Memory Utilization of Processor Allocator for NoC-based Chip Multiprocessors with Mesh Topology

Dawid Zydek*, Henry Selvaraj*, Laxmi Gewali**

**Department of Electrical and Computer Engineering, University of Nevada, Las Vegas, USA*
Las Vegas, NV, USA (e-mails: zydekd@unlv.nevada.edu, selvaraj@unlv.nevada.edu)
***School of Computer Science, University of Nevada, Las Vegas, USA*
Las Vegas, NV, USA (e-mail: laxmi@cs.unlv.edu)

Abstract: Chip MultiProcessors (CMPs) have become the primary method of build high-performance microprocessors. Besides speed, major elements such as processing elements and network on chip, allocation and management of on-chip processors are also important factor to achieve high efficiency of future CMPs. In this paper, the authors study a Processor Allocator (PA), especially the issue of its memory utilization. All known important processor allocation schemes with busy array, busy list and free list are presented and compared based on their memory aspects. The presented results show that the PA implemented using busy array uses significantly less amount of memory, that is the key issue in implementing a fast, energy and space efficient PA on the same die as CMP.

Keywords: CMP, mesh topology, processor allocator, allocation algorithms, memory utilization.

1. INTRODUCTION

Technology scaling enables integration of billions of transistors on a chip and that has led to an increase in the number of processing elements on a single die. Tiled CMP architectures consisting of many cores, connected through a Network on Chip (NoC) are becoming main computing platforms today in research and computing centers, and in the near future will emerge as the main computing platform in industry as well (Held et al., Vangal et al., Zydek and Selvaraj).

Besides the on-chip network, the operating system is an important factor that impacts CMPs' performance. Nodes in CMPs can be used by many unrelated applications as well as by a single program. Processor allocation and job scheduling problems are two major issues associated with the design of operating system. Allocation of processors is done by PA, which selects a set of processors for a given job (which can include many tasks). Job scheduling is done by Job Scheduler (JS) that deals with the selection of the job to be executed next (Fig. 1).

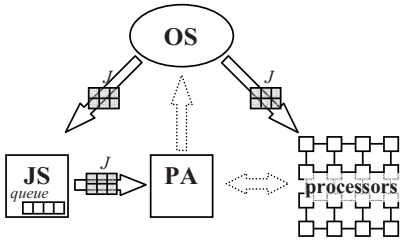

Fig. 1. Processor management system.

In a mesh topology based CMP, an incoming job is described by the size of the subgrid it requires. The JS selects the next job from the system queue for execution using a proper scheduling policy. For a job selected by the scheduler, the PA tries to find an available subgrid. If such a free subgrid does not exist, the scheduler handles the job according to the implemented policy (e.g. the JS waits until a submesh is released or a smaller job is sent from the queue to the PA) (Fig. 1). The jobs are allocated in such a manner that they do not overlap with each other, and if they are allocated, they run until completion. In this paper authors focus on a PA structure and the approaches to processor allocation algorithms.

There are two major processor allocation strategies:

- Contiguous allocation – where the processors allocated to a job are physically adjacent and have the same topology like NoC,
- Non-contiguous allocation – where a job can be executed on multiple disjoint smaller subgrids.

Non-contiguous scheme allows dividing a job rather than waiting until a single subgrid of the requested size is available. But it can generate global traffic, which we would like to avoid in NoC. It is the reason why the authors have chosen to focus on contiguous strategy. The contiguous scheme has a tendency to external fragmentation, which occurs when there are enough free processors for a particular job, but it is not allocated due to lack of contiguously or different topology as the NoC. It leads to a critical property of a processor allocation algorithm - subgrid recognition ability (ability to find free subgrids for incoming jobs). If an algorithm can always find an available subgrid (if it exists) for an incoming request, we say that the algorithm has

complete subgrid recognition ability. However, such implementations with recognition completeness increase the complexity of the processor allocator (Yoo and Das). Another important property of allocation algorithms is its speed. A good solution is supposed to have complete subgrid recognition ability, with low allocation overhead.

A lot of research has been done to increase the efficiency of subgrid recognition by the allocation algorithms. Main difference between these schemes is the way of keeping the status of the processors (if they are free or if they are already processing a job). There are three approaches: i) A busy array (bit map) with allocation status of each processor (Zhu); ii) A busy list – list with busy subgrids (Chmaj et al., Chuang and Tzeng, Das Sharma and Pradhan, Ding and Bhuyan, Yoo and Das, Yoo and Youn); iii) A free list – list with free subgrids (Ababneh, Kim and Yoon, Liu et al.).

Zydek and Selvaraj have presented an idea of hardware implementation of JS and PA, and integrating them on one die together with the processing elements. Even with very positive prospects of transistors scaling in CMP design process, we have to take care of the on-chip resource usage. As a result of implementing PA on the CMP, resources used by the PA are becoming important in order to keep the chip space, energy, power and heat at reasonable levels (Vangal et al., Zydek et al. (2008)). In this paper, we debate the PA memory consumption. All known and above mentioned algorithms are considered and analyzed. The reminder of this paper is arranged as follows: Nomenclature and terms used are described in Section 2. Most important allocation algorithms are presented in Section 3. Section 4 contains memory utilization issues for allocation schemes. Final remarks are presented in Section 5.

2. NOMENCLATURE

A k-ary 2-mesh (2D-mesh) topology, denoted by *M(w, h)*, consists of $w \times h$ nodes arranged in a $w \times h$ 2D grid. Each node in the mesh refers to a processor. The node in column c and row r is identified by address $<c, r>$, where $0 \le c < w$ and $0 \le r < h$. A non-boundary node $<c, r>$ is connected by direct communication channel to its neighboring nodes, $<c \pm 1, r>$ and $<c, r \pm 1>$. A boundary node has two or three neighboring nodes depending on its location within the entire mesh.

Definition 1. A 2D submesh *S(p, q)* in the mesh *M(w, h)* is a subgrid *M(p, q)* such that $1 \le p \le w$ and $1 \le q \le h$. A job requesting a submesh $p \times q$ is denoted by *J(p, q)*. A submesh *S* is identified by its base (lower left node) and end (upper right node) and is denoted as *S[<xb, yb><xe, ye>]*.

Definition 2. A node is busy if it is allocated to any job. A busy submesh β is a submesh where all of its nodes are allocated to job.

Definition 3. A node is free if it is not allocated to any job. A submesh is free when all of its nodes are available (are free).

Definition 4. A busy array of a mesh *M(w, h)* is a bit map *B[w, h]*, in which element *B[c, r]* has a value *1* or *0* if node $<c, r>$ is busy or free respectively.

Definition 5. A busy list is a set of all busy submeshes in the system. Similarly, a free list is a set of all free submeshes in the system.

Definition 6. The coverage of a busy submesh β with respect to a job *J* is denoted by $\xi_{\beta,J}$ and it is a set of processors such that use of any node in $\xi_{\beta,J}$ as the base of free submesh for the allocation of *J* will cause the job *J* to be overlapped with β. The coverage set with respect to *J* is denoted by C_J and it is the set of the coverages of all busy submeshes.

Definition 7. The reject area or submesh with respect to a job *J*, denoted by R_J , is a set of processors such that use of any node in R_J as the base of free submesh for the allocation of *J* will cause the job *J* cross the boundary of the mesh.

Definition 8. The sink of the reject area is the processor with coordinates $<w - p + 1, h - q + 1>$.

Definition 9. A base block with respect to a job *J* is a submesh which nodes can be used as base for free submeshes to allocate job *J*. A set of disjoint base blocks is called the base set.

Definition 10. External fragmentation is the ratio of the number of free processors to the total number of processors in the mesh, when the allocation of incoming task fails but there is a sufficient number of free processors.

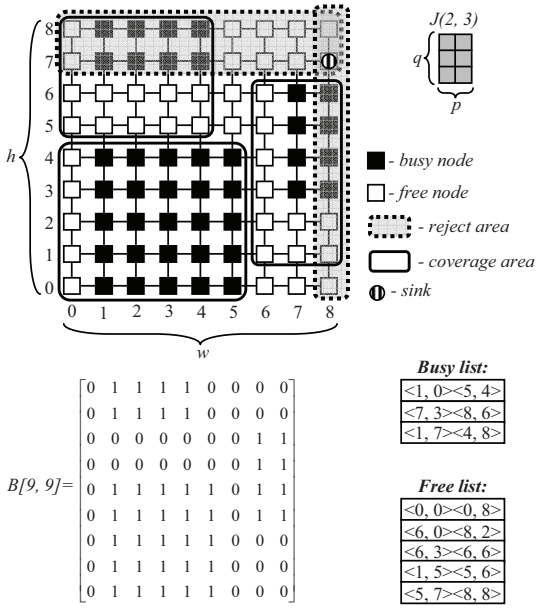

Fig. 2. A mesh M(9, 9), busy and free nodes, sink, coverage areas, reject areas, busy array, busy list and free list.

As an example, a mesh *M(9, 9)*, busy and free nodes, sink, coverage areas, reject areas, busy array, busy and free lists with respect to *J(2,3)* are presented in Fig. 2. The busy nodes

are marked using black color while free using white color. The reject area is presented by the shaded region with doted edges. The coverages of busy submeshes β_1=[<1, 0><5, 4>], β_2=[<7, 3><8, 6>] and β_3=[<1, 7><4, 8>] are $\xi_{\beta1,J}$=[<0, 0><5, 4>], $\xi_{\beta2,J}$=[<6, 1><8, 6>] and $\xi_{\beta3,J}$=[<0, 5><4, 8>] respectively. The coverage set $C_J = \xi_{\beta1,J} \cup \xi_{\beta2,J} \cup \xi_{\beta3,J}$. The reject area R_J = [<0, 7><8, 8>]∪[<8, 0><8, 8>] and its sink has coordinates <8, 7>. The base set is [<5, 5><5, 6>] ∪[<6, 0><7, 0>]. Both internal and external fragmentations in presented case are equal zero.

For a given job J, $C_J \cup R_J$ represents the set of processors, which can not be the base of the free submeshes. Thus the base set for J is $Z - C_J - R_J$, where Z refers to the set of all processors in the system. It is important to note, that the base set with respect to $J(p, q)$ is different from this with respect to $J(q, p)$, when $p \neq q$.

3. ALLOCATION ALGORITHMS

3.1 A Status of Processors as Busy Array

The solution with a bit map representing the allocation status of processors in the mesh was presented by Zhu. Based on that idea, two allocation schemes were presented: The First Fit (FF) and the Best Fit (BF). With respect to an incoming job, the busy array B (without considering reject area) is scanned in to create a coverage array C_T, which is a bit map representing a coverage set. In order to form the C_T in efficient way, each coverage $\xi_{\beta,J}$ is divided into three regions: job coverage, left coverage and bottom coverage (Fig. 3).

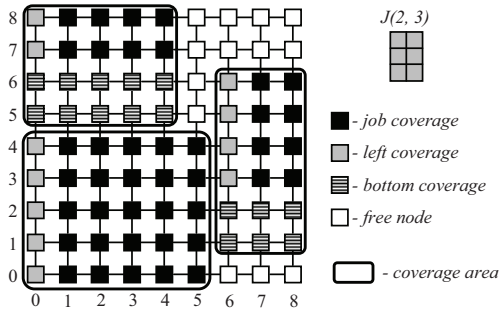

Fig. 3. Coverages and their three regions with respect to J(2, 3) .

Then, two scans are necessary: 1) All rows from right to left (determining a job and left coverages); 2) All columns (left to right) from top to down (creating a bottom coverage). The FF strategy returns the first available node that does not belong to the coverage set while the BF selects the base which has the maximum number of busy neighbors. Thus, in the BF strategy in each case, the bit map is scanned two times while for the FF in some cases second scan can be interrupt (when base node is found). Simulation experiments show that system utilization and external fragmentation are almost the same for both schemes. However the FF algorithm is more efficient and simple, that led to the selection of FF in practical implementations (Zhu).

With the FF solution, the achieved time complexity is $O(wh)$ for both allocation and deallocation. The drawback of the algorithm is lack of recognition completeness, which is the result of considering only the fixed orientation of tasks.

Deallocation of processors in the systems with a busy array reduces to clearing of all the elements in bit map B. It is also done in $O(wh)$.

3.2 A Busy List

The replacement of bit map by a list with busy subgrids was proposed by Chmaj et al., Chuang and Tzeng, Das Sharma and Pradhan, Ding and Bhuyan, Yoo and Das, and Yoo and Youn. As in the bit map case, each technique with busy list is based on the fact that for any job J none of the nodes inside $C_J \cup R_J$ can serve as the base. So, for each J, creating C_J and R_J is a common first step for algorithms with busy list. Because of the fact that allocation status of processors is maintained here using busy list, a C_J is also in a list form and it is constructed by scanning the busy list, and for each β in the list $\xi_{\beta,J}$ is constructed. For a given β=[<x_b, y_b><x_e, y_e>], its coverage with respect to $J(p, q)$ is $\xi_{\beta,J}$=[<x_c, y_c><x_e, y_e>], where $x_c=max(0, x_b - p + 1)$ and $y_c=max(0, y_b - q + 1)$. R_J is found by calculating the sink.

Finding the node that is not in $C_J \cup R_J$ and the strategy of choosing one node (if many) is the next step that differs from the above mentioned solutions. The first strategy with busy list was Frame Sliding (FS) – Chuang and Tzeng. The FS for the given job $J(p, q)$ maintains a frame of size $p \times q$, which is identified by lower left corner. The frame slides through the mesh, starting from the lowest left available node. When nodes in the currently examined frame are not available, the frame slides over the mesh by taking horizontal and vertical steps equal to the width and height of the frame respectively. The frame concept in the FS is applied by checking if lower left corner of the frame is not in $C_J \cup R_J$ (the coverage list is searched for each considered node) and if it fails (lower left corner is in $C_J \cup R_J$) by moving the corner according to the size of the frame. With the FS solution, the achieved complexity is $O(whB)$, where B is the length of the busy list. The FS technique does not have complete submesh recognition ability due to two factors: 1) Fixed strides of frame; 2) Fixed orientation of tasks.

The advanced version of the FS algorithm with recognition completeness is proposed by Ding and Bhuyan. The Adaptive Scan (AS) strategy for every requested job J goes through the mesh, node by node, taking horizontal and vertical steps respectively, that differentiates it from the FS. Each node, belonging to $C_J \cup R_J$ is tested by going through the whole coverage list. Membership to $C_J \cup R_J$ causes the algorithm to check another node while in the other case, node can be a base for a given J. Recognition completeness is achieved through node by node verification and by considering two job orientations: If allocation of $J(p, q)$ fails then $J(q, p)$ possibility is checked. Similar to FS, the complexity of AS is $O(whB)$.

Another development in FS and AS strategies is presented in the Quick Allocation (QA) algorithm (Yoo and Youn). In the

QA scheme individual checking of the nodes was replaced by testing only each row in the mesh. It is achieved by introducing an one dimensional array called *last_covered*, which remembers the horizontal coordinate (x-coordinate) of the rightmost covered node for each row. Thus, instead of going through each node in the row, it is enough to check, if horizontal coordinate in the *last_covered* for the node under consideration is not in R_J. If so, the *last_covered* value for the row is the base for the requested job *J*. A key issue here is the method of creating and updating the *last_covered* array. It is done by going through the coverage list and analyzing the horizontal coordinates of $\xi_{\beta,J}$. In order to make this analysis possible in a single pass of the coverage list, it is necessary to sort coverages $\xi_{\beta,J}=[<x_c, y_c><x_e, y_e>]$ in the increasing order of x_c. The QA technique is recognition complete and according to computer simulations (Yoo and Das, Yoo and Youn), is faster and more efficient than the earlier described algorithms based on busy list. The complexity of the QA solution is $O(hB)$.

All the algorithms so far presented find the base for the job by maintaining a busy list and scanning the nodes or rows of the mesh. The ADJacency (ADJ) strategy (Das Sharma and Pradhan) initiates another approach, which eliminates the need for scanning. For requested job *J* the ADJ compares the corners of *J* with the corners of each $\xi_{\beta,J}$ in the coverage list. If a submesh *J* overlaps with some of the allocated subgrids, the submesh *J* slides along its boundary. It adjoins the busy submeshes as much as possible, that reducing the chances of external fragmentation. The ADJ is recognition complete and as computer simulations showed (Das Sharma and Pradhan), only the QA scheme from former busy list algorithms is faster than the ADJ (Yoo and Youn). It is important to note that the ADJ is the first technique, where the allocation time does not depend directly on the size of the mesh, but it varies along with the size of the busy list (we don't have scanning of mesh structure, we are searching the busy list). The complexity of the ADJ is $O(B^3)$.

The idea from ADJ of getting rid of scanning the nodes or rows and considering only the coordinates of submeshes was the foundation for creating the innovative allocation scheme – the Stack-Based Allocation (SBA) algorithm (Yoo and Das). The SBA uses coordinate calculations and spatial subtractions in order to find a base. To increase the efficiency of the algorithm, main steps of the scheme are performed using a stack. For a requested job *J,* the initial candidate base block I_J is determined and it gives the first set of candidate blocks B_J: $I_J=[<0, 0><x_s - 1 , y_s - 1>]$, where $<x_s, y_s>$ is the sink. The coverages $\xi_{\beta,J}$ in C_J in form of a busy list are then spatially subtracted from the B_J : if the first coverage $\xi_{\beta,J}$ interests with any block in B_J, then the $\xi_{\beta,J}$ is subtracted from the intersecting block of B_J, that gives a new blocks for B_J (the B_J is updated). Afterwards, the next $\xi_{\beta,J}$ is checked (if it intersects with any block in the updated B_J) and if so, it is subtracted from B_J , and so on. This continues until all $\xi_{\beta,J}$ in C_J are considered. If B_J is not empty after subtracting all $\xi_{\beta,J}$, any node from B_J can be a base for *J*. The key idea of the algorithm that makes it very efficient is to implement B_J as a stack. A candidate block on the top of the stack is always compared with next $\xi_{\beta,J}$ to see if they intersect with each other. When new blocks for B_J are generated from a spatial subtraction, they are pushed onto stack, replacing the top element. Each block on the stack has a pointer to the next $\xi_{\beta,J}$ that the block should be compared with. When a block on the stack with a null pointer appears on top of the stack, the desired base block is obtained. If the stack becomes empty, then the allocation fails.

The complexity of the SBA was $O(B^2)$. But it is proved by Zhang, that the actual time complexity is $O(B^3)$. The SBA technique was improved by Chmaj et al., where the Improved Stack Based (ISBA) algorithm is proposed. Both, the SBA and ISBA algorithms use manipulation of job orientation to obtain complete submesh recognition ability. However, when job $J(p,q)$ has both *p* and *q* sizes equal ($p=q$) there is no need to change job orientation – because it causes that the algorithm is executed two times with the same job $J(q,q)$. This drawback is eliminated in the ISBA. In systems with busy list, deallocation of a job requires removal of an element from the busy list. It can be done in constant time by implementing pointers from allocated jobs to corresponding elements in the busy list. Thus, the time complexity of deallocation is $O(1)$.

3.3 A Free List Solution

Another approach to the processor allocation problem is not keeping in memory the information about subgrids that are busy, but maintaining a list with free subgrids (keep the list of processors that can be allocated for a requested job) – Ababneh, Kim and Yoon, Liu et al. The allocation seems to be simple and fast. To allocate a job, the free list is scanned to find a free submesh, which is large enough to accommodate the job. Problem appears while deallocating, when a subgrid is released and can be joined with another, that would create greater subgrid (in this way we could accept bigger job if requested). Such an expansion is necessary in order to preserve recognition completeness of scheme.

The Free List (FL) algorithm (Liu et al.) maintains a free list, where subgrids can overlap and they are sorted in the no decreasing order of their size. So, for a requested job *J,* the free list is searched from the beginning for the first free candidate submesh whose size is equal or larger than *J*. Both orientations of *J* are checked. In order to reduce external fragmentation, four corners of the candidate submesh are considered and one with the highest boundary value is actually allocated (boundary value is calculated similar to the ADJ – Das Sharma and Pradhan). The deallocation process involves expansion of free submesh with free subgrids in the free list. This operation is hard and as it is reported by Ababneh, the FL technique can miss sometimes the biggest submeshes that cause the scheme to lack recognition completeness. The allocation and deallocation complexity of the FL is $O(F^2)$, where *F* is the length of a free list.

The drawbacks of the FL are addressed by Kim and Yoon. The Free Submesh List (FSL) strategy maintains two lists: the free list with no overlapping free submeshes and the busy list with busy subgrids. For an allocation request, the FSL

scans the free list (it is ordered in the no increasing order) and generates a sublist with candidates of the desired size. Then, the algorithm evaluates candidates by using the reservation factor, which decreases external fragmentation. In the deallocation process, the requested job *J* is removed from the busy list and the free list is updated to one initial free submesh I_F=[<0, 0><w – 1 , h – 1>]. Afterwards, based on the I_F and busy list, new free subgrids are generated. In this way, the largest possible submeshes are always on the free list and their expansion is not required. Furthermore, the FSL scheme is recognition complete. Drawbacks of the solution are the necessity of two lists (busy and free) and higher time complexities: $O(F^2)$ for allocation and $O(F^3)$ for deallocation.

Disadvantages in both the free list techniques, lack of recognition completeness in the FL and high complexity in the FSL case were foundations for creating the newest scheme – the Compacting Free List (CFL) algorithm (Ababneh). The CFL maintains unordered list of possibly overlapping free submeshes. For requested job *J* the first free subgrid that is large enough to accommodate *J* is selected. Both orientations of *J* are considered and also four corners of aspirant subgrid are tested as the base – one with the highest boundary is chosen. If *J* was smaller than a candidate submesh, it is subtracted from the submesh and results are added on the head of list. Also, the allocated submesh is subtracted from the remaining free submeshes that overlap with it – results are put on the head of list. Deallocation is divided onto two phases: in the first, the deallocated submesh is expanded into elements in the free list, in the second the members of the free list are expanded into subgrids expanded in phase one. Together with two types of proposed expansions, the CFL does not miss maximal free subgrids, that was sometimes the case in FL, thus the CFL is recognition complete. The time complexity of the CFL is linear for both allocation and deallocation, and equals $O(F)$.

All presented techniques with implementation of free list are complicated in both allocation and deallocation phases. Computer simulation results show that even the best of free list strategies, the CFL scheme with linear time complexity, is not better than previous busy array and busy list strategies (Ababneh). Necessity of often expansion, scanning and sorting (the FL) makes solutions with list of free subgrids not efficient and less attractive in comparison to the others.

4. MEMORY UTILIZATION

4.1 Busy Array

The algorithm based on bit map maintains two busy arrays: i) Mesh bit map – with allocation status for each node; ii) Coverage bit map – created for each incoming job and informing which node can be the base for the task. Size of array depends on size of mesh, thus it is $w \times h$. Because it is a bit map, size of each cell is *1 bit*, so, finally size of required memory is: $2 \times w \times h \times 1$ *[bits]*.

4.2 List Solutions

The busy list schemes need two lists: i) Busy list – list with coordinates of busy submeshes; ii) Coverage list – created for each requested job and keeping coordinates of coverages. We need to assume, that in the worst case each node hosting one separate job, that implies separate busy submesh for each node, thus the length of busy and coverage list is $w \times h$. In the case of the free list algorithms, also two lists are necessary. For the FL and CFL techniques, we need two lists with free subgrids: i) Free list – with coordinates of free lists; ii) Temporary free list – created for each considered job. For the FSL procedure, we need two free lists (main and temporary) and one busy list.

It is important to notice, that on the list we have coordinates. The size of each coordinate depends on the horizontal and vertical size of mesh *w* and *h* respectively. To encode the addresses of nodes using natural binary codes, we need *n* and *m* bits for *w* and *h* sizes respectively, where: $2^n \geq w - 1$ and $2^m \geq h - 1$. Let's consider the mesh $w = 12$ and $h = 8$. We need to be able place in a memory addresses of nodes from <0, 0> to <11, 7>. Using natural binary codes, we need $n = 4$ and $m = 3$ bits to store one horizontal and vertical coordinate respectively.

Structure of the list itself becomes a sensitive factor. In the Fig. 4, two list structures are shown. In the case (a), beside coordinates of busy subgrid, each element includes pointers to previous and next element. Such a solution ensures very effective memory utilization from data structure point of view, also scanning of the list limits only to valid memory entries – cells without busy submesh are not read. Size of each cell in this case consists of: i) $n + m$ bits for both pointers *prev* and *next*; ii) $n + m$ bits for lower left and upper right corner of subgrid. Size of the whole the busy list is $w \times h$ times each entry, thus finally with the consideration of two lists that are required for the algorithms, we have: $2 \times w \times h \times (4 \times n + m)$ *[bits]*. Case (b) on the Fig. 4 presents memory minimized structure, where with each element of the list a validation bit is associated. If the element includes busy subgrid, the bit is set to one, while for empty entries, the bit is set to zero. In this case, each memory cell has to be scanned, but only one bit is checked in order to get info about validation of the memory cell. Size of each cell contains one validation bit and $n + m$ bits for lower left and upper right corner of subgrid, thus for two busy lists we have: $2 \times w \times h \times (1 + 2 \times n + m)$ *[bits]*.

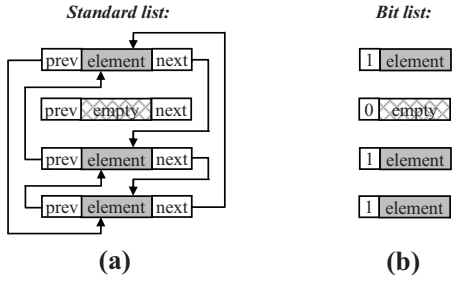

Fig. 4. 3-element list structures: (a) Standard list; (b) List with bit validation position.

4.3 Experimental Results

In Fig. 5, size of the memory as a function of mesh size for the three solutions is plotted. The solution with the bit map outperforms the remaining approaches in terms of the size of memory usage. Also, as we could expect, the list implemented with validation bit takes less space than the standard list. In all the considered cases, the difference becomes more significant with increasing size of mesh.

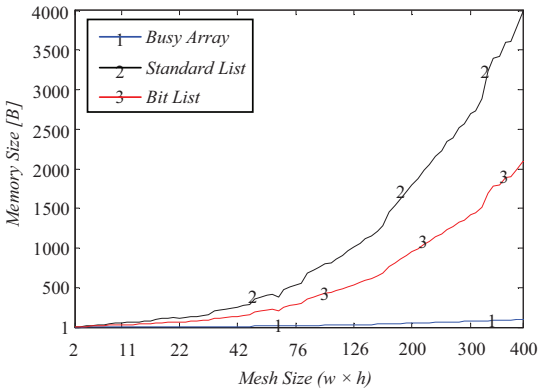

Fig. 5. Size of using memory in the system with size of mesh M(w, h).

5. FINAL REMARKS

Processors currently available on the market contain few cores integrated on one die. The CMPs with more cores, with the network implemented on chip, are expected to become the main strategy for high efficiency, future processors. Technology scaling allows integration of billions of transistors on a chip, however reasonable management of available resources provides better performance, space and energy utilization and heat dissipation. Through the concept of integrating the JS and the PA in one die with the CMP, the operating system is made redundant and better efficiency is achieved. In this paper, we investigated memory consumption caused by the PA. All former important processor allocation techniques were studied and discussed in terms of their memory utilization.

Memory analysis reveals the huge advantage of the busy array as an architecture to be used to hold the status of processors in use. The standard list, which is the best solution from data structure point of view, consumes the largest amount of memory. The bit list structure gives better prospects for list solutions, however its results are weak in comparison to bit map strategy. The experimental results shown in this paper demonstrate that the PA based on a busy array uses less amount of memory, that can provide lower area and energy consumption on a chip.

REFERENCES

Ababneh, I. (2006). An efficient free-list submesh allocation scheme for two-dimensional mesh-connected multicomputers. *Journal of Systems and Software*, vol. 79, no. 8, pp. 1168-1179.

Chmaj, G., Zydek, D., Koszalka, L. (2004). Comparison of task allocation algorithms for mesh-structured systems. *In Comp. Systems Engineering Theory & Applications, 4th PBW*, pp. 39-50.

Chuang, P.J. and Tzeng, N.F. (1991). An Efficient Submesh Allocation Strategy for Mesh Computer Systems. *Proc. Inter. Conf. Distributed Computing Systems*, pp. 256-263.

Das Sharma, D. and Pradhan, D.K. (1993). A Fast and Efficient Strategy for Submesh Allocation in Mesh-Connected Parallel Computers. *Proc. 5th IEEE Symp. Parallel and Distributed Processing*, pp. 682-689.

Ding, J. and Bhuyan, L.N. (1993). An Adaptive Submesh Allocation Strategy for Two-Dimensional Mesh Connected Systems. *Proc. Inter. Conf. on Parallel Processing*, vol. 2, pp. 193-200.

Held, J., Bautista, J., Koehl, S. (2006). From a Few Cores to Many: A Tera-scale Computing Research Overview. *White Paper, Intel Corp.*

Kim, G. and Yoon, H. (1998). On Submesh Allocation for Mesh Multicomputers: A Best-Fit Allocation and a Virtual Submesh Allocation for Faulty Meshes. *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 2, pp. 175-185.

Liu, T., Huang, W.-K., Lombardi, F., Bhuyan, L.N. (1995). A Submesh Allocation Scheme for Mesh-Connected Multiprocessor Systems. *Proc. 1995 Inter. Conf. Parallel Processing*, vol. II, pp. 159–163.

Vangal, S. et al. (2007). An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. *IEEE International Solid-State Circuits Conference (ISSCC 2007)*, pp. 98–589.

Yoo, B.S. and Das, C.R. (2002). A Fast and Efficient Processor Allocation Scheme for Mesh-Connected Multicomputers. *IEEE Transaction on Computers*, vol. 51, no. 1, pp. 46-60.

Yoo, S.-M. and Youn, H.Y. (1997). An Efficient Task Allocation Scheme for Two-Dimensional Mesh-Connected Systems. *IEEE Trans. Para. & Distr. Systems*, vol. 8, no. 9, pp. 934-942.

Zhang, L. (2003). Comments on "A Fast and Efficient Processor Allocation Scheme for Mesh-Connected Multicomputers". *IEEE Trans. on Computers*, vol. 52, no. 2, pp. 255-256.

Zhu, Y. (1992). Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *Journal of Parallel and Distr. Computing*, vol. 16, no. 4, pp. 328-337.

Zydek, D. and Selvaraj, H. (2009). Processor Allocation Problem for NoC-based Chip Multiprocessors. *Proc. of 6th International Conference on Information Technology: New Generations (ITNG 2009)*, pp. 96-101.

Zydek, D., Shlayan, N., Regentova, E., Selvaraj, H. (2008). Review of Packet Switching Technologies for Future NoC. *Proc. of 19th International Conference on Systems Engineering (ICSEng 2008)*, pp. 306-311.