Power aware data type refinement for the HIPERLAN/2

GREGORY DIMITROULAKOS[1], ATHANASIOS MILIDONIS[2], MICHALIS D. GALANIS[3], CH. YKMAN-COUVREUR,[4] ATHANASSIOS KAKAROUNTAS[5], FRANCKY CATTHOOR[6], COSTAS E. GOUTIS[7]
VLSI Design Laboratory, Electrical & Computer Engineering Department
University of Patras, Rio Campus, GREECE

Interuniversity Micro Electronics Center (IMEC), Kapeldreef 75, B3001 Belguim

Abstract: - This paper considers a domain specific methodology that has been employed to derive a cost optimized on-chip memory architecture for network protocols such as the Data Link Control layer of the HIPERLAN/2 protocol. The performed design flow was based on a well established methodology script which is appropriate for network protocol applications. The methodology consists of a sequence of steps that take as input the initial code specification. Initially the application's data types are identified automatically and then the crucial ones in terms of power are optimized. Finally, for the optimized code specification the methodology exctract as output an optimized on-chip memory architecture. During the optimization process, the time constraints are also taken into account as they are crucial in wireless network protocol applications. As the experimental results show, the application of the methodology on the HIPERLAN/2 application reduces the power consumption up to 37% comparing to that imposed by the initial code specification.

Key-Words: - Wireless Protocols, HIPERLAN/2, Dynamic Memory Management, Memory Architecture.

1 Introduction

The rapid growth in the complexity and diversity of networks and telecom systems, along with the ever increasing user demand in networking services, has imposed the need for efficient protocol processing [1]. One of the fundamental functionalities, and also most demanding parts in terms of real-time requirements and power consumption in high-speed network processors, is data queuing. It includes real-time memory (de)allocation, buffering, retrieving and forwarding of the incoming data packets. Its implementation must be highly optimized for combining high execution speed, low power, and high memory bandwidth.

Two major factors influence the implementation of a data queuing system: the size of the memory and the number of memory accesses. In data queuing systems where data reuse opportunities are absent, the reduction of the memory size is a major way to reduce the energy dissipated by the memories. This can be achieved by optimizing the memory architecture imposed by the initial system specification to an architecture were the majority of the memory accesses is performed on smaller memories. For statically allocated arrays several approaches exist to find a distributed memory organisation (see related work in section 2) but for dynamic data types this is much more difficult and it requires an adapted approach, as we will discuss in section 2 and 4.

In this paper a methodology for optimizing the initial code specification in terms of power and deriving an optimized on chip memory architecture for wireless network applications is presented. The methodology specifically targets on dynamic applications with data reuse absence. By applying static and dynamic analysis on the initial specification code the crucial data types in terms of memory access and storage are identified automatically. Then proper data structures are selected and an efficient memory architecture is derived meeting the time constraints and reducing the energy of the system's memories. Experimental results show a reduction of the memory energy consumption up to 37% compared with the energy consumption of the memory architecture imposed by the initial specification code.

The remainder of this paper is organized as follows: In section 2 the related work is presented. Section 3 gives an overview of the HIPERLAN/2 application. In section 4 the proposed methodology steps are described while conclusions are drawn in section 5.

2 Related Work

Several approaches [2]-[5] exist for defining a distributed memory organization dealing with statically allocated data, like arrays. But these are not directly applicable in dynamic applications because prior to the distributed memory organization issues, the dynamically allocated data should be transformed at the algorithmic/system level. In our approach we have integrated these algorithmic data type transformations and the memory assignment to obtain better results.

In [1] a systematic Dynamic Memory Management (DMM) methodology has been proposed to implement data queuing systems by handling efficiently all dynamic memory (de)allocations in terms of memory access and time. The first step of the DMM is the Abstract Data Type (ADT) refinement. The goal of the ADT is to select an optimal data structure realization for the given application, since as referred in [1] there is an important difference in power consumption and performance between different realizations of the same data type. This methodology does not consider the cases where data reuse opportunities are absent and also does not consider the extraction of an optimized on chip memory architecture.

In [6] the MATISSE framework was proposed to perform analysis on the system's code specification in order to collect the information required for the optimization and verification stages. It is a design environment intended for developing systems characterized by a tight interaction between control and data flow behavior, intensive data storage and transfer, and stringent real-time requirements.

In [7] a case study is presented, for a high-speed Queue Manager for ATM systems. The virtual memory is partitioned into blocks to store data packets and all dynamic data structures and queues. The manager enables high-speed data transfer to and from system memory. An approach for developing such a manager is described and also, an architecture is provided with implementations in hardware and software for embedded systems. The implementation of the selected memory architecture by off-chip DRAMs at [8]. However, this work is a case study and it cannot be applied in general for network applications.

In [9], the queuing module of the PRO system is used to demonstrate the effectiveness of a new system level exploration method for optimizing the memory performance in dynamic memory management. Compared with the conventional memory management technique for embedded systems, this exploration method can reduce the bank conflicts, which allows improving worst-case memory performance of data queuing operations. In this work, no on-chip memory architecture extraction is included.

3 Demonstrator

According to the HL/2 functionality, the data queuing system of the Data Link Control (DLC) sublayer is among the most crucial part and so it is used as a test vehicle. Data packets belonging to various network connections and with specific priorities must be routed to appropriate destinations through the data queuing system.

Two kinds of terminals exist: the Access Point (AP) and the Mobile Terminal (MT). The AP manages the network resources and also handles the data packet routing. When an MT is connected at the radio cell the relevant AP allocates memory space for seven queues. These queues correspond to the seven different priorities, which the protocol supports. The MT communicates with other MTs by sending data packets via one or more APs. Also in an MT there are seven queues, which correspond to the seven priorities. The data packets are buffered according to two address fields to separate queues. The first one is called MAC_ID address and identifies the network connection while the second one called DLC_ID identifies the priority associated to each packet. Both comprise the DLCC_ID address which uniquely corresponds to a specific queue.

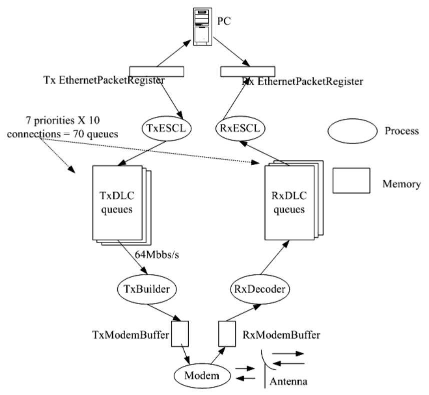


Fig. 1. DLC part's process diagram

Fig. 1 illustrates the process diagram of the DLC sublayer of the HL/2. Considering the sender part, the PC sends Ethernet packets to the HL/2 protocol stack and the received packets are stored in the Ethernet packet buffer as it is shown in the left upper side of Fig. 2. Afterwards, the process TxESCL breaks the Ethernet packet into a number of 48 byte pieces and attaches a 4 byte serial number in front of each piece. These 52 byte data chunks called LCHs are then forward to the DLC queues. Next, the process Tx Builder is activated to retrieve the data stored in the queues, to assemble them into HL/2 packets, and send the packets to the modem. A similar procedure stands for the receiver parts, as illustrated in the right side of Fig. 2.

Studying the initial source code it was deduced that a straightforward implementation of the DLC queues requires a static array of 6Mbs for each of them. Thus the goal is to optimize these queues and to derive an efficient implementation in terms of power consumption.

4 Proposed Methodology Flow

The first step of our methodology considers analysis and preprocessing where the critical data types in terms of power are identified automatically [10]. Afterwards, the MATISSE framework is used for refining the implementation of the identified crucial data types. And the last step considers the on-chip memory architecture derivation step. In the following these steps are described in detail.

4.1 Analysis-Preprocessing

Profile information needs to be extracted before memory management exploration on the Hiperlan/2 application. Initially, the motivation of the analysis is to get a good knowledge on which of the application's data types are most frequently accessed and which require a lot of memory space to be stored. By applying memory management techniques only on those data types there will be a great impact on minimizing the application's total power consumption and execution time. Also, the design time will be significantly minimized since many solution paths that don't lead to attractive results from memory management perspective will be avoided. For this task, two kinds of analysis are performed, namely the static and dynamic ones. Next, the analysis step concentrates on deciding which will be the best suited and refined data types based on the application's behaviour concerning the number of memory accesses, that will be stored in the system's memory architecture.

During static analysis all data structures and their static size are identified. The static size of a structure is considered as the total memory space that is needed for storing an instance of that structure. The important data structure for the analysis from memory management perspective is the array and its static size is the sum of all its elements' memory storing space.

Continuing, dynamic analysis needs to be performed to find out which of the data types are most frequently accessed. This is accomplished by placing an increment instruction of a variable underneath each instruction of the initial application's source code at which an array is accessed. For each array, a unique variable is declared for counting its accesses. Additional instructions are placed inside the initial source code for printing the total number of accesses for each array before the application's execution ending. Next, the code is executed and all information about the accesses of each array is extracted.

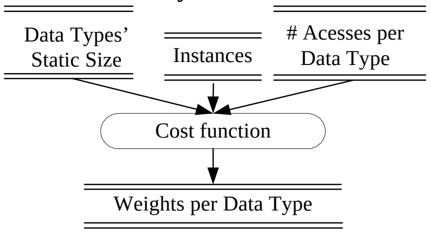


Fig. 2. Weight Assignment Procedure

All the above information is used for deciding which data types are the crucial ones from memory management perspective. For that reason, a cost function is employed that uses the extracted information from static and dynamic analysis and will assign to each data type a weight. In that way, a measure of how crucial each array is is obtained. Fig.2 shows the assignment of weights on the data types.

In order to decide that a data type is crucial for data management perspective, its static size and the number of accesses during the execution time is taken into account. This information is important since the data management optimization's phase that will take place afterwards in order to contribute significantly to the system's total power consumption needs to focus on arrays that require a large amount of memory storage and they are accessed very frequently. Eq. (1) gives the simple cost function that is used for this task.

$$Weigth = number \quad of \quad accesses \ * \ static \quad size \quad (1)$$

Table 1. Analysis results for Hiperlan/2

Data types	# of Accesses	Storage Size (bits)	Weight
EC_Tx_unack	7,454,748	1,216,833	9,07E+12
EC_Rx_unack	7,454,744	1,216,833	9,07E+12
ModemBuffer	1,500	1,216,833	1,83E+09
EthernetPacketRegister	1,519	1,155,991	1,76E+09
ConnectionTable	1,157	170,000	1,97E+08

After the weight assignment to each instance of the initial code's data types, the threshold should be determined beyond which each array instance is considered as crucial. Applying the above to the Hiperlan/2 source code [11], static and dynamic analysis information is extracted for each array. The profiling was performed for 500 HL/2 packets, which in real time is 1 sec. Table 1 shows the number of accesses for each array during execution time and the number of bits required for their memory storage. Continuing, during the data type assignment step a weight is assigned to each array according to a weight function. The results are shown in Table 1.

As can be seen, the weights of *EC_Tx_unack* and *EC_Rx_unack* arrays are a thousand times larger than those of the next crucial data type *Modem Buffer*. For that reason, the threshold that will distinguish the crucial data types from the non-crucial ones is placed under the first two data types and all memory management techniques will be applied only to them. It must be noticed that the initial source code in C++ of the Hiperlan/2 application contains 70000 lines. By focusing only on the code, which handles the data types *EC_Tx_unack* and *EC_Rx_unack* the corresponding number of lines are now 1930. By this it is implied that the complexity on deciding the memory management solution paths, on applying transformations is seriously decreased. For that reason, the debugging and verification time of the transformed code is now much less than the ones needed on a code with transformations applied on many of its data types. Thus, the total application's design time is drastically decreased.

4.2 Analysis with Matisse

The refinement of the initial code's specification was done using the MATISSE framework [6]. The MATISSE language allows the designer to define the application's dynamic data types in terms of MATISSE internal Abstract Data Types. In this way, the application's behavior is explored during runtime for different realizations of its data types so as to arrive in an optimized implementation of them. Based on the optimized realizations of the application's data types a subsequent step of MATISSE gives the ability to implement an optimized memory manager for the allocation and deallocation of data.

As mentioned earlier, with this tool set the potential dynamic data type refinement solutions are automatically integrated and tested to assure the validity of a choice in the implementation of the system. In the following, the analysis results from measurements taken with the MATISSE framework will be illustrated. These results are used to identify the optimized realization of the application's data types in terms of power and performance.

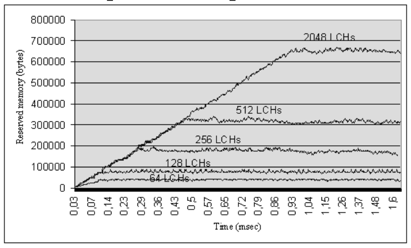


Fig. 3. Reserved memory over time for different queue sizes

In the first measurement, the reserved memory over time during the terminal's operation was monitored. The experiments refer to scenarios that correspond to different queue sizes. The experimental results in terms of the reserved memory are illustrated in Fig. 3 from which it can be deduced that the system operates for every scenario in two different modes. The terminal's queues fill quickly with packets reaching a situation where the reserved memory space ripples near its maximum value, which is the queue size.

The rate of read and write accesses is given in Fig. 4 for the case where the terminal's memory size is 2048 LCHs. It is clear that during the transient state, where the queues are filled with data, the write rate is twice the read rate, while in the steady state these rates are equal. Based on the above experiments, the following results were derived: a) concerning the reserved memory over time, the system has two modes of operation namely the transient and steady state. The time where the system is in transient state is negligible comparing to the time it is in the steady state, b) regarding the reserved memory, the system has no dynamic behavior since it constantly uses the full amount of the terminal's queue and c) no data reuse exists because firstly the read and write rates are equal in the steady state and secondly according to the system's specifications the Ethernet data are written once in the queue and read once, when the HL/2 packets construction takes place. The absence of data reuse makes the use of a memory hierarchy inappropriate.

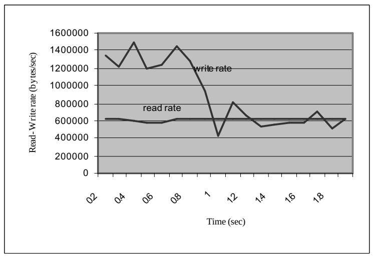


Fig. 4. Rate of read and write accesses over time

Since the queue of the terminal is always full, and according to the methodology presented in [1] the only appropriate data type is the static array. The implementation of the application's data types with a link list has no advantage in this case. This is because the system has no dynamic behavior from which it could exist potential advantages by the use of a link list based data structure. However, further exploration is required to determine the size of the array taking into consideration the system's operation and requirements.

Thus, the following exploration addresses the determination of the static array size for allocating the incoming packets along with memory architecture. For this reason a new experiment was performed for exploring the system's performance based on different scenarios concerning the queue size. The experiment outputs the total number of packets serviced by the terminal for a fixed amount of time for six different scenarios concerning the queue size. The experimental results are presented in Fig. 5. It is deduced, that the system's throughput remains unaffected from the queue size. Therefore, regarding the queue size it is proposed that the memory size of the queues must be as small as possible to achieve reduced cost per access. Hence the best size in terms of power is 64 LCHs.

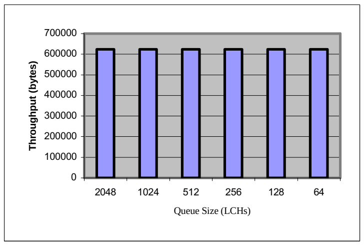


Fig. 5. System's throughput for different queue sizes

4.3 Derived Memory Architecture

Based on the previous analysis results, it is derived that the time constraints can be met by using smaller queues. Due to the fact that retransmissions are supported by the HL/2 64 LCHs is a quite small memory. According to the standards specification the TxDLC queue unit contains at least 1024 LCHs for retransmission purposes. However, retransmission seldom occurs. Therefore, what is required is to exploit the opportunity of building a system with the smallest possible memory size satisfying the constraints derived by retransmission. For this reason, two scenarios were explored.

In scenario I, a pool of 1024 LCHs size for each queue was assumed. The pool is accessed for every incoming Ethernet packet and every out coming HL/2 packet meeting the retransmission constraints. If more space than the minimum set of 1024 LCHs is required, which is defined by the standard, then it is essential that the maximally allocated data pool size is adopted. In scenario II, two pools are assumed for each queue. The sizes of these pools are 64 and 1024 LCHs. The pool of 1024 LCHs holds the data required for retransmission, while the memory of 64 LCHs is used during the system's normal operation.

In particular, concerning scenario II, the incoming Ethernet packets are stored in both pools but only the pool of 64 LCHs is read during the HL/2 packet construction. Since retransmission rarely occurs, the majority of read access is performed in a small pool. However, the number of write accesses in the second scenario is doubled because the Ethernet-packet data are stored in both pools. Fig.6 illustrates the two scenarios together with the original along with the accesses for each pool.

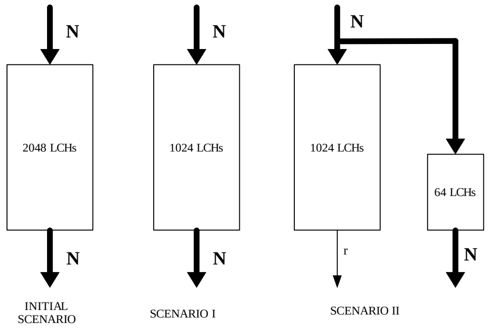


Fig. 6. Data pool organisation scenarios

We should now also define the distributed memory organisation. In this case, we will assume that each pool is assigned to a separate memory, though this is not necessarily the case. For power reasons, this is however usually (not always) the best option, at least if the memory communication overhead is negligible. But it comes at the price of a larger memory size so a trade-off is present. In the first scenario one memory is used to write and retrieve data from the queues. As a consequence one memory bank is used with size 1024 LCHs instead of 2048 LCHs that was the initial size; hence a reduction of size is accomplished by 50%. In this case if N is the number of bytes inserted in the queues in a fixed time period T then the number of times memory is accessed during this period is $2*N$.

In the second scenario, two memory banks are employed. The first is a 1024 LCHs off-chip memory and the second is a 64 LCHs on-chip memory. The off-chip memory exists only to offer retransmission services when requested. This means that data are stored in the big memory and they are retrieved only if there is a need for retransmission.

The on-chip memory is now used for the normal system operation (no retransmissions), which is the most probable. Therefore if N is the number of bytes inserted in the queues in a fixed time period T then the number of times that the off-chip memory is accessed is now $N+r$ (r: number of accesses for retransmission) instead of $2*N$ in the scenario I, while the on-chip memory is accessed $2*N$ times. From Fig. 6 it is deduced that a 47% reduction in the size of memory is achieved but by introducing $N+r$ additional memory accesses.

The two different scenarios will be compared and the most efficient will be used in the system implementation. Assuming that w_1 and w_2 are the energy per access for the off-chip and the on-chip memory, respectively the energy consumption is:

$$W_1=2\cdot N\cdot w_1 \quad (1)$$

for the first scenario and

$$W_2=N\cdot w_1+r\cdot w_1+2\cdot N\cdot w_2 \quad (2)$$

for the second scenario. Comparing the energy consumptions we have:

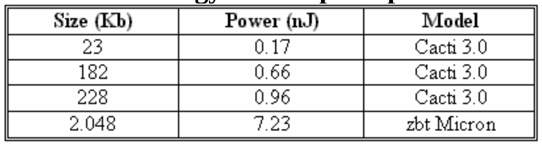
$$W_1-W_2=(N-r)\cdot w_1-2\cdot N\cdot w_2 \quad (3)$$

In the typical case $N>>r$, hence

$$W_1-W_2=N\cdot(w_1-2\cdot w_2) \quad (4)$$

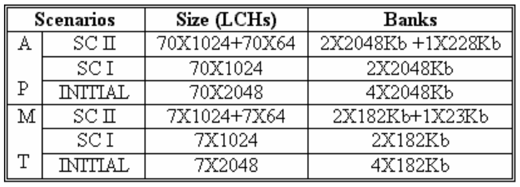
where w_1, w_2 corresponds to the power consumption of the 1024 and 64 LCHs memories respectively. In Table 2 the size and power consumption per access is shown for the memory banks employed to implement the DLC queues. The power estimations for the on-chip srams were based on the Cacti 3.0 power model while for the off-chip SRAMs on the power model for ZBT SRAMS provided by MICRON.

Table 2. Energy consumption per access



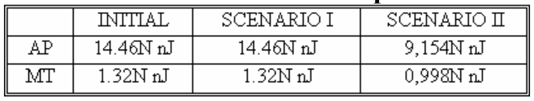
Size (Kb)	Power (nJ)	Model
23	0.17	Cacti 3.0
182	0.66	Cacti 3.0
228	0.96	Cacti 3.0
2.048	7.23	zbt Micron

Table 3. Memory configuration for different scenarios



Scenarios		Size (LCHs)	Banks
A	SC II	70X1024+70X64	2X2048Kb +1X228Kb
	SC I	70X1024	2X2048Kb
P	INITIAL	70X2048	4X2048Kb
M	SC II	7X1024+7X64	2X182Kb+1X23Kb
	SC I	7X1024	2X182Kb
T	INITIAL	7X2048	4X182Kb

Table 4. Power consumption



	INITIAL	SCENARIO I	SCENARIO II
AP	14.46N nJ	14.46N nJ	9,154N nJ
MT	1.32N nJ	1.32N nJ	0,998N nJ

In Table 3 the memory bank configurations for implementing the DLC queues for each scenario and network terminal are presented. According to Table 3 and Table 4 $w_1>2\cdot w_2$ holds for both network terminals hence, scenario II is the best choice compared to the scenario I because it consumes less energy with only 3% overhead in memory size. Table 4 presents the memory power consumption of the initial and optimized implementation.

According to Table 4 a reduction on power consumption of 37% and 24% was achieved between the initial and the optimized implementation for the AP and MT respectively. The calculation was based on the assumption that the required number of retransmissions is kept very low as is in most of the real cases. Further improvements in power consumption can be achieved by using FIFOs instead of RAMs as data are accessed sequentially. So in this case the address generation becomes trivial and only a 1-bit INC/DEC signal has to be generated/transmitted.

5 Conclusions

A power aware dynamic data management solution for the DLC of the HIPERLAN 2 standard has been presented. It has been proven that identifying and studying the crucial dynamic data types of the system, very efficient low-energy memory architectures can be derived while still meeting all the time and functional constraints of the system.

References:

[1] S. Wuytack et al, "Memory management for embedded network applications", IEEE Trans. On CAD, Vol. 18, Num. 5, pp. 533-544, May 1999.

[2] P.R.Panda, N.D.Dutt, A.Nicolau "Memory data organization for improved cache performance in embedded processor applications" (ISSS) , pp.90-95,Nov 1996

[3] L.Benini,A.Macii,E.Macci,M.Poncino,"Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation",IEEE Design and Test of Computers Vol.17, No.2, pp74-85, April 2000

[4] A.Vandecappelle, M.Miranda, E.Brockmeyer, F.Catthoor, D. Verkest, "Global Multimedia System Design Exploration using Accurate Memory Organization Feedback", 36th ACM/IEEE DAC, ,pp.327-332,June 1999

[5] F.Balasa, F.Catthoor, H.De Man, "Dataflow-driven Memory Allocation for Multi-dimensional Signal Processing Systems",ICCAD, pp.31-34 Nov 1994

[6] D. Verkest et al, "Matisse: A system-on-chip design methodology emphasizing dynamic memory management", Proc, of the Annual Workshop on VLSI, April 1998.

[7] D. N. Serpanos, P. Karakonstantis, "Efficient Memory Management for High-Speed ATM Systems", Design Automation for Embedded Systems, pp. 207–235, April 2001, Kluwer Academic Publishers.

[8] A. Nikologiannis and M.Katevenis, "Efficient Per-Flow Queueing in DRAM at OC-192 Line Rate using Out-of-Order Execution Techniques", Int. Conf. on Communications, pp. 2048-2052, 2001.

[9] Ch. Ykman-Couvreur et al, "System-level performance optimization of the data queueing memory management in high-speed network processors", DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

[10] A. Milidonis, G. Dimitroulakos, M.D. Galanis, G. Theodoridis, C. Goutis and F. Catthoor, "An automated C++ Code and Data Partitioning Framework for Data Management of Intensive Appilcations", 8th International Workosho[on Software and Compilers for Embedded Systems SCOPES 2004 Amsterdam.

[11] http://easy.intranet.gr/