Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Journal of Systems Architecture 53 (2007) 417–436

JOURNAL OF SYSTEMS ARCHITECTURE

www.elsevier.com/locate/sysarc

Systematic methodology for exploration of performance – Energy trade-offs in network applications using Dynamic Data Type refinement ☆

Stylianos Mamagkakis [a,d,*], Alexandros Bartzas [a], Georgios Pouiklis [a], David Atienza [b,c,1], Francky Catthoor [d,1], Dimitrios Soudris [a], Antonios Thanailakis [a]

[a] *VLSI Design and Testing Center-Democritus University, Thrace, 67100 Xanthi, Greece*
[b] *DACYA/UCM, 28040 Madrid, Spain*
[c] *LSI/EPFL 1015 Lausanne, Switzerland*
[d] *IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium*

Received 30 January 2006; received in revised form 20 October 2006; accepted 4 December 2006
Available online 7 February 2007

Abstract

Modern network applications require high performance and consume a lot of energy. Their inherent dynamic nature makes the dynamic memory subsystem a critical contributing factor to the overall energy consumption and to the execution time performance. This paper presents a novel, systematic methodology for generating performance-energy trade-offs by implementing optimal Dynamic Data Types, finely tuned and refined for network applications. Our systematic methodology is supported by a new, fully automated tool. We assess the effectiveness of the proposed approach in four representative, real-life case studies and provide significant energy savings and performance improvements compared to the original implementations.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Dynamic data type refinement; Embedded systems; Low-energy consumption

1. Introduction

In the last years, there is a trend towards networks and network applications implemented with embedded consumer devices. The complexity of modern wired and wireless networks is increasing to support a wide variety of services. Additionally, increased interaction with the environment (e.g., in wireless networks) has increased the dynamism of the data access pattern in network applications

☆ This work is partially supported by the European Union funded project AMDREL IST-2001-34379, http://vlsi.ee.duth.gr/amdrel, the Spanish Government Research Grant TIN 2005/05619 and E.C. Marie Curie Fellowship contract HPMT-CT-2000-00031. A small part of this work is published in DATE'06.

* Corresponding author.
E-mail addresses: smamagka@ee.duth.gr (S. Mamagkakis), ampartza@ee.duth.gr (A. Bartzas), gpouikli@ee.duth.gr (G. Pouiklis), datienza@dacya.ucm.es (D. Atienza), catthoor@imec.be (F. Catthoor), dsoudris@ee.duth.gr (D. Soudris), thanail@ee.duth.gr (A. Thanailakis).

[1] Also professor at the Katholieke Univ. Leuven, Belgium.

1383-7621/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2006.12.001

[13]. Such complex systems require a combination of hardware and middleware components; in order to deliver the required functionalities without increasing complexity too much and keeping a short time to market. This has led to an increased reliance on Dynamic Data Types (DDTs from now on). DDTs are structures, which allow data to be dynamically allocated and deallocated at run-time (releasing the memory they occupied back to the Operating System, when it is no longer needed) and provide an easy way for the designer to connect, access and process data [29]. They can cope, in the most efficient way, with the variations of run-time needs (e.g., network traffic, user interaction) and the massive amounts of data processed and stored.

Various factors contribute to the dynamism of network applications. Depending on the input of the application, the different functions and the number of times that they are executed differ. The size of the data storage, needed for these functions, varies as well. Therefore, dynamically adjustable storage is necessary, allowing to free memory when it is no longer needed. A static memory allocation at compile time is not efficient at all, because the worst case situation has to be assumed in the beginning and implemented for the whole execution time. Therefore, especially in embedded systems, where memory is a scarce resource, dynamic memory allocation and management is required. If not, far more memory space would be occupied than really needed at run-time. The data structures used to implement dynamic memory allocation are the Dynamic Data Types (DDTs). The most common examples of DDTs are single and double linked lists.

On the one hand, inefficient use of the DDTs results in performance losses by adding computational overhead to the internal DDT usage mechanisms. On the other hand, each access of the DDTs to the physical memory (where the data is stored) consumes energy. Energy consumption is the limiting factor in the amount of functionality that can be placed in these devices, because portable computers like PDAs and notebooks using wireless communications rely on limited battery energy for their operation. These two factors cannot be optimized with the same DDT implementation (i.e., a fast DDT is not always the most energy efficient, as is shown in the experimental results in Section 5) and the designer must be able to choose a balanced DDT implementation, in order to achieve the required performance in every case, while minimizing the energy consumption.

In the embedded systems community, designers deal with data management at different levels of abstraction, ranging from abstract data types and dynamic memory allocation down to the storage of data at the register level. In order to achieve large reductions in energy consumption, memory footprint and/or execution time, optimizations at the data structure level are needed. Apart from the memory used to store the data records, the internal mechanisms of the DDTs add memory overhead to create their own internal structures and access mechanisms. Nevertheless, they can achieve great memory footprint gains in comparison to a statically allocated, compile-time memory solution [18].

A dynamic network application consists of various functions and concurrent tasks. Each function (or task) accesses and processes its own set of data in different ways and patterns, leading to a complex overall dynamic behavior. Each set of data is assigned to a specific DDT. The final decision about the optimal combination of the different DDTs that should be implemented in the network application, is influenced of this complex algorithmic-based dynamic behavior. Therefore, no general, domain-specific, optimal solution exists but only custom, application-specific ones. Additionally, the different configurations available to networking applications add one more layer of complexity and demand further customization of the DDTs to achieve optimal results. Thus, the decision should be in accordance to both the application's algorithmic-based dynamic behavior and the dynamic behavior influenced by the network configuration. Finally, the system design restrictions also have to be met. This decision requires a very complex exploration task. A systematic, step-by-step methodology is needed to help the designer to make the right trade-off choice for each DDT in the networking application.

This exploration should not be taken lightly without the proper methodology and tool automation support. To develop and test n different DDT implementations manually, one would have to rewrite the source code of an application n times. This procedure is not applicable to realistic applications consisting of thousands of code lines. In addition, a way to profile and extract metrics for a cost function is also needed in order to choose optimal solutions. Our choice of optimal solutions is based on Pareto points, which represent an optimal implementation only if no other implementation has better results in all the metrics, which are explored. Thus, a point is said to be Pareto-optimal, if it is no longer possible to improve upon one cost factor without worsening any other [11].

In this paper, we present a systematic and automated methodology to perform DDT refinement of any given network application, with any network configuration, using Pareto-optimal trade-offs. This gives the designer a way to improve performance and energy consumption at a high abstraction design level without considering any changes in the hardware and the functionality of the application. More specifically the innovations in this paper are focused on:

- Network configuration sensitivity: Our proposed methodology provides for the first time optimizations, which are sensitive to the specific configuration of the network (and not just to the application).
- Multi-objective optimizations: Instead of focusing on a single metric, we provide for the first time optimization solutions for (up to) four different metrics, including energy consumption, memory accesses, performance and memory footprint.
- Fully automated exploration: For the first time, we produce a systematic exploration framework which is consistently automated by a set of tools.

With the use of our proposed methodology and automation support, we manage to reduce significantly the targeted metrics. Also, we manage to reduce dramatically the design time of our exploration by pruning the available solution space during one of the steps of our proposed methodology (according to the metric that we want to optimize). The remainder of the paper is organized as follows. In Section 2, we provide an overview of the related work. In Section 3, we define the basic search space of available DDT solutions. In Section 4, we present our proposed systematic Dynamic Data Type refinement methodology for network configurations. In Section 5, the case studies are introduced and the obtained simulation results are analyzed. Finally, in Section 6 we draw our conclusions.

2. Related work

Data management and data optimizations for traditional (i.e., non-dynamic) embedded applications have been extensively studied in the related literature [6,12,15,28]. The work presented in [5,24] are good overviews about the vast range of proposed techniques to improve memory footprint and decrease energy consumption in statically allocated data. Also, from the methodology viewpoint, several approaches have been proposed to tackle this issue at the different levels of abstraction (e.g., memory hierarchies), such as the Data Transfer and Storage Exploration (DTSE) methodology [7]. However, in modern dynamic applications the behavior of many algorithms is heavily determined by the input data. This often means that multiple and completely different execution paths can be followed, leading to complex, dynamic data usage according to the behavior of the users. Therefore, our approach focuses on optimizing the Dynamic Data Types for modern network application with run-time memory allocation needs, contrary to the aforementioned static (i.e., compile-time) data allocation optimizations.

In general-purpose software and algorithms design [1,3,29], primitive data structures are commonly implemented as mapping tables. They are employed to accomplish software implementations with high performance or with low memory footprint. Additionally, the Standard Template C++ Library (STL) [26] or other proposed templates [19] provide many basic algorithms and data structures needed for implementing dynamic data structures in a general context. For embedded system software, several transformations of data structures for compilers have simplified local loops in imperative programs [22]. Nevertheless, none is suitable for exploration of complex dynamic data structures used in modern wired and wireless network applications, because they handle only very simple data structures (e.g., arrays or pointer arrays), which mostly focus on performance.

More attention is paid to energy consumption and other embedded systems criteria in [9,14,17]. A fast, stepwise, cost-driven, and automated exploration and refinement is proposed in [2,17] for multimedia applications at system level, which operate on large and irregular data structures that typically exist in this application domain. We tried to use the same exploration for network applications (which is the context of this paper) but we realized that contrary to multimedia applications, we could not find a universally good DDT solution per application (regardless of the configuration of the application). Contrary to multimedia applications, which mostly have a sequential access behavior of the DDTs, in network applications the access behavior of the DDTs changes considerably (in some cases even completely) with different configuration implementations. Thus, we had to do an additional step of exploration, which was related to network-sensitive criteria in order to arrive at optimal Dynamic Data Types. Some of these criteria include the number of nodes in a network, throughput, etc. (see Section 4 for more details).

The work presented in this paper is more related to [30–32]. However, there are major differences to the work presented there. First of all, we concentrate on data structure optimizations in the context of various network applications. We extend the range of applications, without being limited to a specific hardware network router. In fact, in this paper we use a network routing application, a context switching application, a network firewall and a network scheduling application, which are independent of specific hardware implementations. Secondly, we analyze the software implementation of energy efficient data structures, as opposed to explicitly designing and using specifically configured physical memories. In our context, we assume that the embedded platform is already designed and that our Dynamic Data Types, which are tuned to the network application, are incorporated in the middleware on top of the given platform hardware. Thirdly, we extend considerably the exploration of the Dynamic Data Type design space. This extension is done by adding the factor of network configuration to explore the search space more consistently (see Section 4 for more details). Our approach studies the optimizations and possible trade-offs related to additional design metrics that are key factors in embedded systems, such as low-energy consumption and memory accesses, on top of the usual ones, namely performance and memory footprint. Finally, we support the whole methodology flow for the first time with fully automated tools, which reduce significantly the design time needed for the optimizations.

3. Dynamic data type search space

A DDT is a software abstraction by means of which we can manipulate and access data. The implementation of a DDT has two main components:

- It has storage aspects that determine how data memory is allocated and freed at run-time and how this memory is tracked.
- It involves an access component. This means that once the data is stored in the memory, it still needs to be accessed from within the algorithm. These two components have a big influence on performance.

Even though the number of DDT basic building blocks is as such limited, developers tend to write custom DDTs for each application. Therefore, the number of alternatives are limited to the developer's skill and the available time to implement each one of them. We have developed and used 10 different DDT implementations for our exploration and final refinement. We chose not to use the tree data type in our exploration, because previous experimental results [17] have proven that their poor locality provides very bad energy efficiency results. Also, we explored only data types that are allocated and deallocated dynamically at run-time (through `malloc/free` and `new/delete` operands). The more complex DDTs were created by using the basic DDTs, their variations and multi-layer combinations of them:

Basic DDTs:

- Array (AR): is a set of sequentially indexed elements of type *T*. Each element of the array is a record of the application.
- Single Linked List (SLL): is a single linked list of vectors of type *T*. Each element of the list is connected with the next element through a pointer.
- Double Linked List (DLL): is a double linked list of vectors of type *T*. Each element of the list is connected with the next and the previous element with two pointers, one pointing to the previous element and one to the next.

Variations of Basic DDTs:

- Pointer (P): in the pointer variation of each basic DDT, the record of the application is stored outside the DDT and is accessed via a pointer. This leads to a smaller DDT size but also to an extra memory access to reach the actual data. All DDTs used in our exploration comply to this variation except the simple array.
- Roving Pointer (O): The roving pointer is an auxiliary pointer helping us to access a particular element of a list with less accesses. For example, for an array if you access element $n + 1$ immediately after element n, your average access count is $1 + 1$ instead of $n/2 + 1$.

Then, these simple DDTs can be combined in multi-layered structures that offer a compromise

Table 1
The different DDT implementations used

Abbreviation	Description
SLL(AR)	"Pointer" single linked list of arrays
SLL(ARO)	"Pointer" single linked list of arrays (roving pointer)
DLL(AR)	"Pointer" double linked list of arrays
DLL(ARO)	"Pointer" double linked list of arrays (roving pointer)
SLL	"Pointer" single linked list
SLL(O)	"Pointer" single linked list (roving pointer)
DLL	"Pointer" double linked list
DLL(O)	"Pointer" double linked list (roving pointer)
AR	Simple array
AR(P)	"Pointer" array of pointers

between flexibility, memory use and access time (as can be seen in Table 1). For instance, a "pointer" linked list of 100 elements has an access count of 51, while a "pointer" single linked list of arrays (10 list elements and 10 array elements) has an access count of 3.5. The trade-off of the second DDT, which is faster, is that it requests more memory from the system and is more difficult to be maintained. Table 1 presents the DDTs, which are used in our exploration. Their corresponding average size and average number of accesses, which are needed to access a random element of the DDT (i.e., random access count) or to access the next element in the DDT (i.e., sequential access count), are presented in Table 5 in Appendix A.

Generally, the factors that influence the overall performance of the memory system are the amount of memory accesses, the optional auxiliary mechanisms to access the data (e.g., pointers, roving pointers) and the access pattern based on the implemented algorithm and its configuration. The dissipated energy is related to the accesses to memory and the memory size that DDTs occupy [20]. The energy consumption per memory access increases with the size of the memory. Hence, energy can be saved either by reducing the number of memory accesses, or by reducing the memory footprint of the DDTs, thus storing data into smaller (less energy hungry) memories. Of course, by doing both at the same time, the energy consumption is reduced much more.

The memory accesses measured are the ones made by the DDTs to access the allocated data of the application [8]. No accesses to instruction memories are measured in the context of this paper. Separate, complementary methodologies can be used, in addition to our work, to reduce the energy of the instruction memory [4]. In the context of this paper, the energy estimations are calculated using the model in [25] and concern the energy dissipated by the memory. This model depends on memory footprint factors (i.e., memory size, internal structure of banks and sub-banks, memory leaks, working time of the memory and technology) and energy consumption factors created by memory accesses (i.e., number of memory accesses, energy consumption in active mode, size of the memory and technology). In the results presented in this paper (Section 5), we have employed the 0.13 μm technology node. Nevertheless, it is possible to easily change, in a modular way, this feature and others related to the memory hierarchy (e.g., memory sizes, number of memories, etc.) and memory model. In the context of this paper, we calculate energy consumption for a flat memory hierarchy (i.e., consisting of a single on-chip SRAM).

4. Dynamic data type refinement methodology

4.1. Methodology overview

Our methodology enables the systematic refinement of Dynamic Data Types for network applications, implemented in embedded systems. Our systematic approach is supported by a number of tools to fully automate the whole process (as shown in Fig. 1). It is able to handle the exploration of more than one data structure in each application, thus providing a proposed combination of DDTs for the implementation of each data structure (e.g., for an application with three different data structures our framework might propose a combination of: a single linked list for the first one, a pointer double linked list for the second one and a simple array for the third one). The whole framework enables us for the first time to automatically explore a big number of DDT combinations for various network configurations for real-life network applications and to be able to arrive at truly optimal solutions.

The designer just inputs in our DDT exploration framework:

(1) The desired network application.
(2) The design constraints of the embedded system where the application runs and our optimizations objectives.
(3) The different network configurations that the application uses.

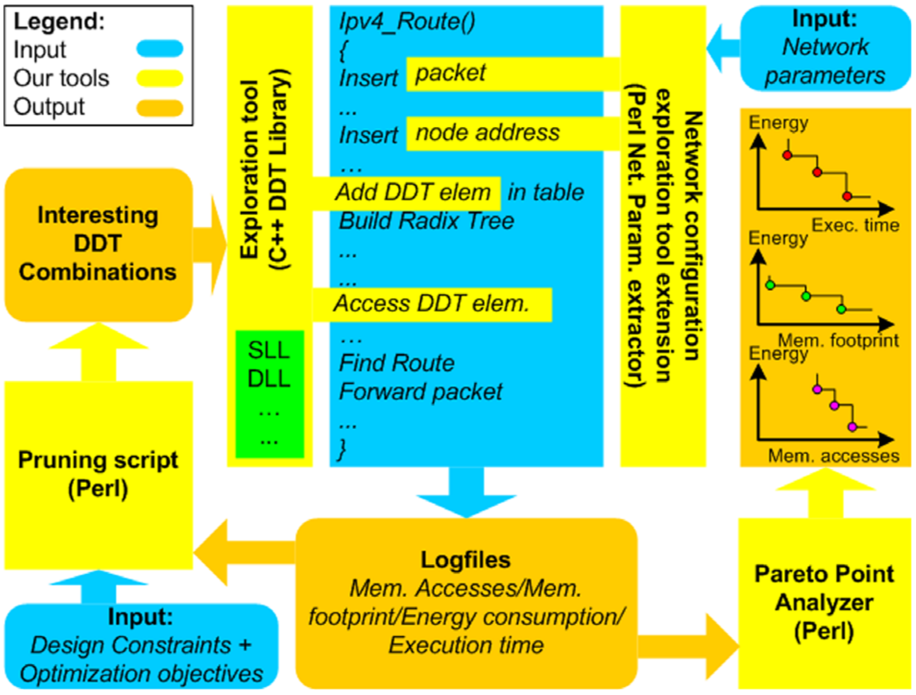

Legend:
Input
Our tools
Output
Interesting DDT Combinations
Exploration tool (C++ DDT Library)
SLL
DLL
...
...
Ipv4_Route()
{
Insert packet
...
Insert node address
...
Add DDT elem in table
Build Radix Tree
...
...
Access DDT elem.
...
Find Route
Forward packet
...
}
Network configuration exploration tool extension (Perl Net. Param. extractor)
Input: Network parameters
Energy
Exec. time
Energy
Mem. footprint
Energy
Mem. accesses
Pruning script (Perl)
Input: Design Constraints + Optimization objectives
Logfiles Mem. Accesses/Mem. footprint/Energy consumption/ Execution time
Pareto Point Analyzer (Perl)

Fig. 1. Tool support of the Dynamic Data Type Refinement methodology flow.

The output of our DDT exploration framework is the optimal combination of DDTs that should be implemented.

The goals of the proposed optimization methodology and automated exploration framework are:

- Optimized Dynamic Data Types: We refine the implementation of the Dynamic Data Types in the applications of the embedded system and thus reduce their memory footprint, their number of memory accesses, their energy consumption and increase their performance (compared to the original implementation).
- Automated exploration: We automate the refinement, which is done by exploring a set of DDT implementation from our library. The goal is to provide automated optimizations to the embedded system designer without requiring his expertise in the field of Dynamic Data Type refinement.
- Reduced design time: We prune the search space of potential solutions with a stepwise approach, thus reducing the design time that our proposed tool requires to provide an optimized DDT solution.

In the following subsections we analyze in detail the steps of the proposed optimization methodology (as shown in Fig. 2) and the way that each step is linked with the tools of the automated exploration framework (as shown in Fig. 1).

4.2. DDT exploration at application-level

In the first step of our proposed methodology (as shown in the upper part of Fig. 2), we explore the DDTs at the application-level, in order to find optimal DDT combinations for the dynamic data access behavior of the application under study. This means that we find a set of DDT combinations that have the potential to give the best results for the application under study regardless of the network configuration of the final implementation. The reason that we can pinpoint these DDTs is because there are some algorithmic characteristics of the application, which favor some DDTs more than others. For example, the algorithmic characteristic and consequent memory accesses of a LIFO queue favors double linked lists more than single linked lists, because the last record of the list is always accessed. Since we want to automate the process of pinpointing these DDTs (without wasting the valuable time of the designer), we choose to do an exhaustive exploration of the search space of the available DDT solutions.

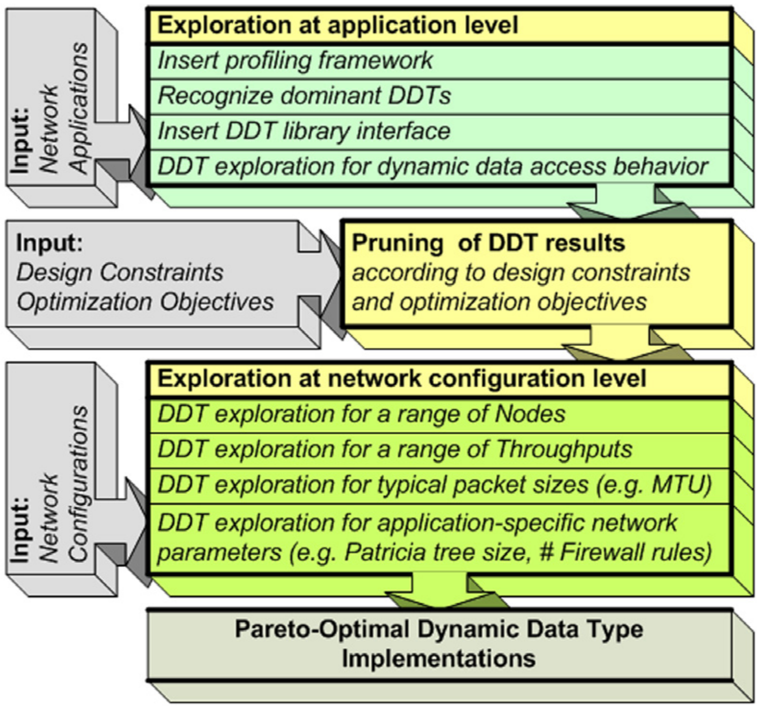

Input: Network Applications
Exploration at application level
Insert profiling framework
Recognize dominant DDTs
Insert DDT library interface
DDT exploration for dynamic data access behavior
Input: Design Constraints Optimization Objectives
Pruning of DDT results according to design constraints and optimization objectives
Input: Network Configurations
Exploration at network configuration level
DDT exploration for a range of Nodes
DDT exploration for a range of Throughputs
DDT exploration for typical packet sizes (e.g. MTU)
DDT exploration for application-specific network parameters (e.g. Patricia tree size, # Firewall rules)
Pareto-Optimal Dynamic Data Type Implementations

Fig. 2. Dynamic Data Type Refinement methodology flow.

To assist and automate this exploration step, we developed a library-based exploration framework similar to [18,26]. We attach to each candidate DDT of the network application a profile object and run the application for one typical input trace. The profiling reveals the *dominant* data structures of the application, which are then explored (typically 2–4 DDTs are accessed dynamically more than 90% of the time, see Section 5 for more details). We need to insert just once inside the source code the adequate instrumentation in order to link each dominant data structure of the network application with our C++ DDT library (as seen in the left part of Fig. 1). The instrumentation of the code is the only step not automated, requiring the designer to manually intervene in the code. The C++ DDT library is comprised of 10 different DDTs (as explained in Section 3).

The instrumentation consists of typical functions to operate the DDTs (e.g., add one more record, access the record or remove the record). The exploration is done in an automatic way by keeping the same instrumentation and changing the DDT implementation for each dominant data structure. All the DDTs in our C++ library (and combinations of them) are used in the exploration and are simulated and profiled at run-time. The whole procedure takes from 0.8 up to 64 s per simulation for a single DDT combination according to the application. By using the term simulation we mean an execution of the application under study using as input any network trace. At this point we do not need to evaluate different network configurations because we want to exploit only the inherent algorithmic characteristics of the network application.

We simulate all combinations of DDTs for the chosen network applications. For example, if there is one dominant data structure, we have to simulate 10 times, one time for each different DDT. If there are 2 dominant data structures, then we have to simulate 100 times (i.e., 10 different DDTs for the first dominant data structure combined with 10 different DDTs for the second dominant data structure). If there are 3 dominant data structures, then we have to simulate 1000 times (i.e., 10 different DDTs for the first dominant data structure, combined with 10 different DDTs for the second dominant data structure, combined with 10 different DDTs for the third dominant data structure).

4.3. Pruning of the DDT results

After the exhaustive exploration of DDTs at the application-level, we prune the best DDT solutions (as shown in the middle part of Fig. 2) according to our embedded system constraints and according to the metric (or metrics for multi-objective optimizations). This means that we select only a small subset of DDT combinations out of all the available DDT combinations. We apply two criteria in order to discard DDT combinations:

(1) If an implementation of a DDT combination exceeds the constraints of our embedded system design, this combination is discarded. For example, if we have a constraint of 4 MB memory in our embedded system design and the exploration at the application-level supplies a combination of DDTs, which uses 5 MB, we reject this combination.
(2) If we want a single objective optimization (e.g., just energy efficiency), we can discard 90% of the DDT combinations that give the worst results for this metric (i.e., energy). For multi-objective optimizations of two, three or four optimization objectives we can discard 80%, 60% and 20%, respectively of the DDT combinations that give the worst results for all the metrics that are involved in the optimization. For example, if we want to optimize for energy efficiency and performance, we discard 80% of the DDT combinations that have the worst energy efficiency and the worst execution time.

This pruning step reduces the available search space that is explored in the next step (i.e., exploration at network configuration-level) and is needed in order to reduce the total design time of the whole exploration. For example, if we evaluate in the first step that there are 2 dominant DDTs in our application, then we do the exploration at application-level and simulate the combination of 10 different DDT solutions for each dominant DDT. Therefore, the exhaustive exploration at the application-level requires 100 simulations, which provides us with 100 DDT combinations. With the use of our pruning script (as seen in the lower left part of Fig. 1) we discard 80 DDT combinations (for an energy and performance optimization). Therefore in the end we remain with 20 possible solutions, which we explore further in the next step of our methodology (i.e., exploration at network configuration level).

Note that the percentages of the DDT combination that we can prune (according to the number of objectives that we want to optimize concurrently) is not random. After exhaustive experimental results, we evaluated that the best 10%, 20%, 40% and 80% of the available DDT combinations always give us the best results for a specific application doing optimizations for one, two, three and four metrics respectively. We realized that the other DDT combinations are non-optimal regardless of the network configuration. Still we do not know which specific DDT implementation (out of the DDT combinations that were not pruned) is the optimal for a specific network configuration.

4.4. DDT exploration at network configuration-level

In the last step of our proposed methodology (as shown in the lower part of Fig. 2), we explore the DDTs at the network configuration-level. Even though we manage to prune heavily the available DDT search space in the previous step, we do not manage to select the optimal DDT combination for our multi-objective optimization. Although the algorithmic characteristics play a significant role on the optimal DDT selection (i.e., these characteristics enable the pruning in the first place), the network configuration has the final word on which DDT combination we should choose.

Different network configurations (in addition to algorithmic characteristics of the application) account for the second reason why network applications access data in different ways (and thus require different optimal DDT combinations). For example, a routing application accesses data differently; if it is configured for a classroom with 30 seats (i.e., up to 30 network nodes) or if it is configured for a conference room with 4 seats (i.e., up to 4 network nodes). In the context of this paper, when we refer to a specific network configuration we mean a range of network parameters. Therefore, we explore between the configuration of a small network (i.e., 2–4 nodes) and a big network (i.e., 20–30 nodes). However, the exact exploration of parameters (e.g., exploring for an exact number of nodes), which can easily change during run-time, is out of the context of this paper.

The general network configurations, which are important for the DDT exploration, are: the maximum number of nodes in the network, the maximum throughput of the network and the typical packet sizes used (e.g., Maximum Transmission Unit packet size and Acknowledgement packet size). Finally, other network parameters, which are used for the DDT exploration, are application-specific. For example, the Patricia tree size (or Radix tree size) is an important parameter for the IPv4 routing application and affects greatly the DDT exploration. Other examples are the level of fairness used in the Deficit Round Robin scheduling application (i.e., Quantum) and the number of rules activated in a firewall application (case studies are presented in Section 5). Note that all these configurations are known at design time or are calibrated periodically and do not change at run-time.

This exploration step requires input traces, which are typical for the network configuration. These network traces usually contain the activity of a whole week for a certain network and consist of several Gigabytes of data. With the help of our tool, we parse these input traces and extract automatically the parameters of the network. In order to test the validity of our results, we use a total of 10 traces from 8 different networks. The first three networks came from the National Laboratory for Applied Network Research [23] and originate from the total campus and satellite buildings activity. The rest come from Dartmouth University's collection of wireless network traces [16] on the corresponding campus buildings (an example of these traces can be seen in Fig. 3).

The above network configurations, and consequently their traces, come from wired and wireless networks. Also, the number of maximum nodes differ from trace to trace, as does the time when the traces were taken and the data transfer rates. This results in different traffic patterns that influence the dynamic behavior of the tested applications, contributing to a wider and more appropriate exploration. For example, a MTU (i.e., Maximum Transmission Unit) packet of bigger size can lead to bigger DDT elements, while more users (i.e., nodes in the network) could lead to more elements, depending always on the functionality of each application. Some of the variations between the network configurations are presented in Table 2.

To automate this exploration step we have developed an extension to our DDT exploration framework. The first part of the tool (written in Perl) can recognize automatically the differences between the various network configuration implementations. This is done by parsing the available network traces and extracting the network parameters from the raw data in the traces. Then, the second part of the tool (written in C++) can automatically do the DDT implementation for all the different network configurations in the application.

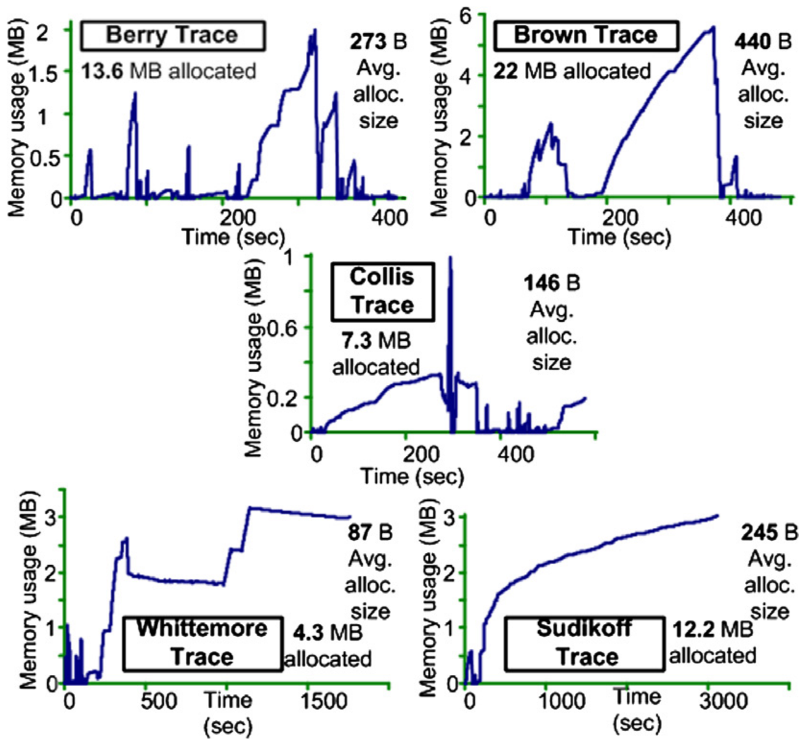


Fig. 3. Wireless network traces from five different Dartmouth University's buildings.

Table 2
Range of network configurations that were explored

Trace characteristics	Range
Max. number of nodes	16–100
Average daily traffic	1.8–12.4 GB
Average packet size	405 B–22 KB
Data transferred by 50,000 packets	36 MB–1.1 GB
Access points (in wireless networks)	2–13
Transfer rate (of the 50,000 packets used)	3.3–101.2 Kbps (wireless networks)

More specifically, we take the remaining DDT combinations of the previous step and simulate each one of them for the different network configurations. For example, this means that we perform approximately 80 simulations, if we explore for 4 different network configurations in an application with 2 dominant data structures for 2 optimization objectives (i.e., 20% of 100 DDT combinations equals 20 DDT combinations. Twenty DDT combinations times 4 different network configurations equals 80 simulations). If we had not reduced the exploration space in the previous step, we would have had to perform 400 simulations. Reduction of total simulations needed to explore the design space for 2 optimization objectives for all the network applications can be seen in Table 3.

Table 3
Reduction of total simulations needed to explore the design space for 2 optimization objectives (namely, energy vs. performance or mem. footprint vs. mem. accesses)

Network applications	Exhaustive simulations	Reduced simulations	Pareto-optimal
1. Route	1400	271	7
2. URL	500	110	4
3. Firewall	2100	546	6
4. Scheduling	500	60	3

4.5. Output of our exploration framework

The output of our exploration framework is either a single optimal DDT (if a single optimization objective was requested, for example energy) or a set of Pareto-optimal DDTs (if a multi-objective optimization was requested, for example energy vs. performance as seen in Fig. 4). Hence, instead of giving the designer a single DDT combination, our Pareto Point Analyzer (as seen in the lower right part of Fig. 1) parses all the DDT combinations from the previous step and gives a Pareto-optimal set, represented by a Pareto curve (also known as Pareto frontier). Every point in the set is better than any other solution in at least one metric. That is, it consumes, for example, the least energy under a given time constraint or it finishes earlier under a given memory footprint constraint of the embedded system. Thus, design constrains can be implemented directly in the exploration approach and get the best trade-offs from the final DDT implementation. The achieved trade-offs that we produce by applying our methodology in the four applications (see Section 5) are presented in Table 4. Therefore, Table 4 is an overview of all the experimental results, within the context of this paper.

The Pareto space of all the solutions are selected and our tool produces graphically the Pareto curves

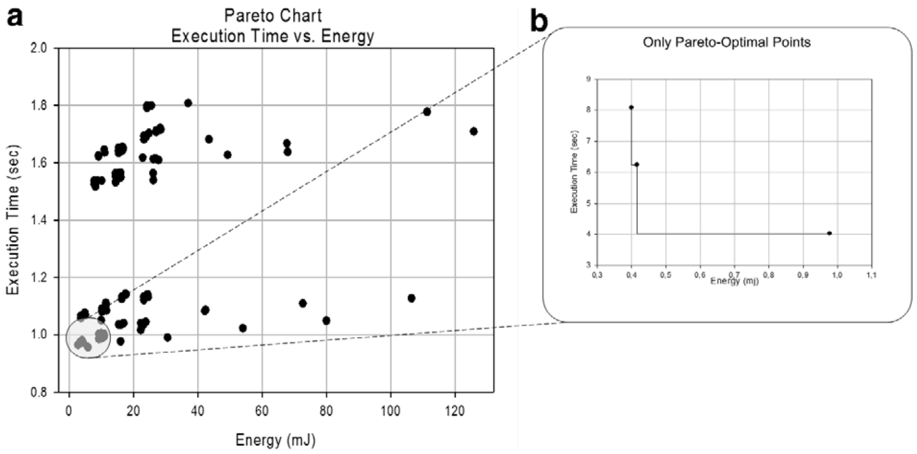


Fig. 4. (a) Performance vs. energy Pareto space of the URL application, (b) Pareto-optimal points for performance vs. energy graph.

Table 4
Trade-offs achieved among Pareto-optimal points

Application	Energy (%)	Execution time (%)	Memory accesses (%)	Memory footprint (%)
1. Route	90	20	88	30
2. URL	52	13	70	82
3. IPchains	38	3	87	63
4. DRR	93	48	53	80

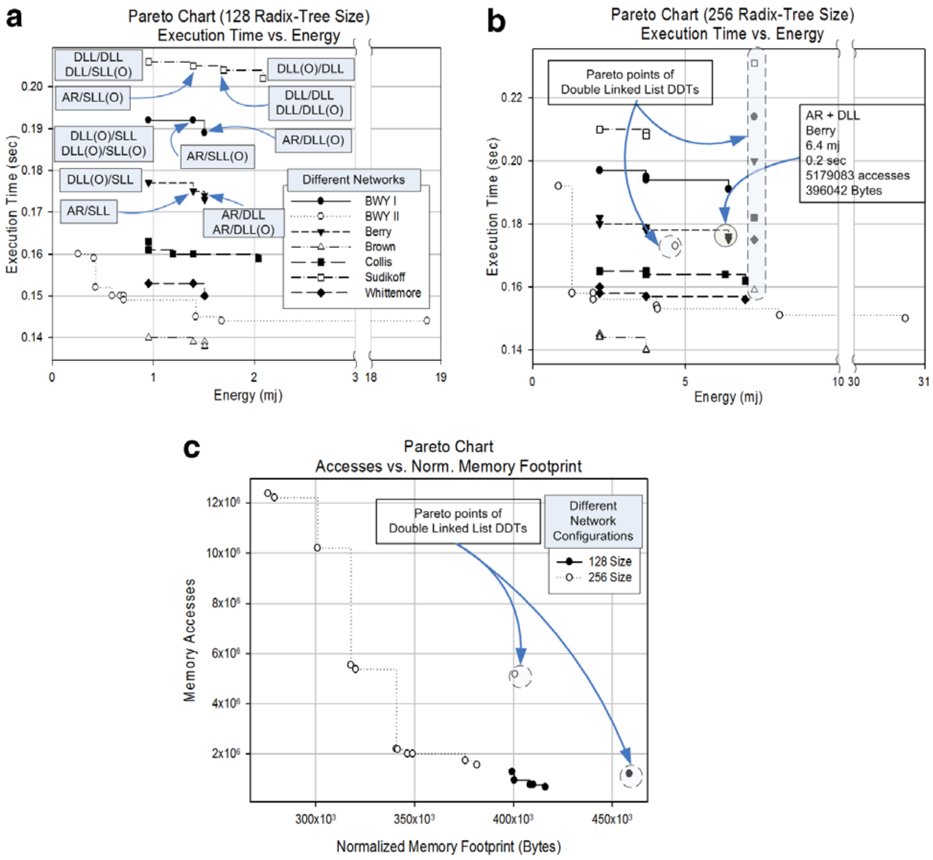


Fig. 5. Pareto charts for the route application: (a) execution time vs. energy (table size 128), (b) execution time vs. energy (table size 256), (c) accesses vs. memory footprint (BWY I).

that the embedded designer requested in his multi-objective optimization. For example, three Pareto curves can be seen in Fig. 5. The rest of the DDT combinations are discarded (i.e., the discarded DDT combination are not Pareto-optimal for the specific network configuration and the specific optimization objectives). Now the designer has a set of application-tuned, configuration-tuned Pareto-optimal DDT implementations (usually 3–4 different DDT combinations per network configuration), which are within his design constraints and fulfill the needs of his multi-objective optimization. The algorithm, that we implement in order to evaluate the Pareto Curve can be found in [11]. Our experimental results validate the need of this step, because we show that different network configurations have different Pareto-optimal DDT combinations and thus there is no single, general DDT combination that serves equally well all the network configurations.

5. Case studies and simulation results

We apply the proposed methodology to four realistic case studies, representing different modern network applications, selected from a broad variety:

the first one is a routing application, the second one is a context switching algorithm, the third one is a firewall application and the fourth one is a scheduling application. All applications are taken from the NetBench Benchmarking suite [21].

In the following subsections, we briefly describe the different dynamic data access behavior of the four case studies (due to the different algorithmic characteristics). Then, we apply the proposed methodology to explore the combinations of the DDTs that provide the Pareto-optimal solutions for each network configuration, in order to minimize the memory footprint, the number of memory accesses, the energy consumption and the execution time. In all of our experiments, we request a set of two multi-objective optimizations, namely energy vs. performance and memory accesses vs. memory footprint. We have chosen these 2 sets, because these are the most contradicting optimization factors and thus the hardest to achieve manually. The Pareto-optimal combinations for these two sets can be seen in Tables 6 and 7 in Appendix A. To validate the correctness of our approach, we compared the results of our methodology with the results obtained from a truly exhaustive exploration (i.e., without any pruning) and we found that they match.

The results were obtained with gcc-3.3.3 on a Pentium4 1.6 GHz with 512 MB RAM running Linux kernel 2.6.4. The 512 MB RAM that we use for our simulation framework limit the number of traffic traces that we can use during the simulation of our applications. Every memory access is logged during the simulation and this means that some application simulations (namely DRR and URL) run out of memory before they can finish the simulation of some memory access intensive traffic traces. All the performance results presented here are average values after a set of 10 simulations for each application, where all the final values were very similar (variations of less than 2%). A set of 10 simulations are needed because the internal processes of the Linux kernel sometimes alter the measured execution time of the simulated applications.

5.1. Methodology applied to a network routing application

The first case study is the IPv4 routing algorithm [10]. It implements the table lookup along with the internet checksum for the header. It makes the necessary changes to the IP header (e.g., to the Time-To-Live value), then fragments the packet if necessary and forwards it. The routing table uses a radix tree structure (also known as Patricia tree), which is the data structure to hold both host addresses and network addresses. The address being searched for and the addresses in the tree are considered to be sequences of bits.

Each internal node represents a bit position to test. This allows the same functions to maintain and search one tree containing fixed-length 32-bit Internet addresses. An entry in the routing table matches a search key, if we perform the logic function "AND" to the search key with the network mask of the entry and it equals the entry itself. A given search key might match multiple entries in the routing table, with a single table for both network route and host routes, the table must be organized so that more specific routes are considered before less specific routes.

In the first step, we determine, which are the dominant DDTs of the application (i.e., the ones that are accessed the most). Two dominant DDTs are present in the Route application. The `radix_node` structure that forms the nodes of the tree and the `rtentry` structure, which holds the route entries and contains other useful pointers. This means that we simulate automatically a maximum of 100 combinations of DDTs (i.e., 10 different DDTs for the first dominant data structure combined with 10 different DDTs for the second dominant data structure) to do the exploration at the application-level.

After the pruning for 2 optimization objectives, explorations are done at the network configuration level. For the simulation, 7 network configurations are used, utilizing 7 different networks (i.e., 2 wired traces from [23] and 5 wireless from [16]). Additionally, we do the exploration for 2 different network configuration, which are application-specific, namely the size of the Radix Tree (i.e., for 128 entries and for 256 entries). The maximum number of simulations for an exhaustive exploration would have been 1400 ($\times 7$ different networks $\times 2$ application-specific configurations $\times 100$ DDT combinations). The number of simulations using our methodology is reduced to 271 due to the pruning implemented according to our 2 optimization objectives (we have chosen not to apply any design constraints). During the simulation, the log files are created, containing detailed information concerning the behavior of the DDTs: the number of memory accesses, the memory footprint, the dissipated energy and the execution time.

For the graphical output, we parse the log files with the postprocessing tool and we produce the Pareto-optimal points for memory accesses vs. memory footprint and execution time vs. energy (i.e., our chosen optimization objectives). Then, the Pareto curves are drawn (Fig. 5), giving the designer a visual aspect of the solutions that we found. In Fig. 5, the Pareto-optimal curves are depicted. These curves can be produced for each combination of metrics essential to the designer. For example, in Fig. 5b the Pareto-optimal curve is depicted, for routing a table size of 128 elements and for 7 different networks. Each network configuration is represented by a curve and each point represents the combination of DDTs along with the corresponding values concerning memory accesses, memory footprint, dissipated energy and execution time. For example, in Fig. 5b the Pareto-optimal point within the solid circle is the combination of AR/DLL DDTs, with energy dissipation 6.4 mJ, execution time 0.2 s, memory footprint 396,042 Bytes and 5,179,083 accesses. For comparison reasons, if double linked lists were used, the application would demand 68.8% more memory footprint, 12% more energy and 12.5% gains in execution time from the best Pareto-optimal point of each corresponding metric (Fig. 5b and c). We do not provide comparative results with the initial DDT of the IPv4 application (i.e., a tree). As we mention explicitly in Section 3, the tree DDT was not modeled in our design search space (due to its very poor locality attributes); and thus we cannot provide consistent comparisons with the rest of the modeled DDTs.

Trade-offs can be achieved up to 90% for the dissipated energy, 20% for the execution time, 88% for the memory accesses and 30% for the memory footprint. Comparing these solutions with the remaining Pareto points, which do not belong to the Pareto-optimal curve, the gains become bigger. Particularly, we experience a reduction in memory accesses up to a factor of 8, for memory footprint up to a factor of 12, for dissipated energy up to a factor of 11 and for execution time up to a factor of 2.

5.2. Methodology applied to a context switching algorithm

The second case study is URL-based switching [21] (referred to as URL from now on), which is a commonly used context-switching mechanism. In URL, all the incoming packets to a switch are parsed and switched according to the URL requested by it. The algorithm works as follows: a table with patterns is formed. During the initialization phase, each pattern corresponds to a specific kind of data (e.g., to be served by a particular server). Therefore, there is a route indicating where packets containing similar patterns should be forwarded to. This route is also written to the table of patterns. During the normal execution phase of the application, the header of each packet is parsed and searched to find possible similarities to a pattern in the aforementioned table. Then, the packet is forwarded to the route corresponding to the pattern match that was found.

Our proposed methodology is applied to URL. In the first step, the dominant data structures of the application were located. The first one is class `StrTreeNode` (in the initial implementation it was formed with a single linked list). The second dominant data structure is a class named `PatternNode`, also forming a single linked list. Then, again the profiling instrumentation of our proposed framework is inserted and the interface functions implementing and alternating DDT combinations and input followed. Without the application of our pruning step (again for 2 optimization objectives), we would simulate the application a maximum of 500 times (100 different DDT combinations $\times$ 5 different networks) to explore the DDTs at network configuration level. The number of simulations using our methodology is reduced to 110. In the third step of our methodology, the simulation results were again filtered to get Pareto-optimal points, which are depicted in Fig. 6.

One can compare Fig. 6 to Fig. 4 to see the reduction of the Pareto-optimal points compared to the whole spectrum of available DDT solutions at the beginning of the third step. Combinations of DDTs that are not superior in at least one of the metrics have been rejected (i.e., because they are not Pareto-optimal). In URL the best combination of DDTs in terms of energy gives a 52% reduction in comparison to the most energy-consuming Pareto-optimal point. This percentage is the average of the five traces. The corresponding reduction percentage of time is 13%, of memory footprint 70% and of memory accesses 82%. Other combinations can take up to double execution time to run, consume up to 30 times more energy, have up to 4 times more memory accesses and allocate up to 5 times more memory footprint comparing to the average of the Pareto-optimal.

A comparison with the initial Netbench DDT implementations is worthwhile (both DDTs were

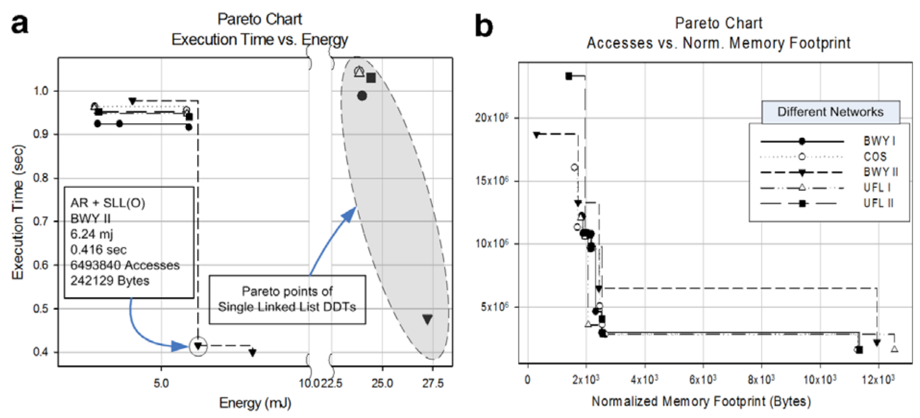


Fig. 6. URL application simulated for five different networks (curves). Pareto charts for: (a) execution time vs. energy, (b) memory access vs. memory footprint.

implemented as single linked lists). The execution time is reduced by 20% and energy by 80%. In the case of this application, it is very hard to pinpoint the reason for the bad performance of the initial DDT implementation, because for each network configuration the combination of optimal DDTs is very different (as can be seen in Tables 6 and 7 in Appendix A).

5.3. Methodology applied to a network firewall

The third case study is the IPchains firewall application [21]. The IPchains algorithm inserts and deletes rules from the packet filtering section of the kernel. The kernel starts with three lists of rules; these lists are called chains. Each chain is a checklist of rules. Each rule says "if the packet header matches certain criteria, then perform a set of actions to the packet". When the rule does not match the packet, the next rule in the chain is consulted. Finally, when there are no more rules to consult, the kernel looks at the chain policy to decide what to do. In a security-conscious system, this policy tells the kernel to reject or deny the packet.

After we insert the profiling instrumentation of our proposed framework, we determine the dominant DDTs. Two dominant data structures are present in IPchains, the structure `ip_fw` which contains fields to be filled when adding or replacing a rule (i.e., the source and destination IP address, name of the interface via which the packet is received, TOS value) and the structure `ip_fwuser`, which is used when the application is calling some of the commands of the aforementioned structure. This results in 100 DDT combinations to be simulated for these two structures. Then we insert the interface functions that enable the automated alteration of DDTs combinations and network traces.

Seven different networks and three application-specific configurations (i.e., 100, 200 and 500 rules for the firewall) were used to perform exploration at the network configuration level. Thus, for an exhaustive exploration 2100 DDT combinations would have been simulated (i.e., × 7 networks × 3 application-specific configurations × 100 DDT combinations), without implementing our pruning. The number of simulations using our methodology is reduced to 546. Initially, simulation with 2 wired traces from [23] for all the combinations of DDTs, is performed. Then, the 5 wireless traces from [16] were used. After using our tools, results were obtained for each combination of DDTs and with the use of the developed parsing tool, Pareto-optimal points for memory accesses vs. memory footprint and execution time vs. energy have been produced. In Fig. 7 the Pareto-optimal curves are depicted. These curves can be produced for each combination of metrics essential to the designer. For example, in Fig. 7c the Pareto-optimal curve is depicted, implementing 500 rules and 7 different networks. For each different network we have the Pareto-optimal curve and for each point the corresponding values concerning memory accesses, memory footprint, dissipated energy and execution time. For example, in Fig. 7c the Pareto-optimal point within the solid circle is the combination of SLL/DLL DDTs, with energy dissipation 35.92 mJ, execution time 0.893 s, memory footprint 4,831,790 Bytes and 4,292,601 accesses.

For comparison purposes, opposed to the double linked list DDT implementation, the application

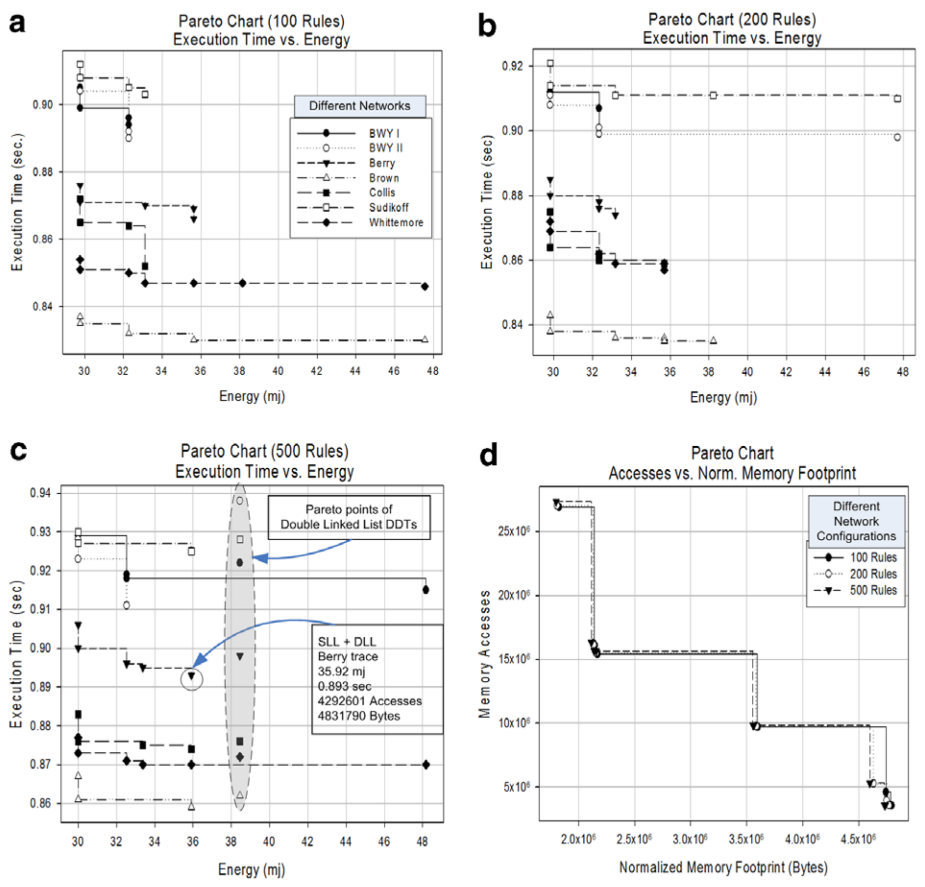


Fig. 7. IPchains application. Pareto charts for execution time vs. energy: (a) 100 rules, (b) 200 rules, (c) 500 rules and (d) accesses vs. memory footprint (BWY I).

demands 2.5 times more memory footprint and 21% more energy from the best Pareto-optimal point of each corresponding metric (Fig. 7c). In the case of this application, it is very hard to pinpoint the reason for the bad performance of the initial DDT implementation, because for each network configuration (and more specifically for each number of rules) the combination of optimal DDTs is very different (as can be seen in Tables 6 and 7 in Appendix A).

Nevertheless, we can achieve trade-offs up to 38% for the dissipated energy, 3% for the execution time, 87% for the memory accesses and 63% for the memory footprint. Apparently, when we compare these solutions with the remaining Pareto points, which do not belong to Pareto-optimal curve, the gains become bigger, several degrees of magnitude. Particularly, for memory accesses up to 39, for memory footprint up to 8, for dissipated energy up to 63 and for execution time up to 8.

5.4. Methodology applied to a network scheduling application

The fourth case study is the Deficit Round Robin (DRR from now on) fair scheduling algorithm that is commonly used for scheduling according to available bandwidth [27]. The algorithm is implemented in various switches currently available (e.g., Cisco 12000 series). In the DRR algorithm, the scheduler visits each internal non-empty queue, increments a variable called `deficit` (representing the amount of data the queue can transmit) by the value `quantum` and determines the number of bytes in the packet at the head of the queue. If the variable `deficit` is less than the size of the packet at the head of the queue, then the scheduler moves on to service the next queue. If the size of the packet at the head of the queue is less than or equal to the variable `deficit`, then the variable `deficit` is reduced by the number of bytes in the packet and

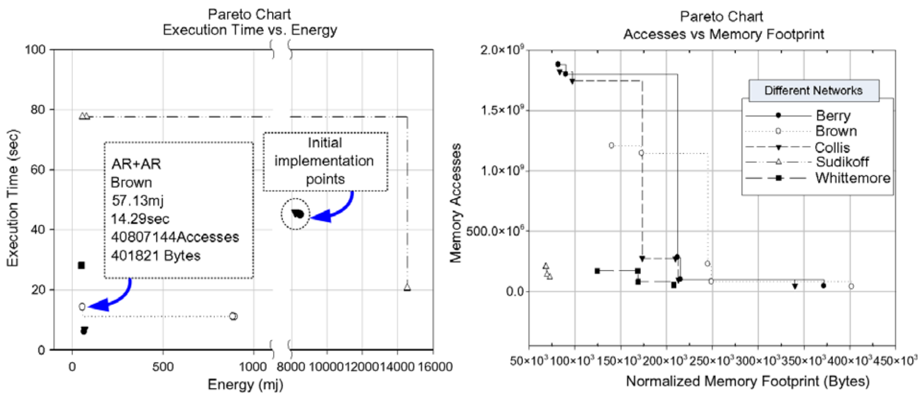


Fig. 8. DRR application simulated for five different networks (curves). Pareto charts for: (a) execution time vs. energy, (b) memory access vs. memory footprint.

the packet is transmitted on the output port. The scheduler continues this procedure to the rest of the queues, traversing them in a round robin way.

The dominant data structures in DRR are two: the first is the class `Packets`, which is used to create and encapsulate the information of the packets to be scheduled in queues. The second one is the class `Deficit_node`, used to create the queue nodes, in which the packets are stored and scheduled.

In this case study, we would simulate the application 500 times in order to explore the DDT combinations exhaustively (i.e., $\times$ 5 networks $\times$ 100 DDT combinations). The number of simulations using our methodology is reduced to 60. In each simulation 50,000 packets were scheduled, and using five network configurations taken from five traces of network traffic from Dartmouth University [16]. After the exploration at application-level, the pruning for 2 optimization objectives and the exploration at network configuration level, the simulation results were again automatically filtered to get the Pareto-optimal points depicted in Fig. 8.

In DRR in terms of energy, a 93% reduction was achieved among Pareto-optimal points. The corresponding percentage of execution time reduction is 48%, of memory footprint 53% and of memory accesses 80%. Note that these statistics come from comparisons only among the optimal DDTs. Other combinations can take more than double the time to run, 30 times more energy (consumed in the dynamic memory subsystem) and 4 times more memory accesses than the average of the Pareto-optimal. Seventy-nine combinations have been left out of the above statistics calculations due to their unacceptably high memory footprint (we considered 40 MBytes as a design constraint).

Finally, a comparison with the initial Netbench implementation is interesting (both data structures were initially implemented as single linked lists). The execution time is reduced by 22% and the energy consumption is reduced by 80%. The reason for the bad initial implementation is that the scheduling application creates huge lists of many packets and thus the memory footprint overhead of the 'next' pointers in each record of the linked list becomes very big. Most of our proposed DDT combinations (which are arrays or arrays of lists) do not have such a big memory footprint overhead and thus can be put in smaller memories (and smaller memories consume less energy per access).

6. Conclusions

In this paper, we presented a systematic approach to explore all possible implementations and combinations of DDTs using a novel methodology and supporting an automation framework, which has led to significant improvements in terms of energy consumption, execution time, memory accesses and memory footprint. This methodology shows, that the choice of an optimal implementation of a Dynamic Data Type can be flexibly tuned to the specific needs of each application (according to algorithmic characteristics), each network configuration (according to the range of network parameters), each optimization objective and each embedded system constraint. The design flow methodology was verified under various conditions, traces and DDT implementations. Furthermore, it was shown that we can reach, with the use of our methodology, significant energy savings and performance speedup compared to the original benchmarks implementation without any increase in memory footprint and memory accesses. Finally, trade-offs among the Pareto-optimal choices provide alternative solutions to the designer. In the future, we plan to extend the exploration for energy

Table 5
The different DDT implementations used

Abbreviation	Sequential access count	Random access count	Average size
SLL(AR)	$3 \times N_e + N_a$	$\frac{N_e}{2\times N_a} + 3$	$5s_w + \frac{N_e}{N_a}$
SLL(ARO)	$3 \times N_e + N_a$	$\frac{N_e}{2\times N_a} + \frac{3\times N_a}{N_e} + 1$	$7s_w + \frac{N_e}{N_a} \times (3s_w + s_T)$
DLL(AR)	$3 \times N_e + N_a$	$\frac{N_e}{4\times N_a} + 3$	$6s_w + \frac{N_e}{N_a} \times (4s_w + s_T)$
DLL(ARO)	$3 \times N_e + N_a$	$\frac{N_e}{4\times N_a} + \frac{N_a}{5\times N_e} + \frac{5}{4}$	$8s_w + \frac{N_e}{N_a} \times (4s_w + s_T)$
SLL	$3 \times N_e$	$\frac{N_e}{2} + 1$	$4s_w + N_e(2s_w + s_T)$
SLL(O)	$3 \times N_e$	$\frac{N_e}{2} + \frac{1}{N_e}$	$6s_w + N_e(2s_w + s_T)$
DLL	$3 \times N_e$	$4N_e + 1$	$5s_w + N_e(3s_w + s_T)$
DLL(O)	$3 \times N_e$	$\frac{N_e}{4} + \frac{2}{N_e} + \frac{1}{4}$	$7s_w + N_e(3s_w + s_T)$
AR	N_a	1	$N_a \times s_T$
AR(P)	$2 \times N_a$	2	$N_a(s_T + s_w)$

N_e is the number of elements in the DDT, N_a is the number of element in the array, s_w is the width of a word on the architecture and s_T the size of one element of type T.

Table 6
Optimal DDT combination (Pareto point for energy vs. execution time) for each network configuration

Network configurations	Applications			
	Route (128 size)	URL	IPchains (200 rules)	DRR
BWY I	AR/DLL	AR/AR	DLL(O)/SLL(O)	–
BWY II	DLL(O)/AR	AR/AR	DLL(O)/DLL(O)	–
COS	–	AR/AR	–	–
UFL I	–	AR/AR	–	–
UFL II	–	AR/AR	–	–
Berry	AR/DLL(O)	–	SLL/SLL(O)	AR/AR
Brown	AR/DLL(O)	–	DLL/DLL(O)	AR(P)/DLL(O)
Collis	AR/DLL(O)	–	DLL/SLL(O)	AR/AR
Sudikoff	DLL(O)/DLL	–	DLL(O)/DLL(O)	DLL(O)/AR
Whittemore	AR/DLL	–	SLL/DLL(O)	AR/AR(P)

The combinations presented here are the ones with less needed time to finish execution.

Table 7
Optimal DDT combination (Pareto point for memory accesses vs. memory footprint) for each network configuration

Network configurations	Applications			
	Route (128 size)	URL	IPchains (200 rules)	DRR
BWY I	SLL(AR)/SLL(AR)	SLL(AR)/SLL	SLL(O)/SLL	–
BWY II	DLL(ARO)/SLL(O)	SLL(O)/SLL(AR)	SLL(O)/SLL	–
COS	–	DLL/DLL(O)	–	–
UFL I	–	DLL(ARO)/SLL(AR)	–	–
UFL II	–	SLL/DLL(O)	–	–
Berry	SLL(AR)/SLL	–	AR/AR	SLL(AR)/AR
Brown	SLL(AR)/SLL	–	AR/AR	SLL(AR)/AR
Collis	SLL(AR)/SLL(O)	–	AR/AR	SLL(AR)/AR
Sudikoff	SLL(AR)/SLL	–	AR/AR	DLL(ARO)/AR
Whittemore	SLL(AR)/SLL(ARO)	–	AR/AR	DLL(ARO)/AR

The combinations presented here are the ones with less memory footprint.

efficient DDTs with the use of complex memory hierarchies and extend the search space for performance-centric DDTs with the tree data structure.

Appendix A

In the appendix, we summarize in Table 5 the average size, average sequential access count and the average random access count of each DDT in our library. In Table 6, we give the multi-objective Pareto-optimal DDT combinations for all our applications and different network configurations. Our objectives in this case are energy consumption and execution time. In Table 7, we give the multi-objective Pareto-optimal DDT combinations for all our applications and different network configurations. Our objectives in this case are memory footprint and memory accesses.

References

[1] Alfred V. Aho, John E. Hopcroft, Jeffrey Ullman, J.D. Ullman, J.E. Hopcroft, Data Structures and Algorithms, Addison-Wesley Longman Publishing Co., Inc., 1983.

[2] David Atienza, Mark Leeman, J.M. Mendias, Francky Catthoor, V. de Florio, G. Deconinck, Some experiences on dynamic memory management refinement at system-level for multimedia applications, in: DCIS '03: Proceedings of the Conference on Design of Circuits and Integrated Systems, 2003.

[3] Yossi Azar, Yossi Richter, Management of multi-queue switches in qos networks, Algorithmica 43 (1–2) (2005).

[4] Francisco Barat, Murali Jayapala, Tom Vander Aa, Rudy Lauwereins, Geert Deconinck, Henk Corporaal, Low power coarse-grained reconfigurable instruction set processor, in: Proceedings of the 13th International Workshop Field-Programmable Logic and Applications (FPL 2003), September 2003.

[5] Luca Benini, Giovanni de Micheli, System-level power optimization: techniques and tools, ACM Transactions on Design Automation of Electronic Systems 5 (2) (2000) 115–192.

[6] Luca Benini, A. Macii, E. Macii, M. Poncino, Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation, IEEE Design & Test of Computers (April–June) (2000) 74–85.

[7] Francky Catthoor, Eddy de Greef, Sven Wuytack, Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design, Kluwer Academic Publishers, 1998.

[8] Francky Catthoor, Alexander De Graaf, Reinder Nouta, Rene Van Leeuwen, et al., Unified Meta-Flow Summary for Low-Power Data-dominated Applications, Kluwer, 2000.

[9] Angel Dominguez, Sumesh Udayakumaran, Rajeev Barua, Heap data allocation to scratch-pad memory in embedded systems, Journal of Embedded Computing (2005).

[10] FreeBSD. freebsd operating system, 2003. <http://www.freebsd.org>.

[11] Tony Givargis, Frank Vahid, Jorg Henkel, System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip, in: ICCAD '01: Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design, Piscataway, NJ, USA, IEEE Press, 2001, pp. 25–30.

[12] P.A. Green Jr., The art of creating reliable software-based systems using off-the-shelf software components, in: SRDS '97: Proceedings of the 16th Symposium on Reliable Distributed Systems (SRDS '97), Washington, DC, USA, IEEE Computer Society, 1997, p. 118.

[13] J.L. da Silva Jr., M. Sgroi, F. De Bernardinis, S.F. Li, A. Sangiovanni-Vincentelli, J. Rabaey, Wireless protocols design: challenges and opportunities, in: CODES '00: Proceedings of the 8th International Workshop on Hardware/software Codesign, New York, NY, USA, ACM Press, 2000, pp. 147–151.

[14] Julio Leao da Silva Jr., Chantal Ykman-Couvreur, Miguel Miranda, Kris Croes, Sven Wuytack, Gjalt de Jong, Francky Catthoor, Diederik Verkest, Paul Six, Hugo De Man, Efficient system exploration and synthesis of applications with dynamic data storage and intensive data transfer, in: DAC '98: Proceedings of the 35th Annual Conference on Design Automation, ACM Press, 1998, pp. 76–81.

[15] M. Kandemir, J. Ramanujam, A. Choudhary, Improving cache locality by a combination of loop and data transformations, IEEE Transactions on Computers 48 (2) (1999) 159–167.

[16] David Kotz, Kobby Essien, Analysis of a campus-wide wireless network, in: Proceedings of the 8th Annual International Conference on Mobile Computing and Networking, September 2002, pp. 107–118. Revised and corrected as Dartmouth CS Technical Report TR2002-432.

[17] Marc Leeman, David Atienza, Francky Catthoor, Geert Deconinck, Vincenzo de Florio, Jose Manuel Mendias, Rudy Lauwereins, Methodology for refinement and optimisation of dynamic memory management for embedded systems in multimedia applications, in: Proceedings of Signal Processing Symposium (SiPS), Seoul, Korea, August 2003. IEEE Signal Processing Society and IEEE Circuits and Systems Society, pp. 369–374.

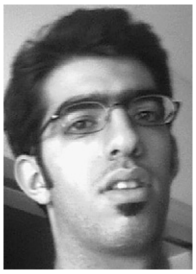
[18] Marc Leeman, Chantal Ykman, David Atienza, Vincenzo De Florio, Geert Deconinck, Automated dynamic memory data type implementation exploration and optimization, in: ISVLSI '03: Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03), Washington, DC, USA, IEEE Computer Society, 2003, p. 222.

[19] M. Herlihy, V. Luchangco, M. Moir, W. Scherer, Software transactional memory for dynamic-sized data structures, in: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, ACM Press, 2003.

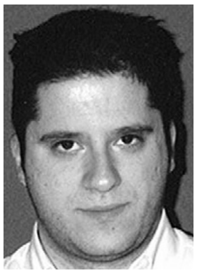
[20] Stylianos Mamagkakis, Alexandros Mpartzas, Georgios Pouiklis, David Atienza, Francky Catthoor, Dimitrios Soudris, Jose Manuel Mendias, Antonios Thanailakis, Design of energy efficient wireless networks using dynamic data type refinement methodology, in: Proceedings of Wired/Wireless Internet Communications (WWIC 2004), 2004, pp. 26–37.

[21] Gokhan Memik, William H. Mangione-Smith, Wendong Hu, Netbench: A benchmarking suite for network processors, in: Proceedings of the 2001 IEEE/ACM International

Conference on Computer-aided Design, IEEE Press, 2001, pp. 39–42.
[22] Steven S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann Publishers Inc., 1997.
[23] NLANR, National Laboratory for Applied Network Research. <http://www.nlanr.net>.
[24] Preeti Ranjan Panda, Franky Catthoor, Nikil D. Dutt, K. Danckaert, Erik Brockmeyer, C. Kulkarni, Data and memory optimizations for embedded systems, ACM Transactions on Design Automation for Embedded Systems (TODAES) 6 (2) (2001) 142–206.
[25] A. Papanikolaou, M. Miranda, F. Catthoor, H. Corporaal, H. De Man, D. De Roest, M. Stucchi, Karen Maex, Global interconnect trade-off for technology over memory modules to application level: case study, in: SLIP '03: Proceedings of the 2003 International Workshop on System-Level Interconnect Prediction, New York, NY, USA, ACM Press, 2003, pp. 125–132.
[26] P. Plauger, A. Stepanov, M. Lee, D. Musser, The Standard Template Library, Prentice-Hall, 1998.
[27] M. Shreedhar, George Varghese, Efficient fair queueing using deficit round-robin, IEEE/ACM Transactions on Networking 4 (3) (1996) 375–385.
[28] S. Steinke, L. Wehmeyer, B. Lee, P. Marwedel, Assigning program and data objects to scratchpad for energy reduction, in: DATE '02: Proceedings of the Conference on Design, Automation and Test in Europe, Washington, DC, USA, IEEE Computer Society, 2002, p. 409.
[29] Derick Wood, Data Structures, Algorithms, and Performance, Addison-Wesley Longman Publishing Co., Inc., 1993.
[30] Sven Wuytack, Francky Catthoor, Hugo De Man, Transforming set data types to power optimal data structures, in: ISLPED '95: Proceedings of the 1995 International Symposium on Low power Design, New York, NY, USA, ACM Press, 1995, pp. 51–56.
[31] Sven Wuytack, Julio Leao da Silva Jr., Francky Catthoor, Gjalt de Jong, Chantal Ykman-Couvreur, Memory management for embedded network applications, in: Readings in Hardware/Software Co-design, Kluwer Academic Publishers, 2002, ISBN 1-55860-702-1, pp. 465–476.
[32] Ch. Ykman-Couvreur, J. Lambrecht, D. Verkest, F. Catthoor, H. de Man, Exploration and synthesis of dynamic data sets in telecom network applications, in: ISSS '99: Proceedings of the 12th International Symposium on System Synthesis, Washington, DC, USA, IEEE Computer Society, 1999, p. 85.



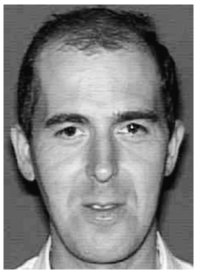
Stylianos Mamagkakis received his Diploma in Electrical and Computer Engineering from the Democritus University of Thrace, Greece, in 2002. He is currently a senior PhD candidate in the VLSI Design and Testing Center in the Democritus University of Thrace. His research interests include optimizations in dynamic memory management on multimedia and wireless network applications for low power and high performance, embedded systems using high-level design optimizations. He has published more than 20 papers in international journals and conferences. He was investigator in four research projects funded from the Greek Government and Industry as well as the European Commission. He is a member of the IEEE.



Alexandros Bartzas received his Diploma in Electrical and Computer Engineering from Democritus University of Thrace, Greece, in 2003. He is currently a PhD Candidate in the VLSI Design and Testing Center of Democritus University of Thrace. His research interests include dynamic memory optimizations, dynamic data assignment and access scheduling for embedded systems using system-level design optimizations. He was investigator in four research projects funded by the Greek Government and Industrial Partners, as well as the European Commission. He is a member of the IEEE.

Georgios Pouiklis received his Diploma in Electrical and Computer Engineering from the Democritus University of Thrace, Greece, in 2003 and his M.Sc. in 2005 from the same institution. He is currently a PhD candidate in the Space Research Laboratory of the Democritus University of Thrace. His research interests include embedded software optimization methodologies and mixed signal IC design for space applications. He has been a researcher in three European research projects funded by the Greek government, the European Commission and the European Space Agency over the last four years.

David Atienza received the M.Sc. and PhD degrees in Computer Science from Complutense University of Madrid (UCM), Spain in June 2001 and June 2005, respectively. Currently he is Post-Doc at the Integrated Systems Laboratory at EPFL, Switzerland. He also holds the position of invited Assistant Professor at the Computer Architecture and Automation Department (DACYA) of UCM. His research interests include several aspects of design technologies for integrated circuits and systems, with particular emphasis on dynamic memory management on embedded systems, flexible Networks-On-Chip (NoC) interconnection paradigms for Multi-Processors System-on-Chip, design automation and low-power design. In these fields, he is reviewer and co-author of various publications in prestigious journals and international conferences: ACM TODAES, IEEE Trans. on VLSI Systems, VLSI Journal, Journal of Embedded Systems, DATE, DAC, etc. Also, he is part of the Technical Program Committee of the IEEE/ACM DATE conference.



Francky Catthoor received a Ph.D. in El. Eng. from the K.U. Leuven, Belgium in 1987. Since then, he has headed several research domains in the area of architectural methodologies and system synthesis for embedded multimedia and telecom applications. His current research activities mainly belong to the field of system-level exploration, with emphasis on data storage/transfer and concurrency exploitation, both in customized and programmable (parallel) instruction-set processors. All this within the DESICS division at IMEC, Leuven, Belgium where he is currently a research fellow. He is also professor at the K.U. Leuven. He has (co-)authored over 500 papers in international conferences and journals, and has worked on 8 text books in this domain. He was the program chair and organizer of several conferences including ISSS'97 and SIPS'01.

Dimitrios Soudris received his Diploma in Electrical Engineering from the University of Patras, Greece, in 1987. He received the Ph.D. Degree in Electrical Engineering, from the University of Patras in 1992. He is currently working as Assoc. Professor in Dept. of Electrical and Computer Engineering, Democritus University of Thrace, Greece. His research interests include low power VLSI design, embedded systems design, and VLSI Signal Processing. He has published more than 150 papers in international journals and conferences. Also, he is co-editor in two books of Kluwer and Springer. He is leader and principal investigator in numerous research projects funded from the Greek Government and Industry as well as the European Commission (ESPRIT II-III-IV and 5th & 6th IST). He has served as General Chair and Program Chair for the International Workshop on Power and Timing Modelling, Optimisation, and Simulation (PATMOS) and he will be General Chair of IFIP-VLSI-SOC 2008. Also, he received an award from INTEL and IBM for the project results of LPGD #25256 (ESPRIT IV) and awards from Int. Conferences VLSI 2005 and ASP-DAC 05 for the results of the project AMDREL IST-2001-34379. Finally, he is a member of the IEEE, the VLSI Systems and Applications Technical Committee of IEEE CAS and the ACM.

Antonios Thanailakis was born in Greece on August 5, 1940. He received B.Sc. degrees in physics and electrical engineering from the University of Thessaloniki, Greece, 1964 and 1968, respectively, and the Msc. and Ph.D. Degrees in Electrical Engineering and Electronics from UMIST, Manchester, UK in 1968 and 1971, respectively. He has been a Professor of Microelectronics in Dept. of Electrical and Computer Eng., Democritus Univ. of Thrace, Xanthi, Greece, since 1977. He has been active in electronic device and VLSI system design research since 1968. His current research activities include microelectronic devices and VLSI systems design. He has published a great number of scientific and technical papers, as well as five textbooks. He was leader for carrying out research and development projects funded by Greece, EU, or other organizations on various topics of Microelectronics and VLSI Systems Design (e.g. NATO, ESPRIT, ACTS, STRIDE).