Platform-aware Dynamic Data Type Refinement Methodology for Radix Tree Data Structures

Thomas Papastergiou*, Lazaros Papadopoulos†, and Dimitrios Soudris‡
School of Electrical and Computer Engineering
National Technical University of Athens, Greece
Email: *el09149@central.ntua.gr, †lpapadop@microlab.ntua.gr, ‡dsoudris@microlab.ntua.gr

***Abstract*—Modern embedded systems are now capable of executing complex and demanding applications that are often based on large data structures. The design of the critical data structures of the application affects the performance and the memory requirements of the whole system. Dynamic Data Structure Refinement methodology provides optimizations, mainly in list and array data structures, which are based on the application's features and access patterns. In this work, we extend various aspects of the methodology: First, we integrate radix tree optimizations. Then, we provide a set of platform-aware data structure implementations, for performing optimizations based on the hardware features. The extended methodology is evaluated using a wide set of synthetic and real-world benchmarks, in which we achieved performance and memory trade-offs up to 29.6%. Additionally, Pareto optimal data structure implementations that were not available by the previous methodology, are identified with the extended one.**

I. Introduction

Nowadays, we experience the constantly increasing computational power of embedded systems. Applications that were previously executed in High Performance Computer (HPC) systems, are now implemented in embedded devices. Embedded servers, multicore heterogeneous architectures that integrate both embedded cores and FPGA programmable logic [1] execute complex and demanding applications that require high computational power. The term High Performance Embedded Computing (HPEC) has been recently used to describe embedded devices with very large processing power, used mostly in aerospace and military applications [2]. The energy efficiency of these systems makes them competitive in the market, over the large power-hungry systems and promising solutions towards the development of the exascale computing [3].

Examples of modern application domains that are executed in high-end embedded devices are databases and stream processing. Such applications are expected to store and process large amounts of data. The data are normally stored in complex data structures, in order to be processed efficiently, with low latency and high throughput. The data inserted and stored in these data structures arrive often in unpredictable ways, thus resulting in highly dynamic systems.

Apparently, these applications rely heavily on data structures. Different data structures favor different application access patterns. Therefore, the data structure implementation greatly affects the performance, the memory utilization and the energy consumption of the whole system. However, the complexity of modern applications makes the data structure design and optimization an error-prone and time-consuming task, especially, when there is a large set of metrics that should be optimized at the same time by the developer. Therefore, there is a need of tools that automate the whole process of selecting and optimizing the data structures of the application.

Dynamic Data Structure Refinement methodology(DDTR) is a set of tools that assist the designer in selecting the optimal data structure for the application under optimization [4]. The methodology proposes a design time exploration, aiming to select the optimal dynamic data structure implementations, from a set of ready-to-use data structures, for dynamic applications under optimization. The main idea is to run the application once for each different data structure implementation and collect profiling result for each execution. Then, a set of Pareto points assists the designer to select the optimal implementation.

In this work, we propose several extensions to the DDTR approach to address its limitations: First, we argue that the optimization of the data structures is not only related with the application access pattern, but also with the underlying hardware specifications in which the application is executed. Therefore, we propose a new set of platform-dependent data structure implementations. Additionally, we extend the data structure library in which the methodology is based, with tree data structures.

The goal of the DDTR methodology is to provide to the application developer a set of tools for designing and optimizing the data structures of dynamic applications. In this work we focus on radix tree data structures, which are used for storing large amounts of data (e.g. IP addresses, dictionaries, etc.). The DDTR library is extended to cover five different radix tree implementations, in which two of them are platform-aware, in the sense that they take advantage of the memory cache, in case it is a part of the underlying platform memory hierarchy. The methodology is evaluated using a wide set of synthetic and realistic datasets and two modern embedded platforms with different hardware specifications.

The rest of the paper is organized as follows: In Section II we provide a sort description of related approaches. The DDTR methodology, along with its extensions is presented in Section III. In Section IV we provide details about the cache conscious data structure implementations. The experimental results are presented in Section V, while in Section VI we draw our conclusions.

978-1-4673-7311-1/15/$31.00 ©2015 IEEE

II. Related Work

The optimization of data structures has been extensively studied in the literature [5], [6], [7]. However, it focuses mainly to static compile-time data allocation optimizations. Our work focuses on data structures of dynamic applications, in which the access pattern may change at runtime, along with the dynamic behavior of the application.

Several radix tree data structure implementations have been proposed to optimize the performance or the memory requirements of the application [8], [9]. The performance of these data structures depends on the dynamic characteristics of the application and more specifically, on the access pattern and on the underlying platform specifications. In other words, different dynamic characteristics of applications would result in different throughput, memory utilization and energy consumption results for each one. Therefore, such data structure implementations can be integrated in our methodology for exploration and evaluation.

There exist several methodologies for the implementation of cache conscious data structures. For instance, the *ccmalloc* library implements a wrapper around malloc to optimize the memory allocation of newly created elements and increase locality [10]. The coloring technique attempts to map contemporaneously accessed elements to non-conflicting cache regions [11]. The clustering technique attempts to pack in a cache block the data structure elements that are accessed contemporaneously. [10]. In this work we extend the DDTR methodology by integrating cache conscious data structures in it that are implemented based on the clustering technique.

This work is mostly related to [4], which describes the DDTR approach, focusing mainly on multimedia and network applications. The library of data structures in which it is based, is limited to simple platform-independent data structures, like lists and arrays. On the contrary, in the present work, the methodology is extended to trees and platform-dependent data structure implementations.

III. Methodology Description

In this Section we provide a short description of the DDTR methodology focusing on the new features we provide in order to extend the methodology to radix trees and to platform-specific data structure implementations.

A. Description of DDTR and Limitations

The flow of the DDTR methodology is presented in Fig.1. It is composed of three steps which are briefly described in this subsection. More information can be found in [4]. The core of the methodology is a library of ready-to-use data structure implementations in C/C++. The library contains an STL-like interface that is inserted in the source code of the application and it replaces its data structures with the ones of the library. Therefore, all the data structure operations, (e.g. push, pop, find etc.) are forwarded in the library's data structures. The second step of the DDTR methodology is the exploration phase. The application is executed once for each different data structure implementation of the library and the execution time and memory utilization are logged. In the last step, these results are presented in Pareto curves. Thus, the designer can select the data structure implementations which fit the design constraints.

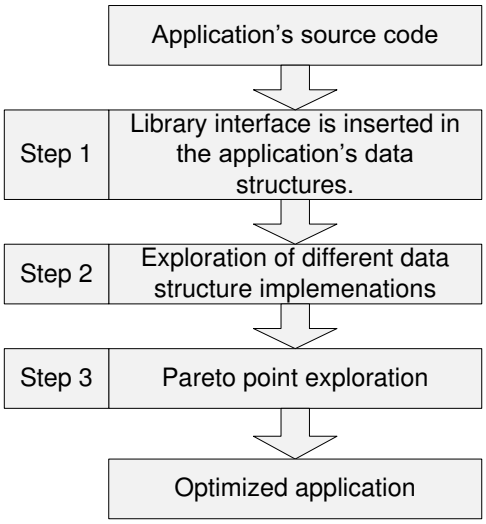

Fig. 1. Flow of the DDTR methodology.

TABLE I. Qualitative Comparison between the existing and new DDTR approach

	Existing DDTR	New DDTR
Library of Data Structures	list, arrays	list, arrays, radix trees
Platform-awareness	Platform-independent implementations	Cache-conscious implementations
Metrics	Exec. time, memory	Exec. time, memory, throughput, latency
Evaluation	x86	ARM-based, Myriad embedded systems

The DDTR methodology, as presented in [4] has a number of limitations, which are summarized, as follows:

- It contains a relatively small number of simple data structures, mainly list and array implementations. More complex data structures, like trees, were never evaluated.
- The DDTR approach was developed under the assumption that data structure implementation constraints are related only to the application dynamic behavior. Therefore, it contains only platform independent data structure implementations. They are built at high level of abstraction and consider a plane memory hierarchy. As a result, they do not consider any memory specification, like caches.
- Finally, the profiling component of the DDTR methodology, logs only execution time and memory utilization. Other important metrics, like throughput and latency of operations are not available.

We address these issues by providing a set of extensions to the methodology, which can make it useful in modern complex applications executed in high-end hardware platforms.

B. New DDTR approach

In these subsection we describe a new approach to the DDTR methodology and discuss the way we faced the limitations that were previously described. A qualitative comparison

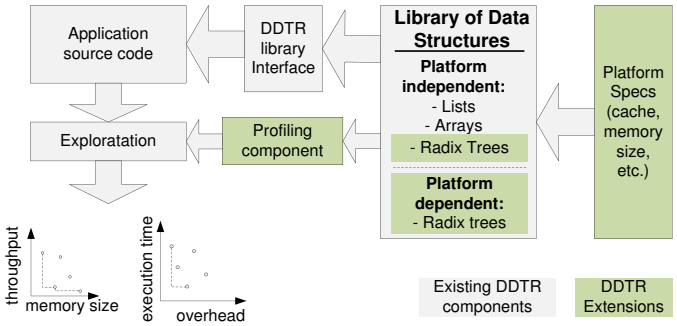

Fig. 2. Tools supporting the DDTR methodology. The new tools and extensions are highlighted.

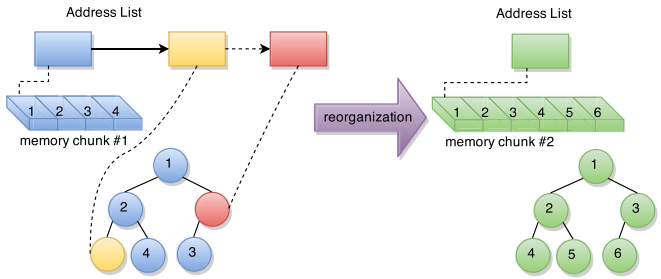

Fig. 3. Tree reorganization process of the cache conscious data structures.

between the existing and the new DDTR approach is summarized in Table I.

As mentioned in the previous subsection, the existing DDTR methodology focuses on lists and array implementations. The new DDTR approach presented in this work extends the existing one by integrating tree data structures, and more specifically, radix tree implementations. Radix trees are extensively used in applications for storing and handling strings, like dictionary data and IP addresses. Several implementations of the radix trees have been proposed in the literature: Apart from the Patricia trie implementation, Ternary tree and Hat-trie [8] are other alternatives, which are integrated in the new DDTR library.

As stated before, in the new DDTR approach, we argue that the optimal data structure implementation is not only affected by the application dynamic behavior (e.g. the access pattern, the dominant operations etc.), but also by the system's architecture and mainly by the existence of cache memory. Therefore, we have extended the data structure implementations of the library by providing implementations that exploit the cache memory. As described in next section in more detail, both the Patricia trie and the Ternary tree can be optimized to become cache conscious. Therefore, the library provides a total of five different radix tree implementations.

The existing DDTR approach provides information for performance and memory utilization of each data structure implementation, during the exploration phase. The new approach extends the profiling information with new metrics, like throughput and overhead information for the cache conscious implementations, which are important for data structures used for storing large amounts of data.

Finally, in contrast with the existing DDTR methodology, the new one is evaluated in real modern embedded devices. Myriad co-processor [13] and a Freescale i.MX6 [12] are two embedded chips with different memory specifications. Therefore, in this work, we extend the DDTR approach, by showing how different hardware characteristics, along with the application dynamic behavior, affect the optimal data structure implementation.

Fig.2 presents the tools composing the DDTR methodology, along with the new tools that extend its features. First, as stated earlier, the extended library contains a new set of radix tree data structures. There is a total of five different radix tree implementations, in which two of them are cache conscious. Additionally, there is an updated profiling component, which logs information about the throughput and the overhead of cache conscious data structures (apart from the execution time and the memory utilization that the existing component logs). Finally, we developed a new component which provides the DDTR methodology with hardware-related information, such as the existence of cache memory. This component enables or disables specific DDTR implementations in the exploration phase. For example, there is no reason to evaluate cache conscious implementations in an embedded system that does not contain cache memories. In this case, the relevant implementations are excluded automatically in the exploration phase, thus reducing the exploration time.

IV. Cache Conscious Radix Tree Implementations

In this section, we describe the algorithms we used for the cache conscious (i.e. platform-aware) radix tree implementations. The two data structures that were optimized in order to be converted to cache conscious are the Ternary tree and the Patricia trie. Another cache optimized data structure that we integrated in the extended DDTR library is the Hat-trie [8].

Each Ternary tree node holds a string character, along with three pointers: (*greater* ,*equal* ,*smaller*). As the string is consumed one character at a time, the nodes are traversed in the following fashion: if the current character is equal to the character of the node the *equal* pointer is followed; otherwise the *greater* or *smaller* pointer is followed, accordingly.

The Patricia Trie is a combination of radix and crit-bit tree. Contrary to the Ternary's tree nodes, Patricia nodes store a complete string, alongside with an integer that marks a specific bit of the string. As a result, instead of examining the whole string, only the critical bit is examined. If the bit is set, the right pointer is followed, otherwise the left one is followed. Insertions are more complex than in the Ternary tree, since the *crit-bit* must be examined before inserting the node to the correct position in the tree.

The cache conscious implementations of the Patricia and the Ternary tree we implemented as part of the extended DDTR library are based on the compression technique, which is described in [10]. The compression technique enables elements to be clustered to the same cache block, by separating the node fields (i.e. structure's data) in those which are accessed frequently and those which are not and by clustering them in the same cache line. Based on this idea, in our cache conscious implementations, we cluster adjacent tree nodes in the

same cache line. Thus, we benefit from hardware prefetching techniques, which most modern processors avail. Additionally, adjacent cache lines are fetched in the cache after cache misses, which also increased the application performance. Even if prefetching techniques are not available, a performance increase is almost always guaranteed as the spatial locality is as strong as possible.

However, the memory fragmentation that occurs during the tree node updates (i.e. insertions and deletions of tree nodes), may eliminate the advantages of the clustering. One simple solution is to initiate a complete tree memory reorganization at specific moments of the execution time. This approach requires the allocation of the following data structures: An array with memory size equal to that of the tree, in which the existing tree nodes are allocated in order to increase the memory locality and a list (*address-list*), which provides the necessary memory chunks for the new nodes that are being created after the reorganization.

To maximize the locality, the adjacent tree nodes are allocated in adjacent positions in the array, so as to reduce the number of possible cache misses. Therefore, the lookup tree operations are expected to be very efficient in the cache conscious implementations.

The reorganization process is the following: Each time a reorganization takes place, a new array is being allocated with size equal to the existing tree size. Then, the tree nodes in the existing array, along with the nodes in the *address-list* are copied in the new array. Finally, the old array is deallocated and the *address-list* becomes empty.

In respect with the *address-list*, which provides memory chunks for the newly created nodes, we performed the following optimization: Instead of providing memory chunks of size equal to a single tree node, we allocate large enough chunks in which more than one tree nodes can be stored. Therefore, the number of the *address-list* elements is reduced. Additionally, this approach requires less *malloc* calls, which often leads to higher performance. Also, the reduced number of *malloc* calls leads to less memory overhead, due to the reduction of the extra system memory that is required by the internals of the OS memory allocator.

The tree reorganization process is further illustrated with an example in Fig.3. Before the tree reorganization, there exist an array with 4 tree elements (memory chunk #1)and the *address-list*, which holds the two elements that were created after the last reorganization. When a specific number of update operations has occurred (i.e. insertions and deletions), the tree needs to be reorganized, in order to increase its locality. In our experiments the reorganization takes place when the number of modifications (insertions and deletions) reach the number of nodes the tree had at the previous reorganization (i.e the tree size has been doubled since the last reorganization). To reorganize the tree, a new array with size equal to the tree size is allocated (memory chunk #2). Then, it is being filled with the adjacent memory blocks of the neighboring nodes of the tree (i.e. with the memory blocks of the memory chunk #1 and the ones stored in the *address-list*). Finally, the previous *address-list* and the previous array are deallocated and a new empty *address-list* is created.

This optimization technique has the obvious drawback of the memory overhead, which can be observed as spikes in the application's memory footprint during the tree reorganization. The spike is observed when a new array is allocated in the memory and ends when the old array is deallocated (i.e. when the memory chunk #1 and memory chunk #2 of 3 coexist in the memory). The execution time overhead depends on the number of tree nodes that are copied, so it is related with the tree size. These overheads may or mayhe two data structures t not be acceptable, depending on the system's memory availability and the real-time constraints. Nevertheless, we consider this approach a fair performance-memory trade-off, since our experiments show large execution time improvements.

Another advantage of this method, is that it can be applied in all pointer-based data structures, in which the node size is comparable to that of the cache line and (aside from the memory footprint spikes), it has almost the same memory requirements with the non-cache optimized versions of the data structures. Based on this technique, we implemented two versions of the aforementioned data structures: a cache conscious Patricia trie and a Ternary tree, which we integrated in the DDTR library.

Hat-trie is based on the idea that a node can be either a simple trie-node or a bucket. Trie-nodes have a flag, like the Ternary node, but instead of three pointers, it contains an array of pointers. The leaves of the Hat-trie are usually buckets that contain part of the strings. A bucket is a practically a hash table in the form of an array to take advantage of spatial locality and to give the data structure efficient cache conscious behavior.

V. Evaluation of the Extended DDTR approach

We evaluated the extended DDTR methodology using two modern embedded chips and a set of real-world benchmarks, along with synthetic ones. First, we describe in brief the specifications of the two chips and the experimental setup. Finally, we proceed with the description of the results.

A. Experimental Setup

Our goal was to evaluate the extended DDTR methodology in platforms with different memory hierarchies. The first platform we used for the evaluation is the Freescale i.MX6 [13], which is a 4-core ARM-based embedded chip. It contains two levels of cache memory and an 1GB DDR3 RAM. The second one is the Myriad chip designed by Movidius Ltd. and normally acts as a low-power co-processor in mobile devices, smartphones and wearable gadgets [12]. It integrates 8 VLIW cores, which access an 1MB shared SRAM memory. In contrast with the Freescale chip, no cache exists between the cores and the shared memory.

The evaluation of the extended DDTR methodology was made through a variety of synthetic and real-world datasets. The synthetic benchmarks consist of custom testcases, of 10 million operations each one. The real-world benchmarks are an IP and a set of dictionary datasets. The IP dataset is composed of requests made to servers for the 1998 World Cup and was taken from the Internet Traffic Archive [14]. It is composed of three million IP addresses, which 4% are unique. The dictionary datasets are taken from real dictionaries of various languages, utilized in the WinEdt text editor [15] and contains unique string entries. In the Myriad experiments, we run only

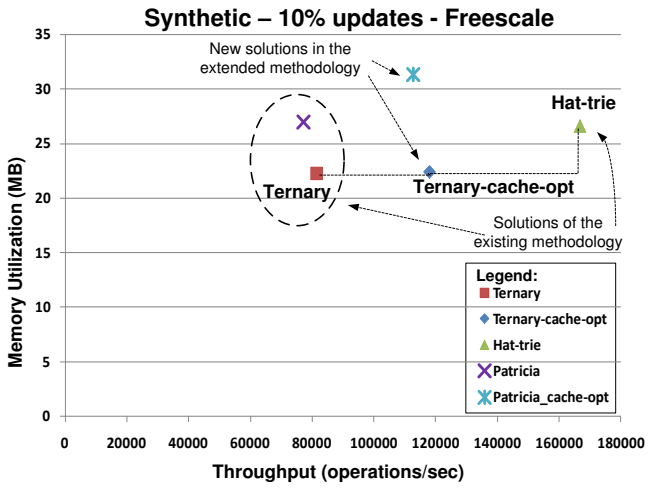

Fig. 4. Throughput vs. memory utilization of the synthetic benchmark with 10% update operations in the Freescale board.

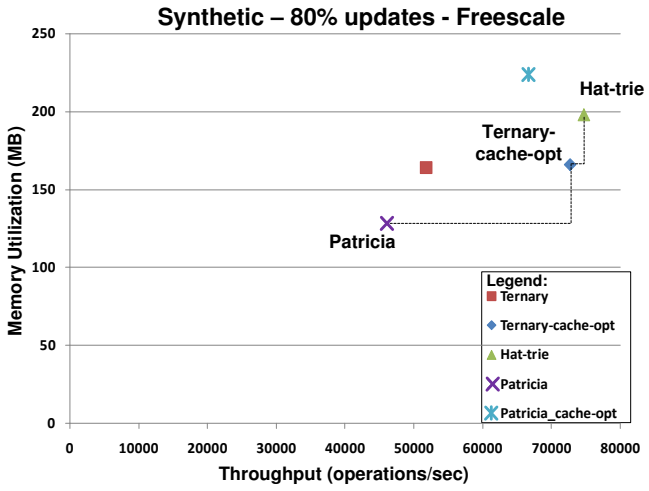

Fig. 5. Throughput vs. memory utilization of the synthetic benchmark with 80% update operations in the Freescale board.

a part of each benchmark, due to low on-chip memory size that is restricted to 1MB.

The metrics we used to evaluate each implementation are the throughput (i.e. operations per second) and the memory footprint. Additionally, we present results in respect with the performance and memory overhead of the cache conscious implementation that occurs during the tree reorganization.

B. Experimental Results

In this subsection we present and analyze the experimental results of applying the extended DDTR methodology in various benchmarks.

1) Synthetic Datasets: We performed two experiments using synthetic benchmarks in the Freescale board, which are presented in Fig.4 and Fig.5. In the first one the updates are 10% of the total operations, while in the second they are 80%.

In the first experiment in Fig.4, there are three optimal implementations, namely the Ternary tree, the cache conscious

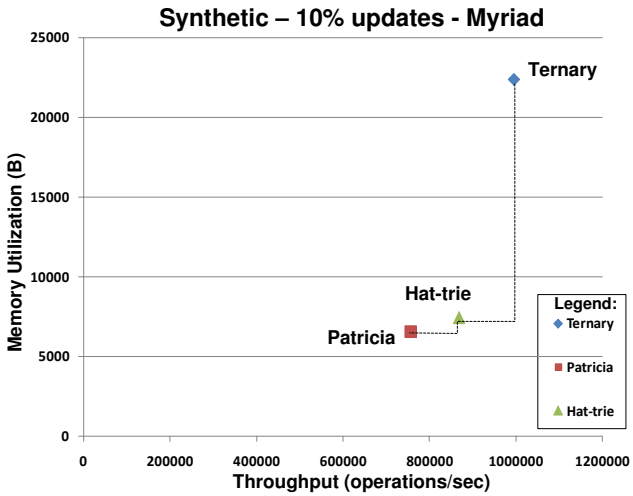

Fig. 6. Throughput vs. memory utilization synthetic benchmark with 10% update operations in the Myriad board.

Ternary tree (Ternary-cache-opt) and the Hat-trie. The highest throughput is achieved by the Hat-trie implementation (35% higher in comparison with the Patricia). The main reason is the Hat-trie provides faster look-up in comparison with the other implementations. However, the Ternary tree provides the lowest memory footprint, by 18.5% in comparison with the Hat-trie. Another important observation is the fact that the cache conscious versions outperform the corresponding non-optimized, by up to 34%. The fact that the specific benchmark has a small number of updates, favors the cache-optimized implementations.

In the second benchmark in Fig.5, the optimal implementations are the Hat-trie, the cache conscious Ternary tree and the Patricia. Hat-trie provides the highest throughput, by 39% in comparison with the Patricia implementation. It is interesting to notice that the cache-optimized implementations throughput is closer to that of the Hat-trie, than in the previous experiment. Indeed, the advantage of the Hat-trie, which is the fast lookups, is lost in this experiment, where the updates are the dominant operations.

The corresponding evaluation of the synthetic benchmarks in Myriad is presented in Fig.6 and Fig.7. The cache-conscious optimizations are omitted in the Myriad experiments, due to the lack of cache memory in the Myriad chip. In the first experiment, where the lookup operations are dominant, the Ternary tree provides the highest throughput (22% in comparison with the Patricia). However, the Patricia implementation is the one with the lowest memory utilization. In the second experiment, in which the update operations dominate, the Ternary tree and the Hat-trie are the optimal implementations. An interesting observation is the fact that the lack of cache memory leads to low Hat-trie performance in both experiments. Therefore, the results a lot different in comparison with the corresponding Freescale board experiments.

2) IP Datasets: The IP benchmark contains 3 million IP addresses, in which 4% are unique. The results on the Freescale chip are presented in Fig.8.

The overwhelming percentage of lookup operations leads

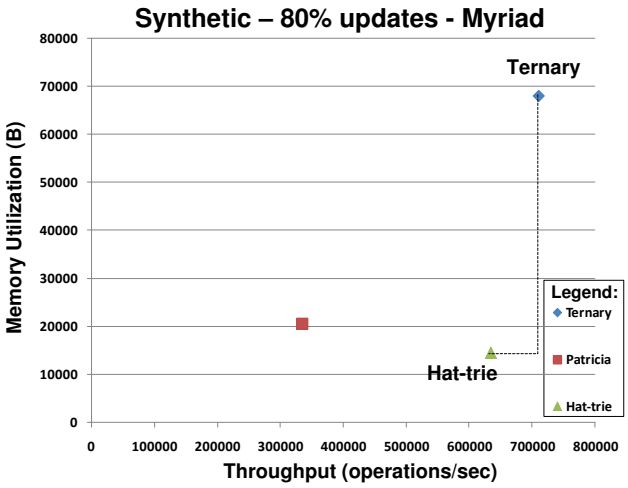

Fig. 7. Throughput vs. memory utilization of the synthetic benchmark with 80% update operations in the Myriad board.

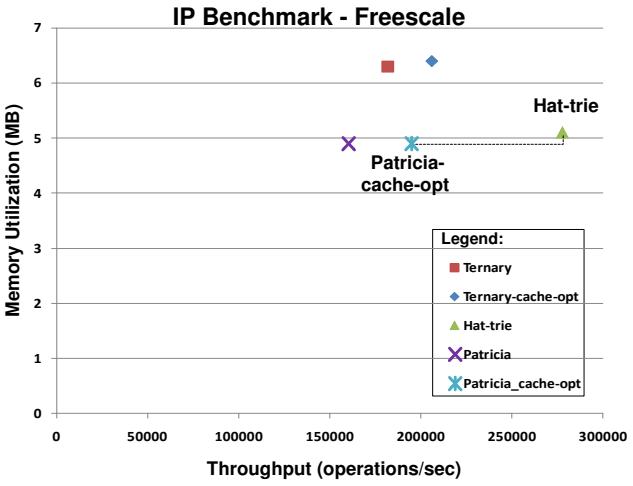

Fig. 8. Throughput vs. memory utilization of the IP benchmark in the Freescale board.

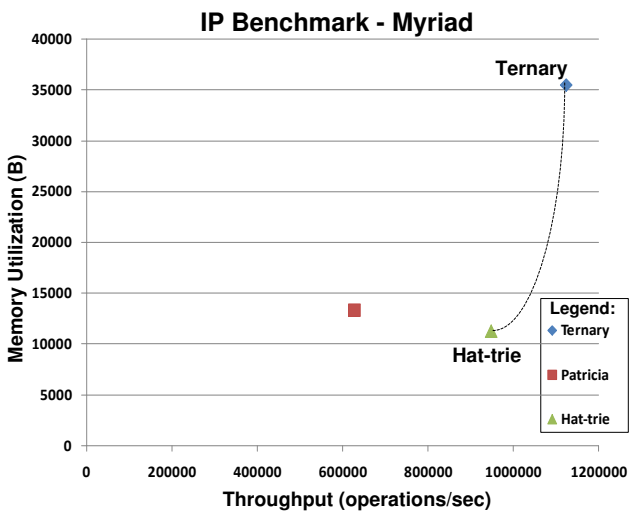

Fig. 9. Throughput vs. memory utilization of the IP benchmark in the Myriad board.

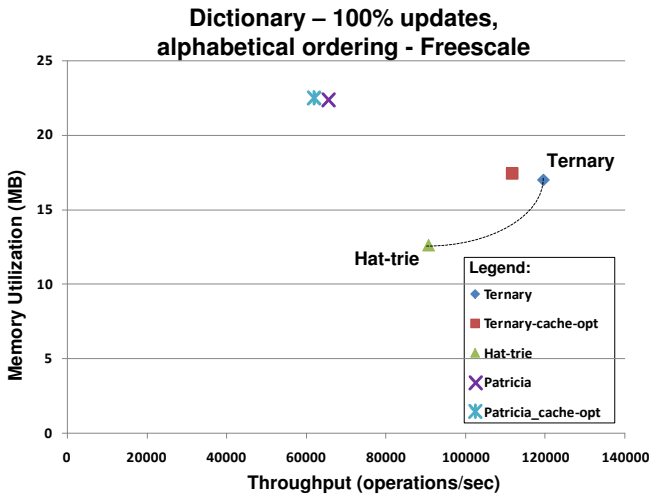

Fig. 10. Throughput vs. memory utilization of the dictionary benchmark with 100% update operations and strings being inserted in alphabetical order, in the Freescale board.

to very high throughput for the Hat-trie implementation (29.6% in comparison with the Patricia-cache-opt). We also notice that the cache-optimized ternary and Patricia implementations lead to 12% and 17% higher throughput, than the non-optimized implementations. The Patricia implementations have the smallest memory footprint: 3.5% lowest memory footprint in comparison with the Hat-trie. Indeed, Patricia tries are capable of storing strings with many similarities (like IP addresses) in a very memory efficient way.

In respect with the Myriad board, the corresponding results for the IP benchmark are presented in Fig.9. As in the synthetic experiments, the Ternary tree provides the highest throughput (18.5% against the Hat-trie), while the Hat-trie, the lowest memory consumption.

3) Dictionary Datasets: We have made several experiments with the dictionary datasets, which are presented in Fig.10, Fig.11 and Fig.12. The first experiment contains only update operations and the word insertion requests are coming in an alphabetical order. In the second one, the order is random, while in the last one, there are 65% lookup operations.

The optimal implementations in Fig.10 and Fig.11, are the Ternary trie and the Hat-trie. In the first experiment, the Ternary tree leads to 23% higher throughput in comparison with the Hat-trie, while the Hat-trie provides 18% lower memory consumption. In the second one (Fig.11), the corresponding differences are 8.9% for the performance, while the memory consumption is the same. Indeed, the insertion order matters only for the performance. When the dataset is sorted, consecutive words have the same prefixes, so they follow the same path in the tree in order to find the correct insertion spot. Since it is very likely that the path is already in the cache, the performance is higher, in comparison with the unsorted dataset in Fig.11. Finally, in Fig.12, where the lookup operations are dominant, the Hat-trie is the optimal implementation both in terms of throughput and memory footprint.

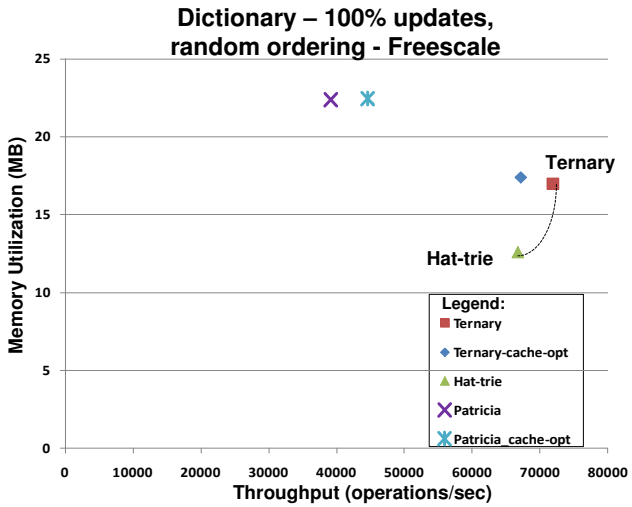

Fig. 11. Throughput vs. memory utilization of the dictionary benchmark with 100% update operations and strings being inserted in random order, in the Freescale board.

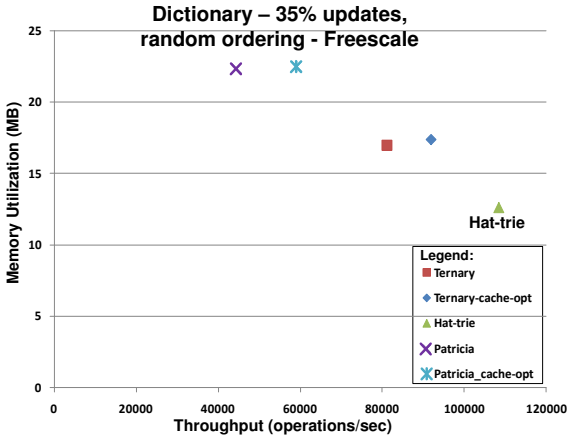

Fig. 12. Throughput vs. memory utilization of the dictionary benchmark with 35% update operations and strings being inserted in random order, in the Freescale board.

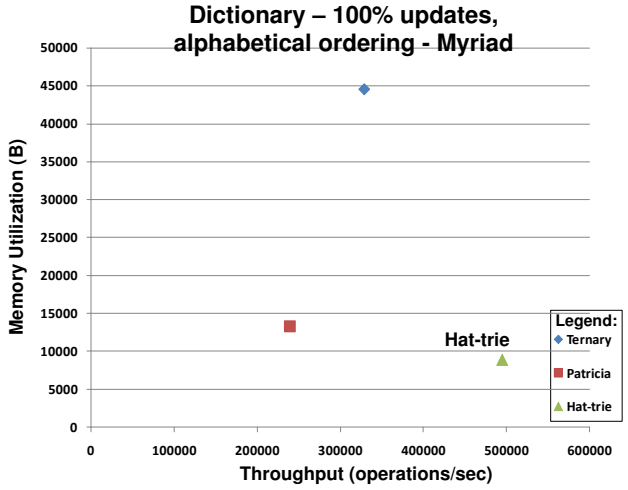

Fig. 13. Throughput vs. memory utilization of the dictionary benchmark with 100% update operations and strings being inserted in alphabetical order, in the Myriad board.

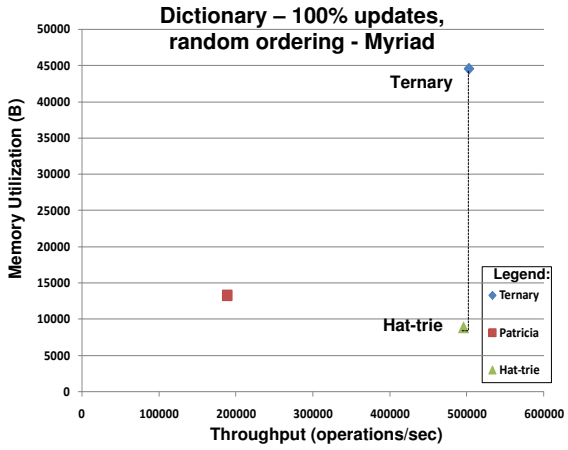

Fig. 14. Throughput vs. memory utilization of the dictionary benchmark with 100% update operations and strings being inserted in random order, in the Myriad board.

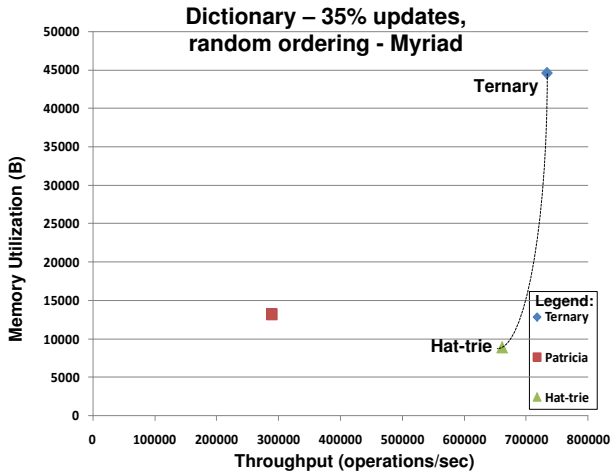

Fig. 15. Throughput vs. memory utilization of the dictionary benchmark with 35% update operations and strings being inserted in random order, in the Myriad board.

The corresponding results on the Myriad board are presented in Fig.13, Fig.14 and Fig.15. As in the previous experiments, the Hat-trie and the Ternary tree provide the highest throughput. However, the Ternary tree requires much more memory than the other two implementations. The high memory requirements of the Ternary tree in Myriad are noticed in all experiments. The reason is mostly related to the application, rather than in the Myriad memory architecture. Since we executed in Myriad only a part of the benchmark, the fact that the Ternary stores in each tree node only a character, leads to increased memory requirements. However, this is not the case with the Patricia, which it can store a whole string in each node.

4) Overhead of Cache Conscious Implementations: Table II presents the execution time and the memory footprint overhead of the cache conscious implementation for all datasets. As stated before, these overheads occur during the tree reorganization. More specifically, the execution time overhead is the percentage of time that is spent for the tree reorganization,

TABLE II. EXECUTION TIME AND MEMORY FOOTPRINT OVERHEAD OF CACHE CONSCIOUS IMPLEMENTATION ON FREESCALE BOARD.

	Execution time overhead		Memory size overhead (MB)	
	Ternary cache-opt	Patricia cache-opt	Ternary cache-opt	Patricia cache-opt
Synthetic 10% upd.	5.7%	5.38%	22	28
Synthetic 80% upd.	20%	18.4%	138	168
IP dataset	4.6%	3.6%	3.7	2.6
Dict. 100% upd. alph. order	35%	15.3%	14.3	18.5
Dict. 100% upd. random order	36%	20.3%	14.3	18.5
Dict. 35% upd. random order	20%	12.1%	16.2	32

during the whole execution of each benchmark. The memory footprint overhead is the memory utilization spike that occurs during the copy of the tree, when the new array is allocated and before the previous array is deleted from the memory.

In respect with the performance overhead, it largely depends on when the reorganization is being instructed. For instance, if the last reorganization was made near the end of the benchmark execution, the performance overhead is high, since there are not enough lookup operation to take advantage of the optimized tree. (This can be observed in the dictionary datasets). However, in the IP dataset, that was not the case and the performance overhead was very small. In respect with the memory overhead, it obviously depends on the size of the tree that is being copied. If the extra amount of memory needed is not a constraint, then the overhead is not an issue.

C. Discussion of Experimental Results

There are apparently some general principles regarding the data structures behavior under different application requirements. For example, we noticed in all experiments that the Hat-trie performs well when there is a large number of lookup operations. Also, in the dictionary benchmarks, the word ordering affects the performance. However, we also noticed the very different behavior of the same implementations between the two platforms with different memory hierarchies. Indeed, the existence of cache plays a major role in the performance results of the data structure implementations.

The improvements made in the DDTR methodology by the integration of the cache-conscious implementations extended the methodology by making it able to adhere to hardware-related constraints. In many cases (like in the IP benchmark and the synthetic ones), some Pareto points would not be "visible" with the previous DDTR methodology. In other words, the methodology is adapted, not only to the application constraints, but also to the hardware constraints and specifications. Thus, the set of data structure implementation solutions increases, providing the developer with more flexibility.

Finally, the cache conscious implementations through the data structure reorganization can be a valid alternative to the generic ones. In many cases (e.g. in the IP benchmark in Fig.8) are optimal in terms of throughput and memory requirements. If the performance and the memory overhead is not a constraint, then such implementations can achieve high performance by the effective cache utilization.

VI. CONCLUSION AND FUTURE WORK

The DDTR methodology provides and efficient way to refine the data structure implementations of dynamic applications. So far, the methodology was limited to generic array and list implementations. However, the extended approach provides tree implementations which are refined according to the hardware specifications of the underlying platform. Therefore, new Pareto optimal data structure implementations that they were not be available in the previous version, appear in the new one. In the future, we plan to extend the DDTR library with more tree implementations and evaluate it in more embedded platforms with various memory hierarchies.

ACKNOWLEDGEMENT

This work was supported by the EC through the FP7-ICT project 612069, HARPA (Harnessing Performance Variability).

REFERENCES

[1] Intel corp., Intel Atom Processor E6x5C Series Product Preview Datasheet, Tech. rep., 2010, URL http://www.arrownac.com/offers/intel/e6xx/E6x5C%20Datasheet.pdf.

[2] W. Wolf, High-performance embedded computing: architectures, applications and methodologies, Morgan Kaufmann, 2007.

[3] N. Rajovic et al "Tibidabo: making the case for an ARM-based HPC system," Future generation computer systems, Elsevier, Amsterdam, 2013.

[4] L. Papadopoulos, C. Baloukas and D. Soudris, "Exploration methodology of dynamic data structures in multimedia and network applications for embedded platforms," Journal of Systems Architecture, Embedded Systems Design 54(11), pp. 1030-1038, 2008.

[5] L. Benini, A. Macii, E. Macii, M. Poncino, "Increasing energy efficiency of embedded systems by application-specific memory hierarchy generation," IEEE Design and Test of Computers, pp. 7485, AprilJune 2000.

[6] S. Steinke et al., "Assigning program and data objects to scratchpad for energy reduction," in Proc. DATE Conference, IEEE Computer Society, Washington, DC, USA, 2002.

[7] M. Leeman et al., "Automated dynamic memory data type implementation exploration and optimization," in Proc. of the IEEE Computer Society Annual Symposium on VLSI, IEEE Computer Society, Washington, DC, USA, 2003.

[8] N. Askitis, R. Sinha, "HAT-trie: A Cache-conscious Trie-based Data Structure for Strings," in Proc. Thirtieth Australasian Computer Science Conference (ACSC), pp.97-105, 2007.

[9] P. Prokopec, N.G. Bronson, P. Bagwell and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," Proc. Symposium on Principles and Practice of Parallel Programming, pp. 151-160, 2012.

[10] T. M. Chilimbi, M. D. Hill, and J. R. Larus, Making pointer-based data structures cache conscious, Computer, vol 33(12): 6775, 2000.

[11] L. Li, L. Gao and J. Xue, "Memory coloring: a compiler approach for scratchpad memory management," Proc. PACT, pp. 329-338, 2005.

[12] D. Moloney "1TOPS/W software programmable media processor," HotChips HC23, Stanford, 2011.

[13] Freescale Semiconductor, i.MX 6Dual/6Quad Applications Processors for Industrial Products Datasheet, Freescale Semiconductor, 2014, URL http://cache.freescale.com/files/32bit/doc/data_sheet/IMX6DQIEC.pdf.

[14] The Internet Traffic Archive, URL http://ita.ee.lbl.gov/.

[15] WinEdt editor, URL http://www.winedt.com/.