AN EFFICIENT PLACEMENT ALGORITHM FOR RUN-TIME RECONFIGURABLE EMBEDDED SYSTEM

Radha Guha, Nader Bagherzadeh, Pai Chou
EECS Dept. of University of California, Irvine, California 92697
USA
{rguha; nader; phchou}@uci.edu

ABSTRACT

In the era of application convergence, the small handheld battery-powered portable devices are required to multiplex their limited hardware resources between many complex applications. Our first contribution in this paper is a modular and block based configuration architecture for modern FPGAs like Xilinx's Virtex-4 and Virtex-5 devices, to increase multi-tasking capabilities, power savings and performance improvement of applications for mobile handsets. Our second contribution is an on-line placement algorithm based on bin packing, called Hierarchical Best Fit Ascending (HBFA) algorithm, which is more efficient than Best Fit (BF) algorithm for mapping a dynamic task list onto the FPGA. The overall time complexity of the proposed on-line placement algorithm, HBFA, is reduced to *O(n)* in comparison to the complexity of BF algorithm which is $O(n^2)$. The placement solution provided by HBFA algorithm is also better than that of BF algorithm.

KEY WORDS

Reconfigurable computing, application convergence, task mapping

1 Introduction

Process technology's continual shrinking trend following Moore's Law makes reconfigurable field programmable gate array (FPGA) very promising for its abundant resources. International Technology Roadmap of Semiconductors (ITRS) [1] predicts one billion transistors on the same chip in 65 nm and under process technology by the end of this decade. But today's embedded system design is becoming harder with the increased capacity and complexity of the integrated circuits (ICs). With the advancement of the wireless technology such as WiMAX and Wi-Fi, the embedded systems in small handheld battery-powered, portable devices will support standard voice functions of a telephone as well as text messaging, sending and receiving photos and videos, gaming, speech recognition and security provisions for sensitive data exchanges [2], [3]. The limited hardware resources of these small devices need to be multiplexed for different functions at different times. All of the functionalities involve fast processing of complex data-intensive algorithms with a strict power budget. Parallel processing and reconfiguration capabilities of FPGAs, its lower design cost and shorter time to market make it a popular technology for embedded system development in contrast to the Application Specific Integrated Circuits (ASICs).

The abundant resources of FPGA alone are not enough for performance boost of applications. We need efficient on-chip application configuration architecture and efficient resource manager and scheduler. A modular and flexible application configuration architecture having better resource utilization is the key to power savings, increased multi-tasking capabilities and overall performance improvement of the applications. A compact placement of tasks on hardware resources can increase multi-tasking capabilities or can save significant amount of power consumption by advanced techniques like clock gating, voltage scaling [4] etc. Thus we have designed a modular and block based flexible configuration architecture for FPGA, named Memory- Aware Run-Time Reconfigurable Embedded System (MARTRES) which will facilitate partial reconfiguration at run time, efficient resource management and resource utilization by our Hierarchical Best Fit Ascending (HBFA) task placement algorithm. As the HBFA placement algorithm works faster than the BF algorithm it contributes to the overall performance improvement of the applications.

In this paper, we focus on hardware resource management between a number of competing tasks in the order they arrive or the tasks which have already been identified for concurrent hardware execution. On the contrary in references [5], [6], their partitioning and scheduling algorithms evaluate many task parameters for task ordering before the actual task placement.

Next we introduce our MARTRES architecture and the bin packing problem related to task mapping problem on hardware resources of FPGA.

1.1 Proposed Configuration Architecture, MARTRES

Several algorithms [7] such as Fast Fourier Transform (FFT), Discrete Cosine Transform (DCT) and Data Encryption Standard (DES) etc. required for real-time audio and video processing are streaming or data intensive and require high-bandwidth data transfer from external memory to on-chip memory. Data intensive computations of the streaming applications can be accelerated by the FPGA's concurrent processors and the remaining control intensive parts are better suited for running sequentially by a general purpose processor. These kind of streaming applications need a general-purpose processor for hosting a real-time operating system (RTOS) for hardware configuration, resource management and task scheduling onto the hardware resources of FPGA. A block diagram of the required hybrid system is shown in Figure 1.

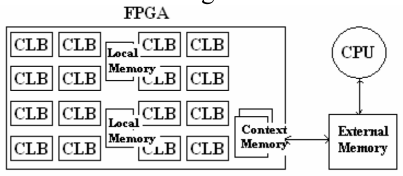

Figure 1: The Hybrid System for Algorithm Acceleration

Our proposed configuration architecture, MARTRES for the FPGA is based on the analysis that to sustain the processing speed of the parallel processors we need distributed on-chip cache memories. Having memory blocks equally distributed throughout the chip reduces data communication cost in terms of time, power consumption and router overhead. Also most of the time the tasks mapped onto FPGA chip are independent of each other and provision of distributed block memory at regular intervals is beneficial for resetting one memory block and filling it with new data without disturbing other memories used by other tasks. Thus the MARTRES configuration architecture for hardware is designed to be modular and block based as shown in Figure 2, to facilitate partial reconfiguration at run time. In each module there is a balance between the amount of logic and memory so that the processing elements (PEs) in each block can sustain their speed by concurrent data access from memory blocks. In reference [8] their "Synchroscalar" architecture is column based, but the research findings of this paper suggest that block based architecture is possible for modern FPGAs like Xilinx's Virtex-4, Virtex-5 [9], [10], where block based partial reconfiguration is possible.

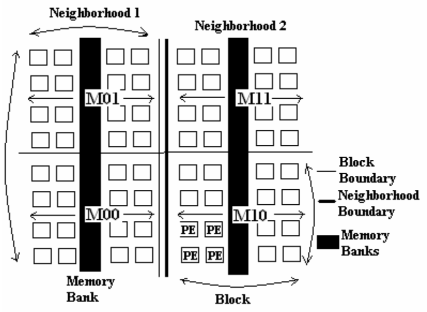

Figure 2: MARTRES Configuration Architecture

As different applications may have different memory requirements, even if all the SRAM cells are not used as memories all the time, they can be used for implementing the control units or the finite state machines (FSMs) [11] for the processing elements more efficiently. FSMs implemented with SRAMs have better power and area savings than the same implementation by Flip Flops (FFs) and programmable lookup tables (LUTs).

In Figure 2, a partial MARTRES configuration is shown for a Xilinx's Virtex-4 XCVSX35 device, which is an array of 96*40 configurable logic blocks (CLBs) [9] with maximum capacity of 240 Kb of distributed RAM or shift registers and total 3.4 Mb of block RAMs where each block is 18 Kb and operating at 500 MHz. Block RAMs can be cascaded to form larger memory blocks. We assume that an array of 5*5 CLBs are required to implement a processing element (PE). Since MARTRES is coarse grained, the basic elements before configuration are PEs. During configuration, data-path widths of each PE or connecting PEs are changed for function-specific optimization purposes. In MARTRES architecture, a group of PEs is called a *block* (*BL*), and a group of *blocks* is called a *neighborhood* (*NH*).

The actual memory requirement for a group of PEs will depend on their frequency of operations, type of connection between them, type of algorithm they are processing and the input data rate. If required 18 Kb dual-ported block RAMs can be cascaded to configure, say 566 Kb RAM to cater for two *blocks* in a *neighborhood*. Memories are identified with column location first and then with port numbers. Two ports M_{00} and M_{01} from the same memory bank cater for two *blocks*, bottom and top in the same *neighborhood* as shown in Figure 2. Routing in each *neighborhood* is more efficient than inter-*neighborhood* routing and higher bandwidths are provided inside the *neighborhood* than that of the routing connecting the *neighborhoods*.

MARTRES architecture is modular and a hardware resource manager can keep track of all the resources in a hierarchical fashion, first by a *neighborhood* number, then by a *block* number in a *neighborhood* and finally by a PE number in each *block*. Suppose MARTRES has 8*8 array of PEs. First the PEs are grouped together as *blocks* and then as *neighborhoods*. Suppose each *block* has 4*4 PE array and each *neighborhood* has 2*1 *block* array or 8*4 PE array. So the whole chip consists of 2 *neighborhoods*, 4 *blocks* and 64 individual PEs. The *block* boundaries are indicated by thin lines and the *neighborhood* boundaries are indicated by thick lines in Figure 2. Each *block* is identified by its participation in which *neighborhood* first and then by its number. So the *blocks* are identified as $block_{i,j}$ where $i = 1$ to N, N being the number of *neighborhoods*, and $j = 1$ to B, B being the number of *blocks* in each *neighborhood*. Each PE is identified by its participation in which *neighborhood* first and then in which *block* and then by its number. So the PEs are identified as $PE_{i,j,k}$ where $i = 1$ to N, N being the number of *neighborhoods*, $j = 1$ to B, B being the number of *blocks* in each *neighborhood* and $k = 1$ to P, P being the number of PEs in each *block*.

We have created MARTRES configuration architecture so that the HBFA placement algorithm takes less time for mapping tasks. To reduce search time by HBFA placement algorithm, the resource manager creates two tables. A smaller *neighborhood header table* is created which is only a list of *neighborhood*s and their unoccupied capacity in terms of number of PEs, sorted in ascending order as shown in Table 1.

Table 1: NH Header

NH No.	# PEs Available
1	32
2	32

Table 2: NH Detail

NH No.	BL No.	# PEs Available	PE No.
1	1	16	1,….,16
1	2	16	1,….,16
2	1	16	1,….,16
2	2	16	1,….,16

A larger *neighborhood detail table* is created to list all the *blocks* in each *neighborhood* and their unoccupied capacity in terms of number of PEs as shown in Table 2. The header and detail table can be implemented as arrays and from the header table a particular *neighborhood* in detail table can be accessed in constant time by its index. It can be shown that the MARTRES configuration architecture facilitates efficient resource management and the HBFA placement algorithm evaluates faster than the BF placement algorithm for mapping a dynamically arrived task onto FPGA. Before presenting the HBFA algorithm we introduce the bin packing problem first.

1.2 Bin Packing Problem

The problem of placing objects of various sizes in bins of equal capacity is called bin packing problem. Object size varies from zero to maximum bin capacity and is indivisible. The objective of the bin packing problem is to find the minimum number of bins to accommodate all the objects. A second objective of the bin packing problem is to find the packing solution fast. The bin packing problem as an optimization problem is NP-hard which can not be solved in polynomial time**.** Approximation algorithms are used to tackle the NP-hard problems if a sub-optimal but good solution is found in polynomial time.

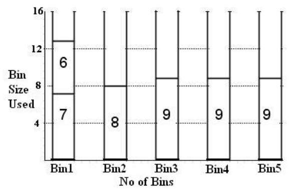

Figure 3: Bin Packing by BF Algorithm

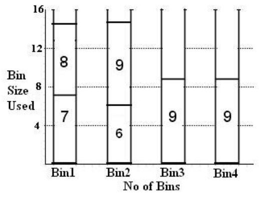

Figure 4: Bin Packing Optimal Solution

In Figure 3, packing or placement solution for a list of objects L = {7,6,8,9,9,9} is shown to take 5 bins of equal capacity of 16 units by Best Fit algorithm which places a object in a bin which will leave minimum unused space. The off-line optimal solution for the same problem takes 4 bins as shown in Figure 4. The quality of an approximation algorithm is defined by the ratio of the number of bins required by it to that of optimal solution.

Bin packing algorithms can be divided into two categories. In the off-line variety the object list is static and the objects can be ordered in descending order of their size to get better placement solution. In the on-line variety the object list is dynamic and the objects come one by one and no ordering can be done. There are several approximation algorithms [12], [13], [14], [15] for bin packing, such as on-line variety First Fit (FF), and Best Fit (BF) and off-line variety First Fit Decreasing (FFD) and Best Fit Decreasing (BFD). The packing generated by off-line algorithms FFD and BFD use no more than (11/9 *OPT* + 4) bins, where *OPT* is the optimal number of bins. The packing generated by on-line algorithms FF and BF use no more than (17/10 *OPT* + 2) bins.

Placement of tasks on hardware resources is similar to bin packing prblem. In this paper the objects are computation modules or functional blocks known as tasks and the bins are hardware resources. In practice the bin size is determined by configuration granularity and other implementation options of the chip. Here the bins are of equal size of a *block*, which is a group of 16 PEs. The tasks are one dimensional and measured only by number of PEs instead of having fixed width and height of PEs. The solution provided by BF algorithm for a task list in Table 3 is as shown in Figure 5. BF algorithm will use the *neighborhood detail table* where the bins of a size of a *block* are listed and sorted in ascending order.

Table 3 Task Table

Task No.	Task Size (No. of PEs)
1	8
2	12
3	14
4	4

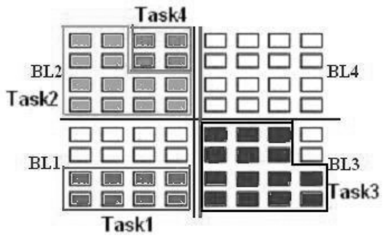

Figure 5: 4 Consecutive Tasks Placement by BF

Table 4: NH Detail after 4 Tasks Placement by BF

NH No.	BL No.	# PEs Available	PE No.
1	2	0	
2	3	2	15,16
1	1	8	9,…..,16
2	4	16	1,2,…..,16

By BF algorithm, Task 1 is placed in *block* 1, which occupied 8 PEs and still has 8 PEs unoccupied. After each task placement the *neighborhood detail table* is updated and sorted in ascending order. Task 2, which requires 12 PEs, is placed in *block* 2. Task 3, which requires 14 PEs, is placed in *block* 3. Task 4, which requires only 4 PEs is placed in *block* 2. After four consecutive task placements by BF algorithm the *neighborhood detail table* will look like Table 4.

Task placement on hardware resources varies from the classic bin packing problem in the following manner. For a hardware with resource constraint or limited number of bins, if a new task can not be placed in the chip the scheduler has to wait for the time when some area will be available after the execution of already placed task instead of adding a new bin. The tasks can also be divided between two bins to get better placement solution if the introduced communication overhead due to splitting of the task is not too high. Communication overhead depends on the amount of communication between the tasks and the length of the distance it has to traverse.

1.3 Related Work

There are many application areas of classic bin packing problem such as loading trucks, recording of music on CDs etc. where the objects are indivisible. To reduce the time complexity of the traditional algorithms like BF, FF etc. in reference [13] they have designed a Harmonic_M bin packing algorithm whose time complexity is $O(n)$. They have classified each object according to its size and put it in a specific bin for that class of objects. The number of classes, M, are not to exceed 12 in practice. If M> 6 and the sizes of objects are equally distributed for each class then the Harmonic_M solution is better than Next Fit and First Fit solution but never better than Best Fit solution.

In reference [14], [15] they have hybridized iterative genetic algorithm (GA) with traditional bin packing algorithms like FF, BF etc. to get better placement solution at the cost of additional time. In reference [16] they have developed an efficient placement algorithm based on Bazargan's [17] approach, that requires partitioning the free space of FPGA into non-overlapping rectangular areas after each task placement. With the help of a binary tree and a hash-table data structure they keep track of the free rectangles. There approach is able to find a free space in constant time but the subsequent updating of the data structures takes more time. In reference [18] they have proposed a Partitioned Best Fit Decreasing (PBFD) algorithm where they have divided a task between bins if only it does not fit in any single bin. Their algorithm can split the task between any two bins irrespective of whether they are physically adjacent or not. The time complexity of their PBFD algorithm is same as BFD algorithm.

In contrast our HBFA algorithm is for on-line task list which can not be ordered beforehand. We also split a task but only between physically neighboring bins to reduce communication overhead. By hierarchical organization of the hardware resources the time complexity of HBFA algorithm is reduced to *O(n)*. HBFA has a truly hierarchical best fit approach where a task is first tried to be confined in a *block* then in a *neighborhood* and then in the chip.

The rest of the paper is organized as follows. Section 2 presents the HBFA placement algorithm. Section 2.1 proposes different search strategies for different task sizes. Section 2.2 presents the placement solution provided by HBFA algorithm and its pseudo-code. Section 2.3 analyzes the time complexity of HBFA in comparison to BF algorithm. Section 3 concludes the paper by pointing out the combined benefits of MARTRES architecture and HBFA placement algorithm and their possible applications.

2 Proposed Placement Algorithm, HBFA

A novel variety of bin packing algorithm, named, Hierarchical Best Fit Ascending (HBFA) algorithm is proposed in this paper as a placement algorithm for MARTRES architecture. The algorithm is so called because the resources or bins are listed heirarchically and searched heirarchically and the higher level bins are sorted in ascending order. The placement algorithm first searches the higher level bins listed in the smaller *neigborhood header table* to ensure available capacity in a *neighborhood* is enough for a task and then goes to the lower level bins listed in larger *neigborhood detail table* for determining exact physical location for that task in that *neighborhood*.

HBFA algorithm is based on the following analysis. For the bin packing problem we argue that, if the bin capacity is comparatively larger than individual object sizes then the total number of bins, *OPT*, required for optimal solution is smaller than the solution obtained when the object sizes are larger. This is because more objects can be packed in the same bin and there is less chances of larger unoccupied wasted space in each bin. If *OPT* is small then approximate solution of BF algorithm which is less than or equal to (17/10 *OPT* + 2) is also small. Thus difference between an optimal solution and an approximate solution varies depending on average relative size of bin capacity and object sizes. Analytically either the object size should be equal to the bin size or between 0 to 30% of the bin size so that there will be less % of each bin remaining unutilized after placement of several objects.

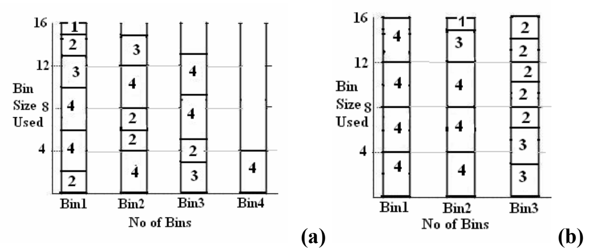

Figure 6: (a) BF vs. (b) Optimal Solution for Smaller Task Sizes

To prove the above analysis, Figure 6 maps a task list with relatively smaller size tasks such as L= {2,4,4,3,2,4,2,2,4,3,3,2,1,4,4,4} with BF algorithm and compares it with the optimal solution. Total task size in that list is 48 and optimal solution took 3 bins and approximate solution took 4 bins of size 16 units each. Figure 3 and Figure 4 mapped a task list with relatively larger size tasks such as L= {6,7,8,9,9,9} also totaling to 48 with BF algorithm and compared it with the optimal solution before. In that case optimal solution took 4 bins and approximate solution took even larger number of 5 bins. So to increase the bin size in HBFA algorithm, we group a number of smaller bins to make a larger bin. In the lower level the bin size is smaller and is labeled as a *block* (*BL*). In the higher level the bin size is larger and is labeled as *neighborhood* (*NH*) which is a group of *blocks*. Both NH and BL bin sizes are measured in number of PEs.

2.1 Allowing Different Task Sizes by HBFA

If we can control the task size we make the task size much smaller than the *neighborhood* size to make our approximate solution close to optimal solution. For mapping a task, if the task size is smaller than a *neighborhood* size the *neighborhood header table* will be searched in forward order starting from the top of the list. In the first try no task will be split between two *neighborhoods* to avoid inter-*neighborhood* communication delay. But if a task can not be placed in any *neighborhood* then it can be split between *neighborhoods*. If task related parameters like communication cost of tasks are available and two tasks have many communications between them then their combined size can be considered so that they both can be placed in the same *neighborhood* to reduce communication cost.

As the tasks get mapped onto the FPGA fabric, the area gets fragmented after a few tasks placement which amounts to poor resource utilization as discussed in detail in reference [5]. To prevent area fragmentation we split a task between the *blocks* in the same *neighborhood* if no single *block* can contain the task. This increases resource utilization in a *neighborhood*.

If we can not control task sizes then we allow all different task sizes even larger than a *neighborhood* size. If the task size is larger than a *neighborhood* size then search in the *neighborhood header table* is done in the reverse order from the bottom of the list to the top of the list. Everytime a *neighborhood* is selected for that task placement the available area of that *neighborhood* is deducted from the original task size. If the modified task size is still greater than a *neighborhood* size then the next *neighborhood* in the reverse search order in the *neighborhood header table* is selected. Whenever the modified task size becomes smaller than a *neighborhood* size the *neighborhood* header table is searched in forward order to meet best fit criteria. After the task is completely placed the *neighborhood header table* is sorted again in ascending order. Whenever a task is split between *neighborhoods* the inter-*neighborhood* communication increases. Thus if the average task sizes are larger than *neighborhood* size we have to strengthen inter-*neighborhood* communication interconnects.

2.2 Placement Solution and Pseudo Code of HBFA

We map the dynamic task list in Table 3 with our HBFA algorithm this time. Four consecutive task placements are shown in Figure 7. Task 1 is placed in *neighborhood* 1, which occupied 8 PEs and still has 24 PEs unoccupied. After each task placement the *neighborhood* header and detail tables are updated and the *neighborhood header table* is sorted in ascending order. Task 2, which requires 12 PEs, is placed in *neighborhood* 1 by splitting it between *block* 1 and *block* 2. Task 3, which requires 14 PEs, is placed in *neighborhood* 2. Task 4, which requires only 4 PEs is placed in *neighborhood* 1. After four consecutive task placements, the *neighborhood* header and detail tables look like Table 5 and Table 6 respectively. After completion of each task execution the HW available area is put back in the *neighborhood* header and detail tables. HBFA placement algorithm pseudo code is shown in Table 7.

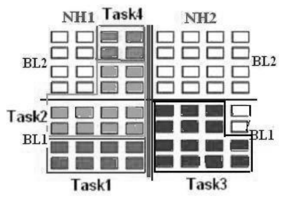

Figure 7: 4 Consecutive Tasks Placement by HBFA

Table 5: NH Header after 4 Tasks Placement by HBFA

NH No.	# PEs Available
1	8
2	18

Table 6: NH Detail after 4 Tasks Placement by HBFA

NH No.	BL No.	#PEs Available	PE No.
1	1	0	
1	2	8	9,……,16
2	1	2	15,16
2	2	16	1,…...,16

Table 7: HBFA Algorithm Pseudo Code

```
Create array Task_List[Task_1, …Task_m]  //m number of tasks
Create array NH_Header[NH_1,…NH_N]  //sorted in ascending order of
available capacity, N number of neighborhoods
Create array NH_Detail[NH_1, …NH_N][ BL_1,…BL_B]
//B, Number of blocks configured in each neighborhood
Create variable max_NH //max neighborhood size

HBFA(Task_list, NH_Header, NH_Detail)
for i ← 1 to length[Task_list]
 if Task_list [i]<= max_NH then //forward search
  Forward_Search:
   for j ← 1 to length[NH_header]
    if (Task_list [i]<= NH_header[j])   //task is placed
in jth neighborhood
    then  NH_header[j] = NH_header[j] - Task_list [i]
      SORT (NH_header) //ascending order for Best Fit
      TASK_PLACEMENT(Task_List[i], NH_Detail[j][])
 else //reverse search
  modified_task_size= Task_list [i]
  Reverse_Search:
  for j ← length[NH_header] to 1
   if modified_task_size> max_NH then
    TASK_PLACEMENT(NH_header[j], NH_detail[j][])
     modified_task_size -= NH_header[j]
     NH_header[j] = 0
   else Task_list [i] = modified_task_size
      go to Forward_Search

TASK_PLACEMENT(Task_list[i], NH_detail[j][])
//task can be split between blocks
for k ← 1 to B //B, Number of blocks configured in each
neighborhood
 if  Task_list[i] > NH_detail[j][k]  then
   Task_list[i] = Task_list[i] - NH_detail[j][k]
   NH_detail[j][k]= 0;
 Else NH_detail[j][k]= NH_detail[j][k]- Task_list[i]
```

2.3 Complexity Analysis of HBFA Algorithm

By the hierarchical relations of the *neighborhood* header and detail tables, the search time of the HBFA placement algorithm is reduced. If there are total M PEs on the chip, and each *block* has P PEs, and each *neighborhood* has B *blocks*, then we have a total of N *neighborhoods*, where $N = M/(B*P)$ and here the *neighborhood* or bin size is $(B*P)$. So, in the first level, the number of bins to search is $N = M/(B*P)$. In the second level of search, the *block* or bin size is P and the number of *blocks* or bins to search is B. So, in the worst case total search time is $(M/(B*P)) + B$ for each task placement. Where as in one level of BF placement algorithm where the bin size is P or a *block*, the number of bins to search is M/P. After each placement the bin list is sorted in ascending order to meet best fit criteria, and in the worst case sorting takes $O(n^2)$ time, where n is the number of items in the list. In our heirarchical HBFA algorithm only the *neighborhood header table* is sorted and it takes $O((M/(B*P))^2)$ time. In one level BF algorithm the sorting time is $O((M/P)^2)$. From the above example we can formulate the complexity of HBFA algorithm and can compare it with that of BF algorithm.

HBFA execution time = search time + sort time

$= (M/(B*P)) + B + O((M/(B*P))^2)$

$= N + B + O(((N*B*P)/(B*P))^2)$

$= N + B + O(N^2)$

$= 2N + O(N^2)$; if N and B are made equal in number

$\approx O(N^2)$

If we add the cost of an ad-hoc sort for the selected *neighborhood* to meet best fit criteria in that neighborhood it will take additional $O(B^2)$ time. HBFA execution time will still remain $O(N^2)$. But for a small *neighborhood* with a few *blocks* we may not require to sort it as we are splitting a task between *blocks* anyway. A large *neighborhood* with many *Blocks* may be sorted to meet the best fit criteria.

BF execution time = search time + sort time

$= M/P + O((M/P)^2)$

$= (N*B*P)/P + O(((N*B*P)/P)^2)$

$= N*B + O((N*B)^2)$

$= N^2 + O(N^4)$; if N and B are made equal in number

$\approx O(N^4)$

Now if we say $n = N^2$; then HBFA is a $O(n)$ algorithm and BF is $O(n^2)$ algorithm. Thus the overall complexity of HBFA algorithm is reduced to $O(n)$ in comparison to BF algorithm which is $O(n^2)$ for each task placement.

In our case M=64, N=2, B=2 and P=16. HBFA algorithm takes 2+2=4 units of search time and one level of BF algorithm also takes 4 units of search time. Sorting time of HBFA algorithm is 2^2=4 units where as in BF it is 4^2=16 units. Total execution time saving is (4+16) – (4+4) = 12 units. For larger values of M, N and B more savings of time will be observed. The cummulative time saving for a number of tasks placement over a time period will be considerable.

Table 8: HBFA vs. BF for Different Block and Neighborhood Sizes

# PE M	# NH N	# BL /NH B	# PE /BL P	HBFA Map. Time /Task (cycles)	BF Map. Time /Task (cycles)	Mapping 8 Tasks With HBFA (cycles)	Mapping 8 Tasks With BF (cycles)
64	8	4	2	76	1056	608	8448
64	4	4	4	24	272	192	2176
128	16	4	2	276	4160	2208	33280
128	8	4	4	76	1056	608	8448
128	4	8	4	24	1056	192	8448
128	4	4	8	24	272	192	2176
128	8	8	2	80	4160	160	33280

Table 8 compares the impact of varying HW resources M, and different *neighborhood* and *block* size on HBFA and BF algorithms. If the original bin size, a *block* of P number of PEs, is fixed then smaller *neighborhood* size will give better placement solution as the task need not be split over many *blocks*. Whereas small number of *neighborhoods* will make the HBFA algorithm work faster. Thus we have to strike a balance between the number of *neighborhoods* and the *neighborhood* size.

Table 9: HBFA vs. BF Algorithm under Different Implementations

		HBFA		BF
		Header Table	Detail Table	Detail Table
Memory (Priority Queue DataStructure, Array implementation)		$O(N)$	$O(N^2)$	$O(N^2)$
Linear serach-time		$O(N)$	$O(N)$	$O(N^2)$
Insertion Sort	Mem	-	-	-
	Time (worst case)	$O(N^2)$	-	$O(N^4)$
Mearge Sort	Mem	$O(N)$	-	$O(N^2)$
	Time (worst case)	$O(N\lg N)$	-	$O(N^2\lg N^2) \approx O(N^2\lg N)$
Quick Sort	Mem	-	-	-
	Time (worst case)	$O(N^2)$	-	$O(N^4)$
Heap Sort	Mem	-	-	-
	Time (worst case)	$O(N\lg N)$	-	$O(N^2\lg N^2) \approx O(N^2\lg N)$

In Table 9 HBFA and BF algorithms are compared under different search and sort implementations [19] and under worst case time complexity of the search and sort algorithms. HBFA algorithm use a *neighborhood header table* as well as *neighborhood detail table*. BF algorithm use only the *neighborhood detail table*. The length of the header table is N, where N is the number of *neighborhoods*. The length of the detail table is M/P, where M is the number of PEs in the whole chip and P is the number of PEs per *block*. For calculating time and space complexity we again assume $N = B$ here. The placement algorithm does not require any table to grow or shrink. So they can be implemented by static array data structure. The detail table can be accessed directly from the header table by an index in constant time.

HBFA algorithm has extra memory requirement of $O(N)$ to keep the header table information in addition to the detail table information. HBFA algorithm needs to search and update both header and detail tables and sort only the header table. BF algorithm needs to search, update and sort only the detail table. Search and sort can be implemented by different algorithms. Different algorithms need different data structures. Different algorithms have different memory (mem) overhead and execution time as listed in Table 9. In Table 9, maximum benefit in running time by HBFA algorithm over BF algorithm is observed when sorting is done by Insertion Sort and Quick Sort, without any memory overhead difference for sorting. Minimum benefit in running time is observed when sorting is done by Merge Sort and Heap Sort. Memory overhead for sorting by Merge Sort in

HBFA is $O(N)$. Memory overhead for sorting by Merge Sort in BF is $O(N^2)$. For Heap Sort there is no memory overhead either by HBFA or by BF algorithm for sorting.

3. Conclusion

The MARTRES architecture is very flexible and efficient for resource management purposes and for evaluating the HBFA, placement algorithm for a dynamic task list. BFA algorithm saves on search time as well as on sort time as the size of the *neighborhood header table* is much smaller than the *neighborhood detail table*. The HBFA algorithm can be applied to many kind of bin packing problems where object size can be divisable between physically neighboring bins. HBFA algorithm can have more than two levels to reduce the search and sort time further if the placement solution and the expense of extra memory to keep all the header informations in each level are acceptable. In contrast to the "Divide-and-Conquer" paradigm of many recursive algorithms, HBFA algorithm can be touted as "Unite-and-Conquer" approach, where a number of smaller bins are grouped together to know their combined capacity to reduce search and sort time.

Besides the time benefit of the HBFA algorithm over BF algorithm, there are several other benefits of HBFA algorithm. HBFA facilitates partial reconfiguration at run time as the whole task is tried to be kept in the same *neighborhood*. Confining a task in the same *neighborhood* also reduces the communication cost in a task. In BF algorithm the bin size is a *block* and a single task is not split between the *blocks* in the first try thus it has poor resource utilization in comparison to HBFA algorithm where a task is split between *blocks* to reduce fragmentation of the area in a *neighborhood*.

The HBFA, placement algorithm is also extensible for configuration reuse purposes by adding some extra task specific information in the header and detail table as required. As for example to avoid reconfiguration overhead a functional description can be added to a group of PEs which have been relinquished by an old task and is currently available for a new task with the same functionalities.

References

[1] ITRS, International Technology Roadmap for Semiconductors, 2005 EDITION.

[2] P. Heysters, G. Smit and E. Molenkamp. A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems. *The Journal of Supercomputing, Volume 26,* Issue 3, November 2003.

[3] J. Chen, J.Yeh, Y. Lan, L. Lin, F. Chen and S. Hung. RAMP: Reconfigurable Architecture and Mobility Platform, In *Proceedings of IEEE Globecom,* 2005.

[4] J. Flynn and B. Waldo. Techniques for Power Optimization. *Compilers* @ Synopsys Inc. 2007.

[5] S. Banerjee, E. Bozorgzadeh and N. Dutt. Physically-Aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration. *DAC 2005*, June 2005.

[6] B. Mei, P. Schaumont and S. Vernalde. A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems. In *Proceedings of ProRISC*, 2000.

[7] E. Caspi, M. Chu, R. Huang, J. Yeh, Y. Markovskiy A. DeHon and J. Wawrzynek. Stream Computations Organized for Reconfigurable Execution (SCORE). *Conference on Field Programmable Logic and Applications,* August 2000.

[8] J. Oliver, R. Rao, P. Sultana, J. Crandall , E. Czernikowski, L. W. Jones IV, D. Franklin , V. Akella, and F. T. Chong. Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor. *2004 International Symposium on Computer Architecture*. 2004.

[9] Xilinx. Virtex-4 User Guide UG070 (v2.2) April 10, 2007.

[10] Xilinx. Virtex-5 User Guide UG190 (v3.0) Feb 2, 2007.

[11] A. Tiwari and K. A. Tomko. Saving Power by Mapping Finite-State Machines into Embedded Memory Blocks in FPGAs, *2004 IEEE.*

[12] AMS. Bin Packing . www.ams.org.

[13] C. C. Lee and D.T. Lee. A Simple On-Line Bin-Packing Algorithm. *Journal of the Association for Computing Machinery, Vol. 32*, No. 3, pages 562-572, July 1985.

[14] E. D. Goodman, A. Y. Tetelbaum, and V. M. Kureichik. A Genetic Algorithm Approach to Compaction, Bin Packing, and Nesting Problems. TECHNICAL REPORT # 940702*, Case Center for Computer-Aided Engineering and Manufacturing*, Michigan State University.

[15] C. Reeves. Hybrid Genetic Algorithms for Bin Packing and Related Problems. Published in *Annals of OR*, 63, pages 371-396.

[16] H. Walder, C. Steiger, M. Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03).*

[17] K. Bazargan, R. Kastner and M. Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. In *IEEE Design and Test of Computers*, volume 17, pages 68–83, 2000.

[18] D. de Niz and R. Rajkumar. Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems. *International Journal of Embedded Systems*, 2005.

[19] T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to Algorithms* (Prentice Hall, India, 2003).