Analysis of Cache Sensitive Representation for Binary Space Partitioning Trees

Vlastimil Havran
Czech Technical University in Prague
Dept. of Computer Science and Engineering
Faculty of Electrical Engineering
Karlov nám. 13, 12135 Prague
Czech Republic
Phone: +420 2 24357432, Fax: +420 2 298098
E-mail: havran@fel.cvut.cz

Keywords: binary tree, binary space partitioning, \(kd\)-tree, cache, spatial locality

Edited by: Maria A. Cobb and Frederick E. Petry

Received: December 12, 1992 Revised: February 9, 1993 Accepted: March 1, 1993

Binary trees are commonly used data structures. Several variants of binary search trees were designed to solve various types of searching problems including geometrical ones as point location and range search queries. In complexity analysis we usually abstract from the real implementation and easily derive that the time complexity of traversal from the root of a balanced tree to any leaf is \(O(\log m)\), where \(m\) is the number of leaves. In this paper we analyse a new method for memory mapping of a binary tree aiming to improve spatial locality of data represented by binary trees and thus performance of traversal algorithms applied on these data structures.

1 Motivation

The basic motivation for this research comes from binary search trees for computer graphics applications, where they are used to accelerate ray-shooting. To solve this problem space subdivision schemes are used, for survey see e.g. [Wat92].

One space subdivision is an orthogonal binary space partitioning tree (BSP tree or BSPT in the following text). It is often referred to as \(kd\)-tree in the context of computational geometry [Berg97]. It was initially developed to solve the hidden surface problem in computer graphics [Fuchs80].

A BSPT is a higher dimensional analogy to a binary search tree. The BSPT for a set \(S\) of objects in \(\mathbb{R}^n\) is a binary tree defined as follows. Each node \(v\) in BSPT represents a non-empty box (rectangular parallelepiped) \(R_v\) and set of objects \(S_v\) that intersects \(R_v\). The box associated with the root node is the smallest box containing all the objects from \(S\). Each interior node of BSPT is assigned cutting plane \(H_v\) that splits \(R_v\) into two boxes. Let \(H_v^+\) be the positive halfspace and \(H_v^-\) the negative halfspace bounded by \(H_v\). The boxes associated with the left and the right child of \(v\) are \(R_v \cap H_v^+\) and \(R_v \cap H_v^-\), respectively. The left subtree of \(v\) is a BSPT for a set of objects \(S_v^- = \{s \cap H_v^- | s \in S_v\}\), the right subtree is defined similarly. The leaves of the BSPT are either occupied by the objects or vacant.

The BSPT is constructed hierarchically step by step until termination criteria given for leaf are reached. There are usually two termination criteria. First, maximum depth of BSPT is specified. Second, a node becomes a leaf if the number of objects associated with the node is smaller than a constant. The cutting plane \(H\) is for ease of computing search queries perpendicular to one of coordinate axes (orthogonal cutting). The example of BSPT in \(\mathbb{R}^2\) space is depicted in Fig. 1.

The most important operation carried out for any BSPT in any application is exhaustive traversal; for example the traversal in depth-first-search (DFS) order. It occurs when BSPT built for \(n\)-dimensional data is used for point location and range search queries [Samet90]. Several vari-
ants of BSPT are thus extensively used in GIS and other spatial database systems.

This decade I/O efficient algorithms and data structures for external memory have acquired noticeable research interest. The design of these data structures is driven by properties of external memory hierarchy. Some achieved results are surveyed for example in [Chiang95].

In this paper we do not deal with external memory data structures, but with the memory hierarchy between the processor and the main memory including either on--chip cache or second--level cache. The main difference between this hierarchy and external memory one is the size of and the access time to one data block. Those for external memory hierarchy are much larger than the ones between processor cache and the main memory.

Second, techniques for external memory data structures were developed mostly for one-dimensional search problems. For example, well known B--tree [Cormen90] cannot be used to decrease time complexity of the BSPT traversal for $n > 1$, since the B--tree cannot represent $n$--dimensional data. In this paper we analyse novel methods to increase spatial locality of data in cache and thus to decrease the time complexity of any algorithm that uses BSPT. For the sake of simplicity we assume traversing BSPT in DFS order from root to a leaf.

2 Preliminaries

In this section we recall the facts necessary to understand the concept of BSPT nodes memory mapping. This includes memory allocation techniques and the structure of the memory hierarchy.

2.1 Memory Allocation

The key idea of this paper is mapping BSPT nodes to addresses in the main memory. Allocation of dynamic variables is always provided by a memory allocator. Let us suppose the contiguous block of the unoccupied memory is assigned to the memory allocator at the beginning. This is used to assign the addresses within the block to the variables allocated so the variables do not overlap. We call this memory block a memory pool. Since the mapping is crucial for the main contribution of this paper, we discuss it in detail.

Common solution is to use a general memory allocator. Each BSPT node is then represented as a specially allocated variable. Let $S_I$ denote the size of memory to store information in a node. This is the position and the orientation of the splitting plane. Let $S_P$ be the size of a pointer. Then the size required to represent one interior BSPT node is $S_{IN} = S_I + 2.S_P$. Use of the general memory allocator requires to store two additional pointers with each allocated variable that are used later to free this variable from memory pool.

In this paper we also use another strategy to allocate the BSPT node. We use a special memory
**allocatro** described in [Stroustrup91] to allocate variables of the same type and thus of the same **fixed size** $S_y$. We use BSPT nodes of fixed size and dedicate them a special memory pool. During building up BSPT the nodes are allocated from the memory pool as from an array in linear order.

### 2.2 Memory Hierarchy

The time complexity of a traversal algorithm using BSPT is connected with the hardware used. Let us recall the organization and the properties of the memory hierarchy. For analysis we suppose Harvard architecture with separated caches for instructions and data. Let $T_{MM}$ denote latency of the main memory (time to read/write one block of data to/from processor).

The larger the memory and the smaller the access time, the higher the cost of the memory. The instruction/data latency of processors is smaller than $T_{MM}$. That is why a cache is placed between the memory and the processor. The cache is a memory of relatively small size with respect to the size of the main memory. The cache latency $T_C$ is smaller than $T_{MM}$. This solution is economically advantageous; it uses **temporal and spatial locality** of data exposed by a typical program and the average access time can be significantly reduced. Data between the cache and the main memory are transferred in blocks. The size of the block transferred is referred to as **cache line size** $S_{CL}$. Typical memory hierarchy is depicted in Fig. 2.

<table>
<thead>
<tr>
<th>Size [words]</th>
<th>Microprocessor chip</th>
<th>Access time [cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 to 256</td>
<td>Registers</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>8 K</td>
<td>Cache 1</td>
<td>1 to 2</td>
</tr>
<tr>
<td>256 K</td>
<td>Cache 2</td>
<td>5 to 15</td>
</tr>
<tr>
<td>32 M to 256 M</td>
<td>Main Memory</td>
<td>40 to 100</td>
</tr>
</tbody>
</table>

![Figure 2: Typical memory hierarchy](image)

In this paper we use for the analysis only the one cache placed between processor and the main memory. We denote the time consumed by operations in terms of cycles. Let $T_W$ denote the average processing time on a BSPT node to decide whether to follow its left or right descendant.

Typical values for today’s superscalar processors and typical application are $T_{MM} = 55, T_C = 4, T_W = 5, S_{CL} = 128$ Bytes for MIPS R8000. These values are taken from [SGI96]. They are used further in the paper. Note that for a typical search algorithm on BSPT holds $T_W \ll T_{MM}$.

### 3 BSPT Representations

As we already stated the BSPT is actually represented by a binary tree. In general, a binary tree does not represent a valid instance of BSPT, since the splitting plane has to intersect the bounding box associated with the node. This is one of the reasons why the decomposition induced by BSPT cannot be simply replaced by B-trees or some hashing scheme commonly used for one-dimensional search problems. The information stored in BSPT node is the orientation and the position of splitting plane. The axis aligned bounding box is known explicitly for a root node only. The axis aligned bounding boxes associated with interior and leaf nodes are not stored explicitly in these nodes, but they can be derived by traversing down the tree.

Let us recall some terminology concerning binary trees. The *depth* of a node $A$ in the tree is the number of nodes on the path from the root to the node $A$. The depth of root node is zero. We call a binary tree *complete* if all its leaves are positioned in the same depth $d$ from the root node and thus the number of leaves is $2^d$. An *incomplete* binary tree is the one that is not complete. Let $h_C$ define complete height of a binary tree $A$ as the maximum depth, for which the binary tree constructed by the nodes of $A$ is complete. The same definitions hold for BSPT.

The next four subsection gives details of BSPT representations in the memory. This includes a usual method to represent BSPT nodes using general memory allocator. We call this *random* representation. The less used method is *DFS order* representation. Finally, we describe two forms of a *subtree* representation that we designed.
to decrease further the average traversal time on BSPT tree.

### 3.1 Random Representation

A common way to store the arbitrary BSPT in the main memory is to represent each node as a special variable using general memory allocator. The representation is depicted in Fig. 3 (a).

This representation requires additional memory for pointers used by general memory allocator for each allocated variable, but it is the simplest technique to implement. The addresses of nodes in the main memory have no connection with their location in the BSPT. Assume that two additional pointers are needed to allocate the variable. Then the memory size $M_S$ consumed by random representation to store $N_{NO}$ nodes of BSPT is:

$$M_S^{\text{random}} = N_{NO}.(4.S_P + S_I) \quad (1)$$

### 3.2 Depth–First–Search (DFS) Representation

A DFS representation is implemented by using the allocator for fixed size variables described in subsection 2.2. In this representation the nodes are put subsequently in the memory pool in linear order, when BSPT is built up in the DFS order, see Fig. 3 (b). The size of the memory consumed to represent $N_{NO}$ nodes of BSPT is:

$$M_S^{\text{DFS}} = N_{NO}.(2.S_P + S_I) \quad (2)$$

Then $2.N_{NO}.S_P$ of memory taken by pointers to implement general memory allocator is saved in comparison with random representation.

### 3.3 Subtree Representation

The main goal of this paper is the analysis the representation of BSPT proposed originally in [Havran97] to reduce the time complexity of ray–shooting query performed on BSPT. Let us describe the representation in detail.

We also use allocator for fixed size variables, but the size of one allocated variable is equal to cache line size $S_{SL}$. The variable is subsequently occupied by the nodes organised into subtree. The whole BSPT is then decomposed to subtrees, see Fig. 3 (c). Once the subtree is read to the cache, the access time to its nodes is equal to cache latency $T_C$. The subtree need not be complete. We distinguish between two subtree representations, see Fig. 4.

An ordinary subtree has all nodes of the same size, with two pointers to its descendants, regardless of whether the descendant lies in the subtree or not.

A compact subtree has no pointers among the nodes inside the subtree because their addressing is provided explicitly by a traversal program. The pointers are needed only to point between the subtrees. The leaves in an incomplete subtree have to be marked in a special variable stored in each subtree (one bit for each node).

The size of the memory described by both subtree representations is given in the next section.

### 4 Time Complexity and Cache Hit Ratio Analysis

In this section we analyse the time complexity of a DFS order traversal for all the BSPT representations described in previous section. The theoretical analysis assumes that the BSPT nodes data stored in the main memory are not loaded into the cache, i.e., cache hit ratio $C_{HR} = 0.0$. Further, we suppose that the BSPT is complete and its height is $h$. An incomplete BSPT requires to compute its average depth $\tilde{h}$ and substitute it for $h$.

These simplifications enable us to express the average traversal time $T_A$ on BSPT in DFS order from its root to a leaf. We compute the $T_A$ for an example of BSPT of height $h = 23$. Further, we suppose random traversal with the probability that we turn left in a node is equal to $p_L = 0.5$.

If some data are already located in the cache ($C_{HR} > 0.0$), the analysis can be very difficult or even infeasible. The interested reader can follow e.g. [Arnold90]. Since the cache has asynchronous behaviour, we analysed the case by means of simulation.

#### 4.1 Random Representation

As we suppose $C_{HR} = 0.0$ during the whole traversal, i.e., the processing time of each BSPT node is $T_M + T_W$. As we know that the number of nodes along the traversal path from root to the
Figure 3: BSPT representations (cache line size $S_{CL} = 3.\text{size(node of BSPT)}$) (a) Random (b) DFS (c) Subtree

$$T_A = (h_l + 1).(T_{MM} + T_W)$$  \hspace{1cm} (3)

For values given above ($T_{MM} = 55$, $T_W = 5$, $h_l = 23$) we obtain $T_A = 1392.0$ cycles.

4.2 DFS Representation

The DFS representation increases the cache hit ratio by involuntary reading the descendant nodes for next traversal step(s) if traversal continues to the left descendant(s) of the current node. Assuming the size of the BSPT node is $S_{IN} = S_I + 2.S_P$, we derive the average traversal time $T_A$ as follows:

$$T_A = (h_l + 1).[p_L.T_{MM}.\frac{S_{IN}}{S_{CL}} + T_W$$

$$+T_C.(1 - \frac{S_{IN}}{S_{CL}}) + (1 - p_L).T_{MM}]$$  \hspace{1cm} (4)

For $S_{IN} = 4 + 2.4 = 12$ and $p_L = 0.5$ we obtain $T_A = 859.1$ cycles.

4.3 Ordinary Subtree Representation

Assume that $S_{CL}$ and $S_{IN}$ are given. Let $S_{ST}$ be the size of the memory needed for each subtree used to represent subtree type identification. We express the size of the memory taken by a complete ordinary subtree of the height $h$:

$$M(h) = (2^{h+1} - 1).S_{IN} + S_{ST}$$ \hspace{1cm} (5)

$$M(h) \leq S_{CL}$$

From Eq. 6 we derive the complete height of the ordinary subtree $h_C$:

$$h_C = [-1 + \log_2(\frac{S_{CL} - S_{ST}}{S_{IN}} + 1)]$$ \hspace{1cm} (6)

The number of nodes in the incomplete ordinary subtree in the depth $d = h_C + 1$ is then:

$$N_{ODK} = \left\lfloor \frac{S_{CL} - (2^{h_C+1} - 1).S_{IN} - S_{ST}}{S_{IN}} \right\rfloor$$ \hspace{1cm} (7)

The average height of the subtree $h_A \geq h_C$ for $N_{ODK} > 0$ is computed as follows:
\[ h_A = -1 + \log_2(2^{h_c+1} + N_{ODK}) \]  

(8)

Finally, the average traversal time of the whole BSPT of height \( h_i \) is:

\[ T_A = (h_i + 1)(T_W + \frac{T_{MM} + T_C.h_A}{h_A + 1}) \]  

(9)

The subtrees are placed in the main memory so they are aligned with the cache lines when read to the cache. Each subtree corresponds to one cache line. The size of the unused memory in the cache line is then:

\[ M_{\text{unused}}^{OSR} = S_{CL} - (2^{h_c+1} - 1 + N_{ODK})S_{IN} - S_{ST} \]  

(10)

For \( S_{IN} = 12, S_{ST} = 4 \), we get \( h_C = 2, N_{ODK} = 3, h_A = 2.46, M_{\text{unused}}^{OSR} = 4 \), and the average traversal time \( T_A = 553.9 \) cycles.

### 4.4 Compact Subtree Representation

Let \( S_I \) be the size of the memory to represent the information in the BSPT node, \( S_P \) the memory taken by one pointer. The size of the memory consumed by a complete subtree of the height \( h \) is expressed as follows:

\[ M(h) = (2^{h+1} - 1).S_I + 2^{h+1}.S_P + S_{ST} \]  

\[ M(h) \leq S_{CL} \]  

(11)

The complete height \( h_C \) of subtree is from Eq. 12 derived similarly to Eq. 6 as follows:

\[ h_C = -1 + \left[ \frac{S_{CL} + S_I - S_{ST}}{S_I + S_P} \right] \]  

(12)

In the same way as for the ordinary subtree representation we derive the number of nodes \( N_{ODK} \) located in the depth \( d = h_C + 1 \) in the subtree:

\[ N_{ODK} = \left[ \frac{S_{CL} - 2^{h_c+1}.(S_I + S_P) + S_I - S_{ST}}{S_I + S_P} \right] \]  

(13)

The unused memory for one subtree in the cache line can be derived similarly as for ordinary subtree:

\[ M_{\text{unused}}^{CSR} = S_{CL} - (2^{h_c+1} - 1 + N_{ODK})S_N - 2.S_P(N_{ODK} + 2^{h_c} - N_{ODK}/2) - S_{ST} \]  

(14)

The average height of the subtree \( h_A \) and the average traversal time \( T_A \) are computed using Eq. 8 and Eq. 9. Given \( S_P = 4, S_I = 4, S_{ST} = 4 \) we compute \( h_C = 3, N_{ODK} = 0, h_A = 3.0, M_{\text{unused}}^{CSR} = 0 \), and \( T_A = 510.0 \) cycles.

The \( h_C, N_{ODK}, h_A \) as the function of the cache line size for ordinary and compact subtree representations and \( T_A \) for all BSPT representations are depicted in Fig 5.

### 5 Simulation Results

We implemented a special program simulating the data transfer in a typical memory hierarchy for the DFS traversal on a complete BSPT. The simulation was carried out for the same memory hierarchy and BSPT properties as in previous section: \( T_{MM} = 53, T_C = 4, T_W = 5, h_I = 23, S_P = 4 \) Bytes, \( S_I = 4 \) Bytes, \( S_{ST} = 4 \) Bytes, four-way set associative cache with cache line size \( S_{CL} = 2^7 = 128 \) Bytes, the size of the cache was \( 2^{20} \) Bytes. The cache placement algorithm and its structure correspond to those found in current superscalar processors, e.g., MIPS R8000 or MIPS R10000 (see [SGI96]).

The theoretical, simulated times, and their ratio are summarised in Table 1. The parameter \( C_{HR} \) is the average cache hit ratio to access a BSPT node in the cache during traversal. The average cache hit ratio for the node as the function of its depth in BSPT is in Table 2.

Note that the compact subtree is for \( S_{CL} = 128 \) complete, so the cache hit ratio for all the nodes at the same depth in the BSPT is equal. This is the reason why \( C_{HR} \) for depth 12, 16, and 20 are quite different from neighbour values, since these BSPT nodes are often read from the main memory. The probability that these nodes are already loaded in the cache is smaller with the increasing depth.

The average traversal times obtained by simulation correlate well with those computed theoretically. It is obvious that the times obtained by
Figure 5: The analysis: (A) Average traversal time $T_A(S_{CL})$ for all BSPT representations, (B) $h_A(S_{CL})$, (C) $N_{DK}(CL)$, (D) $h_C(CL)$ for subtree representations; Representation (a) Random (b) DFS, (c) Ordinary subtree, (d) Compact subtree

the simulation are smaller than these derived theoretically, since the theoretical analysis supposes in each step an initial value of $C_{HR} = 0.0$.

6 Conclusion

In a previous paper [Havran97] we showed experimentally that ordinary subtree representation can decrease traversal time for ray-shooting using BSPT by 40% in a ray tracing application. In this paper we have analysed the time complexity and cache hit ratio of different BSPT representations for DFS order traversal in detail. We have shown the time complexity of traversing of a BSPT is reduced by organising its inner representation which matches better the memory hierarchy. The subtree representation decreases the traversal time for DFS order by 62% and increases hit ratio from 35% to 90% for a given example of common memory hierarchy. Moreover, proposed representation decreases the memory required to store BSPT in the main memory by 57%.

7 Future Work

The presented technique is widely applicable to other hierarchical data structures as well. Future work should include research of variants of multi-dimensional binary trees and hierarchical data structures in general. Dynamization of these data structures with regard to cache sensitive representation is also interesting topic to be researched.
<table>
<thead>
<tr>
<th>Representation</th>
<th>Random</th>
<th>DFS</th>
<th>Ordinary subtree</th>
<th>Compact subtree</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_A ) (theoretical)</td>
<td>1392.0</td>
<td>859.1</td>
<td>555.9</td>
<td>510.0</td>
</tr>
<tr>
<td>( t'_A ) (simulated)</td>
<td>987.1</td>
<td>629.4</td>
<td>445.6</td>
<td>379.3</td>
</tr>
<tr>
<td>( \text{ratio} = t_A / t'_A )</td>
<td>1.41</td>
<td>1.36</td>
<td>1.24</td>
<td>1.34</td>
</tr>
<tr>
<td>( CHR[%] )</td>
<td>35.8</td>
<td>69.8</td>
<td>83.5</td>
<td>90.3</td>
</tr>
</tbody>
</table>

Table 1: The times computed theoretically and obtained by the simulation

<table>
<thead>
<tr>
<th>Depth</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( CHR ) (Random)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>97</td>
<td>91</td>
<td>62</td>
<td>52</td>
<td>39</td>
<td>25</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>( CHR ) (DFS)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>93</td>
<td>79</td>
<td>84</td>
<td>58</td>
<td>56</td>
<td>63</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>( CHR ) (Ordinary subtree)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>97</td>
<td>73</td>
<td>90</td>
<td>85</td>
<td>53</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>( CHR ) (Compact subtree)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>69</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Depth</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>( CHR ) (Random)</td>
<td>21</td>
<td>19</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( CHR ) (DFS)</td>
<td>57</td>
<td>59</td>
<td>47</td>
<td>59</td>
<td>54</td>
<td>48</td>
<td>51</td>
<td>49</td>
<td>47</td>
<td>54</td>
<td>43</td>
<td>54</td>
</tr>
<tr>
<td>( CHR ) (Ordinary subtree)</td>
<td>80</td>
<td>64</td>
<td>66</td>
<td>79</td>
<td>66</td>
<td>70</td>
<td>72</td>
<td>74</td>
<td>61</td>
<td>75</td>
<td>74</td>
<td>62</td>
</tr>
<tr>
<td>( CHR ) (Compact subtree)</td>
<td>7</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The cache hit ratio \( CHR[\%] \) as the function of node depth in BSPT

ACKNOWLEDGMENTS

I would like to thank Jan Hlavicka for delivering the subject of Advanced Computer Architectures and thus compelling me to write a report on this topic. Further, I wish to thank Pavel Tyrdik and all the anonymous reviewers for their remarks on the previous version of this paper.

This research was supported by Grant Agency of the Ministry of Education of the Czech Republic number 1252/1998 and by Internal Grant Agency of Czech Technical University in Prague number 309810103.

References


