System-Level Exploration of Association Table Implementations in Telecom Network Applications

CH. YKMAN-COUVREUR and J. LAMBRECHT
IMEC, Leuven
A. VAN DER TOGT
Technische Universiteit Delft
and
F. CATTHOOR and H. DE MAN
IMEC, Leuven and Katholieke University, Leuven

We present a new exploration and optimization method at the system level 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 average memory power, although it can also be driven by other cost functions such as memory size and performance. Compared with existing methods, for large dynamic data sets, it can save up to 90% of the average memory power, while still saving up to 80% of the average memory size.

Categories and Subject Descriptors: B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids

General Terms: Memory, Performance

Additional Key Words and Phrases: System-level exploration, memory management

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

This paper describes work undertaken in the context of the IST-1999-11419 Protocol Processor Project, a 2.5 years research and development project. The IST program is partially funded by the Commission of the European Union.
Authors' addresses: Ch. Ykman-Couvreur, J. Lambrecht, F. Catthoor, and H. De Man, IMEC, Kapeldreef 75, B3001 Leuven, Belgium; A. van der Togt, Technische Universiteit Delft, the Netherlands.

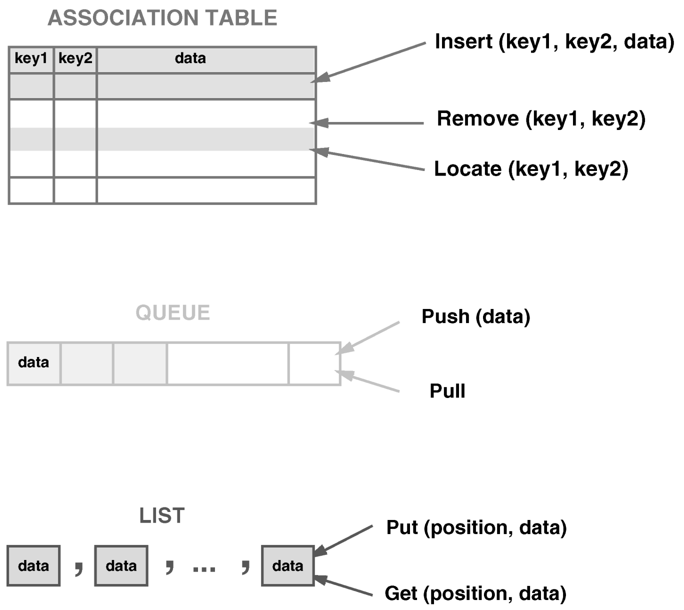


Fig. 1. Typical dynamic data sets.

level of the initial system specification must be raised, so that the designer is not burdened unnecessarily by low-level details in the final design. Also in view of embedded implementations, 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 dynamic queues, lists, association tables, and timer pools. Such behavior is also encountered in database and multimedia applications. These sets, illustrated in Figure 1, are called *dynamic data sets* in the sequel.

In embedded implementations of such telecom network applications, *a dominant bottleneck is the implementation of the dynamic data sets used*. Indeed:

(1) For many of these applications, major area and power are not involved in the data paths and the controllers, but in global communication and memory organization [Catthoor et al. 1994; Wuytack et al. 1994]. Indeed, to store these sets, large storage capacities are required, and a large part of the chip area is due to memory units [Boudec 1992; Therasse et al. 1993]. Hence, area optimization in system design should mainly concentrate on the area optimization of the memory storing such dynamic data sets.

(2) Telecom network applications also involve a number of basic services such as (1) *memory management* to dynamically (de)allocate memory, to detect memory overflow, and to prevent one dynamic data set from consuming all available memory; (2) *set management* to insert, locate, or remove data

from a set, whatever the set may be; (3) *timer management* to increment all active timers, check them for expiration, and generate timeouts for expired timers. These services also play a very important role, and may consume up to 80% of the processing time of the chip [Clark et al. 1989; Watson and Mamrak 1987]. For each of these services, many mechanisms are possible, whose efficiency depends on the application characteristics. To preserve high speed in telecom networks, an intelligent implementation of these services is required [Heddes 1995; Meleis and Serpanos 1992]. To this end, parallelism can be significantly exploited, since many operations in these services can occur in parallel.

(3) Finally, due to intensive data storage and transfer required by these services, the power of the chip is dominated by the huge amount of memory accesses, as demonstrated by recent work at IMEC [Catthoor et al. 1994], at Princeton University [Tiwari et al. 1996], at Stanford University [Meng et al. 1995], and in the IRAM project [Patterson et al. 1997]. This yields a significant amount of heat dissipation, which is a major problem in network switches. Moreover, all these memory accesses cannot be performed sequentially without violating the real-time requirements. Very high I/O memory bandwidth is needed, and some memory accesses need to be done in parallel through either multiple memories or a multiport memory [Wuytack et al. 1999]. Also, using heterogeneous memory architectures with small memories for frequently accessed data enables drastic reduction in power.

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 Wuytack et al. [1996]. In terms of area, performance, power, and depending on the application characteristics, a huge difference in cost between all these implementations has been experienced. 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 [Wuytack et al. 1996]. It fits in both system synthesis approaches, Matisse [da Silva Jr. et al. 1998; Verkest et al. 1999; and Ykman-Couvreur et al. 1999], where it is shown that incorporating dynamic data set synthesis enables us to achieve more efficient system designs. Several real-life telecom network applications are used to illustrate the efficiency of our method. Two of them are components in ATM switches: the multiplexer (MUX) core [Horn 1998], and one operation and maintenance component [Hemani et al. 1995], called F4. The third one is an important component in ATM backbone networks: the Segment Protocol Processor (SPP) [Therasse et al. 1993]. The fourth one is a base station component that implements the Automatic Repeat ReQuest (ARQ) protocol for error control in modern telecom systems [Schuler and Mateescu 1999]. Our method can 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 article is organized as follows. Section 2 summarizes the related work. Section 3 presents the MUX core and describes its association table, which is used to illustrate our method in the next sections. Section 4 overviews the cost function driving our exploration and optimization method. Section 5 characterizes the association table implementations considered in our search space. Section 6 presents our method, and Section 7 discusses the results. Section 8 applies our method to the other applications mentioned previously: F4, the SPP, and the ARQ component. Finally, conclusions are drawn in Section 9.

2. RELATED WORK

Apart from Matisse [da Silva Jr. et al. 1998; Verkest et al. 1999; Ykman-Couvreur et al. 1999], mentioned previously, and which integrated our method, very little support for dynamic data set synthesis is provided in current system design practice. We have found only two approaches for dynamic data set synthesis, presented in more detail later. The first one is followed in programming theory, whereas the second one is a preliminary method for telecom network applications, which we extend in this article.

In programming theory [Aho et al. 1983], 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 to reach software implementations either with high performance or with low memory size, but not with low memory power. Moreover, neither exploration nor optimization is automated. The cost factor in programming theory is different with our application domains, where memory size and power are more dominant cost factors, and performance must be treated as a hard constraint. This heavily influences the required exploration and optimization method.

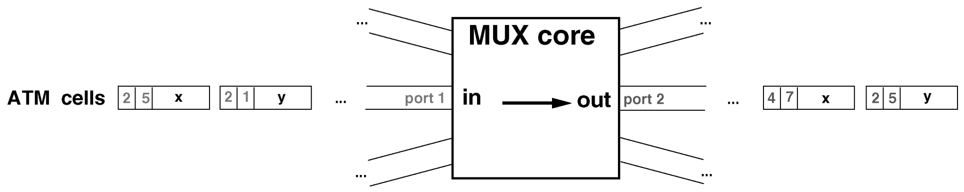
For telecom network applications, a preliminary exploration and optimization method is presented in Wuytack et al. [1996]. This method selects layered implementations for association tables of records indexed by keys. These layered implementations are obtained by combining the primitive data structures previously mentioned and optimized for power. The proposed method is suitable for hardware and software implementations. However, this has several major limitations. First, the search space in the method is too restrictive, and in several telecom network applications, the selected implementation is far from being optimal. Hashing and key splitting/merging 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. Moreover, the method supports at most three keys in practice. Second, the cost of any implementation in the search space is incorrectly estimated, and erroneous decisions are made through the exploration. This is due to the following reasons: (1) Dependencies between keys in the initial application specification are not taken into account. (2) All key values are assumed to be uniformly distributed. If not, keys must be hashed before applying the method. (3) A key (resp. a pointer) used in the data structures is assumed to occupy one memory word. This assumption has become unnecessary. Indeed, our dynamic data set synthesis fits in the context of Matisse, and relies on the subsequent synthesis step for data splitting/merging into memory words [Ellervee et al. 1999], before generating an optimized distributed memory architecture wherein dynamic data sets are stored. (4) The cost function relies too much on storage efficiency, compared to memory accesses. It does not model the power of the memories to be used in the final architecture. Hence, it is not really suited for power-driven exploration. (5) The cost function does not take all relevant application characteristics into account, that is, it assumes that the major operation on the association table is only locating a record given its key. Nevertheless, other operations, e.g., iterating on all records, may also have an important impact on the cost function.

2.1 Our Contribution

In this article, a new exploration and optimization method is proposed, which extends the previous method [Wuytack et al. 1996] as follows. It supports hashing and key splitting/merging, takes key dependencies and key value distributions into account, handles any number of keys, and removes all previous assumptions on the keys. All the extensions supported by our method give rise to a huge search space. Hence, to efficiently implement our method, we characterize it by a minimization problem that can be potentially solved using tools such as Matlab, Simulated annealing, Tempered annealing, Hill climbing, etc. In this article, this minimization problem is solved using a symbolic formulation in Matlab [MathWorks].

Our method is driven by a cost function that estimates the average memory power of each explored table implementation. To this end, the cost function takes all relevant application characteristics into account, to accurately estimate the needed memory size and the number of memory accesses. It also extrapolates at the system level the low-level power model of the memories to be used in the final embedded system design. Nevertheless, our method can also be driven by other cost functions such as memory size and performance.

The implementation of the services related to memory (e.g., allocation, memory overflow detection, and garbage collection) and set management (e.g., insert, locate, remove data) of these dynamic data sets, and the generation of a customized distributed memory architecture to meet the required performance, are part of subsequent synthesis steps in Matisse. They are outside the scope of this article. More details about these steps can be found in Wuytack et al. [1999].



key1	key2	key3	record
VPI-in	VCI-in	port_in	VPI-out, VCI-out, port-out
8 bits	16 bits	8 bits	32 bits
2 = 0000 0010	5 = 0000 0000 0000 0101	1 = 0000 0001	4 = 0000 0100, 7 = 0000 0000 0000 0111, 2 = 0000 0010
2 = 0000 0010	1 = 0000 0000 0000 0001	1 = 0000 0001	2 = 0000 0010, 5 = 0000 0000 0000 0101, 2 = 0000 0010
...	...	...	...

Association table of active connections

Fig. 2. MUX core overview.

3. MUX CORE DESCRIPTION

We now describe one component in the ATM switches, the MUX core, whose association table is used to illustrate our exploration and optimization method in the next sections.

The MUX core constitutes one block of a network terminal interfacing to both public access network and private Customer Premise Network (CPN). A detailed description of the MUX core can be found in Horn [1998]. As shown in Figure 2, the MUX core receives ATM cells, translates the connection identifiers (VPI, VCI) in their headers, and deliver them to their correct destination on either the CPN side or the access network side. ATM cell header translation and delivery are performed by looking up the association table, which stores information about all active connections. This table is indexed by the VPI (8 bit size), the VCI (16 bit size), and the input port (8 bit size) of the incoming ATM cells.

The following operations are performed on the table: *locate* a record with given keys to check whether a connection is already established or not; *insert* a new record with given keys whenever a new connection is established; *erase* a record with given keys whenever a connection is released; *is_empty_VPI* for a given VPI to check whether no further connection exists for the considered VPI; and *get_destination* for a given record to read information about the considered connection and to translate an incoming ATM cell header. These operations must be executed in real time: since the transmission rate in an ATM network is 155 Mb per sec, and since an ATM cell consists of 53 bytes, the timing budget to handle an incoming ATM cell and to execute all related operations is 2734 ns.

This table, if storing information about all possible active connections, would have one entry for each possible VPI/VCI/port combination, that is, $2^8 * 2^{16} * 2^8 = 2^{32}$ entries. Assuming that each entry consists of at least the VPI, the VCI, and the output port identification (i.e., $8 + 16 + 8 = 32$ bits $= 4$ bytes), the size of the complete table would therefore be $2^{32} * 4$ bytes $\simeq 17000$ MB. To execute all needed operations previously mentioned at real time, from simulation and profiling, about 12 memory accesses per ns would be performed.

Evidently, this table constitutes a major bottleneck for an embedded design of the MUX core. Optimization is required to generate an efficient implementation of the table. To this end, characteristics of the network where such MUX cores are implemented and characteristics of the needed operations must be taken into account in the optimization.

4. COST FUNCTION

4.1 System-Level Cost Parameters

In our embedded application domain, as explained in Section 1, memory performance must be treated as a hard constraint, whereas memory size and power are crucial cost factors and must be optimized.

For on-chip memories, the energy consumption of one memory access increases with the memory size, that is, bit-width and number of words. The dependency is between linear and logarithmic, depending on the library used. Several power models such as capacitance models [Landman and Rabaey 1994] and empiric models, provided by memory manufacturers, exist. The power of the internal interconnect and of the address calculation is still small (less than 20%) compared to that of the internal memories. Hence, it can be neglected in the system-level power estimations and taken into account only at the processor level. For off-chip memories, the energy consumption of one memory access can be considered more or less independent of 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 [Catthoor et al. 1998; Wuytack et al. 1999].

To this end, in the synthesis of the dynamic data sets, system-level optimizations should mainly aim at minimizing memory size, memory access, and memory power. Moreover, processor-level optimizations should mainly aim at further minimizing the power of the memories, of the internal interconnect, address calculation, and meeting the real-time requirements. Meeting the performance is taken at the processor level as follows: (1) To preserve high speed in telecom networks, the dynamic data sets require a hardware implementation. An intelligent implementation of the services accessing data is also needed by exploiting operation parallelism. (2) Due to intensive data storage and transfer required by these services, all memory accesses cannot be performed sequentially without violating the strict real-time requirements. Very high I/O memory bandwidth is needed, and some memory accesses need to be done in parallel through either multiple memories or a multiport memory.

4.2 System-Level Memory Power Model

In this article we assume that on-chip SRAMs, in 0.35 micron CMOS technology with one read/write port, are used in the final embedded system design. For this memory type, we use the following memory power model extrapolated from data sheets [STMicroelectronics]:

$$power = acc * 10^{-9} * (9.93 * bw \; + \; 0.0203 * w + 0.017 * bw * w \\ + \; 6.73 * log_2(bw * w + 18.5)), \qquad (1)$$

where bw is the bit-width of the used memory, w is the number of words in the memory, acc is the number of memory accesses per second, and the power unit is mW.

We focus on average memory power optimization at the system level. Our cost function is defined by Eq. (1), where we assume that $bw = 32$, w is the needed memory size (in bits) divided by bw, and acc is the average number of memory accesses in the execution of the needed operations. Nevertheless, any other memory model can also be used. In Ykman-Couvreur et al. [2002], a multiobjective cost function is used, and the exploration method not only optimizes memory power, but it trades off the average memory size, number of memory accesses, and memory power.

For any association table of records indexed by keys, the main application characteristics relevant for this cost function are: (1) the record size and the average number of records stored in the table; (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; (3) the operations performed on the table and the number of operation executions per second; (4) for any table implementation explored and for each operation, the average number of memory accesses in accessing the table. These characteristics are available either in early documents or in profiling information at the system level. This information is automatically generated using the simulation environment of our Matisse system synthesis approach.

MUX ***core application.*** In this representative application (see Section 3), one table is used, wherein records (32 bit size) are indexed by three keys independently of each other: the VPI (8 bit size), the VCI (16 bit size), and the port (8 bit size). We assume that the MUX core is used in a network where only the 10 least significant bits are used in the VCI, and only one port is needed. Both VPI and VCI (restricted to 10 bits) values are uniformly distributed. From profiling information, for each operation performed on the table, the average number of executions per second is as follows: 205925 for *locate*, 1829 for *insert*, 1829 for *erase*, 1463 for *is_empty_VPI*, and 155084 for *get_destination*. The average number of memory accesses per operation in accessing the table is of course dependent on the table implementation. This is considered in Section 5. In the sequel, experiments are reported relative to two different telecom networks where such MUX cores are implemented: In network 1, the table stores 2^{10} records, whereas in network 2, the table stores 2^{16} records on average. Hence, at most 10 bits (resp. 16 bits) are needed to specify a pointer of any table implementation in network 1 (resp. network 2). All these application characteristics are required to estimate the average memory power from Eq. (1).

5. TABLE IMPLEMENTATIONS

This section characterizes the layered implementations considered in our search space, with their respective costs computed from Eq. (1). To this end,

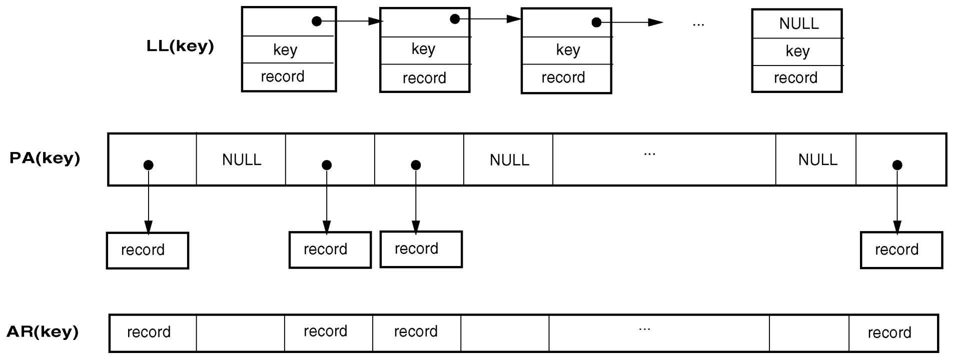


Fig. 3. One-layer implementations.

Table I. Memory Sizes for One-Layer Implementations in MUX Core

One-Layer Implementation	Average Memory Size (Mb)	
	network 1	network 2
LL(k)	0.06	4.3
PA(k)	2.7	6.3
AR(k)	8.4	8.4

the following notation is 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; and $size_{ptr}$ = the record pointer size. We also assume that the size unit is the bit.

5.1 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 3). In the future, ordered linked lists and binary trees will also be considered.

In the unordered linked list, denoted *LL*(k), elements are dynamically (de)allocated. Each of them stores the key value, 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 $(size_{ptr} + size_k + size_{rec}) * avg_{rec}$.

The pointer array, denoted *PA*(k), is an array of pointers to records. The pointer array stores a record pointer for each active key value. Key values do not need to be stored since the key value corresponds to the position of the pointer in the array. The average memory size is $size_{rec} * avg_{rec} + 2^{size_k} * size_{ptr}$.

The array, denoted *AR*(k), reserves memory for each record it can store. Hence, many memory locations are wasted if the average number of records stored in the array is relatively small. The memory size is $2^{size_k} * size_{rec}$.

MUX ***core application.*** First, for each one-layer implementation, the average memory size in both network 1 and network 2 is given in Table I. Then,

Table II. Memory Accesses for One-Layer Implementations in MUX Core

	Average Memory Accesses		
Operation	$LL(k)$	$PA(k)$	$AR(k)$
locate	$avg_{rec} - 1$	1	1
insert	2	2	1
erase	$avg_{rec} + 1$	1	1
is_empty_VPI	$avg_{rec} * \frac{sizeVPI}{size_k} - 1$	$2^{size_k - size_{VPI} - 1}$	$2^{size_k - size_{VPI} - 1}$
get_destination	1	1	1

Table III. Memory Power for One-Layer Implementations in MUX Core

	Average Memory Power (mW)					
	network 1			network 2		
Operation	$LL(k)$	$PA(k)$	$AR(k)$	$LL(k)$	$PA(k)$	$AR(k)$
locate	317.7	9.7	30.6	1035663.9	22.9	30.6
insert	0.006	0.2	0.3	0.03	0.4	0.3
erase	2.8	0.09	0.3	9198.9	0.2	0.3
is_empty_VPI	1.3	35.4	111.2	4087.6	83.5	111.2
get_destination	0.2	7.3	23.0	11.9	17.3	23.0
Total	322.0	52.7	165.4	1048962.3	124.3	165.4

for each one-layer implementation and for each operation performed on the MUX core table, the average number of memory accesses in accessing the table is given in Table II. For example, the average number of memory accesses in the *locate* operation is $avg_{rec}/2$ key accesses + $(avg_{rec}/2 - 1)$ pointer accesses (i.e., $avg_{rec} - 1$, for the $LL(k)$ implementation) and only one access to check whether the record is empty or not in both $PA(k)$ and $AR(k)$ implementations. Also, the average number of memory accesses in the *erase* operation for the $LL(k)$ implementation is the number of accesses to locate the record + 2 pointer accesses, that is, $avg_{rec} + 1$. Finally, for each one-layer implementation and for each operation, again, the average memory power derived from Eq. (1) in both network 1 and network 2 is given in Table III. Evidently, $LL(k)$ may not even be implemented using on-chip SRAMs.

Hence, among these three primitive data structures, finding the best implementation relative to memory power is already not evident. This is due to trade-offs between needed memory size and memory accesses. Moreover, these trade-offs depend heavily on the application characteristics.

5.2 Two-Layer Implementations

In our search space, two-layer implementations are obtained by combining primitive data structures, as illustrated in Figure 4. *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 3 and 4.[1] *AR(k_2)LL(k_1)* is equivalent to *PA(k_2)LL(k_1)*. If only k is known in the application specification, and k_2 and k_1 must be computed, then *AR(k_2)PA(k_1)* is worse

[1]*AR(k_2)LL(k_1)* is the same as *PA(k_2)LL(k_1)*. *AR(k_2)PA(k_1)* is the same as *PA(k_2)PA(k_1)*. *AR(k_2)AR(k_1)* is equivalent to *AR(k_2+k_1)*, but it needs more control logic to calculate two keys instead of only one.

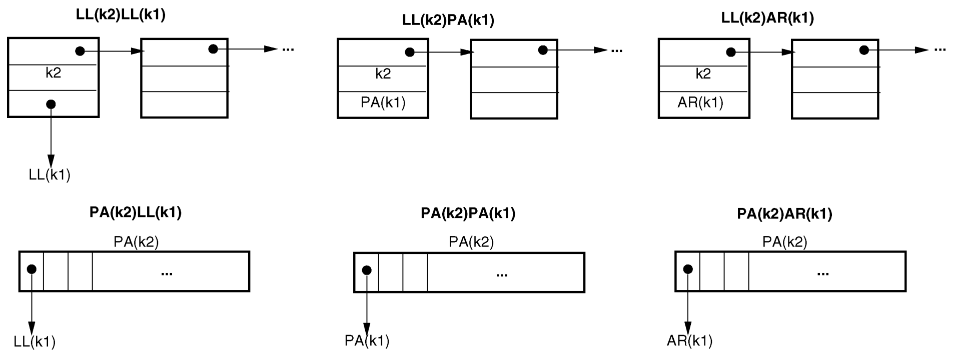


Fig. 4. Two-layer implementations.

Table IV. *LL(VPI)PA(VCI)* Cost in MUX Core

Operation	Average Memory Accesses	Average Power in network 1 (mW)
locate	avg_{VPI}	22.4
insert	$\frac{avg_{VPI}}{2^{size_{VPI}}} * (avg_{VPI} + 1) +$ $(1 - \frac{avg_{VPI}}{2^{size_{VPI}}}) * (2 * avg_{VPI} + 4)$	0.05
erase	avg_{VPI}	0.2
is_empty_VPI	$avg_{VPI} - 1$	0.2
get_destination	1	0.8
	Total	23.7

than *PA(k)*, and *AR(k_2)AR(k_1)* is worse than *AR(k)*. Otherwise they are equivalent.

For each two-layer implementation, the cost can be easily derived. Let us illustrate it for *LL(k_2)PA(k_1)*. Its average memory size is $(size_{ptr} + size_{k_2} + average\ size\ of\ PA(k_1)) * avg_{k_2}$, that is, $((1 + 2^{size_{k_1}}) * size_{ptr} + size_{k_2}) * avg_{k_2} + size_{rec} * avg_{rec}$.

MUX ***core application.*** For the two-layer implementation *LL(VPI) PA(VCI)*, substantial cost reductions can already be observed compared to one-layer implementations. For example, in network 1, and assuming that $avg_{VPI} = 22$, the average memory size in network 1 is only about 0.3 Mb. Estimations about average memory accesses and power are shown in Table IV. For example, the average number of memory accesses in the *insert* operation is derived as follows. We must check whether *LL(VPI)* contains an element with the given *VPI*. If yes (the probability is $avg_{VPI}/2^{VPI}$), the check requires $avg_{VPI} - 1$ memory accesses on average to locate it, and two additional memory accesses are needed to insert the record in *PA(VCI)*. Otherwise, the check through the complete linked list requires $2 * avg_{VPI}$ memory accesses, and four additional memory accesses are needed to insert a new element in the linked list and to insert the record in the associated *PA(VCI)*.

5.3 r-Layer Implementations

r-layer implementations, $r \geq 2$, are obtained by combining primitive data structures similar to the two-layer implementations. A three-layer implementation

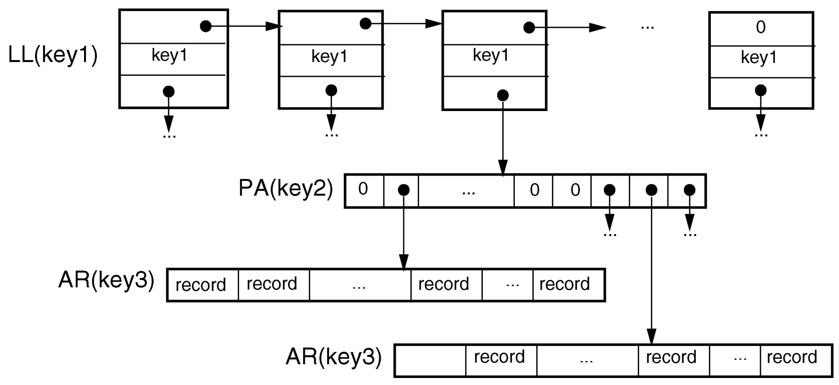


Fig. 5. A three-layer implementation.

is illustrated in Figure 5. A general r-layer implementation is denoted $L_r(k_r)$ $L_{r-1}(k_{r-1})$ ... $L_1(k_1)$, where $L_1(k_1)$ is the lowest layer; $L_r(k_r)$ is the highest one; $L_1(k_1)$ is $LL(k_1)$, $PA(k_1)$, 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 for the two-layer implementations. Its cost can be recursively derived taking into account that: (1) Its average memory size is the average size of $L_r(k_r)$ whose stored records are the (r-1)-layer subimplementations $L_{r-1}(k_{r-1})$ ... $L_1(k_1)$. (2) For each operation *op*, the average number of memory accesses per execution is in general the average number of memory accesses to execute a *locate* operation in $L_r(k_r)+$ the average number of memory accesses to execute *op* in $L_{r-1}(k_{r-1}) \ldots L_1(k_1)$. A systematic way to derive the number of memory accesses for each operation and for each possible r-layer implementation is detailed in the Appendix.

6. EXPLORATION AND OPTIMIZATION METHOD

For any association table of records indexed by n keys $k_1, k_2, \ldots, k_n$ in the initial application specification, the problem is to find the best r-layer implementation in terms of the cost function characterized in Section 4. Several degrees of freedom must be considered: selecting the number r of layers, choosing a primitive data structure for each layer, hashing, ordering, splitting, and conversely merging the keys in the initial application specification to generate the r keys in the r-layer implementation. This gives rise to a huge search space, as is illustrated later.

MUX ***core application.*** For a network making full use of the 8 VPI bits, the 16 VCI bits, and the 8 port bits, we first make the following observations. From the three initial keys, VPI, VCI, and port, there are $3! = 6$ possible key orderings. Each key ordering, after having merged the intial keys, gives rise to a super-key of 32 bits. This super-key can be split into r keys in $31!/((32-r)!(r-1)!)$ possible ways. Hence, the search space for the table consists of (1) three one-layer implementations, being the three primitive data structures defined in Section 5.1; (2) the 1116 two-layer implementations, taking into account the six possible combinations of primitives defined in Section 5.2, the six possible key orderings, and the 31 possible key splittings; (3) the 33480 three-layer implementations; 647280 four-layer implementations, and so on; (4) that is, $18*2^{r-1}*31!/((32-r)!(r-1)!)$ r-layer implementations, for $r \geq 2$, taking into account the $3*2^{r-1}$ possible combinations of primitives, the six possible key orderings, and the $31!/((32-r)!(r-1)!)$ possible key splittings.

Exhaustive exploration of the complete search space of layered implementations is not possible for the following reasons. First, such a search space must be explored for each dynamic data set of the application. Second, dynamic data set synthesis fits in the context of embedded system synthesis, as does our Matisse approach, which is based on fast and successive explorations at all levels of the complete design flow to achieve efficient system designs. Finally, the more refined the application specification becomes in the design flow, the more complex and slower the explorations. Since dynamic data set synthesis is one of the first system-level synthesis steps in the design flow, exploration in this step must be extremely fast.

Hence, an efficient exploration and optimization method based on heuristics and tool support is needed to derive at least near-optimal solutions. In the following, we first overview our method for solving this problem in Section 6.1, and then describe its various steps in detail in Section 6.2.

6.1 Method Overview

The first three steps initialize the search space exploration by ordering, hashing and concatenating the keys in the initial application specification. This enables us to form one super-key from which the r keys in any r-layer implementation of the search space are generated. The next steps successively explore one-layer implementations, two-layer, three-layer ones, and so on, until either the optimal or a near-optimal implementation is reached. Let $Space_r$ denote the search subspace of r-layer implementations explored in our method. Each $Space_r$ is exhaustively explored, but for reasons given later, $Space_r$ becomes more and more restricted while r increases: (1) As illustrated in our applications, the probability is very high that the optimal implementation is 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. (2) From $r \geq 4$, even when the r-layer implementation cost can still be minimized, the decrease is no longer significant. Indeed, the memory access number is still increasing, and is hardly compensated by a sufficient decrease in memory size. Hence, exhaustive exploration becomes useless. (3) Moreover, the number of all possible r-layer implementations increases exponentially with r, as shown previously. Depending on $size_{k_i}$, $1 \leq i \leq n$, $Space_r$ must be restricted accordingly to keep our exploration fast.

MUX ***core application.*** Ranges for needed memory size, average memory accesses per second, and average memory power, among all r-layer $(1 \leq r \leq 6)$ implementations of the table in network 1 are shown in Figure 6. This illustrates that the exploration can stop when no better implementation is reached. The best implementation reached is the optimal one.

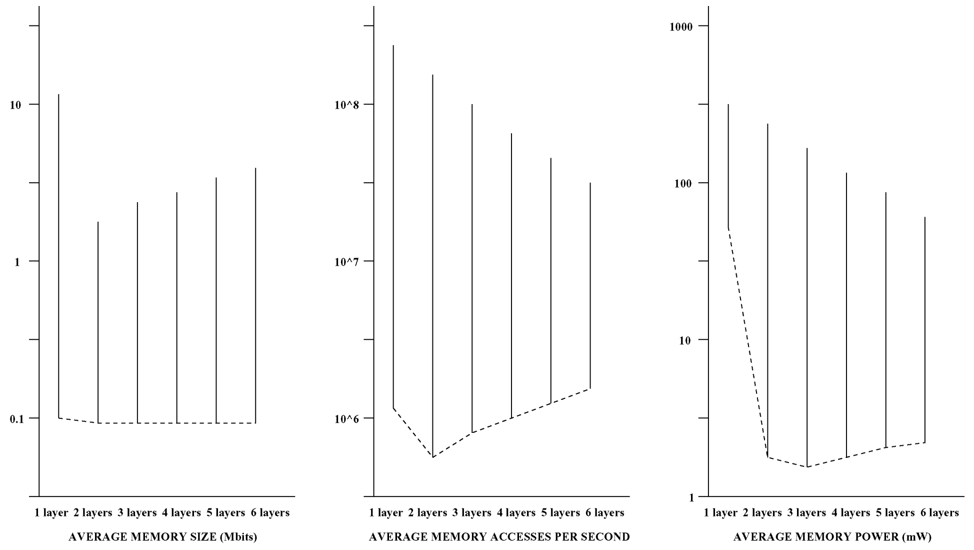


Fig. 6. Cost ranges in search space of the MUX core table.

6.2 Method Description

Our method is characterized by a minimization problem that can be potentially solved using tools such as Matlab, Simulated annealing, Tempered annealing, Hill climbing, etc. In this article, this minimization problem is solved using a symbolic formulation in Matlab [MathWorks], because it is fast and gives very good results (see Section 7). We now describe the different steps of our method, whose global flow is illustrated in Figure 7.

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

(1) o_i depends on $o_j \Rightarrow i < j$. This key ordering[2] naturally yields fewer memory accesses to locate records in the table.

(2) $avg_{o_i}/max_{o_i} > avg_{o_j}/max_{o_j} \Rightarrow i < j$. With this key ordering the memory allocated for the implementation of the lower layers is used as efficiently as possible. Since lower layers generally store more data than higher layers, this reduces the needed memory size of the table.

MUX ***core application.*** In the *is_empty_VPI* operation, VPI is the only accessed key. In the other operations the keys are independent of each other, and their values are uniformly distributed. Hence, the selected higher key is VPI, whereas the ordering between VCI and the port remains irrelevant.

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

[2]For simplicity, in the article 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 most suitable hash function.

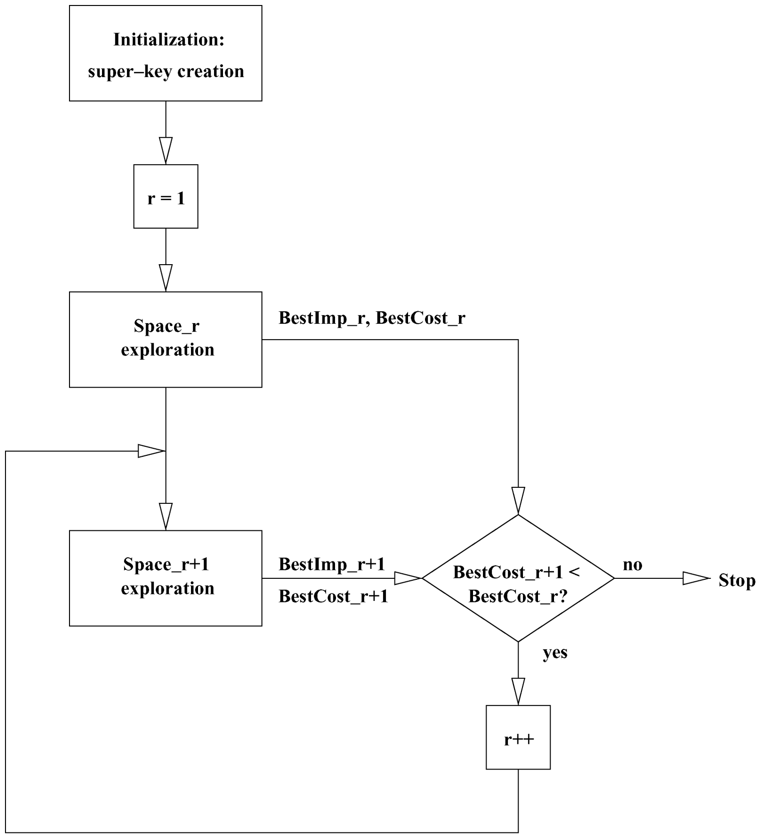


Fig. 7. Exploration and optimization method.

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 application.*** In both networks 1 and 2, only 10 bits are used in the VCI, and only one port is needed. Hence the VCI and the port are hashed as follows: $0 \leq$ hash(VCI) $< 2^{10}$, and hash(port) = constant. This implies that the hash(port) can be left out in the next steps of the method. Also from now on, to simplify the notation, VCI refers to this hashed VCI, rather than to the one used in the initial MUX core specification.

Step 3. Concatenate all keys $h_n, \ldots, h_2, h_1$ to form one super-key K whose $size_K = size_{h_n} + \cdots + size_{h_2} + size_{h_1}$, and $avg_K = avg_{rec}$. At the same time, for each h_i, $1 \leq i \leq n$, derive $sparse_{h_i}$ such that[4]

$$\begin{aligned} avg_{h_i|h_n,\ldots,h_{i+1}} &= 2^{size_{h_i}/sparse_{h_i}}, \quad 1 \leq i < n, \\ avg_{h_n} &= 2^{size_{h_n}/sparse_{h_n}}, \end{aligned}$$

taking into account that $avg_{rec} = avg_{h_n} * \cdots * avg_{h_2|h_n,\ldots,h_3} * avg_{h_1|h_n,\ldots,h_2}$, which can be derived from the application characteristics. These $sparse_{h_i}$, $1 \leq i \leq n$,

[4]See notation introduced in Section 5.

Table V. Selected Implementations in Step 4

Network Characteristics	Selected Table Implementation	Average Memory Size (Mb)	Average Memory Accesses per sec	Average Memory Power (mW)
network 1	PA(18)	2.7	$1.12 * 10^6$	53.0
	PA(12)PA(6)	0.14	$0.59 * 10^6$	1.69
	PA(9)PA(5)PA(4)	0.08	$0.79 * 10^6$	1.49
network 2	PA(18)	6.29	$1.12 * 10^6$	125
	PA(14)AR(4)	3.12	$0.63 * 10^6$	35.1
	PA(10)PA(5)AR(3)	2.90	$0.79 * 10^6$	40.9

are needed to compute the cost of any r-layer implementation. They indicate how sparse the ith layer of the table implementation is.

MUX ***core application.*** $size_K = 18\ bits$. Since VPI and VCI values are independent of each other and uniformly distributed, $sparse_{\mathsf{VPI}} = sparse_{\mathsf{VCI}} = p$, and for any splitting of K into $k_r, \ldots, k_2, k_1$, $sparse_{k_i}, 1 \leq i \leq r, = p$, too. In network 1, $avg_{rec} = 2^{10} = 2^{8/p} * 2^{10/p}$ so that $p = 18/10 = 1.8$, whereas in network 2, $avg_{rec} = 2^{16} = 2^{8/p} * 2^{10/p}$ so that $p = 16/13 = 1.2$.

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 5.1, and select the best one. Then explore the two-layer implementations $L(k_2)L(k_1)$, as defined in Section 5.2, and select the best one, which is the solution to the following minimization problem:

$$MIN_{size_{k_2}+size_{k_1}=size_K}$$
$$\{cost(LL(k_2)LL(k_1)),\ cost(LL(k_2)PA(k_1)),\ cost(LL(k_2)AR(k_1)),$$
$$cost(PA(k_2)LL(k_1)),\ cost(PA(k_2)PA(k_1)),\ cost(PA(k_2)AR(k_1))\},$$

where the cost function is the one defined in Section 4.2. Finally, explore the three-layer implementations $L(k_3)L(k_2)L(k_1)$, as defined in Section 5.3, 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 *BestImp* and *BestCost* denote the best implementation reached so far and its cost. If *BestImp* is a two-layer implementation, then the exploration stops and outputs *BestImp* as the best implementation reached. If *BestImp* is a three-layer implementation, then the exploration goes further, and Step 5 is performed.

MUX ***core application.*** The best one-, two-, and three-layer implementations for both networks 1 and 2 are reported in Table V. For network 1, the exploration needs to go further, whereas for network 2, it stops and selects *PA(14)AR(4)* as the table implementation. It can be shown by exhaustive exploration of the complete search space that this implementation is the optimal one for network 2.

Step 5. Assume that the selected two- and three-layer implementations from Step 4 are $L(k_2)L(k_1)$ and $L(l_3)L(l_2)L(l_1)$. Then in contrast to Step 4, $Space_4$

Table VI. Best Reached implementations for Network 1 in Step 5

Reached Table Implementation	Average Memory Size (Mb)	Average Memory Accesses per sec	Average Memory Power (mW)
PA(7)PA(5)PA(3)PA(3)	0.073	$1.01 * 10^6$	1.72
PA(8)PA(4)PA(4)PA(2)	0.074	$1.00 * 10^6$	1.72
PA(6)PA(6)PA(3)PA(3)	0.074	$1.01 * 10^6$	1.74
PA(7)PA(5)PA(4)PA(2)	0.074	$1.01 * 10^6$	1.74

becomes restricted and consists only of the four-layer implementations:

—generated from *L(k$_2$)L(k$_1$)*, by simultaneously replacing $L(k_i), i = 1, 2$, into two-layer implementations *L(k$_{i2}$)L(k$_{i1}$)*, where $size_{k_{i2}} + size_{k_{i1}} = size_{k_i}$;

—generated from *L(l$_3$)L(l$_2$)L(l$_1$)*, by successively replacing $L(l_i), i = 1, 2, 3$, 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 $Space_4$ also by using Matlab to solve a minimization problem similar to the previous one. 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 the selected four-layer implementation is the next starting point for further exploration in Step 6.

MUX ***core application.*** 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)PA(4), where $size_{j2} + size_{j1} = 9$ *bits*,
PA(9)L(j2)L(j1)PA(4), where $size_{j2} + size_{j1} = 5$ *bits*,
PA(9)PA(5)L(j2)L(j1), where $size_{j2} + size_{j1} = 4$ *bits*.

The best implementations reached are reported in Table VI. The memory size is still decreasing, but only slightly, whereas the number of memory accesses is also increasing, so that the power cannot be decreased any more. Hence, the exploration stops and outputs *PA(9)PA(5)PA(4)* for network 1. Again, it can be shown by exhaustive exploration of the complete search space that this selected implementation is the optimal one for network 1.

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 explored similarly to 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.

7. DISCUSSION OF THE RESULTS

To illustrate that optimized implementations are strongly dependent on the network characteristics, let us compare the average memory power of the best

Table VII. Impact of Network Characteristics

Network Characteristics	Best Selected Table Implementation	Average Memory Size (Mb)	Average Memory Accesses per sec	Average Memory Power (mW)
network 1	PA(9)PA(5)PA(4)	0.08	$0.79 * 10^6$	1.49
	PA(14)AR(4)	0.28	$0.66 * 10^6$	3.4
network 2	PA(14)AR(4)	3.12	$0.63 * 10^6$	35.1
	PA(9)PA(5)PA(4)	3.66	$0.79 * 10^6$	51.1

Table VIII. Impact of Operation Characteristics

Operation Characteristics	Best Selected Table Implementation	Average Memory Size (Mb)	Average Memory Accesses per sec	Average Memory Power (mW)
case 1	PA(9)PA(5)PA(4)	0.083	$0.79 * 10^6$	1.49
	PA(10)PA(5)PA(3)	0.084	$1.58 * 10^6$	3
case 2	PA(10)PA(5)PA(3)	0.084	$0.79 * 10^6$	1.5
	PA(9)PA(5)PA(4)	0.083	$1.07 * 10^6$	2

Table IX. Comparison with Existing Method

Network Characteristics	Exploration Method	Best Selected Table Implementation	Average Memory Size (Mb)	Average Memory Accesses per sec	Average Memory Power (mW)
network 1	Hash + [Wuytack et al. 1996]	PA(VPI)AR(VCI)	0.72	$1.33 * 10^6$	17.4
	Our method	PA(9)PA(5)PA(4)	0.08	$0.79 * 10^6$	1.49
network 2	Hash + [Wuytack et al. 1996]	PA(VCI)AR(VPI)	3.9	$1.11 * 10^6$	115.6
	Our method	PA(14)AR(4)	3.12	$0.63 * 10^6$	35.1

selected implementations when used in both networks 1 and 2. Results are reported in Table VII. Between both implementations, a difference of 56% (resp. 31%) is observed (resp. network 2).

Then to illustrate that optimized implementations are also strongly dependent on the operation characteristics, let us compare the average memory power of the best selected implementations when used in network 1, but with different operation characteristics: (1) the first case is based on the operations characterized in Section 4.2; (2) the second case is based on the *is_empty_VCI* (instead of *is_empty_VPI*) operation with the same average number of executions per second. Results are reported in Table VIII. Observe that VPI is the higher key in *PA(9)PA(5)PA(4)*, whereas VCI becomes the higher key in *PA(10)PA(5)PA(3)*. Here again, between the two implementations, there is a difference of 50% (resp. 25%) in average memory power observed for (resp.) network 2.

To illustrate the efficiency of our method, let us compare it with the one presented in Wuytack et al. [1996]. Results are reported in Table IX. After hashing both VCI and port, the best implementations reached by the method [Wuytack et al. 1996] are *PA*(VPI)*AR*(VCI) for network 1, and *PA*(VCI)*AR*(VPI) for network 2. A difference of 91% (resp. 70%) is observed (resp. network 2) between the two methods. Moreover, in current MUX core design, *PA*(VPI)*AR*(VCI) is the table implementation used whatever the network characteristics are.

Table X. F4 Exploration and Comparison with Existing Method

Best Selected Table Implementation	Average Memory Size (Kb)	Average Memory Accesses per sec	Average Memory Power (mW)
PA(8)	3.3	$0.41 * 10^6$	0.19
PA(7)-AR(1)	2.7	$0.59 * 10^6$	0.26
[Wuytack et al. 1996] AR(8)	14.3	$0.37 * 10^6$	0.24

This experiment shows that systematic exploration of key splitting/merging in selecting optimized implementations is really a must in our telecom network applications.

To illustrate the efficiency of our method, CPU time to run the corresponding Matlab script on a HP 9000 Series workstation has been measured. For network 1, up to four-layer implementations must be explored, and the exploration takes 11.5 sec to output the optimized implementation. For network 2, only up to three-layer implementations must be explored, and the exploration takes 6.7 sec.

8. OTHER APPLICATIONS

This section summarizes the results of our method applied to association tables used in telecom network applications, such as F4, the SPP, and the ARQ component.

8.1 Operation and Maintenance Component (F4)

The OAM functionality [Hemani et al. 1995] of an ATM switch is organized into five hierarchical layers, denoted F1–F5 by ITU. F1–F3 operate at the physical layer, F4 operates at the virtual path layer, and F5 at the virtual channel layer. F4 and F5 are very similar, and in this article we address F4.

F4 operates on both the forward cell-stream from the network to the ATM switch and on the backward cell-stream from the ATM switch to the network. All incoming ATM cells and signals are processed in view of fault management (including alarm surveillance, failure localization in the physical layer and testing), performance monitoring (related to cell loss ratio, cell insertion rate, and effective use of the network resources), or activation/deactivation of performance monitoring. In F4, a small table is used, whose records (56 bit size) are indexed by the virtual path identifier VPI (8 bit size). This table stores status information for all active connections. Since the MUX core and F4 are both ATM switch components, network 1 from MUX core experiments can also be considered here.

The VPI should not be hashed, and from Section 6.2 $sparse_{\mathrm{VPI}} = 1.8$ in network 1. For this F4 table, our method is again compared with the one presented in Wuytack et al. [1996]. Results are reported in Table X. To use our method, we need the average number of executions per second for the different operations on the table. These are 133005 for *locate*, including record reading; 49877 for *insert*; and 49877 for *erase*. The best implementation reached by our method is a pointer array, which also happens to be the optimal implementation.

Table XI. SPP Exploration and Comparison with the Existing Method

Network Characteristics	Best Selected Table Implementation	Average Memory Size (Mb)	Average Memory Accesses per sec	Average Memory Power (mW)
network 1	PA(17)	24.5	$0.20 * 10^6$	87.0
	PA(7)-AR(10)	22.4	$0.29 * 10^6$	116
	PA(7)-PA(1)-AR(9)	22.4	$0.38 * 10^6$	152
	[Wuytack et al.1996] PA(LID)-AR(MID)	22.4	$0.29 * 10^6$	116
network 2	PA(17)	1.2	$0.20 * 10^6$	4.2
	PA(11)-PA(5)	0.17	$0.31 * 10^6$	1.1
	PA(10)-PA(3)-LL(4)	0.13	$0.46 * 10^6$	1.2
	[Wuytack et al.1996] PA(MID)-PA(LID)	0.21	$0.31 * 10^6$	1.3

8.2 Segment Protocol Processor (SPP)

The SPP [Therasse et al. 1993] is an important component in ATM backbone networks, whose functionality is to store and forward user and internal ATM cells, to perform several checks, to issue requests (e.g., routing requests), to process routing replies, and to make garbage collection. In addition, internal ATM cells are exchanged with a coprocessor, and communication takes place with a supervising microprocessor for error and performance monitoring. The ATM cells carry various fields for control and management purposes. Among others, two fields are the local identifier LID (7 bit size) and the multiplexing identifier MID (10 bit size). The LID identifies a connection from a user to a server. The MID identifies a program making use of that connection.

In the SPP, one large table is used, whose records (384 bit size) are indexed by two keys: the MID and the LID. Experiments are also done relative to two different telecom networks: in network 1, the table stores 2^{16} records on average, whereas in network 2, the table stores 2^8 records on average. From profiling information at the system level, it is observed that all MID values and only half of LID values are active simultaneously. Also, the profiled average number of executions per second for the different operations on the table are 73153 for *locate*, including record reading, 18288 for *insert*, and 18288 for *erase*.

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. (1) In network 1, $avg_{rec} = 2^{16} = 2^6 * 2^{10/sparse_{\text{MID}}}$, so that $sparse_{\text{LID}} = 7/6 = 1.2$ and $sparse_{\text{MID}} = 10/10 = 1$. (2) In network 2, $avg_{rec} = 2^8 = 2^6 * 2^{10/sparse_{\text{MID}}}$, so that $sparse_{\text{LID}} = 7/6 = 1.2$ and $sparse_{\text{MID}} = 10/2 = 5$. Exploration results compared with those obtained from Wuytack et al. [1996] are reported in Table XI, similarly to Table X.

8.3 ARQ Component

In a mobile network, the part of the base station that implements the reliable transmission of data coming from multiple mobile terminals having multiple connections with different priorities presents several design bottlenecks, especially when moving to very high data rates. The Automatic Repeat ReQuest (ARQ) protocol is the most robust approach, ensuring reliable data transmission in mobile communications. It consists of both loss detection and loss recovery. For each new up-link connection (from a mobile terminal to the base station) with a given priority, the mobile terminal dynamically starts up a sender ARQ process and the base station starts up a receiver ARQ process. For a down-link connection (from the base station to a mobile terminal), the roles of the mobile terminal and the base station are exchanged. Hence, if the base station must be able to communicate with 64 mobile terminals, and if each mobile terminal can have four connections with their own priorities, the base station may need to implement up to 256 concurrent ARQ processes, whose current status information must be stored dynamically in the base station.

Several ARQ protocols are available, and in this article we consider the Slot Based Selective Repeat ARQ protocol (SBSA) [Schuler and Mateescu 1999]. The ARQ component experimented with in the following consists of the receiver ARQ processes in up-link connections. It uses three tables: (1) The first table keeps track of the active receiver ARQ processes, identified by two keys: the mobile terminal identifier MT-ID (6 bit size) and the connection identifier CID (2 bit size). In this table, each record occupies 266 bits, and the average number of records (64 in the experiment) depends on the average number of active connections. The average number of executions per second for the different operations on this table are 42857 for *locate*, 14286 for *insert*, 14286 for *erase*, and 28571 for *read*. (2) The second table stores timing information about lost packets for each active receiver ARQ process. This allows us to order the packets in the packet stream. In this table, each record is identified by four keys: MT-ID, CID, a wrap counter WC (4 bit size), and a sequence number SN (4 bit size). Each record occupies 64 bits, and the average number of records (128 in the experiment) depends on the channel quality. The average number of executions per second for the different operations on this table are 12500 for *insert*, 12500 for *erase*, and 37500 iterations through the whole table. (3) For each connection, whenever a lost packet is detected, packets following a lost packet are blocked in the base station: they cannot be forwarded because the correct packet ordering must be preserved. To this end, a third table stores these blocked packets (392 bit size) for each active receiver ARQ process. In this table, each record is also identified by the four keys, MT-ID, CID, WC, and SN. The average number of blocked packets (256 in the experiment) also depends on the channel quality. The average number of executions per second for the different operations on this table are 350 for *insert*, 350 for *erase*, and 28000 for *look up* through the whole table with a given MT-ID and CID.

As for the previous applications, exploratory results compared with those obtained from Wuytack et al. [1996] are reported in Table XII. Nevertheless, since Wuytack et al. [1996] support at most three keys, it can only be applied to the first table. Observe the use of the linked list primitive in the selected implementations for both of the last tables. This is due to the significant number of iteration operations.

9. CONCLUSIONS

Generally, major bottlenecks in embedded implementations of telecom network applications are memory size and power. Indeed, a large part of the chip area is due to memory units storing the dynamic data sets of the application, and

Table XII. ARQ Exploration and Comparison with the Existing Method

Table	Best Selected Table Implementation	Average Memory Size (Kb)	Average Memory Accesses per sec	Average Memory Power (mW)
Table I	PA(8)	18.6	$0.11 * 10^6$	0.08
	PA(7)-AR(1)	21.0	$0.17 * 10^6$	0.13
	[Wuytack et al. 1996] PA(8)	18.6	$0.11 * 10^6$	0.08
Table II	LL(16)	11.14	$16.04 * 10^6$	9.70
	LL(5)-LL(11)	10.58	$15.49 * 10^6$	9.21
	PA(1)-PA(4)-LL(11)	10.59	$15.54 * 10^6$	9.24
	PA(1)-PA(1)-PA(3)-LL(11)	10.61	$15.68 * 10^6$	9.33
	[Wuytack et al. 1996] –	–	–	–
Table III	LL(16)	106.5	$14.9 * 10^6$	34.3
	PA(8)-LL(8)	106.5	$0.96 * 10^6$	2.2
	PA(8)-PA(1)-LL(7)	106.4	$0.2027 * 10^6$	0.468
	PA(8)-PA(1)-PA(2)-LL(5)	106.7	$0.2013 * 10^6$	0.465
	PA(8)-PA(1)-PA(2)-PA(1)-LL(4)	107.2	$0.2013 * 10^6$	0.467
	PA(8)-PA(1)-PA(1)-PA(1)-PA(1)-LL(4)	107.3	$0.2020 * 10^6$	0.469
	[Wuytack et al. 1996] –	–	–	–

the power consumption of the chip is heavily dominated by the huge amount of memory accesses. Hence, the storage of the dynamic data sets in the application needs to be already optimized at the system level, where the impact on area, performance, and power is the most important.

To overcome this bottleneck, we have proposed in this article 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, commonly encountered in telecom network, database, and multimedia applications. Our method fits in the context of embedded system synthesis for such applications and enables us to further raise the abstraction level of the initial system specification, where dynamic data sets can be specified without low-level details. The implementation of the services related to memory (e.g., allocation, memory overflow detection, and garbage collection) and set management (e.g., insert, locate, remove data) of these dynamic data sets, and the generation of a customized distributed memory architecture, to meet the required performance, is not in the scope of our method. They are part of subsequent synthesis steps of our Matisse system synthesis approach, wherein our method fits.

In this article, our method is driven by a cost function that estimates at the system level the average memory power, although it can also be driven by any other cost functions such as memory size and performance. Our method is heuristic, but in practice it always reaches the optimal implementation. Compared with existing methods for large dynamic data sets, it can save up to 90% of the average memory power, while still saving up to 80% of the average memory size. As required by system synthesis approaches based on fast and stepwise explorations, our method is very fast. In the future, we intend to extend our method to dynamic data sets encountered in Internet multimedia applications that must run on various platforms (e.g., PCs, laptops, and GSMs).

APPENDIX A: MEMORY ACCESS ANALYSIS

This appendix is organized as follows. First, the number of memory accesses is derived for each operation in each possible one-layer implementation. See Appendix A.1. Then, how to systematically derive the number of memory accesses for each operation is illustrated for some possible two-layer implementations. See Appendix A.2. Finally, how to systematically derive the number of memory accesses for each operation in an r-layer implementation, assuming the ones for an (r-1)-layer implementation are known, is described in Appendix A.3.

Remember that $avg_{k1|k2} = \frac{avg_{rec}}{avg_{k2}}$.

In the following, the memory access analysis is done for the following operations: **locate, insert, erase, get_destination, is_empty_key(k), iterate(k)** (i.e. one look-up inside each record whose first bits of the key are k), with k characterized by the first bits of the super-key sk, and **complete_iterate**.

A.1. One-Layer Implementations

A.1.1. *$LL(sk)$ implementation*

locate	$\rightarrow \frac{avg_{rec}}{2}$ (1 read key + 1 read next) -1
	$= avg_{rec} - 1$
insert	$\rightarrow$ 1 write next + 1 write record
	$= 2$
erase	$\rightarrow$ **locate** accesses + (1 read next + 1 write next) to remove the element from the list
	$= avg_{rec} + 1$
get_destination	$\rightarrow$ 1 read record
is_empty_key	$\rightarrow$ **locate** accesses for given k
	$= \frac{avg_{rec} size_k}{size_{sk}} - 1$
iterate	$\rightarrow avg_{rec}$ (read key + read next) + $\frac{avg_{rec}}{avg_k}$ read record
	$= 2avg_{rec} + \frac{avg_{rec}}{avg_k}$
complete_iterate	$\rightarrow avg_{rec}$ (read record + read next)
	$= 2avg_{rec}$

A.1.2. *$PA(sk)$ implementation*

locate	$\rightarrow$ 1 read PA[sk]
insert	$\rightarrow$ 1 write record + 1 write PA[sk]
	$= 2$
erase	$\rightarrow$ 1 write PA[sk]=NULL
get_destination	$\rightarrow$ 1 read record
is_empty_key	$\rightarrow \frac{2^{size_{sk}-size_k}}{2}$ read PA[sk]
	$= 2^{size_{sk}-size_k-1}$
iterate	$\rightarrow 2^{size_{sk}-size_k}$ read PA[sk] + $\frac{avg_{rec}}{avg_k}$ read record
	$= 2^{size_{sk}-size_k} + \frac{avg_{rec}}{avg_k}$

complete_iterate	$\rightarrow 2^{size_{sk}}$ read PA[sk] $+ avg_{rec}$ read record $= 2^{size_{sk}} + avg_{rec}$

A.1.3. *$AR(sk)$ implementation*

locate	$\rightarrow$ 1 read AR[sk] to check whether the record is empty or not
insert	$\rightarrow$ 1 write record
erase	$\rightarrow$ 1 write AR[sk] $= 0$
get_destination	$\rightarrow$ 1 read record
is_empty_key	$\rightarrow \frac{2^{size_{sk}-size_k}}{2}$ read AR(sk) to check whether the record is empty or not $= 2^{size_{sk}-size_k-1}$
iterate	$\rightarrow 2^{size_{sk}-size_k}$ read AR[sk] $+ \frac{avg_{rec}}{avg_k}$ read record $= 2^{size_{sk}-size_k} + \frac{avg_{rec}}{avg_k}$
complete_iterate	$\rightarrow 2^{size_{sk}}$ read AR[sk] $+ avg_{rec}$ read record $= 2^{size_{sk}} + avg_{rec}$

A.2. Two-Layer Implementations

A.2.1 *$LL(VPI)PA(VCI)$ implementation*

locate	$\rightarrow$ **locate** accesses in $LL(VPI)$ + **locate** accesses in $PA(VCI)$ $= (avg_{VPI} - 1) + 1$ $= avg_{VPI}$
insert	$\rightarrow$ (1) If an $LL(VPI)$ element exists (probability $= \frac{avg_{VPI}}{2^{VPI}} = \frac{2^{\frac{8}{p}}}{2^8} = 0.08$), $=$ **locate** accesses in $LL(VPI)$ + **insert** accesses in $PA(VCI)$ $= avg_{VPI} - 1 + 2$ $= avg_{VPI} + 1$ (2) If not, $=$ Accesses to check that no $LL(VPI)$ element exists + **insert** accesses in $LL(VPI)$ + **insert** accesses in $PA(VCI)$ $= avg_{VPI}$ (read key + read next) $+ 2 + 2$ $= 2avg_{VPI} + 4$ $\rightarrow$ Total $= \frac{avg_{VPI}}{2^{VPI}}(avg_{VPI} + 1) + (1 - \frac{avg_{VPI}}{2^{VPI}})(2avg_{VPI} + 4)$
erase	$\rightarrow$ **locate** accesses in $LL(VPI)$ + **erase** in $PA(VCI)$ $= avg_{VPI} - 1 + 1$ $= avg_{VPI}$
is_empty_VPI	$\rightarrow$ **is_empty_VPI** accesses in $LL(VPI)$ $= avg_{VPI} - 1$

get_destination $\rightarrow$ 1 read record

A.2.2. *$LL(k2)LL(k1)$ implementation*

locate $\rightarrow$ **locate** accesses in $LL(k2)$ + 1 read pointer in $LL(k2)$+ **locate** accesses in $LL(k1)$
$= (avg_{k2} - 1) + 1 + (avg_{k1} - 1)$
$= avg_{k2} + avg_{k1} - 1$

insert $\rightarrow$

(1) If an $LL(k2)$ element exists (probability $= \frac{avg_{k2}}{2^{k2}} = \frac{2^{\frac{k2}{p}}}{2^{k2}}$),
$=$ **locate** accesses $LL(k2)$ + 1 read pointer in $LL(k2)$ + **insert** accesses in $LL(k1)$
$= avg_{k2} - 1 + 1 + 2$
$= avg_{k2} + 2$

(2) If not,
$=$ Accesses to check that no $LL(k2)$ element exists + **insert** accesses in $LL(k2)$ + **insert** accesses in $LL(k1)$
$= avg_{k2}$ (read key + read next) + 2 + 2
$= 2avg_{k2} + 4$

$\rightarrow$ Total $= \frac{avg_{k2}}{2^{k2}}(avg_{k2} + 2) + (1 - \frac{avg_{k2}}{2^{k2}})(2avg_{k2} + 4)$

erase $\rightarrow$ **locate** accesses in $LL(k2)$ + 1 read pointer in $LL(k2)$ + **erase** accesses in $LL(k1)$
$= avg_{k2} - 1 + 1 + avg_{k1} + 1$
$= avg_{k2} + avg_{k1} + 1$

is_empty_key $\rightarrow$

(1) If $size_k \leq size_{k2}$,
$=$ **is_empty_key** accesses in $LL(k2)$
$= avg_{k2}\frac{size_k}{size_{k2}} - 1$

(2) If $size_k > size_{k2}$,
$=$ **locate** accesses in $LL(k2)$ + 1 read pointer in $LL(k2)$ + **is_empty_key**$(k2)$ accesses in $LL(k1)$
$= (avg_{k2} - 1) + 1 + \frac{avg_{rec}}{avg_{k2}}\frac{size_k - size_{k2}}{size_{k1}} - 1$
$= avg_{k2} + \frac{avg_{rec}}{avg_{k2}}\frac{size_k - size_{k2}}{size_{k1}} - 1$

get_destination $\rightarrow$ 1 read record

iterate $\rightarrow$

(1) If $size_k < size_{k2}$,
$=$ **iterate** accesses in $LL(k2)$ + for each element with correct k, 1 read pointer + **complete_iterate** accesses in $LL(k1)$
$= 2avg_{k2}$ + for each element with correct k, $1 + 2avg_{k1|k2}$
$= 2avg_{k2} + \frac{avg_{k2}}{avg_k}(1 + 2avg_{k1|k2})$

(2) If $size_k \geq size_{k2}$,
$=$ **locate** accesses in $LL(k2)$ + 1 read pointer in $LL(k2)$ + **iterate**$(k - k2)$ accesses in $LL(k1)$

$= (avg_{k2} - 1) + 1 + 2avg_{k1|k2} + \frac{avg_{rec}}{avg_k}$
$= avg_{k2} + 2avg_{k1|k2} + avg_k$

complete_iterate → **complete_iterate** accesses in $LL(k2)$ + for each element, **complete_iterate** accesses in $LL(k1)$
$= 2avg_{k2} + avg_{k2}(2avg_{k1|k2})$
$= 2avg_{k2}(1 + avg_{k1|k2})$

A.2.3. *$LL(k2)PA(k1)$ implementation*

locate → **locate** accesses in $LL(k2)$ + **locate** accesses in $PA(k1)$
$= (avg_{k2} - 1) + 1$
$= avg_{k2}$

insert →
(1) If an $LL(k2)$ element exists (probability $= \frac{avg_{k2}}{2^{k2}} = \frac{2^{\frac{k2}{p}}}{2^{k2}}$),
$=$ **locate** accesses in $LL(k2)$ + **insert** accesses in $PA(k1)$
$= avg_{k2} - 1 + 2$
$= avg_{k2} + 1$
(2) If not,
$=$ Accesses to check that no $LL(k2)$ element exists + **insert** accesses in $LL(k2)$ + **insert** accesses in $PA(k1)$
$= avg_{k2}$ (1 read key + 1 read next) + 2 + 2
$= 2avg_{k2} + 4$
→ Total $= \frac{avg_{k2}}{2^{k2}}(avg_{k2} + 1) + (1 - \frac{avg_{k2}}{2^{k2}})(2avg_{k2} + 4)$

erase → **locate** accesses in $LL(k2)$ + **erase** acceses in $PA(k1)$
$= avg_{k2} - 1 + 1$
$= avg_{k2}$

is_empty_key →
(1) If $size_k \leq size_{k2}$,
$=$ **is_empty_key** accesses in $LL(k2)$
$= avg_{k2}\frac{size_k}{size_{k2}} - 1$
(2) If $size_k > size_{k2}$,
$=$ **locate** accesses in $LL(k2)$ + **is_empty_key**$(k2)$ accesses in $PA(k1)$
$= (avg_{k2} - 1) + 2^{size_{k1} - size_k + size_{k2} - 1}$
$= avg_{k2} - 1 + 2^{size_{sk} - size_k - 1}$

get_destination → 1 read record

iterate →
(1) If $size_k < size_{k2}$,
$=$ **iterate** accesses in $LL(k2)$ + for each element with correct k, **complete_iterate** accesses in $PA(k1)$
$= 2avg_{k2}$ + for each element with correct k, $2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$
$= 2avg_{k2} + \frac{avg_{k2}}{avg_k}2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$

(2) If $size_k \geq size_{k2}$,
$= \textbf{locate}$ accesses in $LL(k2)$ + **iterate**$(k2)$ accesses in $PA(k1)$
$= (avg_{k2} - 1) + 2^{size_{k1} - size_k + size_{k2}} + \frac{avg_{rec}}{avg_k}$
$= avg_{k2} - 1 + 2^{size_{sk} - size_k} + \frac{avg_{rec}}{avg_k}$

complete_iterate → **complete_iterate** accesses in $LL(k2)$ + for each element, **complete_iterate** accesses in $PA(k1)$
$= avg_{k2} + avg_{k2} 2^{size_{k1}} + avg_{rec}$

A.2.4. *$LL(k2)AR(k1)$ implementation*

locate → **locate** accesses in $LL(k2)$ + **locate** accesses in $AR(k1)$
$= (avg_{k2} - 1) + 1$
$= avg_{k2}$

insert →

(1) If an $LL(k2)$ element exists (probability $= \frac{avg_{k2}}{2^{k2}} = \frac{2^{\frac{k2}{p}}}{2^{k2}}$),
$=$ **locate** accesses in $LL(k2)$ + **insert** accesses in $AR(k1)$
$= avg_{k2} - 1 + 1$
$= avg_{k2}$

(2) If not
$=$ Accesses to check that no $LL(k2)$ element exists + **insert** accesses in $LL(k2)$ + **insert** accesses in $AR(k1)$
$= avg_{k2}$ (1 read key + 1 read next) + 2 + 1
$= 2avg_{k2} + 3$

→ Total $= \frac{avg_{k2}^2}{2^{k2}} + (1 - \frac{avg_{k2}}{2^{k2}})(2avg_{k2} + 3)$

erase → **locate** accesses in $LL(k2)$ + **erase** accesses in $AR(k1)$
$= avg_{k2} - 1 + 1$
$= avg_{k2}$

is_empty_key →

(1) If $size_k \leq size_{k2}$,
$=$ **is_empty_key** accesses in $LL(k2)$
$= avg_{k2} \frac{size_k}{size_{k2}} - 1$

(2) If $size_k > size_{k2}$,
$=$ **locate** accesses in $LL(k2)$ + **is_empty_key**$(k2)$ accesses in $AR(k1)$
$= (avg_{k2} - 1) + 2^{size_{k1} - size_k + size_{k2} - 1}$
$= avg_{k2} - 1 + (2^{size_{sk} - size_k - 1})$

get_destination → 1 read record

iterate →

(1) If $size_k < size_{k2}$,
$=$ **iterate** accesses in $LL(k2)$ + for each element with correct k, **complete_iterate** accesses in $AR(k1)$

$= 2avg_{k2} +$ for each element with correct k, $2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$
$= 2avg_{k2} + \frac{avg_{k2}}{avg_k} 2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$

(2) If $size_k \geq size_{k2}$,
$=$ **locate** accesses in $LL(k2)$ + **iterate**$(k2)$ accesses in $AR(k1)$
$= (avg_{k2} - 1) + 2^{size_{k1} - size_k + size_{k2}} + \frac{avg_{rec}}{avg_k}$
$= avg_{k2} - 1 + 2^{size_{sk} - size_k} + \frac{avg_{rec}}{avg_k}$

complete_iterate → **complete_iterate** accesses in $LL(k2)$ + for each element, **complete_iterate** accesses in $AR(k1)$
$= avg_{k2} + avg_{k2} 2^{size_{k1}} + avg_{rec}$

A.2.5. *PA(k2)LL(k1) implementation*

locate → **locate** accesses in $PA(k2)$ + **locate** accesses in $LL(k1)$
$= 1 + (avg_{k1} - 1)$
$= avg_{k1}$

insert →
(1) If a $PA(k2)$ element exists (probability $= \frac{avg_{k2}}{2^{k2}} = \frac{2^{\frac{k2}{p}}}{2^{k2}}$),
$=$ **locate** accesses in $PA(k2)$ + **insert** accesses in $LL(k1)$
$= 1 + 2$
$= 3$
(2) If not,
$=$ Accesses to check that no $PA(k2)$ element exists $+ 1$ write pointer in $PA(k2)$ + **insert** accesses in $LL(k1)$
$= 1 + 1 + 2$
$= 4$
→ Total $= 3\frac{avg_{k2}}{2^{k2}} + 4(1 - \frac{avg_{k2}}{2^{k2}})$
$= 4 - \frac{avg_{k2}}{2^{k2}}$

erase → **locate** accesses in $PA(k2)$ + **erase** accesses in $LL(k1)$
$= 1 + avg_{k1} + 1$
$= avg_{k1} + 2$

is_empty_key →
(1) If $size_k \leq size_{k2}$,
$=$ **is_empty_key** accesses in $PA(k2)$
$= 2^{size_{k2} - size_k - 1}$
(2) If $size_k > size_{k2}$,
$=$ **locate** accesses in $PA(k2)$ + **is_empty_key**$(k2)$ accesses in $LL(k1)$
$= 1 + \frac{avg_{rec}}{avg_{k2}} \frac{size_k - size_{k2}}{size_{k1}} - 1$
$= 1 + avg_{rec} \frac{size_k - size_{k2}}{size_{k1} avg_{k2}} - 1$

get_destination $\rightarrow$ 1 read record

iterate $\rightarrow$

(1) If $size_k < size_{k2}$,
$=$ **iterate** accesses in $PA(k2)$ + for each element with correct k, **complete_iterate** accesses in $LL(k1)$
$= 2^{size_{k2}-size_k}$ + for each element with correct k, $avg_{k1|k2} + \frac{avg_{rec}}{avg_k}$
$= 2^{size_k - size_k} + \frac{avg_{k2}}{avg_k} avg_{k1|k2} + \frac{avg_{rec}}{avg_k}$

(2) If $size_k \geq size_{k2}$,
$=$ **locate** accesses in $PA(k2)$ + **iterate**$(k2)$ accesses in $LL(k1)$
$= 1 + 2avg_{k1|k2} + \frac{avg_{rec}}{avg_k}$

complete_iterate $\rightarrow$ **complete_iterate** accesses in $PA(k2)$ + for each non null pointer, **complete_iterate** accesses in $LL(k1)$
$= 2^{size_{k2}} + \frac{avg_{rec}}{avg_{k2}}(2avg_{k1|k2}) = 2^{size_{k2}} + 2\frac{avg_{rec}}{avg_{k2}} avg_{k1|k2}$

A.2.6. *$PA(k2)PA(k1)$ implementation*

locate $\rightarrow$ **locate** accesses in $PA(k2)$ + **locate** accesses in $PA(k1)$
$= 1 + 1$
$= 2$

insert $\rightarrow$

(1) If a $PA(k2)$ element exists (probability $= \frac{avg_{k2}}{2^{k2}} = \frac{2^{\frac{k2}{p}}}{2^{k2}}$),
$=$ **locate** accesses in $PA(k2)$ + **insert** accesses in $PA(k1)$
$= 1 + 2$
$= 3$

(2) If not,
$=$ Accesses to check that no $PA(k2)$ element exists + 1 write pointer in $PA(k2)$ + **insert** accesses in $PA(k1)$
$= 1 + 1 + 2$
$= 4$

$\rightarrow$ Total $= 3\frac{avg_{k2}}{2^{k2}} + 4(1 - \frac{avg_{k2}}{2^{k2}})$
$= 4 - \frac{avg_{k2}}{2^{k2}}$

erase $\rightarrow$ **locate** accesses in $PA(k2)$ + **erase** accesses in $PA(k1)$
$= 1 + 1$
$= 2$

is_empty_key $\rightarrow$

(1) If $size_k \leq size_{k2}$,
$=$ **is_empty_key** accesses in $PA(k2)$
$= 2^{size_{k2}-size_k-1}$

(2) If $size_k > size_{k2}$,
$=$ **locate** accesses in $PA(k2)$ + **is_empty_key**$(k2)$

accesses in $PA(k1)$
$= 1 + 2^{size_{k1} - size_k + size_{k2} - 1} = 1 + 2^{size_{sk} - size_k - 1}$

get_destination → 1 read record

iterate →

(1) If $size_k < size_{k2}$,
= **iterate** accesses in $PA(k2)$ + for each element with correct k, **complete_iterate** accesses in $PA(k1)$
$= 2^{size_{k2} - size_k}$ + for each element with correct k, $2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$
$= 2^{size_{k2} - size_k} + \frac{avg_{k2}}{avg_k} 2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$

(2) If $size_k \geq size_{k2}$,
= **locate** accesses in $PA(k2)$ + **iterate**$(k2)$ accesses in $PA(k1)$
$= 1 + 2^{size_{k1} - size_k + size_{k2}} + \frac{avg_{rec}}{avg_k}$
$= 1 + 2^{size_{sk} - size_k} + \frac{avg_{rec}}{avg_k}$

complete_iterate → **complete_iterate** accesses in $PA(k2)$ + for each non null pointer, **complete_iterate** accesses in $PA(k1)$
$= 2^{size_{k2}} + \frac{avg_{rec}}{avg_{k2}} * 2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$

A.2.7. *$PA(k2)AR(k1)$ implementation*

locate → **locate** accesses in $PA(k2)$ + **locate** accesses in $AR(k1)$
$= 1 + 1$
$= 2$

insert →

(1) If a $PA(k2)$ element exists (probability $= \frac{avg_{k2}}{2^{k2}} = \frac{2^{\frac{k2}{p}}}{2^{k2}}$),
= **locate** accesses in $PA(k2)$ + **insert** accesses in $AR(k1)$
$= 1 + 1$
$= 2$

(2) If not,
= Accesses to check that no $PA(k2)$ element exists + 1 write pointer in $PA(k2)$ + **insert** accesses in $AR(k1)$
$= 1 + 1 + 1$
$= 3$

→ Total $= 2\frac{avg_{k2}}{2^{k2}} + 3(1 - \frac{avg_{k2}}{2^{k2}})$
$= 3 - \frac{avg_{k2}}{2^{k2}}$

erase → **locate** accesses in $PA(k2)$ + **erase** accesses in $AR(k1)$
$= 1 + 1$
$= 2$

is_empty_key →

(1) If $size_k \leq size_{k2}$,
= **is_empty_key** accesses in $PA(k2)$
$= 2^{size_{k2} - size_k - 1}$

(2) If $size_k > size_{k2}$,
= **locate** accesses in $PA(k2)$ + **is_empty_key**$(k2)$ accesses in $AR(k1)$
$= 1 + 2^{size_{k1}-size_k+size_{k2}-1} = 1 + 2^{size_{sk}-size_k-1}$

iterate →
(1) If $size_k < size_{k2}$,
= **iterate** accesses in $PA(k2)$ + for each element with correct k, **complete_iterate** accesses in $AR(k1)$
$= 2^{size_{k2}-size_k}$ + for each element with correct k, $2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$
$= 2^{size_{k2}-size_k} + \frac{avg_{k2}}{avg_k} 2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$
(2) If $size_k \geq size_{k2}$,
= **locate** accesses in $PA(k2)$ + **iterate**$(k - k2)$ accesses in $AR(k1)$
$= 1 + 2^{size_{k1}-size_k+size_{k2}} + \frac{avg_{rec}}{avg_k}$
$= 1 + 2^{size_{sk}-size_k} + \frac{avg_{rec}}{avg_k}$

complete_iterate → **complete_iterate** accesses in $PA(k2)$ + for each non null pointer, **complete_iterate** accesses in $AR(k1)$
$= 2^{size_{k2}} + \frac{avg_{rec}}{avg_{k2}} 2^{size_{k1}} + \frac{avg_{rec}}{avg_k}$

A.3. r-Layer Implementations

In the following, we assume that $r \geq 3$, and $I(r-1)$ denotes a (r-1)-layer implementation.

A.3.1. *$LL(kr)I(r-1)$ implementation with $L(kr-1)$ being LL*

locate → **locate** accesses in $LL(kr)$ + 1 read pointer in $LL(kr)$ + **locate** accesses in $I(r-1)$
$= avg_{kr} - 1 + 1$ + **locate** accesses in $I(r-1)$
$= avg_{kr}$ + **locate** accesses in $I(r-1)$

insert →
(1) If an $LL(kr)$ element exists (probability $= \frac{avg_{kr}}{2^{kr}}$),
= **locate** accesses in $LL(kr)$ + 1 read pointer in $LL(kr)$ + **insert** accesses in $I(r-1)$
$= avg_{kr} - 1 + 1$ + **insert** accesses in $I(r-1)$
$= avg_{kr}$ + **insert** accesses in $I(r-1)$
(2) If not,
= Accesses to check that no $LL(kr)$ element exists + **insert** accesses in $LL(kr)$ + **insert** accesses in $I(r-1)$
$= avg_{kr}$ (1 read key + 1 read next) + 2 + **insert** accesses in $I(r-1)$
$= 2avg_{kr} + 2$ + **insert** accesses in $I(r-1)$
→ Total $= \frac{avg_{kr}}{2^{kr}}$ (avg_{kr} + **insert** accesses in $I(r-1)$)
$+ (1 - \frac{avg_{kr}}{2^{kr}})$ ($2avg_{kr} + 2$ + insert accesses in $I(r-1)$)

erase	$\to$ **locate** accesses in $LL(kr)$ + 1 read pointer in $LL(kr)$ + **erase** accesses in $I(r-1)$ $= avg_{kr} -1 +1+$ **erase** accesses in $I(r-1)$ $= avg_{kr} +$ **erase** accesses in $I(r-1)$
is_empty_key	$\to$ (1) If $size_k \le size_{kr}$, = **is_empty_key** accesses in $LL(kr)$ $= avg_{k2}\frac{size_k}{size_{kr}} - 1$ (2) If $size_k > size_{kr}$, = **locate** accesses in $LL(kr)$ + 1 read pointer in $LL(kr)$ + **is_empty_key**$(k-kr)$ accesses in $I(r-1)$ $= (avg_{kr} - 1) +1+$ **is_empty_key**$(k-kr)$ accesses in $I(r-1)$ $= avg_{kr} +$ **is_empty_key**$(k-kr)$ accesses in $I(r-1)$
get_destination	$\to$ 1 read record
iterate	$\to$ (1) If $size_k < size_{kr}$, = **iterate** accesses in $LL(kr)$ + for each element with correct k, 1 read pointer + **complete_iterate** accesses in $I(r-1)$ $= 2avg_{kr} + \frac{avg_{k2}}{avg_k}$ (1 + **complete_iterate** accesses in $I(r-1)$) (2) If $size_k \ge size_{kr}$, = **locate** in $LL(kr)$ + 1 read pointer in $LL(kr)$ + **iterate**$(k-kr)$ accesses in $I(r-1)$ $= (avg_{kr} - 1) +1+$ **iterate**$(k-kr)$ accesses in $I(r-1)$ $= avg_{kr} +$ **iterate**$(k-kr)$ accesses in $I(r-1)$
complete_iterate	$\to$ **complete_iterate** accesses in $LL(kr)$ + for each element, **complete_iterate** accesses in $I(r-1)$ $= 2avg_{kr} + avg_{kr}$***complete_iterate** accesses in $I(r-1)$

A.3.2. *$LL(kr)I(r-1)$ implementation with $L(kr-1)$ not being LL*

locate	$\to$ **locate** accesses in $LL(kr)$ + **locate** accesses in $I(r-1)$ $= avg_{kr} - 1 +$ **locate** accesses in $I(r-1)$
insert	$\to$ (1) If an $LL(kr)$ element exists (probability $= \frac{avg_{kr}}{2^{kr}}$), = **locate** accesses in $LL(kr)$ + **insert** accesses in $I(r-1)$ $= avg_{kr} - 1 +$ **insert** accesses in $I(r-1)$ (2) If not, = Accesses to check that no $LL(kr)$ element exists + **insert** accesses in $LL(kr)$ + **insert** accesses in $I(r-1)$ $= avg_{kr}$ (1 read key + 1 read next) + 2 + **insert** accesses in $I(r-1)$ $= 2avg_{kr} + 2 +$ **insert** accesses in $I(r-1)$

$\rightarrow$ Total $= \frac{avg_{kr}}{2^{kr}}(avg_{kr} - 1 +$ **insert** accesses in $I(r-1)) +$
$(1 - \frac{avg_{kr}}{2^{kr}})(2avg_{kr} + 2 +$ **insert** accesses in $I(r-1))$

erase $\rightarrow$ **locate** accesses in $LL(kr)$ + **erase** accesses in $I(r-1)$
$= avg_{kr} - 1 +$ **erase** accesses in $I(r-1)$

is_empty_key $\rightarrow$

(1) If $size_k \leq size_{kr}$,
$=$ **is_empty_key** accesses in $LL(kr)$
$= avg_{kr}\frac{size_k}{size_{kr}} - 1$

(2) If $size_k > size_{kr}$,
$=$ **locate** accesses in $LL(kr)$ + **is_empty_key**$(k - kr)$ accesses in $I(r-1)$
$= (avg_{kr} - 1) +$ **is_empty_key**$((k - kr))$ accesses in $I(r-1)$

get_destination $\rightarrow$ 1 read record

iterate $\rightarrow$

(1) If $size_k < size_{kr}$,
$=$ **iterate** accesses in $LL(kr)$ + for each element with correct k, **complete_iterate** accesses in $I(r-1)$
$= 2avg_{kr} + \frac{avg_{k2}}{avg_k} *$ **complete_iterate** accesses in $I(r-1)$

(2) If $size_k \geq size_{kr}$,
$=$ **locate** accesses in $LL(kr)$ + **iterate**$(k - kr)$ in $I(r-1)$
$= (avg_{kr} - 1) +$**iterate**$(k - kr)$ accesses in $I(r-1)$

complete_iterate $\rightarrow$ **complete_iterate** accesses in $LL(kr)$ + for each element, **complete_iterate** accesses in $I(r-1)$
$= 2avg_{kr} + avg_{kr} *$**complete_iterate** accesses in $I(r-1)$

A.3.3. *$PA(kr)I(r-1)$ implementation*

locate $\rightarrow$ **locate** accesses in $PA(kr)$ + **locate** accesses in $I(r-1)$
$= 1 +$ **locate** accesses in $I(r-1)$

insert $\rightarrow$

(1) If a $PA(kr)$ element exists (probability $= \frac{avg_{kr}}{2^{kr}}$),
$=$ **locate** accesses in $PA(kr)$ + **insert** accesses in $I(r-1)$
$= 1 +$ **insert** accesses in $I(r-1)$

(2) If not,
$=$ Accesses to check that no $PA(kr)$ element exists + 1 write pointer in $PA(kr)$ + **insert** accesses in $I(r-1)$
$= 1 + 1 +$ **insert** accesses in $I(r-1)$
$= 2 +$ **insert** accesses in $I(r-1)$

$\rightarrow$ Total $= \frac{avg_{kr}}{2^{kr}}(1 +$ **insert** accesses in $I(r-1)) + (1 - \frac{avg_{kr}}{2^{kr}})$
$(2 +$ **insert** accesses in $I(r-1))$
$= 1 +$ **insert** accesses in $I(r-1) + (1 - \frac{avg_{kr}}{2^{kr}})$
$= 2 +$ **insert** accesses in $I(r-1) - \frac{avg_{kr}}{2^{kr}}$

erase $\rightarrow$ **locate** accesses in $PA(kr)$ + **erase** accesses in $I(r-1)$
$= 1 +$ **erase** accesses in $I(r-1)$

is_empty_key $\rightarrow$
(1) If $size_k \leq size_{kr}$,
$=$ **is_empty_key** accesses in $PA(kr)$
$= 2^{size_{kr} - size_k - 1}$
(2) If $size_k > size_{kr}$,
$=$ **locate** accesses in $PA(kr)$ + **is_empty_key**$(k - kr)$ accesses in $I(r-1)$
$= 1 +$ **is_empty_key**$(k - kr)$ accesses in $I(r-1)$

get_destination $\rightarrow$ 1 read record

iterate $\rightarrow$
(1) If $size_k < size_{kr}$,
$=$ **iterate** accesses in $PA(kr)$ + for each element with correct k, **complete_iterate** accesses in $I(r-1)$
$= 2^{size_{kr} - size_k} + \frac{avg_{kr}}{avg_k} *$ **complete_iterate** accesses in $I(r-1)$
(2) If $size_k \geq size_{kr}$,
$=$ **locate** accesses in $PA(kr)$ + **iterate**$(k - kr)$ accesses in $I(r-1)$
$= 1 +$ **iterate**$(k - kr)$ accesses in $I(r-1)$

complete_iterate $\rightarrow$ **complete_iterate** accesses in $PA(kr)$ + for each element, **complete_iterate** accesses in $I(r-1)$
$= 2^{size_{kr}} + avg_{kr} *$**complete_iterate** accesses in $I(r-1)$

ACKNOWLEDGMENTS

The authors would like to acknowledge the contributions of all their colleagues involved in the project from Lucent Technologies, Hyperstone electronics, IMEC, National Technical University of Athens and Ellemedia Technologies.

REFERENCES

AHO, A., HOPCROFT, J., AND ULLMAN, J. 1983. *Data Structures and Algorithms*, Addison-Wesley, Reading, MA.

BOUDEC, J. L. 1992. The asynchronous transfer mode: A tutorial. *Comput. Networks and ISDN Systems 24*, 279–309.

CATTHOOR, F., FRANSSEN, F., WUYTACK, S., NACHTERGAELE, L., AND MAN, H. D. 1994. Global communication and memory optimizing transformations for low power signal processing systems. In *VLSI Signal Processing*, vol. 7, IEEE Press, New York, 178–187.

CATTHOOR, F., WUYTACK, S., GREEF, E. D., BALASA, F., NACHTERGAELE, L., AND VANDECAPPELLE, A. 1998. *Custom Memory Management Methodology—Exploration of Memory Organisation for Embedded Multimedia System Design*, Kluwer, Boston.

CLARK, D., JACOBSON, V., ROMKEY, J., AND SALWEN, H. 1989. An analysis of TCP processing overhead. *IEEE Communications Magazine* (June 1989), 23–29.

DA SILVA JR., J., YKMAN-COUVREUR, C., MIRANDA, M., CROES, K., WUYTACK, S., DE JONG, G., CATTHOOR, F., VERKEST, D., SIX, P., AND MAN, H. D. 1998. Efficient system exploration and synthesis of applications with dynamic data storage and intensive data transfer. In *Proceedings of the Design Automation Conference*.

Ellervee, P., Miranda, M., Catthoor, F., and Hemani, A. 1999. Exploiting data transfer locality in memory mapping. In *Proceedings of the EUROMICRO*, 14–21.

Heddes, M. 1995. A hardware/software codesign strategy for the implementation of high-speed protocols. Ph.D. thesis, Technische Universiteit Eindhoven.

Hemani, A., Lazraq, T., Postula, A., Svantesson, B., and Tenhunen, H. 1995. Design of operation and maintenance part of the ATM protocol. *Journal on Communications, Special Issue on ATM networks*. Hungarian Scientific Society for Telecommunications.

Horn, W. 1998. Modelling of an ATM multiplexer in a network terminal for a mixed hardware/firmware implementation. M.S. thesis, ESDLab/KTH, Royal Institute of Technology, Kista, Sweden.

Landman, P. and Rabaey, J. 1994. Black-box capacitance models for architectural power analysis. In *Proceedings of the International Workshop on Low Power Design*, 165–170.

MathWorks. http://www.mathworks.com.

Meleis, H. and Serpanos, D. 1992. Designing communication subsystems for high-speed networks. *IEEE Network* (July 1992), 40–46.

Meng, T., Gordon, B., Tsern, E., and Hung, A. 1995. Portable video-on-demand in wireless communication. *IEEE Proceedings, Special Issue on Low Power Electronics 83*, 4, 659–680.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. 1997. Intelligent RAM (IRAM): Chips that remember and compute. In *Proceedings of the International Conference on Solid-State Circuits*, 224–225.

Schuler, C. and Mateescu, M. 1999. Performance evaluation of ARQ protocols for realtime services in IEEE 802.11 and wireless ATM. In *Proceedings of the ACTS Mobile Communications Summit*. Sorrento, Italy.

STMicroelectronics. http://us.st.com/stonline.

Therasse, Y., Petit, G., and Delvaux, M. 1993. VLSI architecture of a SDMS/ATM router. *Ann. Telecommunications 48*, 3–4.

Tiwari, V., Malik, S., Wolfe, A., and Lee, M. 1996. Instruction-level power analysis and optimization of software. *J. VLSI Signal Process. 13*, 223–238.

Verkest, D., da Silva Jr., J., Ykman, C., Croes, K., Miranda, M., Wuytack, S., de Jong, G., Catthoor, F., and Man, H. D. 1999. Matisse: A system-on-chip design methodology emphasizing dynamic memory management. *J. VLSI Signal Process. 21*, 3, 277–291.

Watson, R. and Mamrak, S. 1987. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Trans. Comput. Syst. 5*, 2, 97–120.

Wuytack, S., Catthoor, F., Franssen, F., Nachtergaele, L., and Man, H. D. 1994. Global communication and memory optimizing transformations for low power systems. In *Proceedings of the International Workshop on Low Power Design*, 203–208.

Wuytack, S., Catthoor, F., and Man, H. D. 1996. Transforming set data types to power optimal data structures. *IEEE Trans. Computer-Aided Design 15*, 6, 619–628.

Wuytack, S., da Silva Jr., J., Catthoor, F., de Jong, G., and Ykman, C. 1999. Memory management for embedded network applications. *IEEE Trans. Computer-Aided Design 18*, 5, 533–544.

Ykman-Couvreur, C., Lambrecht, J., van der Togt, A., and Catthoor. 2002. Multi-objective abstract data type refinement for mapping tables in telecom network applications. In *ACM SIGPLAN Workshop on Memory System Performance*. Berlin, Germany.

Ykman-Couvreur, C., Lambrecht, J., Verkest, D., Svantesson, B., Kumar, S., Hemani, A., and Wolf, F. 1999. System exploration and synthesis from SDL of telecom network components. In *Proceedings of the EMMSEC*, 792–798.

Received January 2002; revised June 2002; accepted July 2002