Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 13 (8): 1192-1198, 2013
ISSN 1812-5654 / DOI: 10.3923/jas.2013.1192.1198
© 2013 Asian Network for Scientific Information

A Cache-conscious Structure Definition for List

YuanZhang Li, Yu-An Tan, WenMing Wang, QiKun Zhang and Zuo Wang
School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China

Abstract: A list data structure is a collection of nodes accessible one after another beginning at the head and ending at the tail. The standard way to implement a linked list is to have each node of the list contain both its value and a pointer indicating the location of the next node in the list. However, such design is not suitable for traversal since accessing its loosely distributed nodes may trigger frequent cache replacements. This study presents a novel structure definition for list, the principle of which is to extract the frequent-accessed elements from each node and store them in contiguous memory space. Besides, adjust the layout of contiguous memory space to match the data access type of traversal. With the help of the modified data distribution, a cache miss will load not only the data required for traversing to current node but also the data required for traversing to next nodes and thus reduce the number of cache misses as well as the required cache resource. The experimental results show that, compared with the standard list design, the traversal efficiency is improved significantly.

Key words: Linked list, traversal, cache replacements, contiguous memory space

INTRODUCTION

The standard way of implementing a linked list is to have each node contain both its value and a pointer indicating the location of the next node in the list. Besides the standard way, there are a lot of other list implementations, such as the Linux kernel list [http://kernel.org/pub/linux/kernel] and the Glib list [http://ftp.gnome.org/pub/gnome/sources/glib/2.22/]. As shown in Fig. 1a, although these implementations may differ from each other, almost all of them follow the standard wary to implement list and provide the same operations such as Insert, Remove and Search. Except for inserting or removing an item at the head or the tail, list need carry traversal to find the target location where to insert or the node that need want to remove. Thus traversal is a very important operation. However, for most list implementations, the traversal efficiency is poor.

In the standard way to implement a list, the node structure is pre-defined by users. When users want to insert a new node into list, users call system APIs, such as malloc (size of (_NODE)), to allocate a contiguous space to store the elements defined in the structure _NODE and then launch a traversal to find the target location to insert this new node. For each traversal, there are two important elements: The next Node's Pointer (NNP) which indicates the location of the next node and the Current Node Identification (CNI) which distinguishes one node from others. In fact CNI is not as fixed as NNP since user may choose element A as the comparison tag to search nodes in this traversal while choosing another one in next traversal. Besides, there may be more than one CNI for one traversal.

In order to simply the discussion, assume a list only has one CNI. Let's imagine how traversal works: If processor tries to traverse to current node and current node's CNI does not exist in cache, processor need firstly load the data locating in the same cache line space with CNI into cache, then read CNI from the loaded cache line and perform compare operation. If current node is not the target node and the current node's NNP does not exist in cache, processor also need firstly load the data locating in the same cache line space with NNP into cache, then fetch NNP from the loaded cache line and perform the same action to the next node. It seems that all the above actions follow a logical train of thought. However, think it over, finding that accessing the loosely distributed nodes may cause poor cache utilization: Only a little portion of the loaded cache line will be utilized by traversal. Besides, go a step further, finding that the impact of low cache utilization in traversal is more serious: Frequent cache line replacements will decrease the performance of other applications as well as the total throughout of system. Thus, it need improve the standard implementation of list so as to leverage the hardware cache in modern processors.

A program's performance can be improved by exploiting its data locality (Gorman, 2004) "Field layout restructuring" technique and "pool allocation" technique

Corresponding Author: YuanZhang Li, School of Computer Science and Technology, Beijing Institute of Technology, 5, South Zhongguancun Street, Haidian District, Beijing, 100081, People's Republic of China

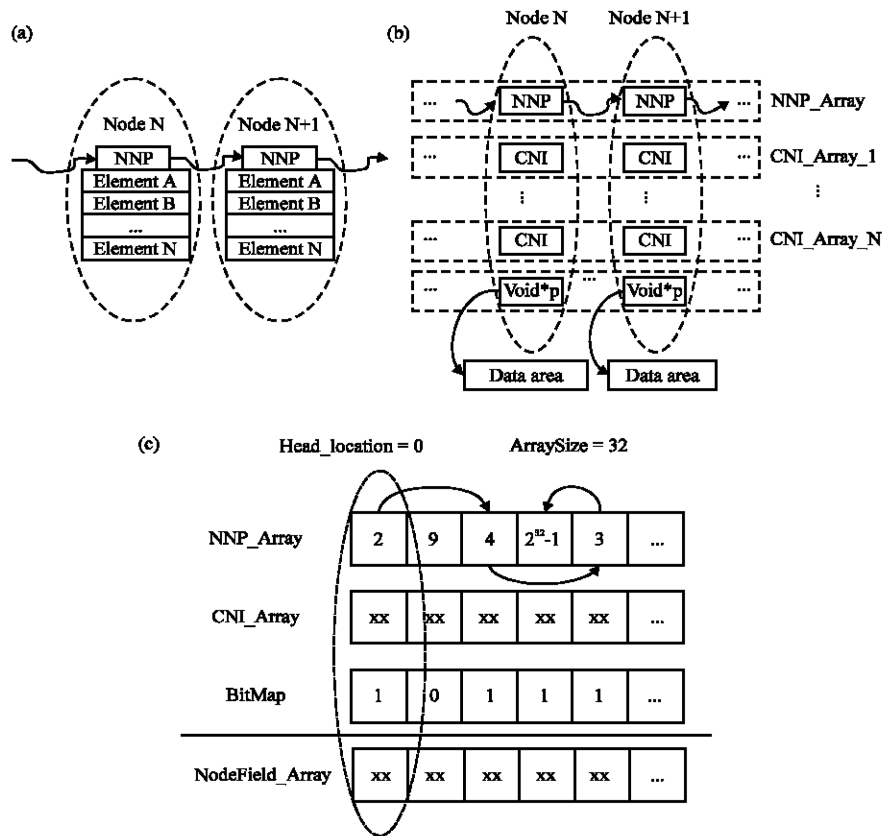

Fig. 1(a-c): Implementation and design of Array List (a) Implement of linked list (b) Frame design of Arraylist and (c) Implementation for a single list

are widely used to modify the heap layout of programs cache (Jeon *et al*., 2007; Yang *et al*., 2008; Lattner and Adve, 2005). The former aims to collocated distinct instances so as to increase reference locality 5 while the later focuses on fields within structures (Mannarswamy *et al*., 2009; Thangaraj *et al*., 2006). Some study (Qin and Mishra, 2009; Lin *et al*., 2007) statically analyzes memory access behaviours at compile time while study (Shin *et al*., 2006) are both profile-based. Profiling is sensitive to the inputs and execution environments and thus is preferred by most embedded systems. For static analysis, extra changes to compiler as well as a longer compile time will be also spent.

All the above works focus on the general process methodology for different data structures and varied data access types. Besides, neither profiling nor static analyses is just a "tool" used to find the data that are simultaneously referred. For some specific data structures such as list, software designers, who rule the program behaviour, may have more precise and detailed data reference information than any tool. Thus, using the experiences of "Field layout restructuring" and "pool allocation" techniques to redefine traditional data structure seems a more efficient choice in some cases. In this study, it demonstrates how to redefine traditional list structure.

PRINCIPLE OF ARRAYLIST DESIGN

Frame design of ArrayList: The principle of ArrayList is to extract the frequent-accessed elements, called "hot data", from node structure and store them in contiguous main memory space. As shown in Fig. 1b, NNP and CNIs are extracted from node structure to create NNP_Array and CNI_Arrays while other elements of node structure are stored in "data area" recorded by a second-level pointers "void *p". Since each cache miss will load the "nearby" data, any cache miss occurs in NNP_Array or CNI_Arrays may load some NNPs and CNIs which is required for traversing to next nodes and thus leverage the hardware cache. It is emphasize two things here. First, a list may have more than one CNI or no CNI. If there is no CNI for a list, traversal loads the data in "data area" into cache and thus can not achieve any improvement in efficiency.

Second, the basic requirement for an element to be a CNI is fixed size. Obviously strings with varied length are not suitable for CNI. Besides, small size CNI is preferred. The smaller the size of CNI is the better efficiency improvement to achieve.

On the assumption that a list has only one CNI, It shows how to design a single list. As shown in Fig. 1c, defining four Arrays in the following:

- **CNI_Array:** This memory space is used to store the CNIs for each node
- **NNP_Array:** This memory space is used to store the NNPs for each node
- **BitMap:** This memory space is used to store the valid bits for each node
- **NodeField_Array:** This memory space is used to store the pointers to "data area" for each node

NNP is not a pointer but an unsigned integer which records the next node's node location. Note that it use "node location" but not "node" because a node location may not really store a node. For example, BitMap [Index] = 1 means that node location Index does store a node which contains its elements: CNI_Array [Index], NNP_Array[Index] and NodeField_Array [Index]. Besides, using an integer variable Head Location to describe the Index of the head node. It also use an integer variable ArraySize to record the number of node locations. As shown Fig. 1c, HeadLocation = 0 means node location 0 store the head node while ArraySize = 32 means there are 32 available node locations. According to BitMap, HeadLocation and NNP_Array, the node sequence (from head to tail) is node location 0→2→4→3.

When initializing an ArrayList instance, users are required to specify the number of CNIs as well as the size of each CNI. For example, if C++ language is used to implement ArrayList, it use "ArrayList object (CNI number, size, size,…size)" to declare an ArrayList instance. Besides, user should specify the size of NPP otherwise ArrayList will use a 4-byte unsigned integer as NNP in which the number of maximal nodes locations is 232-1 (valid NNP range from 0 to 232-2 while 232-1 is used to indicate list tail).

After an ArrayList instance is initialized, a few node locations are allocated. As user carries insert operations continuously, the number of nodes exceeds the maximized node locations which current memory space can provide. At this time, ArrayList firstly allocate a new space in main memory which is twice bigger than the original one, then copy the content of the original space to the new one, at last free the original memory space. The above operations is called "Space Double Procedure".

```
Unsigned Int32 Search_CNI (Target_CNI)
{
Unsigned Int32 Index = HeadLocation;
While (Index<2^32-1)
{
If (CNI_Array [Index] = = Target_CNI)
{Return Index; }
else [Index]; }
}
Return 2^32-1;
}
```

Fig. 2: Initializing an ArrayList instance

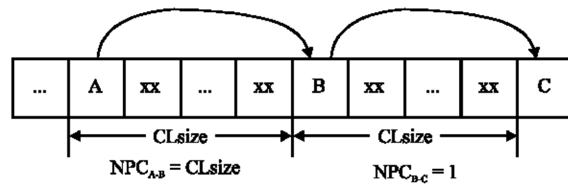

Fig. 3: Poor performance behaviour of traversal ArrayList

As mentioned before, traversal is the most important operation for list, ArrayList simulates the traditional traversal from head to tail. Figure 2 demonstrate the process flow of traversal on the assumption that 4-byte unsigned integer is used as NNP.

Improvement of traversal ArrayList: As shown above, the standard list design may cause poor cache utilization. However, the situation seems different for ArrayList. Taking NNP_Array for example, one cache miss will not only load the target NNP but also load other NNPs which locate in the same cache line space with the target NNP. In such condition, when traversing next nodes, processor directly fetches the required NNP from cache. If define the size of NNP as NNP_Size, the size of CNI as CNI_Size and the size of a cache line as CLSize, the total number of cache misses, triggered by traversing N nodes, can be described as:

$$\text{Total} = N/(\text{CLSize}/\text{NNP_Size})+N/(\text{CLSize}/\text{CNI_Size}) \quad (1)$$

According to Eq. 1, if both NNP_size and CNI_size are 4 while CLSize is assigned to 128, the total cache misses is N/16. Compared to the N cache misses triggered by traversing N nodes in standard list design, the traversal efficiency of ArrayList increase significantly. However, this is not true for some cases. Figure 3 demonstrates the situations in which ArrayList behaves poor performance. Node A, B and C locate loosely in

different cache line space. Using Next Hop Count (NPC) to indicate the distances of the node locations of successive nodes. As shown in Fig. 3, because NPCA-B and NPCB-C is CLsize, traversing from node A to C will trigger 3 cache line replacements. If the data loaded into cache will not be utilized by traversing to next nodes, ArrayList may not achieve performance improvement. Besides, if the data loaded into cache is useful for "far-later" nodes but not for "near-later" nodes, ArrayList may not achieve performance improvement. For example, if the competition for cache resource is serious, frequent cache replacements may replace these loaded cache lines out before utilizing them and thus decrease traversal efficiency sharply. This study proposes two methods to resolve this problem: One method is to make NPCs smaller; the other method is to change traditional traversal.

Improvement for traditional traversal: As mentioned before, NPC plays a very important role in the efficiency of traditional traversal. Thus it present a new design to keep NPC at low level: Try to insert a new node into an empty node location which is near target node's location. The word "near" here means locating in the same cache line space.

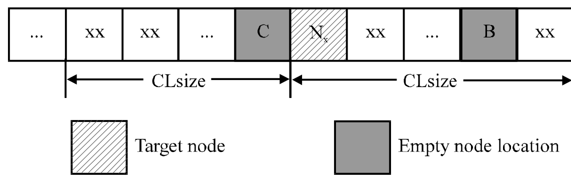

Fig. 4: Method of inserting a new node into an empty node

Inserting a new node into an empty node location, as shown in Fig. 4, location B is near target node Nx while location C is not. Thus, inserting a new node, which is the next-node of target node Nx, into list, then choose location B. However, if there is no empty node location near target node's location, a relocate procedure is launched.

As shown in Fig. 5, try to insert a new node after node A, the flow of operations is described as following:

- Search the empty node location near the target node's location. If there is, insert the new node into the empty node location; if not, jump step 2
- Search an empty cache line space (a cache line space full of empty node locations). If success, jump step 3; if fails, carry "Space Double Procedure" and then jump step 2
- Move next half-line nodes after node location. A into the empty cache line space and then jump step 1
- There is no doubt that the improved Insert operation will keep NPC at low level and thus increase the efficiency of traversal significantly. But relocate procedure seems consume much more main memory space: In order to obtain an empty cache line space, ArrayList may need to carry "Space Double Procedure" even if there are empty node locations. However, it is not true. The new memory space allocated by "Space Double Procedure" is only user memory space but not physical memory. Indeed physical memory is only allocated when several instructions really access it (Gorman, 2004). Thus, there is no need to worry about extra memory cost

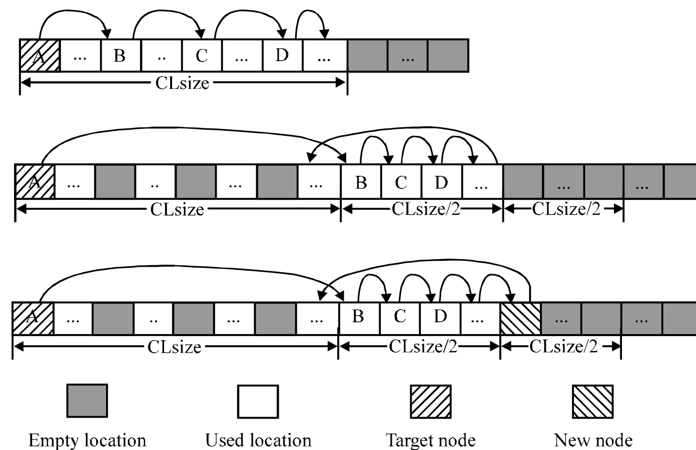

Fig. 5: Process of inserting a new node to the empty cache line space

```
Unsigned Int32 Search_CNI (Target_CNI)
{
 Unsigned Int32 Index = 0;
 While(Index < ArraySize)
 {
  If(CNI_Array [Index] = = Target _CNI && BitMap [Index] = = 1)
  { Return Index; }
  else
  { Index ++; }
 }
 Return 2^32-1;
}
```

```
Unsigned Int32
Search_Location ( )
{
 Unsigned Int32 Index = 0;
 While (Index < ArraySize )
 {
  If (BitMap [Index] = = 0)
  { Return Index; }
  else
  { Index ++; }
 }
 Return 2^32-1;
}
```

Fig. 6: Process of traversal searches the node location

Scheme of direct traversal ArrayList: Unlike traditional traversal to traverse nodes by order, direct traversal searches the node locations one by one, from index = 0 to ArraySize Fig. 6. It shows the process of direct traversal.

Since node locations are searched one by one according to the increasing Index, the data loaded into cache will be inevitably utilized by traversing next nodes. Besides, inserting node into the first empty location will be the best choice to cooperate with direct traversal since it reduces the number of search actions to empty node locations. However, there are a few limits for direct traversal. For example, CNI must be unique. If two or more than two nodes have the same CNI, the search result may differ from traditional traversal.

RESULTS AND DISCUSSION

Efficiency improvement of ArrayList focuses on two fields: One field is the improvement benefit from high cache utilization; the other is the improvement benefit from high data locality (low NPC). First compare traversal time costs between ArrayList and standard list design in which there is no other task or thread competing for cache resource and then evaluate the improved traditional traversal and direct traversal. The scheme platforms a PC with an AMD Dual-core processor with 64 KB L1 D-cache and 512 KB L2 cache (the size of cache line is 128 bytes). Since modern processors are too fast to measure the time cost by normal metrics such as m sec, using the following instructions to get a 64-bit value, such as Fig. 7, which is similar to "CPU cycles".

Using Glib list as the standard list design and implement a ArrayList class using 4-bytes unsigned integer as CNI. Using ArrayList and Glib list to build the same list instances ("same" means the same CNIs and the same node order) with list length varied from 128, 256, 512, 1024, 2048, 4096 and 8192. Generating a sequence of 128*32 Insert and Delete operations and apply them to each instance. Figure 8 compares the time cost of Insert and Delete operations between ArrayList and Glib list in which there is no other task competing for cache resource. The horizontal axis is the average list length while the vertical axis is normalized "CPU cycles" consumed by Insert and Del operations.

```
inline unsigned __int64 GetCycleCount()
{
__asm _emit 0x0F
__asm _emit 0x31
}
```

Fig. 7: Instructions for calculating time cost

As shown in Fig. 8, when list length is small, the time cost of ArrayList is reduced by 14~16%. However, as the average list length increases, the gaps between ArrayList and Glib list increase significantly. The reason is that: Since there is no other task or thread competing for cache resource, all cache resource is available for list traversal; when list length is small, even if cache utilization is poor, L1 D-cache and L2 cache can hold all data used for traversal; however, as list length increases, more and more data can not reside in cache and thus the efficiency decreases sharply.

In the following, it evaluate the improved traditional traversal and direct traversal. Firstly, measuring traversal efficiency in which there is no task or thread competing for cache resource and then run several data-intensive threads simultaneously to simulate the situation in which cache resource is in short supply. Figure 9a and b compare the traversal efficiency between the improved traditional traversal and direct traversal. The horizontal axis is average list length while the vertical axis is normalized "CPU cycles" consumed by Insert and Del operations.

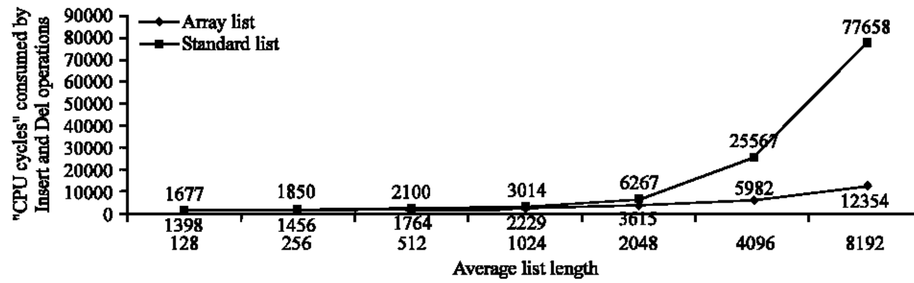

Fig. 8: Comparisons between ArrayList and Standard list

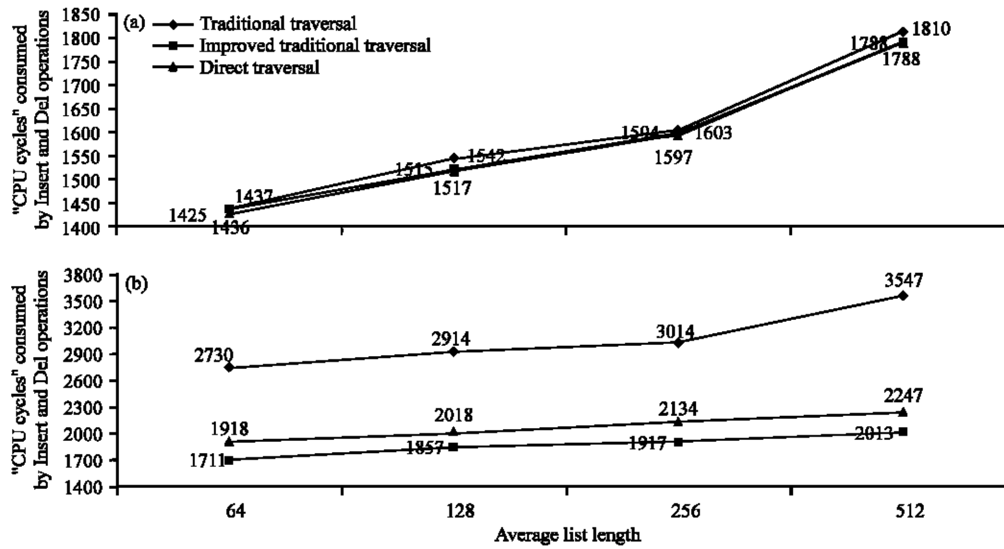

Fig. 9(a-b): Compare of efficiency between traditional traversal and direct traversal, (a) Traversal efficiency comparisons under redundant cache resource and (b) Traversal efficiency comparisons under limited cache resource

As shown in the above two figures, when cache resource is redundant, the efficiency improvement is rather small, probably around 1%; however, when cache resource is in short supply, the efficiency improvement is significant. The reason is simple: High data locality (low NPC) decreases the required cache resource and thus relieves the impact of cache resource on efficiency.

CONCLUSION

In this study, there presents a novel structure definition for list, the principle of which is to extract the frequent-accessed elements from each node and store them in contiguous memory space. Besides, there adjusts the layout of contiguous memory space to match the data access type of traversal. With the help of the modified data distribution, a cache miss will load not only the data required for traversing to current node but also the data required for traversing to next nodes and thus reduce the number of cache misses as well as the required cache resource. The experimental results shows that, compared with the standard list design, the traversal efficiency is improved significantly. Besides, the design of ArrayList can also be applied to other linked data structures such as tree and graph.

REFERENCES

Gorman, M., 2004. Understanding the Linux Virtual Memory Manager. Prentice Hall, Upper Saddle River, New Jersey.

Jeon, J., K. Shin and H. Han, 2007. Layout transformations for heap objects using static access patterns. Proceedings of the 16th International Conference on Theory and Practice of Software, Braga, Portugal, March 26-30, 2007, Springer-Verlag, Berlin, Heidelberg, pp: 187-201.

Lattner, C. and V. Adve, 2005. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Vol. 40, June 6, 2005, ACM, New York, USA., pp: 129-142.

Lin, C.H., Y. Xie and W. Wolf, 2007. Code compression for VLIW embedded systems using a self-generating table. IEEE Trans. Very Large Scale Integration Syst., 15: 1160-1171.

Mannarswamy, S.S., R. Govindarajan and R. Surendran, 2009. Region based structure layout optimization by selective data copying. Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, Raleigh, NC., September 12-16, 2009, IEEE Computer Society Washington, DC, USA., pp: 338-347.

Qin, X. and P. Mishra, 2009. A universal placement technique of compressed instructions for efficient parallel decompression. Comput.-Aided Des. Integr. Circ. Syst., 28: 1224-1236.

Shin, K., J. Kim, S. Kim and H. Han, 2006. Restructuring field layouts for embedded memory systems. Proceedings of the International Conference on Design, Automation and Test in Europe. Vol. 1. March 6-10, 2006, Munich, pp: 937-942.

Thangaraj, S., S. Gummadi and S. Radhakrishnan, 2006. Enhancement in ARM code optimization for memory constrained embedded systems. Proceedings of the International Conference on Advanced Computing and Communications, December 20-23, 2006, Surathkal, pp: 483-486.

Yang, Y., Z. Shao, L. Pan and M. Guo, 2008. ISOS: Space overlapping based on iteration access patterns for dynamic scratch-pad memory management in embedded systems. Proceedings of the 9th International Conference for Young Computer Scientists, Hunan, November 18-21, 2008, IEEE Computer Society Washington, DC, USA., pp: 1360-1366.