Figure 12 demonstrates two dimensional cell marking approach.

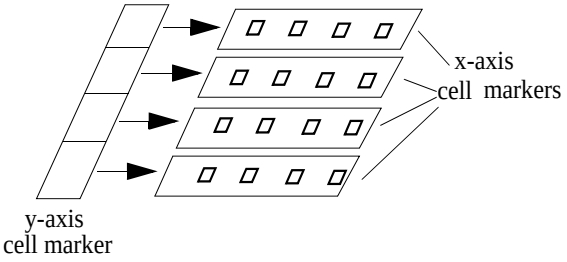



**Figure 12: A two dimensional cell-marker for allocation**

## 4 Conclusion and Future Work

Although there have been several development in the area of reconfigurable computing, the technology of the FPGAs and CPLDs need to be improved to provide better solutions for the wide-range of applications. Advances in hardware technology will reduce the complexity of designs by providing new features for synthesis tools. This paper presented a simple hardware design to allocate logic blocks, saving considerable time over the software approaches. Hardware implemented resource allocation scheme provides fast logic block allocation. Operating Systems may benefit from this approach to manage configurable computing devices as a part of their resources.

Furthermore, the hardware implemented allocation scheme represented in this paper can make feasible hardware implemented resource managers. These on-chip resource managers can reduce the configuration time and provide fast configuration. It may also allow component sharing among the applications without adding any complexity to the system. It is worthwhile investigating resource managers for better performance as well as higher configurability in future research.

## References


[1] B. L. Hutchings, M. J. Wirthlin, “Implementation Approaches for Reconfigurable Logic Applications”, *Fifth International Workshop on Field Programmable Logic and Applications*, Oxford England, August 1995, pp. 419-428.

[2] J. D. Hadley and B. L. Hutchings, “Design Methodologies for Partially Reconfigured Systems”, *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 1995, pp. 78-84.

[3] J. Burns, A. Donlin, J. Hogg, S. Singh, M. Wit, “A Dynamic Reconfiguration Run-Time System”, The 5th Annual IEEE Symposium on Filed - Programmable Custom Computing Machines, 1997.

[4] M. J. Wirthlin and B. L. Hutchings, “A Dynamic Instruction Set Computer”, *Proceedings of IEEE workshop on FPGAs for Custom Computing Machines*, 1995, pp. 99-107.

[5] R. Razdan, K. Brace, and M. D. Smith, “PRISC Software Accelerator Techniques”, *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1994, pp. 145-9

[6] M. Chatter, “Self-Modifying, On-the-Fly-Reconfigurable Logic”, *Fourth Reconfigurable Architecture Workshop*,1997, Geneva, Switzerland.

[7] K. Bazargan, R. Kastner, and M. Sarrafzadeh, “Fast Template Placement for Reconfigurable Computing”, *IEEE Design & Test of Computers Journal, Jan-March 2000*, pp. 68-82.

[8] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck and B. Schmidt, “ Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs”, *Computers and Digital Techniques, IEE Proceedings*, Volume: 147 Issue: 3, May 2000 pp. 181 -188

[9] M. Chang, “An Optimized Two-Dimensional Buddy System for Dynamic Resource Allocation”, *Journal of High Performance Computing*, Vol. 4, No. 1, Dec 1997, pp. 47-54.

[10] K. Shahookar and P. Mazumder, “VLSI Cell Placement Techniques”, *ACM Computing Surveys*, Vol. 23, No. 2, June 1991.

[11] J. M. Chang and F. Gehringer, “A High-Performance Memory Allocator for Object-Oriented Systems”, *IEEE Transactions on Computers*, Vol. 45, No. 3, March 1996, pp. 357-366.

[12] J. M. Chang and S. K. Agun, “Designing Reusable Components in VHDL”, *Proceedings of 13th Annual IEEE International ASIC/SOC Conference*, Sept. 2000, Arlington, Virginia, pp. 165-169.