# Run-Time Dynamic Data Type Transformations[1]

Lazaros Papadopoulos, Alexandros Bartzas and Dimitrios Soudris

School of Electrical and Computer Engineering
National Technical University of Athens, Greece
{lazaros, alexis, dsoudris}@microlab.ntua.gr

**Abstract:** In the last years, complex applications from various domains are implemented in embedded devices. These applications make extended use of the dynamic memory to store dynamically allocated data structures. The implementation of these data structures affects the performance and the memory usage of the embedded system. A methodology for selecting the appropriate data structures at design time is the Dynamic Data Type Refinement (DDTR) methodology. In this paper we present an extension to this approach, by presenting a methodology for adapting the dynamic data structure implementations to the requirements of the embedded system at runtime. By implementing the proposed methodology to a set of various applications from different domains, we achieve a dynamic memory size reduction up to 32%.

## 1 Introduction and Motivation

In the emerging market of embedded systems, an increasing amount of applications (e.g., 3D games, video-players) comes from the general-purpose domain and this software needs to be mapped onto extremely compact and mobile devices, which struggles to execute them. These complex applications hold very different restrictions regarding memory usage features, and more concretely are not concerned with an efficient use of the dynamic (heap) memory. Also, they receive input from and serve directly the end user of the embedded system. This means that the actions of the user have significant impact on the control flow of the algorithms in the applications, thus making the execution dynamic and event-driven.

This has led to an increased reliance on specific data 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. They can cope, in the most efficient way, with the variations of run-time needs (e.g., network traffic, user interaction, controller input) and the massive amounts of data processed and stored. The most common examples of these dynamically allocated data structures are single and double linked lists.

[1] This work is partially supported by the E.C funded FP7-ICT-2009-4-248716 2PARMA Project. Official Website: http://www.2parma.eu.

The data structure implementations affect both the memory consumption and the performance of the application, because each data structure has different characteristics in terms of memory size which it occupies and memory accesses needed to access the data (which affects the performance of the application and the energy consumption of the whole system) [10]. However, the implementation choices made at design time do not take into consideration runtime information that can change during the execution of the application. This information can be derived from the system (e.g. available memory) or from the application (e.g. the current memory size of the data structure).

We argue that the data structure implementations can change at runtime, according the runtime information such as the available memory and the performance requirements of the application. Thus, it is possible to achieve more efficient system resource utilization at runtime. The approach in [10] is the Dynamic Data Type Refinement (DDTR) methodology that provides one optimal data structure implementation for each metric under consideration (i.e. performance and memory footprint). This is accomplished at design time, by inserting the DDTR library interface in the application and then executing the application using some input traces. However, the data structure selected as optimal may not be actually the optimal under certain circumstances.

For example, consider the memory size of two data structure implementations: single linked list (SLL) and dynamic array. The amount of memory that the data structure occupies is obviously affected by the number of objects stored in each data structure and the additional information that the data structure uses to store the objects (i.e. in the case of SLL a pointer to the next object). As a motivation example, Dijkstra application [9] contains a data structure that stores 258 objects of 12 bytes each one, which are accessed by the algorithm. In order to optimize the application in terms of memory footprint, one can use the DDTR approach. According to DDTR the optimal data structure implementation is the SLL. Indeed, Figure 1 displays the comparison between the memory size of SLL and dynamic vector up to 258 objects. However, when the data structure holds between 193 and 256 objects, dynamic vector implementation requires less amount of memory to store the same objects. In this case, we can achieve better memory utilization by changing the data structure implementation from SLL to dynamic array. Thus, in this paper, we examine whether by performing such data structure adaptations, is possible to achieve better memory utilization.

The remainder of the paper is organized as follows. In Section 2, we describe some related work. In Section 3, we analyze the design methodology. In Section 4 our benchmarks are introduced and the experimental results are presented. Finally, in Section 5 we draw our conclusions.

## 2 Related Work

The authors of [1] present a dynamic data type refinement methodology. Using this methodology the designer can make tradeoffs between performance and energy consumption by selecting different data structure combinations from a library of such implementations.

In [2] a set of metadata is presented that could be used to analyze the behavior of the dynamically allocated data structures. The metadata targeted most the access pattern characterization in sequential and random, as well as some other behaviors like the frequency of constructions and copy-constructions.

The Dynamic Data Type Refinement methodology provides a set of Pareto points, at design time, which the designer can use to make trade-offs between performance, memory footprint and energy. Each Pareto point represents a data structure (or a combination of data structures). These data structure implementations are set at design time and remain the same during the execution of the application. In this paper we present a new approach: We argue that by changing the dynamic data type implementation at runtime, we can achieve better resource utilization. The reason that our approach achieves better results that the DDTR, is the fact that it takes into account runtime information that determine which dynamic data type (or combination) is optimal at each point of the execution of the application.

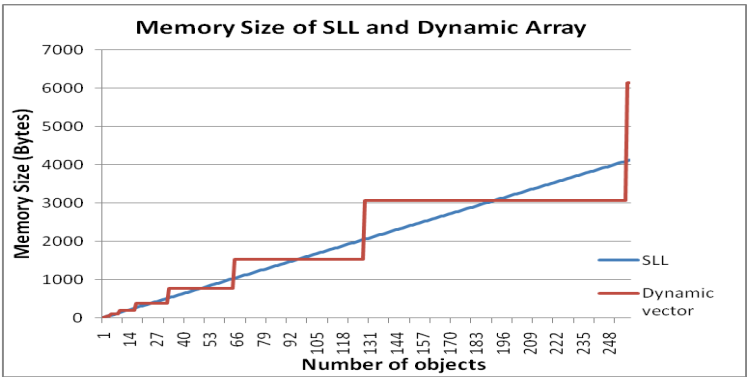


Figure 1: Memory size evolution of single linked list and dynamic array

Several approaches have been proposed for runtime adaptation of applications. For example [3] concentrates in the software adaptation using dynamic change in application components. The authors introduce a framework that monitors changes in the execution environment of applications and performs a dynamic recomposition of the application components, when significant changes in the environment take place.

A few works focus on the dynamic configuration of parallel applications. For instance, in [4] is described a runtime optimization approach that allows the automatic on-the-fly reconfiguration of the parallel simulation code for increasing the performance of the application. The dynamic adaptation is performed by collecting and combining runtime information from the application with static parallel performance models.

There are several tools that focus on runtime software adaptation. For example, Pin [5] is a dynamic binary instrumentation tool that performs in process-level and allows the modification of application instructions prior to the instruction execution. Similar tools, that operate in similar fashion, are Strata [6] and DELI [7].

Runtime adaptation has also been proposed for task migration. For example in [8] is described a mechanism for reducing task migration latency in multi-core architectures, by performing trade-offs between latency and bandwidth. Runtime adaptivity is achieved by using a latency/bandwidth trade-off parameter, which controls the trade-offs. All the aforementioned approaches that refer to the runtime adaptation can be used along with our approach.

# 3 Methodology Overview

The runtime information that our methodology takes into account is the number of objects stored in each data structure. As mentioned before, the amount of data in the data structure affects both the memory size and, in most cases, the number of accesses needed to access each object. We take advantage of this fact, in order to achieve better memory utilization at runtime.

## 3. 1 The modes

Since different implementation types for each data structure exists, we define each different combination of data structure implementations of the application as a different *mode* at which the application can run. For each mode the following information is calculated at design time:

- The size of the data structures for each number of objects
- The memory size and the performance overhead of the transition to each other data structure type implementation for each number of objects.

Table 1: Example of a mode

| **Data Structure** | **Implementation** | **Number of objects** | **Memory size** | **Transition to** | **Performance overhead** | **Memory size overhead** |
|---|---|---|---|---|---|---|
| DDT 1 | SLL | 150 | 128 | DLL | 300 | 200 |
| | | | | Dynamic Array | 320 | 400 |
| DDT 2 | DLL | 100 | 140 | Dynamic Array | 320 | 450 |

Using the aforementioned information, it is possible to compare the modes and keep only the optimal ones, by discarding those for which better one already exist in terms of memory size for a specific number of objects. Modes that violate the designer constraints are also discarded. The modes are defined at design time and the runtime manager handles the transition between the available ones at runtime.

Table 1 presents an example of a mode for an application that uses 2 data structures. The first one is considered to be a single linked list that holds 150 objects and the second a double linked list where 100 objects are stored. For the first data structure a possible transition to double linked list and dynamic array is considered and the necessary overhead information is presented. For the second one, only the transition to single linked list is presented, since any other transformation is supposed to have intolerable overhead. Another mode, for example, would be DDT1 implemented as a vector and DDT2 as a SLL for a different number of objects. The conditions to consider a mode available at runtime are the following:

- The memory size of the mode plus the overhead of the transition is less that the memory size of the mode corresponding to the DDTR data structure combination.

- The performance and the memory footprint overhead of the transition to the specific mode from some other mode, do not violate the constraints set by the designer.

To evaluate each mode, we use as a reference mode, the one corresponding to the DDTR methodology. This is done, in order to prove in this paper that using this methodology we can achieve better results in terms of memory utilization than using only the DDTR methodology.

The condition to make a transition at runtime is the following: The memory footprint of the target mode to be less that the memory footprint of the current mode.

### 3.2 Methodology Description

The methodology is composed of the following three steps and is presented in Figure 2: 1) DDTR exploration; 2) Insertion of the necessary data structure information to the design time manager that automatically detects the available modes; and 3) Execution of the application along with the runtime manager.

The DDTR exploration is exhaustively described at [10]. By implementing the DDTR methodology, the designer obtains a set of optimal data structure implementations, in terms of memory footprint and performance. From this set, the one provided to our tool is the combination that is better in terms of memory footprint. The inputs of the design time manager are the following:

- *Input from the DDTR exploration*. The optimal dynamic data type implementation for each data structure of the application.

- *Object size of each data structure*. This information is necessary to calculate the size of each data structure and obtain the set of available modes.

- *Maximum number of objects for each data structure*. This information is obtained during the DDTR exploration methodology. Although the application is dynamic, and it is not possible to know the exact maximum number of objects in each data structure, the traces used in the DDTR exploration phase, can provide an estimation of the maximum number of objects each data structure holds.

*Designer constraints*. In the case of real time applications performance constraints may exist. Thus, since the transition from one mode to another causes a delay, these constraints determine whether a mode or a transition to a mode is available or not.

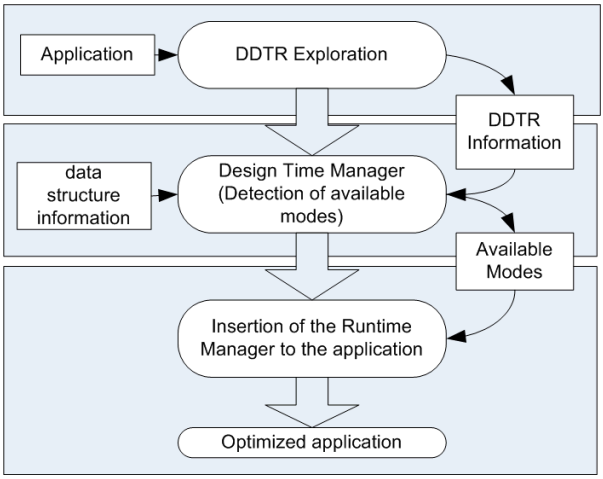


Figure 2: The proposed methodology

The designer inserts the aforementioned information to the design time manager tool. The tool, using the aforementioned inputs, produces a set of candidate modes in which the application can run. Each mode is a different set of dynamic data structure implementations of the application. For all the available modes, which are propagated to the runtime manager, the size of the data structure plus the overhead of the transition to this mode is less that the size of the dynamic data structure selected as optimal by the DDTR methodology. All the dynamic data structure combinations (i.e. modes), for which the aforementioned condition is not valid, are discarded. Thus, from the pool of all the candidate modes, only the optimal ones are provided to the runtime manager.

The input of the runtime manager is the set of the available modes selected by the design time step and the current number of objects in each data structure of the application. The application runs along with the runtime manager. The runtime manager handles the transition between the modes created at the design time phase. Each time an operation takes place in a data structure, the runtime manager checks if transition to another mode is possible.

### 3.3 The overhead of the methodology

In this subsection we examine the overhead of the presented methodology. The overheads are the following:

- *Design time exploration to obtain the available modes*. This overhead is rather trivial, since the whole process is based on mathematical calculations. The duration depends on the number of data structures and the maximum number of objects that each one holds. For example, for 5 data structures of 10,000 objects each one, the exploration takes less that 1 minute.

- *Increased code size (Memory size overhead)*. The code size of the application increases, since the runtime manager is also compiled along with the application. This overhead less than 1 KB of memory, so can be considered trivial, especially for large applications.

- *Performance overhead (runtime overhead)*: The fact that the runtime manager checks whether a better mode or not exists each time a data structure operation takes place, causes a delay in the execution of the application. However, as shown in the experimental results section, this delay is rather trivial, since every calculation needed is made at design time. Also, the routine that the runtime manager uses to check the available modes is very simple.

- *Overhead of mode changes (runtime overhead)*:

    - Performance overhead: This overhead is affected by the number of objects and the type of source and target data structures. (E.g. It is more time consuming to transfer or remove data from a dynamic vector, than from a single linked list, since the dynamic vector is resized). The performance overhead is known at design time. If the overhead is tolerable, then the corresponding mode is provided to the runtime manager. Otherwise, it is discarded.

    - Memory footprint overhead: During the transfer of data from one data structure to another, there are 2 data structures (the source and the destination) that coexist in the memory. Thus, there is a memory size overhead, which is affected by the type of source and destination data structures and the number of objects to be transferred. However, this overhead is calculated at design time and only if it is tolerable, the corresponding mode is forwarded as an input to the runtime manager.

Our tool precalculates the aforementioned overheads and ensures that a mode is available only when the destination data structure characteristics and the overhead of the transformation are better (in terms of memory footprint or performance) from the current data structure.

### 3.4 The tool

The tool which implements the described methodology is composed by two parts. The first one is the design time manager which provides the available modes to the runtime manager. The runtime manager handles the transitions between the different modes.

- *Design time manager*: To use the design time manager the designer provides the necessary input to the tool in text mode. The manager is composed by a set of routines which make the necessary calculations to generate all the possible modes. Then, the manager automatically produces the set of the available modes.

- *Runtime manager*: To use the runtime manager, the designer sets the provided interface to the data structures of the application. This is a straightforward process, especially, if the application uses the STL data structures. The runtime manager contains a set of dynamic data structures, along with a routine which decides when to change the current mode, taken as input the current number of objects of each data structure.

Table 2: Cumulative experimental results

| App. name | Maximum memory size reduction | Average memory size reduction | Maximum Memory footprint overhead | Average memory footprint overhead | Perf. overhead | Code size increase |
|---|---|---|---|---|---|---|
| Dijkstra | 25.1% | 8.6% | 50.9% | 12.16% | 22.4% | 50% |
| 2D Game | 30.3% | 16.4% | 11.3% | 7.1% | 3.4% | 27% |
| 3D Engine | 32% | 5.23% | 10.3% | 6.58% | 18.3% | 14% |
| 3D Game | 25% | 3.32% | 65.4% | 30.38% | 22% | 3% |

## 4 Experimental Results

To validate our approach, we have chosen a wide range of applications from various application domains. By implementing each step of the methodology, we calculated the memory size gains, as well as the overhead added to each application by the tool. More specifically, we calculate the maximum and the average memory size reduction during the execution of the application, in comparison with the memory size of the data structure implementations suggested by the DDTR. All memory size information provided in the experimental results is the size of all data structures of each application.

As far as the overhead is concerned, we calculated the memory footprint overhead which exists when a mode change takes place at runtime. This overhead exists only during the mode change process and is eliminated after the end of the transformation. Performance overhead is compared with the performance of the original application. Finally, the code size shows the increase in the size of the executable of the application due to the tool. The cumulative results of our case studies are presented in Table 2.

### 4.1 Dijkstra application

The first test case is Dijkstra algorithm taken from the Mibench Suite [9], which stores network nodes in a data structure. As mentioned earlier, the optimal data structure implementation in terms of memory footprint according to the DDTR approach is the single linked list. Implementing the DDTR solution (i.e. the mode that corresponds to the DDTR solution), no mode changes take place. However, using the adaptive approach, a number of data structure implementation transformations are being made during the execution of the application, which results in a memory size reduction up to 25.1% in comparison with the memory size occupied by the application using single linked list during the whole execution, (which is the solution proposed by the DDTR approach). This is achieved by implementing 63 mode changes during the execution. The memory size overhead due to mode changes (shown in table 2) is not presented in figure 3. It is considered tollerable, thus only the data structure memory footprint evolution is shown. Figure 3 displays the memory size evolution of the application during the whole execution. It can be seen that the memory size using the adaptive approach can be higher that the memory size using the DDTR approach, at some points of the execution of the application. This is because the transformation to the optimal data structure implementation in terms of memory size has very low benefits or is intolerable, according to the constraints set by the designer. Figure 4 shows the memory size reduction achieved by using the adaptive approach. The maximum memory footprint overhead that takes place during the transformation is 50.9%, which can be considered relatively high. However, this overhead can be decreased by the designer, by setting the appropriate constraints, with a corresponding decrease to the memory size reduction.

### 4.2 Comboling application

A 2D game named Comboling contains a grid of tiles, which are stored in a singly linked list that is filled with tile elements (thus, the memory size is constantly increasing) and accessed in a random pattern [11]. The optimal data structure implementation according to the DDTR approach is the single linked list. Figure 5 displays the memory size comparison between the solution proposed by the DDTR against our approach. Using the adapting approach 8 transformations take place and 30.3% memory size reduction during the execution of the application is achieved. For instance, when the size of the data structure is between 786 and 1024 bytes the vector solution provides less memory size than the single linked list. The memory size gains are presented in Figure 6. The main overhead of our methodology in this application is the code size that seems relatively high. However, the code size of Comboling is less than 1 KB, so the overhead added by our tool can be considered trivial.

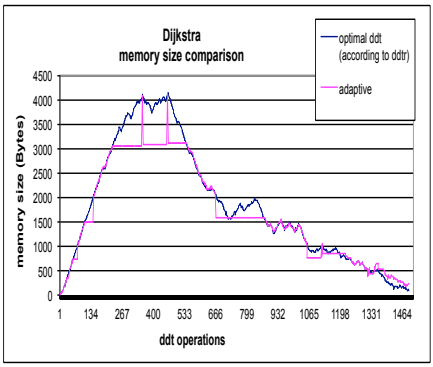


Figure 3: Dijkstra application – run-time evolution of memory footprint

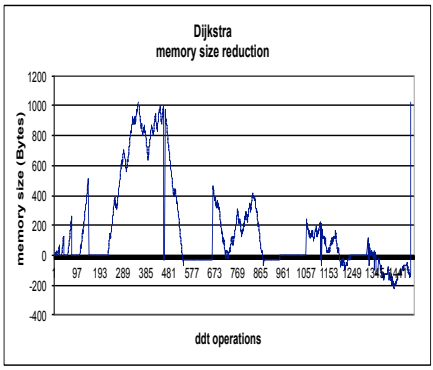


Figure 4: Dijkstra application –reduction of memory footprint

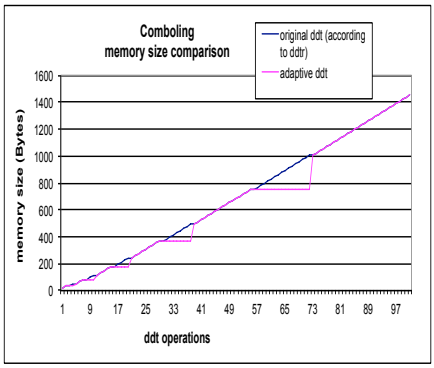


Figure 5: Comboling application – run-time evolution of memory footprint

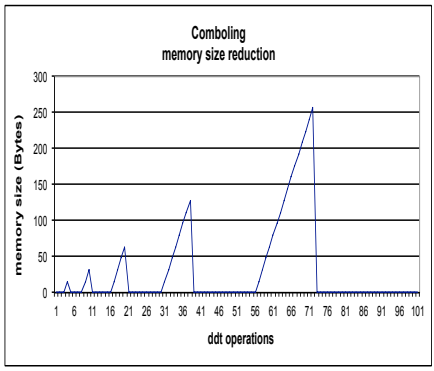


Figure 6: Comboling application – reduction of memory footprint

### 4.3 Simblob application

Simblob is a 3D environment creation engine that utilizes vectors to hold its dynamic data [12]. The optimal data structure in terms of memory footprint according to the DDTR approach is the single linked list. Figure 7 shows the comparison between the DDTR and our approach. Implementing the adaptive approach,4 transformations take place and the maximum memory size reduction is 32%. Figure 8 displays the memory size gains. It can be seen by Table 2 that the overheads of our methodology in this test case are relatively low.

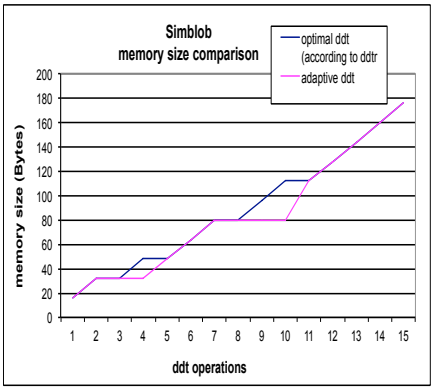


Figure 7: Simblob application – run-time evolution of memory footprint

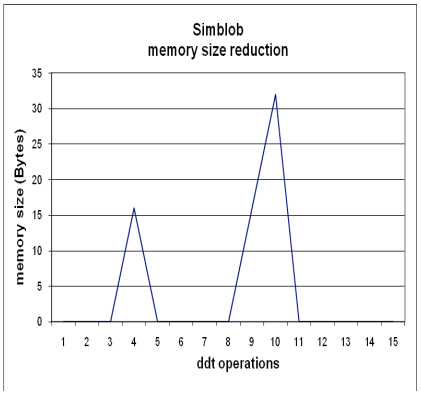


Figure 8: Simblob application – reduction of memory footprint

### 4.4 VDrift application

Vdrift is a 3D open source racing game with realistic physics [13]. The application uses vectors to store its dynamic data for graphics, physics and collisions. The DDTR approach suggests single linked lists as the optimal data structure implementations in terms of memory footprint. Figure 9 displays the memory size comparison between the DDTR approach and the adaptive one. During the adaptive approach 23 mode changes take place and the maximum memory size reduction achieved is 25%. The memory gains are presented in Figure 10.

The memory footprint overhead of the transformations is relatively high, as can be seen in Table 2. However, as mentioned with the Dijkstra test case as well, the overhead can be decreased by the designer, by setting the appropriate constraints. It should be taken into account that in this case, the amount of memory reduction will be also decreased, since some mode transformations will not take place.

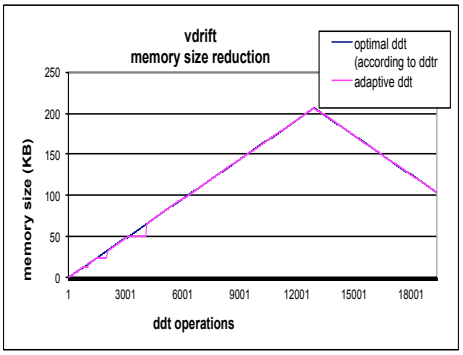


Figure 9: VDrift application – run-time evolution of memory footprint

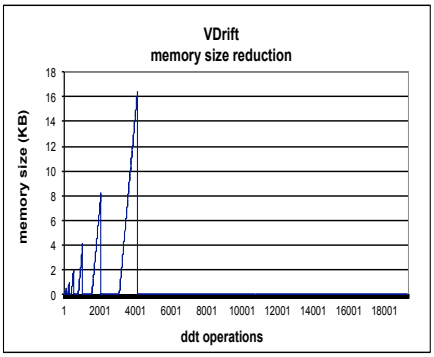


Figure 10: VDrift application – reduction of memory footprint

## 5 Conclusions

In this paper we presented a methodology for adapting the dynamic data structure implementations of an application to the runtime environment at which the embedded system executes the application. We proved that it is possible to achieve better dynamic memory utilization by using a runtime manager that adapts the data structure implementations by adding a tolerable overhead. Our future work addresses the extension of the data structure implementation library of the runtime manager, as well as the further reduction of the overheads of the proposed methodology.

## References


[1] Alexandros Bartzas, Stylianos Mamagkakis, Georgios Pouiklis, David Atienza, Francky Catthoor, Dimitrios Soudris, and Antonios Thanailakis, "Dynamic data type refinement methodology for systematic performance-energy design exploration of network applications," In Proc. DATE, 740-745, EDAA, 2006

[2] Alexandros Bartzas, Miguel Peon-Quiros, Christophe Poucet, Christos Baloukas, Stylianos Mamagkakis, Francky Catthoor, Dimitrios Soudris, and Jose M. Mendias, "Software metadata: Systematic characterization of the memory behaviour of dynamic applications," in J. Syst. Softw. 83, 6, June 2010, Elsevier

[3] Arun Mukhija and Martin Glinz, "Runtime Adaptation of Applications Through Dynamic Recomposition of Components," In Proc. of ARCS, 124-138, Springer, 2005

[4] Xiaoji Ye and Peng Li, "On-the-fly runtime adaptation for efficient execution of parallel multi-algorithm circuit simulation," in Proc. of ICCAD, 298-304, IEEE, 2010

[5] Kim Hazelwood and Artur Klauser, "A dynamic binary instrumentation engine for the ARM architecture," in Proc. of CASES, ACM, 261-270, 2010

[6] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in Proc. of GCO, IEEE Computer Society, 36-47, 2003

[7] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher, "DELI: a new run-time control point," in Proc. MICRO 35, IEEE Computer Society Press, 257-268, 2002

[8] Jahn, J.; Faruque, M.A.A.; Henkel, J., "CARAT: Context-aware runtime adaptive task migration for multi core architectures," in Proc. of DATE, 2011

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: a free, commercially representative embedded benchmark suite", in Proc. of 4th IEEE Workshop Workload Characterization, 10-22, 2001

[10] L. Papadopoulos, C. Baloukas, N. Zompakis, D. Soudris, "Systematic Data Structure Exploration of Multimedia and Network Applications realized Embedded Systems", in Proc. of IC-SAMOS, 58-65, 2007

[11] http://www.Planet-Source-Code.com/vb/scripts/ShowCode.asp?txtCodeId=3222&lngWId=10Sim

[12] http://sourceforge.net/projects/simblob

[13] http://vdrift.net/