Dynamic Data Type Optimization and Memory Assignment Methodologies[⋆]

Alexandros Bartzas[1], Christos Baloukas[1], Dimitrios Soudris[2], Konstantinos Potamianos[3], Fragkiskos Ieromnimon[3], Nikolaos S. Voros[3,4]

[1] ECE Department, Democritus University of Thrace, 67100 Xanthi, Greece
[2] ECE School, National Technical University of Athens, 15780 Zografou, Greece
[3] Intracom Telecom Solutions S.A., Paeania 19002, Athens, Greece
[4] Dept. of Telecommunication Systems & Networks, T.E.I. of Mesolonghi, Greece

Abstract. It is evident that most new computing platforms are becoming more and more complex encapsulating multiple cores and reconfigurable elements. This offers the designers a multitude of resources. It is their responsibility to exploit the available resources in such a way to efficiently implement their applications. Furthermore, the complexity of the applications that run on such platforms is increasing as well. Applications running on such platforms need to cope with dynamic events and have their resource requirements vary during the execution time. In order to cope with this dynamism applications rely in the usage of dynamic data. Applications use containers such as dynamic data types in order to store and retrieve these dynamic data. In this work a set of methodologies that is able to optimize the containers holding the dynamic data and efficiently assign them on the available memory resources is presented. The proposed approach is evaluated in a scheduler for an IEEE802.16-based broadband wireless telecom system and a 3D game application, achieving reductions in the memory energy consumption of 32% and 51% respectively.

1 Introduction

Modern computing platforms are becoming more and more complex encapsulating multiple cores and reconfigurable elements. This makes the design, implementation and mapping of applications into them a challenging task. It is the responsibility of the designers to cope with increased complexity, shorten the design productivity gap, decide which parts of the applications will be in software, which parts will be executed in reconfigurable units and exploit the RTOS services and the multitude of hardware resources offered [1].

Additionally, the complexity of the applications running on such systems is increasing, together with the dynamism of such applications. There are many factors that contribute to the dynamism of modern telecom and network applications. As far as these applications are concerned, the varying size and timing of the packets, that run the network, create an unknown set of requests in data storage and data access. Until recently

[⋆] This work is partially supported by E.C. funded MORPHEUS IST-4-02734 Project, www.morpheus-ist.org

the problem of storing and transferring data to the physical memories was limited to the management of stack and global data statically allocated at design time [2]. Nowadays, increased user control and interaction with the environment have increased the unpredictability of the data management needs of each software application. Moreover, in the future this unpredictability will increase due to the ever-increasing functionality and complexity of telecom systems. This is a turning point for the data management of these applications since the optimal allocation and de-allocation of dynamic data needs to be performed at run-time (through the usage of heap data in addition to stack and global data) [3].

Dynamically allocated data structures, such as linked lists, which are present in telecom and multimedia applications, allocate and access their stored elements in quantities and at time intervals, which can not be known at design-time and become manifest only at run-time. In modern dynamic applications, data are stored in entities called data structures, dynamic data types (DDTs from now on) or simply containers, like arrays, lists or trees, which can adapt dynamically to the amount of memory used by each application. These containers are realized at the software architecture level and are responsible for keeping and organizing the data in the memory and also servicing the applications requests at run-time. Therefore, a systematic exploration is needed, helping the designer to select the optimal data structure implementation for each application. The cost factors that are taken under consideration are the number of memory accesses and the required memory footprint needed by the applications.

In order to optimize the usage of dynamic memory, the designer must choose an efficient implementation for each existing DDT in the application from a design space of possible implementations [4,5] (examples are dynamic arrays, linked lists, etc.). According to the specific constraints of typical embedded system design metrics, such as performance, memory footprint and energy consumption, the designer should explore the DDT design space and then to choose the DDT implementation that performs the best and does not violate the system constraints. This task is typically performed using a pseudo-exhaustive evaluation of the design space of DDT implementations (i.e., multiple executions) for the application to attain the Pareto front, which would try to cover all the optimal implementation points for the aforementioned required design metrics [6]. The construction of this Pareto front is a very time-consuming process, sometimes even unaffordable without proper DDT optimization and exploration methods.

Another equally important aspect of the implementation phase involves the assignment of the application data, both static and dynamic. It is the responsibility of the designer to derive an efficient data assignment on the available resources of the computing platform. In the context of this work the assignment problem is formulated as a knapsack one. Modern platforms offer scratchpad memories apart from caches. These memories are directly addressable and the designer has the full responsibility on placing and removing data from there. Usually scratchpad memories are used to store static data. In this paper a more holistic approach is taken, in contrast to previous works, considering the placement of dynamic data into such memories. This can be achieved by the explicit placement of dynamic memory pools in the address space of the scratchpad memory. We formulate such a problem as a multi-objective knapsack problem. In order to evaluate the proposed approach we use a scheduler for an IEEE802.16-based broadband wireless telecom system and 3D game.

The rest of the paper is organized as follows. An overview of the related work is presented in Section 2. The methodologies for dynamic data type optimization and memory assignment are presented in Section 3. The evaluation of the proposed methodologies is presented in Section 4 and finally the conclusions are drawn in Section 5.

2 Related Work

Regarding DDT optimization, in general-purpose software and algorithms design [4,5], primitive data structures are commonly implemented as mapping tables. They are used to achieve software implementations with high performance or with low memory footprint. Additionally, the Standard Template C++ Library (STL) provides many basic data structures to help designers to develop new algorithms without being worried about complex DDT implementation issues. These libraries usually provide interfaces to simple DDT implementations and the construction of complex ones is a responsibility of the developer. A previous approach in DDT optimization was presented in [7], where a systematic methodology for generating performance-energy trade-offs by implementing dynamic data types, targeting network applications was proposed. The library of DDTs used in that work had several limitations that the authors of [8] addressed and overcame.

An important aspect of assigning data on computing systems is how to perform the partitioning of the data (decide which data structure goes where, evaluate possible trade-offs etc.) [9]. Ideally, data should be allocated so that simultaneous accesses are possible without having to resort to duplicating data. In [10] a dynamic method is proposed for allocating portions of the heap to the scratchpad memory. These portions are called "bins" and are moved between the main memory and the scratchpad, when the data they hold is known not to be accessed in the running phase of the application. For each dynamic data type, a bin is created and only the first instances of it will reside into the bin (and so into the scratchpad), while the rest will be kept into a different pool permanently mapped into the main memory. It is important to place the most frequently accessed data in the closest memories to the processor.

Most of the previous approaches focus on the problem of mapping static data (stack and global variables). This is highlighted in [11–14] where various techniques are presented that map static structures into scratchpad memories. Moreover, [15,16] are good overviews of the available techniques to improve memory footprint and decrease energy consumption in statically allocated data. However, all these approaches focus only at optimizations of global and stack data, which are allocated at compile-time. In this work, we propose optimizations of heap data, which are allocated at run-time. Furthermore, we combine the data assignment step with a first step that is able to optimize the dynamic data type implementations. The aforementioned approaches are compatible with the proposed one and can be combined in order to optimize data of embedded applications, which are allocated both at compile-time and at run-time.

3 Methodology Overview

The overview of the methodology is presented in Figure 1. The application source code is instrumented (insertion of profiling directives and interface to the DDT Library) keeping the functionality intact.

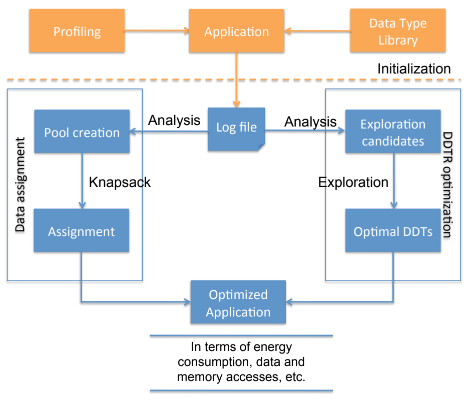

Fig. 1. Methodology Overview

All DDTs are replaced by customized implementations that incorporate a profiling library to log all accesses made within the DDT mechanisms. This insertion is manually executed but a well-defined interface (STL compliant) alleviates the effort of the designer. This is the only manual step of this flow. Afterwards, using typical inputs a platform independent exploration based on the aforementioned cost factors can be performed. In particular, the log file is analyzed by two different modules targeting the optimization of the DDT implementations and the assignment of the memory accesses to suitable memory pools. The exploration phase is fully automated. Finally the results are integrated into the now optimized application.

3.1 Dynamic Data Type Optimization

Choosing an improper DDT implementation can deteriorate the performance of the dynamic memory subsystem of the computing system. On the one hand, inefficient data access and storage operations cause performance issues, due to the added computational overhead of the internal DDT mechanisms. On the other hand, each access of the DDTs to the physical memory (where the data is stored) consumes energy and unnecessary accesses can comprise a very significant portion of the overall power of the system.

A dynamic 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 behaviour. Each set of data is assigned to specific DDT. The final decision about the optimal combination of the different DDTs that should be implemented in the application is influenced of this complex algorithmic-based dynamic behaviour. Therefore, no general, domain-specific, optimal solution exists but only custom, application-specific ones. Thus, the decision should be in accordance to both the application's algorithmic-based dynamic behaviour and the dynamic behaviour influenced by the network configuration. Finally, the system design restrictions also have to be met. This decision requires a very complex exploration task.

A modified version of the DDT Library [8] has been used to implement variations of the dynamic data structures used in the test cases. Differentiated by their access pattern, two types of DDTs, namely a queue and an unsorted linked list, are used in the applications under test. A queue can be implemented either as an array (QAR), or as a singly linked list (QLL); while a linked list in general can have countless implementation variations. Using the componentized architecture of the DDT Library the designer can easily set up nine implementations of a linked list that can be regarded as the most representative ones (see Table 1).

DDT Implementations	Description
AR	a simple dynamic array modeling the STL Vector DDT
SLL	a singly linked list
DLL	a doubly linked list
SLLO	a singly linked list with roving pointer
DLLO	a doubly linked list with roving pointer
SLL(AR)	a singly linked list of arrays
DLL(AR)	a doubly linked list of arrays
SLL(ARO)	a singly linked list of arrays with roving pointer
DLL(ARO)	a doubly linked list of arrays with roving pointer

Table 1. Implementations used in the exploration

The exploration phase of the dynamic data type optimization module involves testing each different DDT implementation and logging related accesses and memory footprint. For each DDT, all available implementations are separately tested. In the end, the implementations that perform best are chosen for each DDT (forming a Pareto solution space). The final combination of implementations for all DDTs is simulated and tested against the original implementations to calculate improvements (see Section 4).

3.2 Dynamic Data Type Assignment

The step that follows the optimization of the DDTs is the data assignment one. In this step the decision on where the data should be placed is taken. The DDTs that were optimized in the previous step serve as a container for the dynamic data handled by the application. So the number of accesses and requested footprint is the one corresponding to these DDT implementations. At this stage the results from the profiling performed at the DDTs of the applications allow the ranking of these data types, according to their size, number of accesses, access pattern, allocation/de-allocation pattern and lifetime, for each cost function relevant to the system. These DDTs are placed into pools which in turn are managed by the system's memory allocators. The main focus is on the internal (on-chip/scratchpad) memory, but data can be allocated, according to the configuration of the memory manager, to the main memory as well (off-chip memory, usually SDRAM). The data assignment provides guidelines about which pools reside in which level of the memory hierarchy.

Taking into consideration all these criteria the assignment problem is modelled as a multi-choice knapsack problem. In order the problem to be solved the knapsack algorithm is fed with the list of data types and with the characteristics of the memory hierarchy (levels of hierarchy, number of modules etc). The solution of this problem is the assignment decisions need to be taken by the system, organizing the data types into pools.

One last important issue remains whether it is convenient or not to split pools across several memory modules. If pools are split over different memory modules, then the assignment problem becomes equivalent to the fractional multiple knap-sack problem and can be solved with a greedy algorithm in polynomial time. The problem with this approach is that it is not possible to know a priori the access pattern to each of the parts of the pools, so it is not easy to quantify the overall impact on cost of the assignment decisions. On the contrary, if pools are not split, the assignment problem is a binary (0/1) multiple knap-sack problem. This problem is much harder to solve as no polynomial-time algorithm can be employed to solve it [17]. Therefore, it is important to keep the size of the input as small as possible. Also, if a problem can be modelled as a fractional knap-sack one, then a better solution can usually be found with this method. However, as pools are not spread over different memory resources at all, it is much easier to quantify the number of accesses that are mapped into the corresponding one.

If pools cannot be split over different memory resources then we define an assignment function $f(i,j)$ that for a given pool P_i and memory resource M_j returns 1 if and only if P_i should be contained into M_j. Then it is possible to formalize the assignment process as a binary multiple knapsack problem. C_j is the cost of the memory module M_j (it can be energy per access, cycles per access etc.), A_i and $Size_i$ are the number of accesses and the size the pool P_i respectively, whereas S_i is the size of the memory module M_j. The target that must be minimized can be characterized according to the following formula:

$$\min\left(\sum_{i=1}^{n}\sum_{j=1}^{m} f(i,j)\times C_j \times A_i\right) \tag{1}$$

The solution must comply with the two following constraints:

1. The total size of all the pools assigned to a given memory resource cannot exceed its capacity

$$\left(\sum_{i=1}^{n} f(i,j)\times Size_i\right) \le S_j, \qquad j = 1...m \tag{2}$$

2. Each pool is assigned exactly once to a memory resource

$$\sum_{j=1}^{m} f(i,j) = 1, \qquad i = 1...n \tag{3}$$

4 Experimental Results

The first application used to evaluate the proposed methodologies is the scheduler of an IEEE802.16-based system terminal and is responsible for establishing connections and then servicing them by inserting and removing cells while supporting interrupts in the scheduling procedure. The number of connections can reach up to 16. Furthermore, the requirements for each connection can be different since several bitrates are supported as well as Quality-of-Service requirements. The designer can set up connections with the desired characteristics and then employ a simulator to check their performance. The application provides detailed information about the current situation of the network for each scheduling cycle. So we can monitor the cell delay, the length of the queues, etc. In this application two are the dynamic data types of interest and these ones are handling data regarding the list of connections and the queues of traffic cells. The connections are being hold in a list, while the traffic cells are being placed in queues. Even though the number of DDT basic building blocks is limited, developers tend to write custom DDTs for each application. Therefore, the number of alternatives is limited to the developer's skill and the available time to implement each one of them. Ten different DDT implementations were developed and used in the exploration and final refinement. In general 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.

In Figure 2 the exploration results are depicted when the queue is implemented as an array (QAR) and when the queue is implemented as linked list (QLL). For the queue we evaluate two implementations and for the list we evaluate the 9 implementations presented in Table 1. In both figures the values for the requested memory footprint and data accesses are normalized to the values of the list data type implemented as array (that is the list holding the connection is implemented as an array). As we can see in both figures the DDTs implemented as linked list exhibit the best behaviour in terms of data accesses and memory footprint. Compared to the performance of the array the single linked list (SLL) implementations exhibit 5% reduced data accesses and using a memory model from MICRON we can see the same implementation can achieve a 6% improvement on energy consumption. Having as goal the combined reduction in data accesses and memory footprint, the SLL implementation is the designers choice.

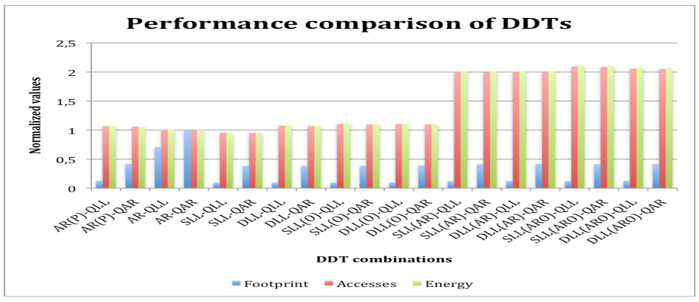

Fig. 2. Comparison of the performance of various implementations for two DDTs

The next step in the flow is the assignment of the dynamic data. The information needed to perform this step is: a) the size of the dynamic data type; the numbers of objects alive during the execution, and c) the number of reads and writes to these data. A tool has been developed to process the profiling information and produce a report of the behaviour of the data types. The elements that reside in the connection list and in the queue of traffic cells are of certain size that is not changing during the execution time. The designer chooses the creation of pools that are capable of handling requests of a specific, predefined size. So the size of each allocation is fixed at the moment the pool is created. The big advantage of using such a pool is the lack of fragmentation, which is a significant degrading factor in the dynamic memory management. Assuming that the system has a 4kB scratchpad and an SDRAM memory we conclude that not all dynamic data can fit into the scratchpad memory, since we have to reserve the space for some scalar variables and static data structures that are more frequently accessed. In this work it was chosen to profile not only the dynamic data types but the static data structures contributing to the resource usage of the application. These structures hold information about the scheduling, the contention, the QoS, etc. Using the extracted information the problem is formulated as a knapsack one and solved using an ILP approach. The outcome of this step provides to the designer a list of static data structures and pools, which can be accommodated into the scratchpad memory and the DRAM.

The case when all the dynamic data are allocated in the SDRAM (solution1) is used as a baseline. As a second solution we place explicitly at the scratchpad memory only the pool that can fit there (the only holding the connections) (solution2). The final solution (solution3) is the one proposed by the mapping algorithm placing dynamic and static data at the scratchpad memory. The reduction in the cycles consumed to access the data is shown in Figure 3.

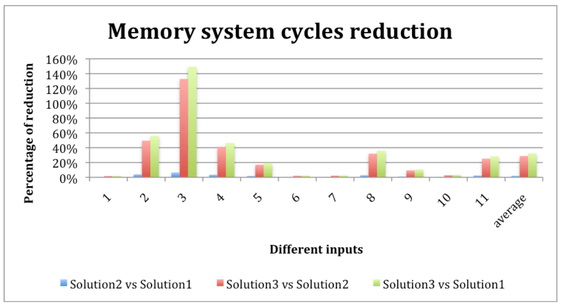

Fig. 3. Memory system cycles reduction

We can see that even a simple pool allocation at the scratchpad memory can achieve gains of 2% on average. The more elaborate solution proposed by the methodology achieves a reduction of 32% on average. Using an energy model from MICRON the improvements in term of energy consumption in the memory subsystem can be calculated. These results are depicted in Figure 4. We can see that the pool assignment in the scratchpad memory can provide us with a 3% reduction in energy consumption and when we test the combined approach we reach a 30% reduction (on average).

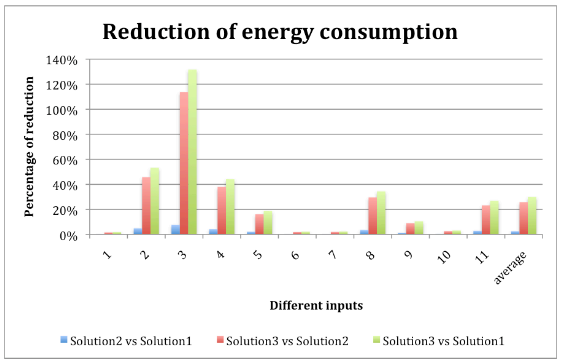

Fig. 4. Reduction of the energy consumption in the memory system

The second application employed to evaluate the proposed approach is from the multimedia domain. It is a 3D game called Vdrift [18]. Vdrift is an open source racing simulator that uses STL Vector to handle its dynamic behaviour. The application uses very realistic physics to simulate the car's behaviour and also includes a full 3D environment for interaction. Vdrift uses 37 dynamic DDTs to hold its dynamic data that are all sequences. The objects put inside the containers vary from wheel objects to float numbers required by the game's physics. During the game, some containers get accessed sequentially, while others exhibit a random access pattern. This means that the applications requests regard objects, which are not successive in the list structure, increasing the complexity of the access pattern. All DDTs were originally implemented using the STL Vector data structure. Vector is a dynamic array, offering fast random access, but also requiring large amounts of memory. The alternative to vector is a list. Using our framework, we explored the behaviour of Vdrift's DDTs testing 9 different implementations as presented in table 1. Vdrift utilizes various access patterns from sequential access of a container to pure random access pattern. Each container can have a different dominant access pattern, a property that renders it difficult for the designers to choose the right data structure for it. The solution usually followed is the implementation of all the DDTs using a single implementation, like the STL vector. However, as it is shown in Figure 5, choosing different implementations for each container individually can have a major positive impact on the number of memory accesses needed (which is also linked to performance) and the total memory footprint. The proposed solution is a combination of different implementations for each DDT.

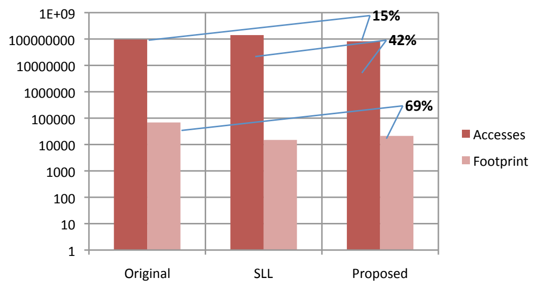

Fig. 5. Reduction in memory accesses and memory footprint comparing to the original implementation

In Figure 5 the proposed solution is compared to the original implementation (an exact model of vector), as well as a singly linked list, as this would be a decent choice for keeping the memory footprint in low levels. A 15% reduction in memory accesses and 69% reduction in memory footprint can be achieved compared to the original application. If all DDTs were implemented as singly linked lists instead of vectors, we could achieve even further reduction in memory footprint, but with a significant loss in performance. Our proposed combination of DDT implementations is 42% lower in memory accesses than the singly linked list implementation. For this case study the solution of the assignment problem is compared to the baseline solution (all heap data to be allocated at the SDRAM memory of the system). Out of the 37 DDTs 7 are to be placed in the scratchpad memory. This assignment manages to reduce the energy consumption to the memory subsystem by 51% and the mean latency per access by 50%.

5 Conclusions

It is evident that most new computing platforms are becoming more and more complex encapsulating multiple cores and reconfigurable elements. This offers the designers a multitude of resources. But not only is the complexity of the platform increasing. The same happens with the applications as they need to cope with dynamic events and have their resource requirements vary during the execution time. In order to cope with this dynamism applications rely in the usage of dynamic data. In this work an exploration framework and methodology is presented that searches the available design space of dynamic data type implementations and offers the designer a number of solutions and the potential trade-offs among them. Furthermore, decisions are taken on placing these

DDTs on the available on-chip memory resoures. The proposed approach has been evaluated using two applications. The first one is the scheduler part of an IEEE 802.16-based terminal broadband telecom system, achieving reduction of the energy consumption of 32%, whereas the second application is a 3D game, achieving reduction of the energy consumption of 51%.

References

1. Thoma, F., Kuhnle, M., Bonnot, P., Panainte, E., Bertels, K., Goller, S., Schneider, A., Guyetant, S., Schuler, E., Muller-Glaser, K., Becker, J.: Morpheus: Heterogeneous reconfigurable computing. In: Proc. of FPL. (Aug. 2007) 409–414
2. Marwedel, P., Wehmeyer, L., Verma, M., Steinke, S., Helmig, U.: Fast, predictable and low energy memory references through architecture-aware compilation. In: Proc. of ASP-DAC. (2004) 4–11
3. Atienza, D., Mamagkakis, S., Catthoor, F., Mendias, J.M., Soudris, D.: Dynamic memory management design methodology for reduced memory footprint in multimedia and wireless network applications. In: Proc. of DATE. (2004)
4. Antonakos, J., Mansfield, K.: Practical data structures using C/C++. Prentice Hall (1999)
5. Wood, D.: Data structures, algorithms, and performance. Addison-Wesley Longman Publishing Co., Inc. (1993)
6. Daylight, E., Atienza, D., Vandecappelle, A., Catthoor, F., Mendias, J.: Memory-access-aware data structure transformations for embedded software with dynamic data accesses. IEEE TVLSI **12**(3) (March 2004) 269–280
7. Bartzas, A., Mamagkakis, S., Pouiklis, G., Atienza, D., Catthoor, F., Soudris, D., Thanailakis, A.: Dynamic data type refinement methodology for systematic performance-energy design exploration of network applications. In: Proc. of DATE. (March 2006)
8. Papadopoulos, L., Baloukas, C., Zompakis, N., Soudris, D.: Systematic data structure exploration of multimedia and network applications realized embedded systems. In: Proc. of IC-SAMOS 2007. (July 2007) 58–65
9. Palermo, D.J., Banerjee, P.: Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In: Proc. of LCPC, Springer-Verlag (1996) 392–406
10. Dominguez, A., Udayakumaran, S., Barua, R.: Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. Journal of Embedded Computing (2005)
11. Kandemir, M., Kadayif, I., Choudhary, A., Ramanujam, J., Kolcu, I.: Compiler-directed scratch pad memory optimization for embedded multiprocessors. IEEE TVLSI **12** (March 2004) 281–287
12. Verma, M., Steinke, S., Marwedel, P.: Data partitioning for maximal scratchpad usage. In: Proc. of ASP-DAC, ACM Press (2003) 77–83
13. Verma, M., Wehmeyer, L., Marwedel, P.: Cache-aware scratchpad allocation algorithm. In: Proc. of DATE. (2004)
14. Udayakumaran, S., Dominguez, A., Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions. Trans. on Embedded Computing Sys. **5**(2) (2006) 472–511
15. Panda, P.R., Dutt, N.D., Nicolau, A.: On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. ACM Trans. Des. Autom. Electron. Syst. **5**(3) (2000) 682–704
16. Benini, L., de Micheli, G.: System-level power optimization: techniques and tools. ACM Trans. Des. Autom. Electron. Syst. **5**(2) (2000) 115–192
17. Khuri, S., Bäck, T., Heitkötter, J.: The zero/one multiple knapsack problem and genetic algorithms. In: Proc. of SAC, ACM (1994) 188–193
18. : Vdrift, `http://vdrift.net/`