Exploration and Synthesis of Dynamic Data Sets in Telecom Network Applications

Ch. Ykman-Couvreur, J. Lambrecht, D. Verkest
IMEC, Kapeldreef 75, Leuven, Belgium
F. Catthoor, H. De Man
IMEC, Kapeldreef 75, Leuven, Belgium. Also prof. at Katholieke Univ. Leuven

Abstract

We present a new exploration and optimization method to select customized implementations for dynamic data sets, as encountered in telecom network, database and multimedia applications. Our method fits in the context of embedded system synthesis for such applications, and enables to further raise the abstraction level of the initial specification, where dynamic data sets can be specified without low-level details. Our method is suited for hardware and software implementations. In this paper, it mainly aims at minimizing the memory power consumption, although it can also be driven by other cost functions such as area or performance. Compared with existing methods, it can save up to 2/3 of the memory power consumption and 3/4 of the memory area.

1 Introduction

To cope with the increasing complexity, the drastic increase in communication speed, and the shortened time-to-market of modern telecom network applications, new system synthesis approaches are needed. The challenge is now to design systems efficiently, fast, and first-time right. To this end, the abstraction level of the initial system specification must be raised, so that the designer is not burdened unnecessarily by low-level details of the final design. Also more efficient system designs must be achieved. This implies that efficient exploration and specification refinement must be provided at the system level where the impact on area, performance, and power is the most important.

For telecom network applications, as encountered in middle layer protocol processing, the behavior is often characterized by algorithms that operate on large and irregular data structures, dynamically allocated and stored in sets, as buffers or association tables. These sets are called *dynamic data sets* in the sequel.

In embedded implementations of such telecom network applications, a dominant bottleneck is the implementation of the used dynamic data sets. Indeed, to store these sets, large storage capacities are required and a large part of the design area is due to memory units [2, 19]. Combined with these sets several basic services such as set management (to insert, locate, or remove data from a set) and memory management (to dynamically (de)allocate memory) play a very important role too. These services may consume up to 80% of the processing time of the design [4, 22]. Finally, due to intensive data storage and transfer required by these services, the power consumption of the design is dominated by the huge amount of memory accesses, as demonstrated by recent work at IMEC [3], at Princeton University [20], at Stanford University [14], and in the IRAM project [15]. However this bottleneck is not sufficiently addressed in a systematic way in current system design practice.

Association tables of records indexed by keys are typical dynamic data sets encountered in telecom network applications. They can be implemented in many different ways. Primitive data structures (i.e. array, pointer array, linked list, and binary tree) can be used. These can also be combined into more complex layered implementations. More details can be found in [24]. In terms of area, performance, and power, and dependent on the application characteristics, a huge difference in cost between all these implementations has been experimented. Therefore, to find the best implementation for an association table in terms of some cost function, the designer has to explore the complete search space. This is not possible without system-level estimations based on the application characteristics, an efficient exploration and optimization method, and tool support.

To overcome this bottleneck, we propose a new exploration and optimization method at the system level to select customized implementations for dynamic data sets, especially oriented to association tables of records indexed by keys. This method is suited for both hardware and software implementations. It extends the preliminary approach of [24]. It fits in the context of our Matisse system synthesis approach [5], where it is shown that incorporation of dynamic data set synthesis allows to achieve more efficient system designs. Three telecom network applications are used to illustrate the efficiency of our method. Two of them are components in ATM switches: the multiplexer (MUX) core [11], and one operation and maintenance component [9], called F4. The third one is an important component in ATM backbone networks: the Segment Protocol Processor (SPP) [19]. However, we believe that our method is general enough to be applied not only to telecom network applications, but also in many other domains, such as database and multimedia applications, where dynamic data sets are used to specify the data storage at a high abstraction

level.

The paper is organized as follows. Section 2 summarizes the related work. Section 3 overviews the cost function used to drive our exploration and optimization. Section 4 characterizes the association table implementations considered in our search space. Section 5 presents our method, and illustrates it on MUX core, whereas Section 6 discusses the results. Section 7 applies our method to both other applications mentioned above, F4 and SPP. Finally conclusions are drawn in Section 8.

2 Related work

2.1 Dynamic data set synthesis

In programming theory [1], the primitive data structures (i.e. array, pointer array, linked list, and binary tree) and hashing considered in our search space are well-known. They are used in order to reach implementations either with high performance, or with low memory, but not with low power requirements. Moreover neither exploration nor optimization is automated.

For telecom network applications, the overall search space and a preliminary exploration and optimization method are presented in [24]. This selects layered implementations for association tables of records indexed by keys, obtained by combining the primitive data structures previously mentioned, optimized for power, and suited for hardware and software implementations. However this still has several major limitations:

• The search space considered in the method is too restrictive, and in several telecom network applications, the selected implementation is far from being optimal. Although hashing and key splitting/merging are introduced to characterize the search space, they are not supported by the method, and the maximum number of layers in the derived implementations is limited by the number of keys in the initial application specification.

• The cost of any implementation in the search space can be incorrectly estimated, and erroneous decisions can be taken through the exploration. This is due to the following reasons:

- Dependencies between keys in the initial application specification are not taken into account.

- All key values are assumed to be uniformly distributed. If not, keys must be hashed before applying the method.

- Any key, any pointer is assumed to occupy one memory word. This assumption has become unnecessary. Indeed, our synthesis of dynamic data sets fits in the context of Matisse, and relies on the subsequent optimization method for data splitting/merging into memory words [6], before generating an optimized distributed memory architecture wherein dynamic data sets are stored.

- The cost function relies too much on the storage efficiency, compared to the memory accesses, and does not exactly model the relative power consumption.

2.2 System design practice

For control dominated applications, support for dynamic memory management is lacking. For data flow applications, current approaches namely concentrate on synthesis of data flow arithmetic. The array signal streams present in DSP applications can be largely analyzed at compile time, and the proposed memory management techniques are not directly suited.

For dynamic data dominated applications, several system synthesis approaches are available. Among [7, 8, 10, 16, 17, 18, 21], several aspects are considered, but no support for system-level synthesis of dynamic data set is provided. Only Matisse [5] supports memory management for dynamic data sets, wherein [24] is fitting.

In this paper, a new exploration and optimization method is proposed, which extends the previous method [24] as follows. It supports hashing and key splitting/merging, takes key dependencies into account, and removes all previous assumptions on the keys. It is driven by a cost function that estimates at the system level the relative average power consumption of an implementation alternative. This cost function takes the application characteristics into account, and efficiently trades off memory size and memory access in order to select an implementation optimized for power.

3 Cost function

3.1 Cost model

In our embedded application domain, performance is a hard constraint to be met, whereas area and power are crucial cost factors and must be optimized. For many of our applications, the major area and power are not involved in the data paths and the controllers, but in the global communication and the memory organization [3, 23]. So it is important to focus on memory area and power in system synthesis of such applications.

For on-chip memories, the power consumption of one memory access increases with the memory size, i.e. bitwidth and number of words. The dependency is between linear and logarithmic depending on the library used. Several power models exist like capacitance models [12] and empiric models provided by memory manufacturers. The power of the internal interconnect and of the address calculation is still small (less than 20%) compared to that of the internal memories and it can be neglected in the system-level power estimations. For off-chip memories, the power consumption of one memory access can be considered more or less independent from the memory size, and a significant portion goes into the off-chip communication. Hence power can be saved either by reducing the number of memory accesses, or by storing data into smaller on-chip memories [25].

3.2 Table implementation cost

The cost of any explored table implementation should be a weighted sum of area and power of the corresponding memory architecture. In the paper we focus on power optimization.

For any table implementation, the average power consumption depends on the record size, the average mem-

ory size of the table, the operations performed on the table, and the average memory access number in accessing any table data. The power model we use is very simple, but good enough for system-level estimations, as illustrated in Section 6. Our method is driven by a cost function, that estimates at the system level the relative average power consumption of a table implementation using the formula: *average memory size of the table* $*$ *(1+ average number of memory accesses to locate any record in the table).* Since in contrast to [24] a key or pointer is not assumed to occupy one memory word any more (see Section 2), the used memory size unit is either the bit or the byte, not the memory word, which allows finer cost estimations.

3.3 Application characteristics

The main application characteristics relevant for our method are available either at compile time or derived from profiling information at the system level. For any used association table of records indexed by keys, these characteristics are: (1) The record size, the average number of records stored in the table, and the average number of accesses to any record in the table during its lifetime. (2) The number of keys in the initial application specification, their size, their value distribution, the average number of their active values in the table, and the key dependencies. These application characteristics are used to prune the exploration and to compute the cost of any table implementation considered in our search space, as illustrated in Sections 5 and 7.

MUX core experiments In this representative application, one table is used, whose records (32 bit size) are indexed by three keys independent from each other: VPI (8 bit size), VCI (16 bit size), and port (8 bit size). In the sequel, experiments are reported related to two different telecom networks where such MUX cores are implemented: in network 1, the table contains 2^{10} records, whereas in network 2, the table contains 2^{13} records on average. Only the 10 least significant bits are used in VCI, and only one port is needed. Both VPI and VCI (restricted to 10 bits) values are uniformly distributed.

4 Table implementations

This section characterizes the layered implementations considered in our search space, with their respective cost function. To this end, the following notations are used: $size_{rec}$ = the record size, avg_{rec} = the average number of records stored in the table, $size_k$ = the size of key k, max_k = the maximum number of possible k values in the table, avg_k = the average number of active k values in the table, $avg_{k_1|k_2}$ = the average number of active k_1 values per k_2 value in the table, $size_{ptr}$ = the pointer size.

4.1 One-layer implementations

Currently in our search space, the one-layer implementations cover the following primitive data structures: unordered linked lists, pointer arrays, or arrays (see Figure 1). Ordered linked lists and binary trees give rise to significant performance overhead during record insertion/removal in the table. They are not well-suited

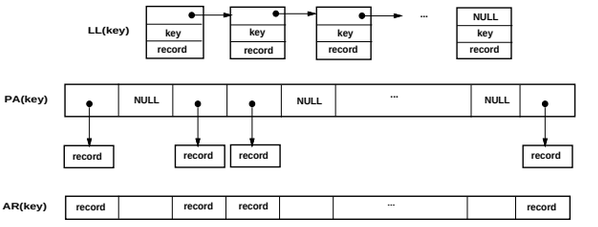

Figure 1: **One-layer implementations**

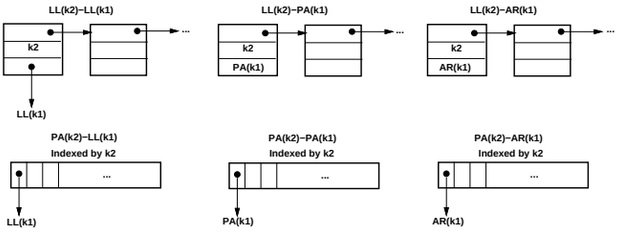

Figure 2: **Two-layer implementations**

for applications characterized by stringent real-time requirements and frequent data insertions/removals.

In the unordered linked list, denoted by LL(key), elements are dynamically (de)allocated. Each of them contains the key, the record itself, and a pointer to the next element of the linked list. Within a long linked list, a large number of memory accesses can be required to locate a record. The average memory size is $(size_{ptr} + size_k + size_{rec}) * avg_{rec}$, and the average number of memory accesses to locate a record is $avg_{rec} - 1$[1], so that the cost function is: $cost(LL(k)) = (size_{ptr} + size_k + size_{rec}) * avg_{rec}^2$.

The pointer array, denoted by PA(key), is an array of pointers to records. The pointer array stores a pointer for each active key value. Key values don't need to be stored since the key value corresponds with the position of the pointer in the array. The average memory size is $size_{rec} * avg_{rec} + 2^{size_k} * size_{ptr}$, and only one memory access is required to locate a record, so that the cost function is: $cost(PA(k)) = (size_{rec} * avg_{rec} + 2^{size_k} * size_{ptr}) * 2$.

The array, denoted by AR(key), reserves memory for each record it can contain. Hence many memory locations are wasted if the average number of records stored in the array is relatively small. The memory size is $2^{size_k} * size_{rec}$, and no memory access is required to locate a record, so that the cost function is: $cost(AR(k)) = 2^{size_k} * size_{rec}$.

4.2 Two-layer implementations

They are obtained by combining primitive data structures, as illustrated in Figure 2. *AR(k_2)-LL(k_1)*, *AR(k_2)-PA(k_1)*, and *AR(k_2)-AR(k_1)* are not considered: they are indeed equivalent or even worse than those shown in Figures 1 and 2. For each two-layer implementation, the cost function can be easily derived. As

[1] $avg_{rec}/2$ key accesses + $avg_{rec}/2 - 1$ pointer accesses.

illustration, $cost(LL(k_2)\text{-}LL(k_1)) = [avg_{k_2} * (2 * size_{ptr} + size_{k_2}) + avg_{rec} * (size_{ptr} + size_{k_1} + size_{rec})] * (avg_{k_2} + avg_{k_1|k_2})$.

4.3 r-layer implementations

They are also obtained by combining primitive data structures, similarly to the two-layer implementations. A general r-layer implementation is denoted by $L_r(k_r)$-$L_{r-1}(k_{r-1})$-...-$L_1(k_1)$, where $L_1(k_1)$ is the lowest layer, $L_r(k_r)$ is the highest one, $L_1(k_1)$ is either $LL(k_1)$, or $PA(k_1)$, or $AR(k_1)$, and $L_j(k_j), 2 \leq j \leq r$, is either $LL(k_j)$ or $PA(k_j)$ ($AR(k_j)$ is not considered as for the two-layer implementations). Its cost function can also be derived in the same way.

5 Exploration and optimization method

For any association table of records indexed by n keys k_1, k_2 ..., k_n in the initial application specification, the problem is to find the best r-layer implementation in terms of the cost function characterized in Section 3. Exhaustive exploration of the complete search space of multi-layer implementations is not possible. An efficient exploration and optimization method based on heuristics and tool support is needed to derive at least near optimal solutions.

Our method is described below. The first three steps initialize the search space exploration. The next steps successively explore one-layer implementations, two-layer ones, three-layer ones, ..., until either the optimal or a near optimal implementation is reached. Let $Space_r$ denote the search sub-space of r-layer implementations explored in our method. Each $Space_r$ is exhaustively explored, but for the reasons given below, $Space_r$ becomes more and more restricted, while r increases:

- As illustrated in our applications, the probability is very high that the optimal implementation is either a one- or a two- or a three-layer implementation. Hence for $1 \leq r \leq 3$, $Space_r$ consists of all possible r-layer implementations.
- From $r \geq 4$, even when the r-layer implementation cost can still be minimized, the decrease is not significant any more. Indeed the memory access number is still increasing, and is hardly compensated by a sufficient decrease of the memory size. Hence exhaustive exploration becomes useless.
- Moreover the number of all possible r-layer implementations is increasing exponentially with r. Depending on $size_{k_i}, 1 \leq i \leq n$, $Space_r$ must be accordingly restricted to keep our exploration tractable.

Our method is characterized by a minimization problem that can be potentially solved using tools as Matlab, Simulated annealing, Tempered annealing, Hill climbing, ... In this paper, our minimization problem is solved using a symbolic formulation in Matlab [13], because it is fast and gives very good results (see Section 6).

Step 1 Taking into account that higher keys are accessed before lower ones while locating records in the table, order $k_i, 1 \leq i \leq n$, into o_n, ..., o_2, o_1 as follows:

1. o_i depends on $o_j \Rightarrow i < j$. This key ordering[2] yields less memory accesses to locate records in the table.
2. $avg_{o_i}/max_{o_i} > avg_{o_j}/max_{o_j} \Rightarrow i < j$. This key ordering allows to use the memory allocated for the implementation of the lower layers as efficiently as possible. Since lower layers generally contain more data than higher layers, this yields lower memory size of the table.

MUX core experiments The key ordering is irrelevant since VPI, VCI, and port are independent from each other, and their values are uniformly distributed.

Step 2 Hash each key $o_i, 1 \leq i \leq n$, whose $max_{o_i} < 2^{size_{o_i}}$, or whose values are not uniformly distributed in the interval $[0, 2^{size_{o_i}}[$[3]. Let $h_i, 1 \leq i \leq n$, denote the hashed keys such that: $2^{size_{h_i}} = max_{o_i}$, and any h_i value is in the interval $[0, 2^{size_{h_i}}[$.

MUX core experiments In both networks 1 and 2, only 10 bits are used in VCI, and only one port is needed. Hence VCI and port are hashed as follows: $0 \leq$ hash(VCI) $< 2^{10}$, and hash(port) = constant. From now on, for simplicity in the notations, VCI refers to this hashed VCI, rather than to the one used in the initial MUX core specification.

Step 3 Concatenate all keys h_n, ..., h_2, h_1 to form one super-key K whose $size_K = size_{h_n} + ... + size_{h_2} + size_{h_1}$, and $avg_K = avg_{rec}$. At the same time, for each $h_i, 1 \leq i \leq n$, derive $freq_{h_i}$ such that

$avg_{h_i|h_n,...h_{i+1}} = 2^{size_{h_i}/freq_{h_i}}, 1 \leq i < n$,

$avg_{h_n} = 2^{size_{h_n}/freq_{h_n}}$,

taking into account that

$avg_{rec} = avg_{h_n} * ... * avg_{h_2|h_n,...h_3} * avg_{h_1|h_n,...h_2}$,

which can be derived from the application characteristics. These $freq_{h_i}, 1 \leq i \leq n$, are needed to compute the cost of any r-layer implementation.

MUX core experiments $size_K = 18\ bits$. Since VPI and VCI values are independent from each other and uniformly distributed, $freq_{VPI} = freq_{VCI} = p$, and for any splitting of K into $k_r, ..., k_2, k_1$, $freq_{k_i}, 1 \leq i \leq r, = p$ too. In network 1,

$avg_{rec} = 2^{10} = 2^{8/p} * 2^{10/p} \Rightarrow p = 18/10 = 1.8$,

whereas in network 2,

$avg_{rec} = 2^{13} = 2^{8/p} * 2^{10/p} \Rightarrow p = 18/13 = 1.4$.

Step 4 Step 4 successively explores $Space_1$, $Space_2$, and $Space_3$, which consist respectively of all one-, two-, and three-layer implementations.

First explore the one-layer implementations among $LL(K)$, $PA(K)$, and $AR(K)$, as defined in Section 4.1, and select the best one. Then explore the two-layer implementations $L(k_2)$-$L(k_1)$, as defined in Section 4.2, and select the best one, which is the solution of the following minimization problem: $MIN_{size_{k_2}+size_{k_1}=size_K}$ *(cost(LL(k2)-LL(k1)), cost(LL(k2)-PA(k1)), cost(LL(k2)-AR(k1)), cost(PA(k2)-LL(k1)), cost(PA(k2)-PA(k1)), cost(PA(k2)-AR(k1)))*. Finally

[2] For simplicity in the paper, it is assumed that the key dependency is an anti-symmetric relation.

[3] In this case further research is still needed to systematically derive the best-suited hash function.

explore the three-layer implementations *L(k3)-L(k2)-L(k1)*, as defined in Section 4.3, and select the best one. This is also the solution of a minimization problem, similar to the previous one, where $size_{k_3} + size_{k_2} + size_{k_1} = size_K$.

Let *BestImpl* and *BestCost* denote the best implementation reached so far, and its cost.

MUX core experiments The best one-, two-, and three-layer implementations for both networks 1 and 2 are reported in Table 1.

	Selected table implementations	Average memory size (bit)	Table cost
netw1	AR(18)	8 388 608	8 388 608
	PA(12)-AR(6)	339 136	678 272
	PA(9)-PA(5)-AR(4)	**161 513**	**484 538**
netw2	AR(18)	8 388 608	8 388 608
	PA(13)-AR(5)	**948 639**	**1 897 277**
	PA(10)-PA(5)-AR(3)	652 755	1 958 264

Table 1: **Selected implementations in Step 4**

Step 5 Assume that the selected two- and three-layer implementations from Step 4 are: *Two(k2)-Two(k1)* and *Three(l3)-Three(l2)-Three(l1)*. Then in contrast to Step 4, $Space_4$ becomes restricted and consists only of the four-layer implementations:

- generated from *Two(k2)-Two(k1)*, by simultaneously replacing $Two(k_i), i = 1, 2$, into two-layer implementations $L(k_{i2})$-$L(k_{i1})$, where $size_{k_{i2}} + size_{k_{i1}} = size_{k_i}$;
- generated from *Three(l3)-Three(l2)-Three(l1)*, by successively replacing $Three(l_i), i = 1, 2, 3$, into a two-layer implementation $L(l_{i2})$-$L(l_{i1})$, where $size_{l_{i2}} + size_{l_{i1}} = size_{l_i}$.

Select the best implementation in $Space_4$, which is again the solution of a minimization problen formulated and solved in Matlab. If the cost is $\geq BestCost$, then the exploration stops and outputs *BestImp* as the best reached implementation. If the cost is $< BestCost$, then *BestImp* and *BestCost* are updated, and this selected four-layer implementation is the next starting point for further exploration in Step 6.

MUX core experiments For network 1, the following implementations are explored:

L(j2)-L(j1)-L(i2)-L(i1), where $size_{j2} + size_{j1} = 12\ bits$ and $size_{i2} + size_{i1} = 6\ bits$,

L(j2)-L(j1)-PA(5)-AR(4), where $size_{j2} + size_{j1} = 9\ bits$,

PA(9)-L(j2)-L(j1)-AR(4), where $size_{j2} + size_{j1} = 5\ bits$,

PA(9)-PA(5)-L(j2)-L(j1), where $size_{j2} + size_{j1} = 4\ bits$.

For network 2, similar implementations are explored. The best implementations reached in each case are reported in Table 2. For both networks 1 and 2, the memory size is still decreasing, but only slightly, so that the cost cannot be decreased any more. Hence the exploration stops, and outputs *PA(9)-PA(5)-AR(4)* for network 1, and *PA(13)-AR(5)* for network 2. It can be shown that both implementations are the optimal ones respectively for networks 1 and 2.

	Reached table implementations	Average memory size (bit)	Table cost
netw1	**PA(7)-PA(5)-PA(3)-AR(3)**	**127 844**	**511 375**
	PA(5)-PA(4)-PA(5)-AR(4)	149 664	598 656
	PA(9)-PA(3)-PA(2)-AR(4)	149 941	599 763
	PA(9)-PA(5)-PA(2)-AR(2)	137 920	551 681
netw2	**PA(9)-PA(4)-PA(2)-AR(3)**	**615 624**	**2 462 497**
	PA(6)-PA(4)-PA(5)-AR(3)	632 356	2 529 424
	PA(10)-PA(2)-PA(3)-AR(3)	623 000	2 491 999
	PA(10)-PA(5)-PA(1)-AR(2)	687 721	2 750 885

Table 2: **Best reached implementations in Step 5**

Step 6 For any $r \geq 4$, assume that *BestImp* is an r-layer implementation. Then $Space_{r+1}$ is restricted to the r+1-layer implementations generated from *BestImp*, by successively replacing each layer into a two-layer implementation. It is similarly explored as in the previous step. The exploration stops when no better implementation is found. However, in practice it is observed that the exploration always stops before executing this step, and the optimal implementation is always reached for cases where an exhaustive search is still feasible to verify this.

6 Discussion of the results

To illustrate that the cost function is a valid estimation of the relative average power consumption in our method, let us compare this cost function with some empiric low-level power model[4]. Results are reported in Table 3. First estimations derived from this model are shown for all selected implementations during the exploration related to both networks 1 and 2. Then the ratio between table costs (from Table 1 and Table 2) and these power estimations is computed, and it remains relatively constant.

		Memory power	Table cost/ memory power
netw1	AR(18)	1 901	4 412
	PA(12)-AR(6)	154	4 407
	PA(9)-PA(5)-AR(4)	**110**	**4 405**
	PA(7)-PA(5)-PA(3)-AR(3)	116	4 401
netw2	AR(18)	1 901	4 412
	PA(13)-AR(5)	**430**	**4 410**
	PA(10)-PA(5)-AR(3)	444	4 410
	PA(9)-PA(4)-PA(2)-AR(3)	558	4 410

Table 3: **Comparison between power and cost**

To illustrate that optimized implementations are strongly dependent on the application characteristics, let us derive from the industrial model both memory area and power of the best reached implementations

[4] Industrial memory area/power model from 1996, assuming an embedded SRAM in 0.7 micron CMOS technology, with 1 read/write port, whose area unit is mm^2 and power unit is mW/sec, and extrapolated above 50 mm^2.

when used in both networks 1 and 2. Results are reported in Table 4. These show that *PA(9)-PA(5)-AR(4)* (resp. *PA(13)-AR(5)*) can really not be used in network 2 (resp. network 1).

		Memory power
netw1	**PA(9)-PA(5)-AR(4)**	**110**
	PA(13)-AR(5)	188
netw2	**PA(13)-AR(5)**	**430**
	PA(9)-PA(5)-AR(4)	459

Table 4: **Impact of application characteristics**

To illustrate the efficiency of our method, let us compare it with the one presented in [24]. Results are reported in Table 5. After hashing of both VCI and port, the best implementations reached by the method [24] are: *PA(VPI)-AR(VCI)* for network 1, and *PA(VCI)-AR(VPI)* for network 2. Both memory area and power are derived from the industrial model, for the best implementations reached by both methods. These show that key splitting/merging in selecting optimized implementations is really a must in our telecom network applications.

			Memory area	Memory power
netw1	Hash + [24]	PA(VPI)-AR(VCI)	601	327
	Our meth.	**PA(9)-PA(5)-AR(4)**	**135**	**110**
netw2	Hash + [24]	PA(VCI)-AR(VPI)	1 047	570
	Our meth.	**PA(13)-AR(5)**	**791**	**430**

Table 5: **Comparison between [24] and our method**

To illustrate the efficiency of our method, CPU time to run the corresponding Matlab script on a HP 9000 Series workstation has been measured. This takes 1.1 sec to output the optimized implementation for both networks 1 and 2.

7 Other applications

This section summarizes the results of our method applied to association tables used in F4 and SPP.

7.1 ATM switch component (F4)

Two small tables are used, whose records (320 bit size and 56 bit size respectively) are indexed by VPI (8 bit size). Since F4 and MUX core are both ATM switch components, network 1 from MUX core experiments can be considered also here.

VPI should not be hashed, and $freq_{\mathsf{VPI}} = 1.8$ (resp. 1.4) in network 1 (resp. network 2). Exploration results compared with those obtained from [24] are reported in Table 6. For both tables, the best implementation reached by our method is a two-layer one, which also happens to be the optimal one.

7.2 Segment Protocol Processor (SPP)

One large table is used, whose records (384 bit size) are indexed by two keys independent from each other:

		Memory area	Memory power
Tab1	PA(8)	13	7
	PA(7)-AR(1)	**11**	**6**
	PA(4)-PA(3)-AR(1)	9.3	8
	PA(4)-PA(2)-LL(1)-AR(1)	9	10
	[24] **PA(8)**	**13**	**7**
Tab2	AR(8)	12	3
	PA(5)-AR(3)	**3.4**	**2.0**
	PA(3)-PA(3)-AR(2)	2.8	2.5
	PA(3)-PA(3)-LL(1)-AR(1)	2.7	3
	[24] **AR(8)**	**12**	**3**

Table 6: F4 **exploration and comparison with [24]**

a multiplexing identifier MID (10 bit size) and a local identifier LID (7 bit size). Experiments are done relatively to two different telecom networks too: in network 1, the table contains 2^{13} records, whereas in network 2, the table contains 2^{16} records on average. From profiling information at the system level, it is observed that all MID values and only half of LID values are simultaneously active.

The selected key ordering is: LID, MID, but none of the keys need to be hashed. $size_K = 17\ bits$. Since only half (i.e. 2^6) of LID values and all MID values are used, in network 1

$avg_{rec} = 2^{13} = 2^6 * 2^{10/freq_{\mathsf{MID}}}$,

so that $freq_{\mathsf{LID}} = 7/6 = 1.2$ and $freq_{\mathsf{MID}} = 10/7 = 1.4$, whereas in network 2,

$avg_{rec} = 2^{16} = 2^6 * 2^{10/freq_{\mathsf{MID}}}$,

so that $freq_{\mathsf{LID}} = 7/6 = 1.2$ and $freq_{\mathsf{MID}} = 10/10 = 1$. Exploration results compared with those obtained from [24] are reported in Table 7. For both networks, the best implementation reached by our method is again a two-layer one, and also happens to be the optimal one.

		Memory area	Memory power
netw1	PA(17)	6 117	1 676
	PA(15)-AR(2)	**4 847**	**1 328**
	PA(12)-PA(4)-AR(1)	3 646	1 498
	PA(9)-PA(3)-PA(4)-AR(1)	3 586	1 965
	[24] **PA(LID)-PA(MID)**	**4 373**	**1 797**
netw2	AR(17)	41 943	5 744
	PA(7)-AR(10)	**20 975**	**5 745**
	PA(4)-PA(3)-AR(10)	20 974	8 618
	PA(1)-PA(3)-PA(3)-AR(10)	20 974	11 490
	[24] **PA(LID)-AR(MID)**	**20 975**	**5 745**

Table 7: SPP **exploration and comparison with [24]**

8 Conclusions

The major bottlenecks in embedded implementations of telecom network applications, are generally the memory area and power. A large part of the area is due to memory units, and the power consumption is heavily dominated by the huge amount of memory accesses. Hence the storage of the used dynamic data sets needs to be optimized already at the system level, where the

impact on area, performance, and power is the most important.

To this end we propose a new exploration and optimization method to select implementations for association tables of records indexed by keys, commonly encountered in telecom network applications. Our method fits in the context of embedded system synthesis for such applications and enables to further raise the abstraction level of the initial specification, where dynamic data sets can be specified without low-level details. Compared with existing realisations, it can save up to 2/3 of the memory power consumption and 3/4 of the memory area. In the future, we intend to extend our method to other dynamic data sets, such as buffers, sets of timers, and pools of buffers shared among active connections with different qualities of services.

Acknowledgments This work is funded by the European commission in the Esprit project No. 21929 (Media). We thank A. Hemani from Royal Institute of Technology Stockholm and G. De Jong from Alcatel, for providing us the specification of the MUX core, the F4, and the SPP. We also thank S. Wuytack from IMEC Leuven, for many insightful discussions.

References

[1] A.V. Aho et al. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] J.-Y. Le Boudec. The asynchronous transfer mode: a tutorial. *Computer Networks and ISDN Systems*, 24, 1992.

[3] F. Catthoor et al. *Global communication and memory optimizing transformations for low power signal processing systems.* In VLSI Signal Processing VII, IEEE Press, New York, 1994.

[4] D. Clark et al. An analysis of TCP processing overhead. *IEEE Communications Magazine*, June 1989.

[5] J.L. da Silva et al. Efficient system exploration and synthesis of applications with dynamic data storage and intensive data transfer. *DAC'98.*

[6] P. Ellervee et al. Exploiting data transfer locality in memory mapping. *Submitted to EUROMICRO'99.*

[7] A. Jantsch et al. Hardware/software partitioning and minimizing memory interface traffic. *EDAC'94.*

[8] M. Heddes. *A Hardware/Software Codesign Strategy for the Implementation of High-Speed Protocols.* PhD thesis, Technische Universiteit Eindhoven, 1995.

[9] A. Hemani et al. Design of operation and maintenance part of the atm protocol. *Journal on Communications, Hung. Sc. Society for Telecommunications, special issue on ATM networks*, 1995.

[10] K. Higuchi et al. Innovative system-level design environment based on FORM for transport processing system. *EDTC'98.*

[11] W. Horn. *Modelling of an ATM Multiplexer in a Network Terminal for a Mixed Hardware/Firmware Implementation.* PhD thesis, ESDLab/KTH, Stockholm, May 1998.

[12] P. Landman et al. Black-box capacitance models for architectural power analysis. *Int. workshop on Low Power Design*, 1994.

[13] MATLAB. http://www.mathworks.com/.

[14] T.H. Meng et al., Portable video-on-demand in wireless communication. *IEEE Proceedings, special issue on low power electronics*, April 1995.

[15] D.A. Patterson et al. Intelligent RAM(IRAM): chips that remember and compute. *IEEE Int. Conf. on Solid-State Circuits*, 1997.

[16] A. Seawright et al. A system for compiling and debugging structured data processing controllers. *EDAC with EURO-VHDL*, 1996.

[17] B. Svantesson et al. An efficient scheme for hardware implementation of processes with multiple active instances. *NORCHIP'97.*

[18] B. Svantesson et al. A methodology and algorithms for efficient interprocess communication synthesis from system descriptions in SDL. *Int. Conf. on VLSI Design*, 1998.

[19] Y. Therasse et al. VLSI architecture of a SDMS/ATM router. *Annales des Telecommunications*, 48(3-4), 1993.

[20] V. Tiwari et al. Instruction-level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13, 1996.

[21] P.H.A. van der Putten et al. Object-oriented co-design for hardware/software systems. *EUROMICRO'95.*

[22] R. Watson et al. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Trans. on Comp. Syst.*, May 1987.

[23] S. Wuytack et al. Global communication and memory optimizing transformations for low power systems. *Int. workshop on Low Power Design*, 1994.

[24] S. Wuytack et al. Transforming set data types to power optimal data structures. *IEEE Trans. on CAD*, June 1996.

[25] S. Wuytack et al. Memory management for embedded network applications. *IEEE Trans. on CAD*, May 1999.