Transforming Set Data Types to Power Optimal Data Structures*

Sven Wuytack, Francky Catthoor, Hugo De Man‡

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

In this paper we present a novel approach to model the search space for optimal set data types in network component realisations. The main objective is to arrive at power efficient realisations of these data types in data structures, but the model can also be used with non-power cost functions. This work also strongly contributes to our overall goal of a higher level of specification and shorter design cycles for table-based memory organisations in network components.

1 Introduction

ATM and network components in general are very important application domains. In the layer 3-6 protocols for most network component applications, the efficient organisation of the large table structures is the most crucial issue [10, 2, 6]. Therefore we propose a novel model to characterise the freedom in specifying and realising these structures. This model allows to calculate what the best table organisations are for a given application and cost function. In our experiments, the cost function is a measure for the power dissipation, but it can be changed to reflect the cost of memory size, the number of memory transfers, or, in practice, a combination of power, memory size and number of transfers. This work fits in a context of system level specification and exploration of network component realisation. It is a step towards higher level modelling of data structures.

We have organised this paper as follows. We start with giving some references to related work in Section 2. Then, we present our novel set data structure model in Section 3. Next, we present, in Section 4, the cost function that we use to obtain power efficient data structure implementations. In Section 5, we explain our optimisation method to arrive at a power optimal solution without having to perform an exhaustive search. In Section 6, we illustrate our methods on a relevant industrial design, namely a Segment Protocol Processor (SPP) [10]. The SPP is an ASIC that implements the common part of AAL3 and AAL4 (ATM Adaptation Layer) as specified by the CCITT standards.

2 State of the Art

There is not much related work on optimising data structures for low power. The basic elements and techniques (e.g. linked lists, arrays, hashing, ...) that we have used in our model are well known in programming theory [1]. However, in the programming community, these techniques are applied to reach high performance solutions or solutions with low memory requirements, but, until now, they are not applied to reach low power solutions. Moreover the decisions there are not automated at all. Most low power oriented research, on the other hand, focusses on much lower levels than the algorithm level, and is therefore not attacking this problem either. Several researchers are looking at the algorithm/architecture level to optimise power [4, 7]. Here, they are focussing their attention mainly on data-paths, and not on memories or memory organisations. Their developments on power estimation on the algorithm/architecture level [5] are very useful for our research, though. There are also some memory related power studies [3, 8], but these are oriented to caches in microprocessors and not for custom network components. Our own previous work was situated at the architectural level [11].

3 Set Data Structure Model

A set of records which are accessed with one or more keys can be represented by many different data structures. All these data structures have different characteristics in terms of memory occupation, number of memory accesses to locate a certain record, power dissipation, and the like. To allow the designer to make a motivated choice, all possible data structures available for a given system level specification have to be represented in the model such that the best solutions for a given application can be searched for. In our model there are a number of primitive data structures that can be combined to create more complex data structures.

3.1 Assumptions

The assumptions that we made in the construction of the current model are: *a key is an integer number within a certain interval* and *every number in that interval is a valid key value.* When these conditions are not met, this can be simply and efficiently solved by realising a coding scheme in hardware. This coding scheme will translate the original values of the keys into an interval of integers. These coded keys are then used to access the records; *All key values have equal proba-*

* This research was sponsored by the IWT HASTEC project.

‡ Professor at the Katholieke Universiteit Leuven

bility. When the distributions of the key values are not uniform, we obviously require statistics which can then be used to influence the weighting in the model; *A key/pointer occupies one memory location*. The model can be easily extended to accommodate keys or pointers that require more than one memory location. Extending the model for keys which require less than one memory location and that can be stored together with other information in one memory location, is less trivial and a topic of current research.

3.2 Primitive Data Structures

Currently, our model incorporates four primitive data structures, which are largely based on what is available in computer science theory [1]. These four primitive data structures are the linked list (LL), the binary tree (BT), the array (AR), and the pointer array (PA), which is an array of pointers to dynamically allocated objects. Most of the primitive data structures have additional parameter options which allow to subdivide them further (e.g. binary vs quad-trees, etc.).

3.3 Combining Primitive Data Structures

In the model, these primitive data structures can be combined into more complex and powerful data structures (Fig. 1). If there is more than 1 key or if the simple key is split up in subkeys, then we can construct a hierarchy of data structures. For example in the Segment Protocol Processor application [10] there is a set of packet records which are accessed by two keys, which we will call key A and key B. In the implementation of the SPP, a 3-layer data structure was chosen to represent this set of records. The first layer, is a pointer array which is accessed with the A key. The pointers point to pointer arrays of the second layer, which are accessed with the B key. The pointers on the second layer then point to the dynamically allocated packet-records which form the third layer. Of course this realisation is only one of the many possible combinations. Finding the best one in terms of a certain cost function is not obvious, because of the high number of possibilities and the dependence on the parameters in the model, as shown in Section 6.

3.4 Key Management

In the previous subsection we assumed that every key corresponds to one layer in a hierarchy of data structures (except for the last layer, which is the record layer and has no key associated with it). This is not necessary, however. It is also possible to split keys into subkeys, or to combine several keys into one super key. This can have a large effect on the cost functions. Also, the order in which the keys are used to access the different data structures is very important. For example, in the SPP application there are two possible key orderings: A - B or B - A. In the implementation of the SPP, the A - B ordering was chosen. The other key ordering is equally valid, however. Which key ordering is the best solution depends on the parameters in the application. Since the key ordering has a large effect on the required memory size, on the average number of memory accesses to locate a certain record, and on the power cost of a data structure, it is important to find the optimal key ordering as well as the optimal number of layers in the hierarchy.

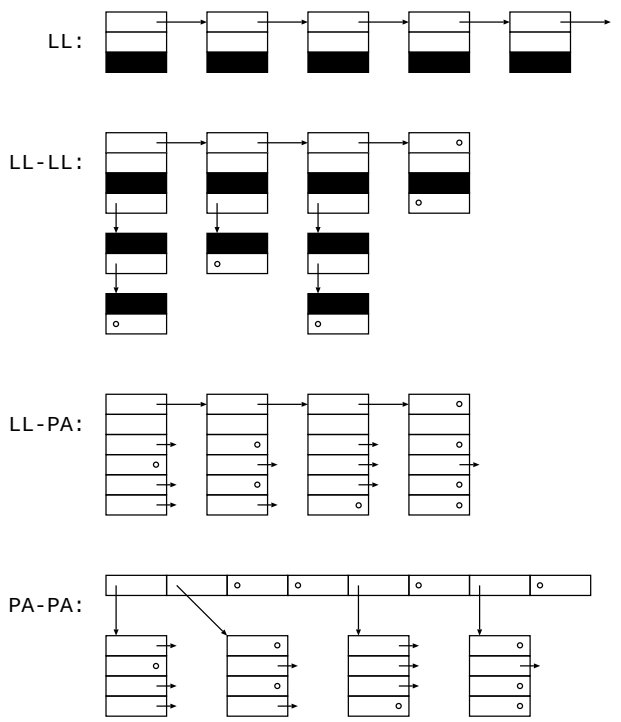


Figure 1: A few combinations of primitive data structures (actual records are not shown).

3.5 Hashing (Key Transformations)

In the model we assume that the key values are uniformly distributed. If this is not the case, then a hashing function can be used to automatically obtain a more uniform distribution. It is important to realise that hashing can, in principle, be applied in combination with any of the primitive data structures. So it provides an “independent” axis of freedom in the search space. Hashing is especially useful in combination with key splitting, because this allows to reduce the (average) size of the data structures. Consider for example a large array of records which contains relatively few elements compared to its size (Fig. 2a). Because there are many empty slots in the array, a lot of memory is wasted in this data structure. This can be solved by applying key splitting. The array of records is converted into a (smaller) array of linked lists of records. Therefore, the large key is split into two parts: one part which is used to access the array and another part which is used to access the linked lists. In the ideal case, the splitting is done such that the small array is fully filled and the linked lists are as short as possible (Fig. 2b). However, if the distribution of the key values is such that most records are contained in only a few linked lists (due to correlation), then the result of the key splitting can be far from optimal (Fig. 2c). Applying a hashing function to obtain a more uniform distribution can avoid this problem (Fig. 2d).

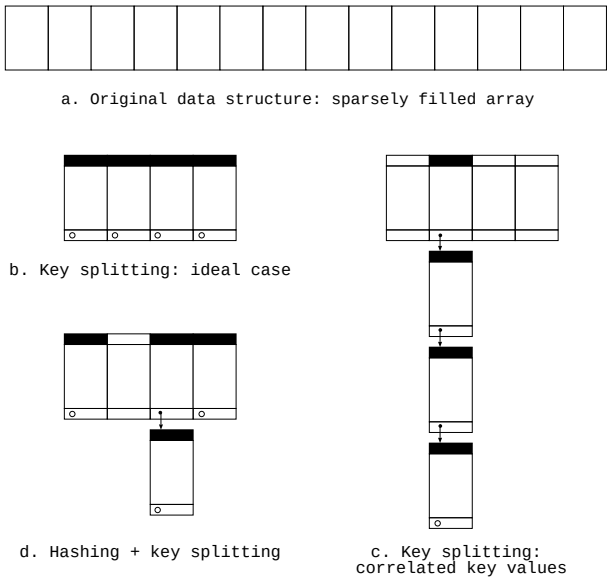


Figure 2: Applying hashing and key splitting to a sparsely filled array.

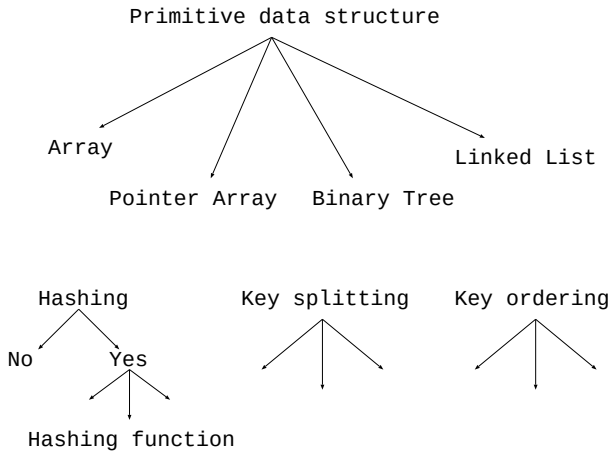


Figure 3: Different orthogonal subtrees in the search space for data structures. These can be chosen independently for each layer corresponding to the original specified keys.

4 Cost Function

For the estimation of the power cost of the different data structures in our model, we use the memory power estimation function of the Stochastic Power Analysis tools of U.C.Berkeley [5]. The power cost function is: $P = C_{read} \cdot V_{dd}^2 \cdot f_{read}$, where C_{read} is the estimated capacitance of the memory for read operations, V_{dd} is the supply voltage of the memory, and f_{read} is the frequency of the read accesses to the memory. Note that, at the moment, we only take the cost of locating a record (read action) into account, not the insertion or removal of records from the set: $C_{read} = n\_RAMs \cdot (C_{r0} + C_{r1}.word\_length + C_{r2}.n\_words + C_{r3} \cdot word\_length \cdot n\_words)$, where n_RAMs is the number of parallel RAMs in the memory architecture. The figures we used in our experiments are $n\_RAMs = 9$, $word\_length = 8$, $C_{r0} = 9707$ fF, $C_{r1} = 108$ fF, $C_{r2} = 1126$ fF, $C_{r3} = 6$ fF, $V_{dd} = 1$ V, and $f_{read} = 0.38 \cdot n\_reads\_per\_frame$ MHz. The capacitance figures are those used in U.C.Berkeley to model SRAMs in their $1.2\mu m$ CMOS technology.

5 Optimisation Methodology

There are many possible data structures within the model that realise a given set of records. As mentioned in Section 3, each of these can be seen as a combination of different choices which are relatively orthogonal (Fig. 3).

Finding the best one for a given application is not so trivial, since it depends on the parameters in the model. Moreover, the full search space is too large to scan it exhaustively. To determine the optimal data structure we have to find the optimal number of layers in the hierarchy, the optimal key ordering, the optimal hashing function for each layer, and the optimal primitive data structure for every layer in the hierarchy. Our experiments showed that all of these optimisations influence each other, so it is not possible to optimise each aspect fully independently to obtain the global optimum. In practice, however, some decisions are much more important than others, and a decision ordering can be proposed which leads to near optimal solutions without exhaustively exploring all combinations.

After explaining the input for the optimisation method in subsection 5.3 and presenting the decision ordering on which it is based in subsection 5.4, we explain each step in the next subsections. This is done using an example, which will be introduced in Section 5.2. But first, we define the *Filling Factor* because it is an important element in our method.

5.1 Definition of the Filling Factor

The filling factor is defined for every layer *i* of the data structure hierarchy. The filling factor of layer *i* is defined as:

$$FF(i) = \frac{\#objects\ on\ layer\ i + 1}{(\#obj\ on\ layer\ i) \times (max\ \#key\ val\ on\ layer\ i)}$$

It is, in fact, the fraction of the slots that would be filled if this layer would be realised with arrays. Obviously, the higher the filling factor of a layer, the more (power) efficient the data in it can be stored (Fig. 4).

This graph shows the power cost function versus the filling factor when the number of objects on layer $i+1$ are kept constant. The power cost function is plotted for the four primitive data types in our model. It is possible to plot a 2D plot of the power cost of an array and a pointer array vs FF(i) when (#obj on layer i) is fixed, because, in this case, the power cost only depends on the product in the denominator of FF(i). Since the power cost functions of a linked list and a binary tree only depend on the number of elements in the data structure, i.e., (#obj. on layer $i+1$)/(#obj on layer i), and not on (max # key val on layer i), they do not depend on FF(i) directly. Therefore, we have drawn these power costs as horizontal lines vs FF(i) for the key ordering that minimizes the

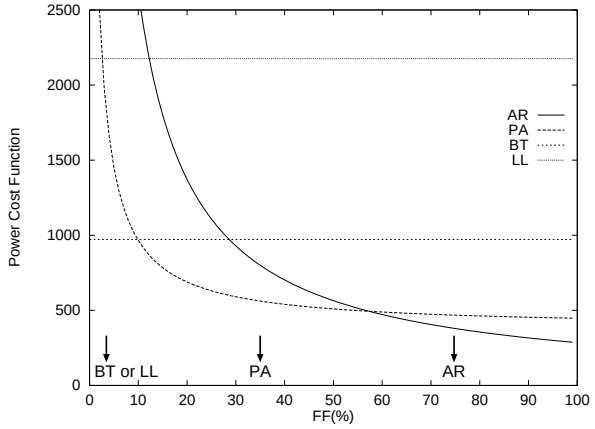


Figure 4: Power cost function versus the filling factor (FF), in which the number of objects on layer $i+1$ is kept constant.

power cost. Clearly, different ranges of FF(i) require different optimal choices (as indicated in Fig. 4).

The actual position of the different curves in this graph depends of course on the parameters in the model.

5.2 Introduction of Illustrative Example

The SPP application has two keys: key A with range $0 \rightarrow 127$ and key B with range $0 \rightarrow 1023$. We will examine the case in which half of the users are active (i.e., 64 of the 128 possible A key values are used) and all B key values are in use. The records are 6 words long, and, the average number of accesses during one frame to the records is 2. We calculate the optimal data structure to store the records in the SPP in terms of our power cost function. We do this for two realistic sets of parameters to show that the optimal solution depends on the parameters in the model. Here, we estimate the best data structure in case the SPP is used with enough external memory to store 8200 records (current SRAM generation). In Section 6, we repeat this calculation in case the SPP is used with enough external memory to store 57300 records (next SRAM generation).

5.3 Input for Optimisation Method

Most of the input to the optimisation method comes from profiling information obtained during system simulation for the network application. Because the result is optimised for the statistics gathered during the simulation, it is important that the simulation is done with realistic input values. See for example Sykas et al. [9] for such studies. If these realistic input values are not known in advance, one can simulate a number of typical scenarios and calculate the optimal data structure and corresponding cost function for each scenario. By comparing the different solutions with their cost functions, and calculating the cost of each solution for every scenario, one can arrive at a good compromise. The information we currently need in the optimisation process for the model of Section 3 - 4 is the following:

- for every original key:
  - key interval $\Rightarrow$ max # different key values
    example: key A: $0 \rightarrow 127 \Rightarrow 128$
    key B: $0 \rightarrow 1023 \Rightarrow 1024$
  - expected # different key values
    example: key A: 64
    key B: 1024
- for every combination of keys:
  - expected # different key combinations
    example: A - B: 8200
- about the records:
  - size (# words) (*example:* 6)
  - avg. # accesses to records (*example:* 2)
- about other data structures in same memory:
  - total size (# words) (*example:* 0)
  - avg. # accesses to other DS (*example:* 0)

5.4 Decision Ordering

Both key transformation and key splitting/merging depend on the statistics of the original keys, and determine the statistics of the new keys. The other optimisations also depend on the key statistics, but they don't alter them. Therefore, key transformation and key splitting/merging has to be performed before optimisation of the key ordering and primitive data structure assignment to optimise the filling (see subsection 3.5). It is very important to have optimal filling factors for the different layers because this determines how efficient the data can be stored. The assignment of the primitive data structures to the layers is highly related to these filling factors, but does not influence the filling factors of the layers. The key ordering, on the other hand, has a large impact on the filling factors of the different layers. Therefore, key ordering has to be decided before the assignment of the primitive data structures to the layers. The information of the filling factors can then be used to assign primitive data structures to the layers.

5.5 Optimising the Key Ordering

Because lower layers in the hierarchy contain in general more data than the higher ones, it is important to store the lowest layers as efficient as possible, i.e., with high filling factors. The key ordering has a large impact on the filling factor of each layer. Therefore we have to select the key ordering which gives high filling factors for the lowest layers. This can be done by calculating the filling factors of every layer for all key orderings and then selecting the one with the highest filling factors for the lowest layers. However, there is a better way of calculating the optimal key ordering. It is based on a property of the filling factor. As mentioned above, the filling factor of a layer depends on the key ordering. However, it only depends on *which* keys are used on layers above it, on *which* key is used on the layer itself, and on *which* keys are used on layers below it. So, it

does not depend on the actual *ordering* of the keys before or after it. Therefore, because we want to calculate the key ordering which gives the highest filling factor on the lowest layer, we can calculate the filling factor for the lowest layer for every key in the last position, and then select the key that leads to the highest filling factor. Once we know the last key in the key ordering, we can calculate the one but last key in the same way. Then, we can repeat this to calculate the key before that, and so on. So, the optimal key ordering is efficiently calculated backwards.

SPP example:

- Last key:

$$key = B \quad \Rightarrow \quad FF = \frac{8200}{64 \times 1024} = 13\%$$
$$key = A \quad \Rightarrow \quad FF = \frac{8200}{1024 \times 128} = 6\%$$
$$13\% > 6\% \Rightarrow last\ key\ =\ key\ B$$

- First key: *only 1 key left* $\Rightarrow$ *first key = key A*
- Optimal key ordering: *key A - key B*

5.6 Optimising the PDS Assignment

In this stage of the optimisation process, we have fixed the subkeys, their statistics, and their ordering. The only thing left is the assignment of primitive data structures to every layer in the data structure hierarchy. As can be seen in Fig. 4, there is a correlation between the *FF* of a certain layer and the optimal primitive data structure for that layer. This could be used to assign a primitive data structure to each layer. However, the actual boundaries between the optimal data structures vary with the parameters in the model and with the assignment of the primitive data structures to the other layers. Therefore, it is not always easy to decide based on the value of the filling factor alone. In this case one can calculate the cost function for the most promising primitive data structures in the model and select the one which yields the lowest cost.

SPP example:

$$\begin{aligned} key = B \quad &\Rightarrow \quad FF = \frac{8200}{64 \times 1024} = 13\% \\ &\Rightarrow \quad PA(B) \\ key = A \quad &\Rightarrow \quad FF = \frac{64}{1 \times 128} = 50\% \\ &\Rightarrow \quad PA(A) \end{aligned}$$

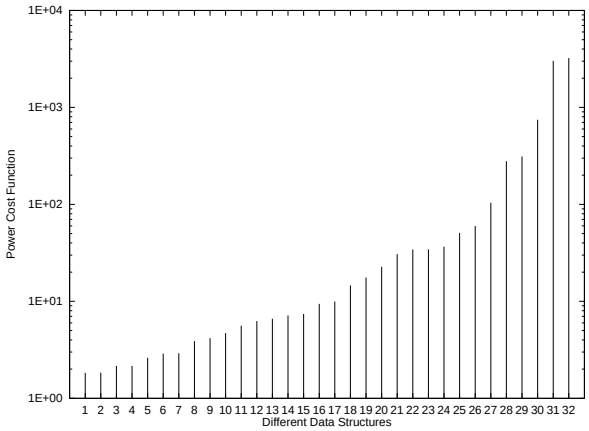
The estimated optimal data structure is thus $PA(A) - PA(B)$ The analysis shows that the difference between an A - B and a B - A ordering is rather small (the difference between 13 % and 6 % is small). Therefore, the best result of the B - A ordering should be almost as good as the best A - B ordering. This will be confirmed in Fig. 5.

6 Examples

First example: ***large external memory***

This is the example which we have analysed in the previous subsections. In this example we assumed that there are 8200 records in the external memory of the SPP. A graph of the cost of a number of alternative data structures (without key transformation and splitting/merging) in the model is shown in Fig. 5. In practice there are even more options and the estimate for each candidate can be quite costly to compute. The alternatives are sorted on the value of the power cost function. The best realisation is a pointer array accessed by key A, which points to a pointer array accessed by key B. However, this solution is only slightly better than an array of binary trees, in which the array is accessed with key B, and the binary trees with key A. This is exactly what we found without exhaustive enumeration in our analysis of the example in Section 5.

Note also the huge difference in the values of the cost function which could be obtained with bad choices for the data structures.



1	PA(A) - PA(B)	12	AR(A, B)	23	PA(A) - LL(B)
2	AR(B) - BT(A)	13	BT(A) - BT(B)	24	BT(A) - LL(B)
3	PA(A, B)	14	BT(B) - BT(A)	25	LL(A) - LL(B)
4	PA(B) - BT(A)	15	BT(B) - LL(A)	26	BT(B) - AR(A)
5	AR(B) - LL(A)	16	PA(B) - AR(A)	27	LL(A) - AR(B)
6	PA(B) - PA(A)	17	BT(A, B)	28	LL(B) - LL(A)
7	PA(B) - LL(A)	18	BT(B) - PA(A)	29	LL(B) - BT(A)
8	AR(A) - BT(B)	19	BT(A) - AR(B)	30	LL(B) - PA(A)
9	PA(A) - BT(B)	20	LL(A) - BT(B)	31	LL(A, B)
10	PA(A) - AR(B)	21	LL(A) - PA(B)	32	LL(B) - AR(A)
11	BT(A) - PA(B)	22	AR(A) - LL(B)		

Figure 5: Example 1: Power cost function for different data structure implementations.

Now, we will look at the same application in case a larger memory is used.

Second example: ***very large external memory***

In this example, we assume that there are 57300 records in the external memory. A graph of the power cost of the same data structures as in the first example is shown in Fig. 6. The graph is again sorted by the value of the power cost function

for every solution. The best realisation this time is a pointer array accessed by key A, which points to an array of records which is accessed by key B. This solution is about 30 % better than the pointer array to pointer array solution obtained with the previous parameters. Remark also the differences between the ranking of the realisations for both sets of parameters! This clearly shows that it is not trivial to predict which is the best realisation for a given application. Let us now calculate the optimal data structure with the proposed optimisation method:

1. Second layer:

$$\begin{aligned} key = B &\Rightarrow FF = \frac{57300}{64 \times 1024} = 87\% \\ &\Rightarrow AR(B) \\ key = A &\Rightarrow FF = \frac{57300}{1024 \times 128} = 44\% \\ &\Rightarrow PA(A) \end{aligned}$$

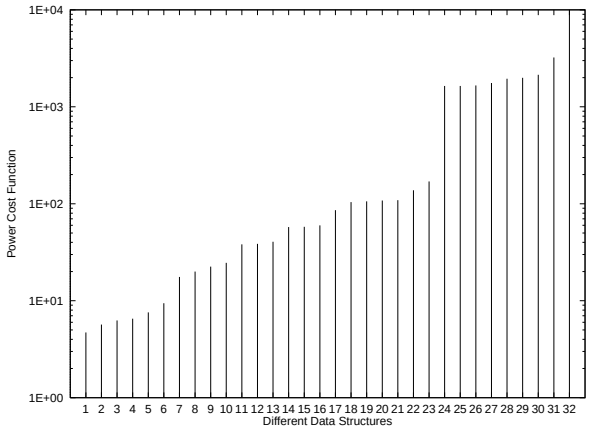
$$87\% > 44\% \Rightarrow second\ layer = AR(B)$$

2. First layer:

$$\begin{aligned} key = A &\Rightarrow FF = \frac{64}{1 \times 128} = 50\% \\ &\Rightarrow PA(A) \end{aligned}$$

3. Conclusion:

$$optimal\ DS = PA(A)\ -\ AR(B)$$



1	PA(A) - AR(B)	12	AR(A) - BT(B)	23	LL(A) - BT(B)
2	PA(A, B)	13	PA(A) - BT(B)	24	AR(A) - LL(B)
3	AR(A, B)	14	BT(A) - BT(B)	25	PA(A) - LL(B)
4	PA(A) - PA(B)	15	BT(B) - BT(A)	26	BT(A) - LL(B)
5	PA(B) - PA(A)	16	BT(B) - AR(A)	27	LL(A) - LL(B)
6	PA(B) - AR(A)	17	BT(A, B)	28	LL(B) - PA(A)
7	BT(A) - AR(B)	18	LL(A) - AR(B)	29	LL(B) - LL(A)
8	BT(A) - PA(B)	19	AR(B) - LL(A)	30	LL(B) - BT(B)
9	AR(B) - BT(A)	20	PA(B) - LL(A)	31	LL(B) - AR(A)
10	PA(B) - BT(A)	21	LL(A) - PA(B)	32	LL(A, B)
11	BT(B) - PA(A)	22	BT(B) - LL(A)		

Figure 6: Example 2: Power cost function for different data structure implementations.

The estimated optimal data structure is again the same as the one obtained by exhaustively calculating the cost function of every possible data structure.

7 Conclusions

We have proposed a novel set data structure model which structures the choices in the search space and that can be used to effectively calculate the best data structure implementation for a set of records in a given application. We have also presented an efficient optimisation method for finding the implementation with minimum power consumption without an exhaustive scan of the search space.

Acknowledgements

We gratefully thank our colleagues from IMEC and Alcatel for the interesting discussions.

References

[1] A.V.Aho, J.E.Hopcroft, J.D.Ullman, "Data Structures and Algorithms", Addison-Wesley, 1983.

[2] A.Alles, "ATM in Private Networking: Tutorial", *INTEROP '93*, 1993.

[3] J.Bunda, W.Athas, D.Fussell, "Evaluating Power Implications of CMOS Microprocessor Design Decisions", *1994 International Workshop on Low Power Design*, Napa Valley CA, pp. 147-152, Apr. 1994.

[4] A.Chandrakasan, M.Potkonjak, J.Rabaey, R.Mehru, R.W.Brodersen, "Optimizing power using transformations", accepted for *Transactions on CAD*, 1994.

[5] P.Landman, J.Rabaey, "Black-Box Capacitance Models for Architectural Power Analysis", *1994 Int. Wkshp on Low Power Design*, Napa Valley CA, pp. 165-170, Apr. 1994.

[6] J.-Y.Le Boudec, "The Asynchronous Transfer Mode: a tutorial", *Computer Networks and ISDN Systems 24*, pp. 279-309, 1992.

[7] T.Noll, "Low-Power Strategies for High-Performance CMOS Circuits", *Proc. ESSCIRC '94*, pp. 72-83, 1994.

[8] C.Su, C.Tsui, A.Despain, "Low Power Architecture Design and Compilation Techniques for High Performance Processors", *Proceedings of IEEE COMPCON*, Feb. 1994.

[9] E.Sykas, K.Vlakos, K.Tsoukatos, E.Protonotarius, "Performance Evaluation of Analytical Models for Effective Bandwidth Allocation in ATM Networks", *Eur.Trans. on Telecommunications*, Vol. 5, no 3, pp. 391-396, May 1994.

[10] Y.Therasse, G.H.Petit, M.Delvaux, "VLSI architecture of a SMDS/ATM router", *Annales des Télécommunications*, 48, no 3-4, pp.166-180, 1993.

[11] S.Wuytack, F.Catthoor, F.Franssen, L.Nachtergaele, H.De Man, "Global communication and memory optimizing transformations for low power systems", *1994 Int. Wkshp on Low Power Design*, Napa Valley CA, pp.203-208, Apr. 1994.