Chapter 2
Analysis and Characterization of Dynamic Multimedia Applications

As discussed in Chap. 1, in nomadic embedded systems an increasing amount of applications (e.g., 3D games, video-players) coming from the general-purpose domain need to be mapped onto a cheap and compact device. However, embedded systems struggle to execute these complex applications because they come from desktop systems, holding very different restrictions regarding memory use features, and more concretely not concerned with the efficient use of the dynamic memory. Today, a desktop computer typically includes at least 2–8 GB of RAM memory, as opposed to the 256–1024 MB range present in low-power respectively high-end nomadic embedded systems. Therefore, one of the main steps during the porting process of multimedia applications (that were initially developed on a PC) onto embedded multimedia systems, involves the optimization of the dynamic memory subsystem.

The rest of this chapter is organized as follows. In Sect. 2.1, the multimedia application domain is analyzed including the typical features this type of application possesses, motivating the need for dynamic data optimizations. Then, in Sect. 2.2, dynamic data and its specific properties are presented. Next, in Sect. 2.3, the method flow to tackle the different optimization steps is explained in more detail, covering both the steps and the reasons for the selected ordering of these steps. Finally, in Sect. 2.4 some conclusions are discussed.

2.1 Characteristics of Multimedia Applications

The optimizations presented in this book exploit the characteristics of modern multimedia and communication embedded applications. Where traditional software would only have global or stack data, modern day applications require the use of heap data (or *dynamic data*). In this section, we describe how dynamic data is used in modern multimedia applications, and how the DDTs are used to manage this data. The set of analyzed multimedia and communication applications includes 3D image reconstruction applications [45, 139], video rendering applications as the MPEG-4 *Visual*

D. Atienza Alonso et al., *Dynamic Memory Management for Embedded Systems*,
DOI 10.1007/978-3-319-10572-7_2

Texture Coder (*VTC*) [59, 125], 3D games [71, 122], the URL-based context switching application [116], IPv4 routing algorithms [91] and firewall applications [116].

Due to the fact that these applications deal with data that is not quantifiable at compile-time, they require heap data (or dynamic data) to store the application-required information. For instance, in 3D image reconstruction, studied in the Sect. 2.1.1, two images are compared to find points that match. Since the amount of points that may match is not known at compile-time, a dynamic data-structure is required for this.

2.1.1 Example: 3D Image Reconstruction System

The previously outlined characteristics are illustrated using a modern 3D image reconstruction application [161]. In particular, we focus on one of the internal algorithms that works like 3D perception in living beings, where the relative displacement between several 2D projections is used to reconstruct the 3rd dimension [139]. This software module heavily uses dynamic memory and is one of the basic building blocks in many current 3D vision algorithms: *feature selection and matching*, which involve multiple sequential accesses and input data filtering operations. The algorithm studied has been extracted from the original code of the 3D image reconstruction system (see [161] for the full code of the algorithm with 1.75 million lines of high level C++), and creates the mathematical abstraction from the related frames that is used in the global algorithm. The algorithm selects and matches features (corners) on different subsequent frames and the relative offsets of these features define their spatial location (see Fig. 2.1). The operations performed on the images are particularly memory intensive, e.g., each image with a resolution of 640×480 uses about 2.5 MB, and the accesses of the algorithm to the images are randomized. Thus, classic image access

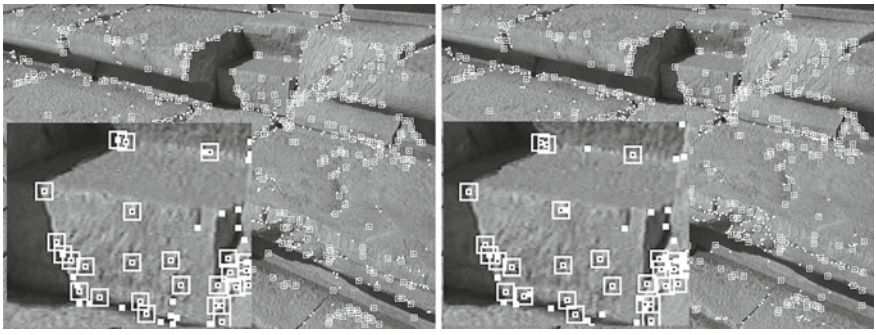

Fig. 2.1 Initialization of the matching of corners on two images of the steps of amphitheater (archaeological site): based on neighborhood search. Already most matches seem to be correct (partially due to the minor difference between the images, which can be seen at the *right hand bottom corner*). Part of the center is enlarged

optimizations such as row-dominated accesses versus column-wise accesses are not applicable.

The number of generated candidate matches is highly dependent on a number of factors. First, the *image properties* affect the generation of the matching candidates. Images with highly irregular or repetitive areas will generate a large number of localized candidates, while a low number will be detected in other parts of the image. Second, the *corner detection parameters* have a profound influence on the results (and, consequently, on the input to the subsequent matching algorithm) because they affect the sensitivity of the algorithm used to identify the interesting features in the images/frames [72]. Finally, the *corner matching parameters* that determine the matching phase have a profound influence and are changed at run-time. For example, the acceptance and rejection criterion changes over time as more 3D information is retrieved from the scene.

Taking all the previous factors into account, the possible combinations of parameters in the system make an accurate estimation of the memory cost, memory accesses and energy dissipation at compile time very hard or even impossible. This difficult to analyze and hence *unpredictable* memory behavior can be observed in many state-of-the-art 3D vision algorithms and makes them very difficult to optimize with traditional compile-time optimization methods for embedded systems. Nevertheless, since this metric 3D-reconstruction from video algorithm can perform the reconstruction of 3D scenes from images and requires no other information apart from multiple frames, it is especially useful for situations where extensive 3D setup with sensitive equipment is difficult, e.g., crowded streets or remote locations, or impossible as when the scene is no longer available [45, 60]. Hence, it is extensively used for quick on-site visualization, and speeding up the application to be able to process more frames for a more detailed reconstruction is a very important problem, which demands extensive code transformations and optimizations of the dynamic memory subsystem. Also, energy consumption is paramount for hand-held visualization devices and needs to be optimized as well. In the following paragraphs the internal DDTs of this case-study are explained and their respective dynamic allocation behaviors are carefully analyzed.

The algorithm internally uses several DDTs whose sizes cannot be determined until the system is running because they depend on factors (e.g., textures in the images) determined outside the algorithm (and uncertain at compile-time). Furthermore, due to the image-dependency related data, the initial DDT implementations for the variables do not fit in the internal memory of current embedded processors. These DDTs are the following:

- `ImageMatches (ImageMatches)` is the list of pairs where one point in the first image, matches another one on the second image based on a neighborhood test [139].
- `CandidateMatches (CandidateMatches)` is the list of candidates that must go through a normalized cross-correlation evaluation of a window around the points [161].

- `MultImageMatches (MultiMatches)` is the list of pairs that pass the aforementioned evaluation criterion. Still one point from the first image can be listed in multiple candidate pairs for a later best match selection.
- `BestMatches (BestMatches)` is the subset of pairs where only the best match for a point (according to the test already mentioned) is retained.

These DDTs were originally implemented using double linked lists and exhibit an unpredictable memory size, typical in many state-of-the-art 3D vision systems [161]. That is due to their use of some sort of *dynamic candidate selection* by traversing the input data in a sequential way, followed by *a criterion evaluation*. Figure 2.2 shows this behavior in the generation order of the DDTs. First, `ImageMatches` is generated after a neighborhood test is applied to pairs of corners detected in two images. Then, `CandidateMatches` is created using the information acquired from previous images. Finally, `MultiMatches` and `BestMatches` are generated with the pairs that pass a normalized cross correlation evaluation [139].

Although the global interaction between these DDTs is defined by the algorithm (see Fig. 2.2), it is not clear how each DDT affects the final system figures (e.g., memory footprint) until the application is profiled. Therefore, the method presented in Sect. 2.3 has been used to explore the behavior of the different DDTs. After inserting the profiling library and running the tools presented in Chap. 3, profiling information is obtained. Then, a memory use graph is automatically generated and memory accesses, memory footprint and energy dissipation figures are calculated (see Table 2.1).

When looking at the memory behavior of the application in Fig. 2.3, it is shown that only very few allocation sizes are very popular. This is of course a factor of the DDTs that were originally used in the application. The block sizes used in the DMM, that affect its design, are in turn affected by the DDTs basic memory requests. Because of this property, DDTs should always be decided upfront and they should take into account the restrictions on the available memory organisation. On an existing platform, which is the most common case now, that memory organisation is namely predefined. Hence it forms a constraint that we should incorporate in each of the

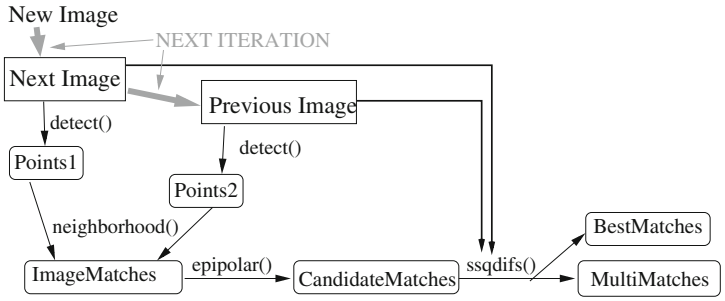

Fig. 2.2 Execution flow of the DDTs in the 3D image reconstruction application

Table 2.1 Original DDT in the 3D image reconstruction application

Variable	DDT accesses	Memory footprint (B)	Energy 0.13 μm tech. (μJ)
`ImageMatches`	1.20×10^6	5.14×10^2	0.18×10^3
`CandidateMatches`	8.44×10^5	2.75×10^5	3.03×10^3
`CMCopyStatic`	6.24×10^4	1.08×10^5	4.48×10^4
`MultiMatches`	1.84×10^4	3.62×10^2	0.02×10^1
`BestMatches`	1.66×10^4	3.07×10^2	0.02×10^1
`Total`	2.14×10^6	3.86×10^5	4.80×10^4

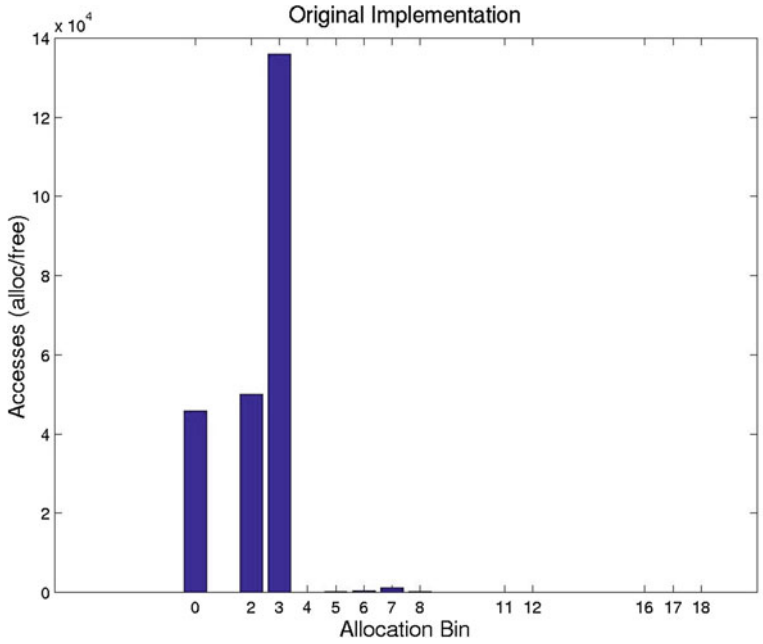

Fig. 2.3 Study of block-size allocation bins of the Kingsley DM manager requested by the original 3D reconstruction application

stages and steps. The DMM basic sizes will then be later decided during the DMMR step, but due to the early DDT decisions that step then has a better starting point.

In order to account for the varying memory use during the program run, normalized memory is used to get an estimation of the overall contribution of each DDT to the energy dissipation shown in Table 2.1. This attributes more accurate contributions to energy cost estimates and avoids that, for instance, DDTs with very short and high memory usage distort and hide the memory contribution from other DDTs. For the energy estimations, the embedded SRAMs memory model described in [84] is used, considering a 0.13 μ technology node for the memory hierarchy implementation. This model depends on memory footprint factors (e.g., size, internal structure or leaks)

and factors originated by memory accesses (e.g., number of accesses or technology node used). Note that any other energy model can be used just by replacing that module in the tools.

The analysis of the profiling information (Table 2.1) shows how the DDTs affect the dynamic memory subsystem. The energy numbers are calculated based on a memory that is large enough to encompass only that single DDT. The DDTs and their different dynamic behaviors can be derived from the internal subdivision of the global algorithm in different sub-algorithms previously described, which include different traversal and filtering phases. First, `CandidateMatches` is the largest DDT in this system. Secondly, `ImageMatches` has frequent accesses. Finally, `CMCopyStatic` is a dynamic array implementation, which has a much faster access to specific stored data, that only keeps a copy of the content of `CandidateMatches`) consumes an important part of the energy used by the system. The reason it has such a high energy cost is that certain accesses, namely adding new elements to it incur an O(N) memory access cost. As this analysis shown, the DDTs in multimedia applications interact as local variables (with very limited lifetime) that store the intermediate data generated between the different sub-algorithms that conform the global algorithm (e.g., `ImageMatches` or `CMCopyStatic`), or as variables that store the filtered input data between the different sub-algorithms. These are used in the end to provide the output data of the current module to the next software modules of the overall applications, due to modularity in software engineering, as it is the role of `BestMatches` and `MultiMatches`. A final characteristic of multimedia and communication applications is the repeating cycle of their dynamic allocation and de-allocation patterns. In particular, in the considered 3D image reconstruction module, this previously analyzed dynamic behavior is similar for each pair of consecutive frames, as in each new iteration of the algorithm, the oldest frame is replaced by the next one in the input set of frames and the algorithms are applied in the same way.

2.1.2 Potential for Optimizations

In this book we look at an important subset of the multimedia application domain, namely image processing (image analysis, image interpretation and image reconstruction), video processing (namely multiview video reconstruction) as well as 3D reconstruction and rendering. In general, the more standard image processing (filtering and coding) part of the applications manipulate quite static stack data [111]. However, more advanced computer vision and image understanding algorithms contain a significant part of dynamic data that have to be managed on the heap [45, 99]. The dynamic data types used for this are either based on libraries like the Standard Template Library (STL) [146, 158] or on own code. However, in both cases, the dominant DDTs are fully deterministic (sequences) in their access behavior and not semi-random data-dependent. Hence, we can conclude from this that the majority of these DDTs would be suited for handling by our approach.

Additionally, many image and video processing algorithms operate on data structures which are more complicated than arrays and which are dynamically constructed (e.g., with data dependent record sizes) [124]. For instance segmentation algorithms operate on data structures which can be organized as linked lists. Nevertheless, the most efficient algorithms aim to access such data in a regular fashion and avoid random access where possible. For instance, Philips has developed such algorithms, which iterate over variable-length line segments, but in a line-scan fashion. The main application domain is 3D image segmentation.

We also look in more detail at the communication protocol applications for nomadic embedded systems. More specifically, in [18], it is shown how communication protocol applications contain data structures that are dynamic in nature. More specifically, packet scheduling algorithms have to deal with a variety of queues of packets, and these are by definition dynamic in nature, as it is not known in advance how many packets will fall into each scheduling queue.

For these two specific domains, the range of possible applications is in theory initially very broad. However, for most of the embedded systems the number and types of multimedia and communication protocol applications to be included in the final design (at least to a large extent) are known at design-time. Thus, it is feasible to analyze the types of dynamically-allocated objects that exist in each of them (e.g., points, triangles, 3D faces, acknowledgment or frequent packets sizes, etc.) and to design the most convenient DDT implementation for each variable, as well as the most convenient DMM for the application in question.

Furthermore, our experience has shown that, in each application, the range of sizes used by the dynamic elements is very limited and it can be determined at design time by, first, a static analysis of the source code of the application and, second, a profiling analysis of the application under exploration with a reduced number of input data sets (i.e., in general no more than 10 variations). Note that the size of the dynamic elements is known a priori, but not the actual number to allocate, which can vary significantly from one possible representative input to another; thus, the use of dynamically allocated memory is justified and extensively used in these application domains. For example, in order to analyze an MPEG-4 video player, different system configurations, such as screen resolution or visualization size, need to be studied according to the target embedded device: a smartphone, a Portable Digital Assistant (PDA), a portable video-game station, etc. This set of configurations needs to be explored for a representative set of input frames (e.g., with a different number of rendered objects and textures, etc.), while taking into account the probability distribution of the different types of inputs for the final working conditions of the target embedded device, as these features will influence the final dynamic data structure to be used for the final application.

In addition, it is necessary to recognize the dominant dynamically-allocated elements of each application. In fact, each application can contain a large number of allocated elements, but in most of the multimedia and communication protocol applications few variables (i.e., between 5 and 20) tend to account for a very large percentage of the overall memory accesses and/or memory footprint used for dynamic memory storage: in general between 50 and 70 %.

2.2 Dynamic Data Handling

As shown in Sect. 2.1, in modern multimedia and communication protocol applications, data is stored in entities called *Dynamic Data Types* (*DDTs*) or simply containers, like vectors, lists or trees, which can adapt dynamically to the amount of memory used by each application [174]. These containers are realized at the software architecture level [99] and are responsible for keeping and organizing the data in the memory and also servicing the application's requests at run-time. These services require to include abstract data type operators for storing, retrieving or altering any data value, which is not tied to a specific container implementation and which shares a common interface (as in STL [146]).

Dynamic data is commonly referred to as *heap data*, while static data usually either resides in the global data segment or on the *stack*. The difference between dynamic data types and dynamic data structures is that the former provide an interface with a given set of operations, while the latter provide a specific implementation that adheres to the interface. For example, STL provides *sequences* as dynamic data types. Vectors and lists are then two specific data structures with different complexity trade-offs that implement the sequence data type.

To understand the difference between dynamic data types and traditional static data types, it is necessary to understand how individual elements stored in these data types are mapped to the actual memory hierarchy. The mapping of an element to its actual place in memory for data types, both static as well as dynamic, consists of two layers. The first layer, from now on referred to as *element mapping*, maps the location of an element in a specific data structure to a location in the memory block(s) that are employed by the data structure. The second layer, from now on referred to as *block mapping*, maps the data structure's memory block(s) to the memory pools that can represent the physical memory hierarchy. The presence of a memory management unit (MMU) adds a third, hardware, layer, that maps virtual memory addresses to physical memory addresses.

The difference between dynamic and static data types is how these mappings are realized. The differences are first detailed at element mapping layer. Then, the difference are detailed at the block mapping layer.

At the element mapping layer, the mapping is fixed for static data types. The most typical static data structure, the array, demonstrates this quite clearly. The memory-offset of an element in the memory-block used by the array is a linear mapping of its index (namely by multiplying by the size of the element). This mapping is static and defined by the platform's application binary interface (ABI) [30].

For dynamic data types, on the other hand, the mapping from element to location in the memory block is defined by the implementation of the specific data structure. Additionally, through the operations provided by dynamic data types, this mapping can change at run-time. For instance, a vector, which is most similar to an array, maps elements to the memory block it uses in a similar fashion to an array. However, a vector allows for the insertion of elements at any index. This insertion affects all elements after the element inserted at, and changes their location. The fact that dynamic data

types allow for the insertion, as well as removal of elements, has implications for the block mapping layer. It is necessary to replace smaller memory blocks for bigger memory blocks as more elements are added to a dynamic data structure. As such, for dynamic data types, the block mapping layer is necessarily dynamic.

For static data types, it is still possible to have a dynamic block mapping layer, in the case that they are allocated at run-time. However, because there is no dynamism in the element mapping, this allocation is necessarily not required as often, as it is only required to create entirely new data structures, not to deal with the dynamism at the element layer mapping.

The actual implementation of dynamic data structures only encodes the element mapping layer. The dynamism at the block mapping layer is handled by the *Dynamic Memory Manager* (*DMM*) which takes care of not only servicing memory allocation requests but also ensures this is done in an optimal manner.

With the combination of both the index-layer mapping as well as the block-layer mapping, elements of DDTs are thus stored into memory pools. After this mapping, it is then possible to map these memory pools, along with the pool dedicated to stack data and global data onto the actual real memory hierarchy. This step combines the dynamic and static data types and is complimentary to and compatible with the index-layer mapping and block-layer mapping. Therefore, dynamic data must be tackled first before it can be combined in terms of memory pools with the remaining static stack pool.

As shown, two components are required to enable the design of applications that use dynamic data. First of all, the implementation of the operators of a DDT can have significant impacts on its access and storage patterns. Secondly, the implementation of the DMM can have an impact on where these accesses and storage reside in the memory hierarchy. These two are detailed further below.

2.2.1 Dynamic Data Structure Optimization Opportunities

The design of DDTs is dictated by a specific interface composed of a set of operators. Existing and widely-used libraries have a good set of operators that provide the required flexibility without posing too many constraints on the implementation of the DDTs (e.g., STL [146] and Java Collections). Before we look at these operators, it is important to realize that also a variety of different types of DDTs exists. And the choice of which DDT is ideal, is usually based on the application in question. As such, a wide set of libraries exist that give different classes of DDTs. Some libraries, like STL [146] are more standardized and come with most C++ compilers. Other libraries are more fragmented, and are thus ad-hoc standards or built for specific purposes. These two categories of libraries are studied in more detail.

2.2.1.1 STL

The classification, as defined by STL, consists of the following top-level concepts. This hierarchy, along with the implementations is also presented in Fig. 2.4. The darker elements are the different classes of DDTs, while the lighter elements specify specific implementations provided by the STL library.

- **Container** A Container is an object that stores other objects (its elements), and that has methods for accessing its elements. In particular, every type that is a model of Container has an associated *iterator* type that can be used to iterate through the Container's elements. [146].
- **Sequences** A variable-sized container whose elements are arranged in a strict linear order. Supports insertion and removal of elements.
- **Associative Containers** A variable-sized container that supports efficient retrieval of elements based on keys. Supports insertion and removal of elements, but differs from sequences that it does not provide a mechanism for inserting an element at a specific position.

Besides these generic categories, STL also provides some specific data-structures, mostly to provide functionality on top of the above-described set of containers, or to address the specific needs of handling strings. Due to the fact that these are either based on top of the existing containers, or address a very specific need (string handling) that is less relevant to the multimedia domain, they are not further treated in this book.

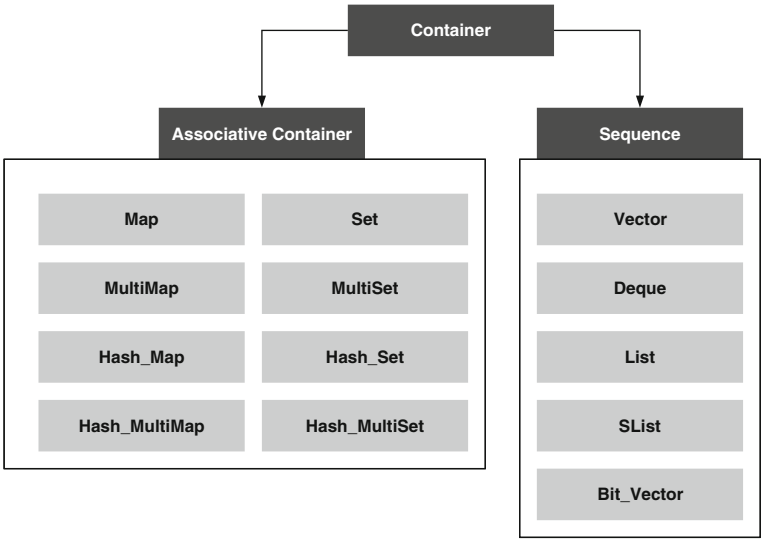

Fig. 2.4 Hierarchy of the different DDTs as classified by STL (in *darker color*), along with the implementations (in *lighter color*) that come with STL

- **Ropes** These are designed to be scalable string implementations and are specifically designed for operations that involve a string as a whole.
- **Bitset** Very similar to a vector of booleans, it contains a collection of bits and provides constant-time access to each bit. Unlike a vector, however, the size of a bitset cannot be changed. As such, this makes it much more akin to an array. Bitsets are not covered in this book as they are not defined to be an STL container, since they lack iterators for traversal.
- **Container Adapters** These are not technically containers, but instead adapters that can be placed on top of other containers to restrict the interface.

Relevant is the fact that accesses to the containers are achieved through iterators. Where static data structures like arrays employ the use of pointers for accessing and navigating through the elements, this is only possible because the memory locations of subsequent elements are contiguous. Iterators are a generalization of pointers. Iterators are central to generic programming because they are an interface between containers and algorithms: algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators [146]. The iterators are categorized by STL, as shown in Fig. 2.5.

- **Trivial Iterator** An object that may be dereferenced to refer to some other object. The simplest example in the case of static data types is a simple pointer into an array.
- **Input Iterator** An iterator that may be dereferenced to refer to some object, and that may be incremented to obtain the next iterator in a sequence.
- **Output Iterator** An iterator that provides a mechanism for storing (but not necessarily accessing) a sequence of values.
- **Forward Iterator** An iterator that corresponds to the usual intuitive notion of a linear sequence of values.

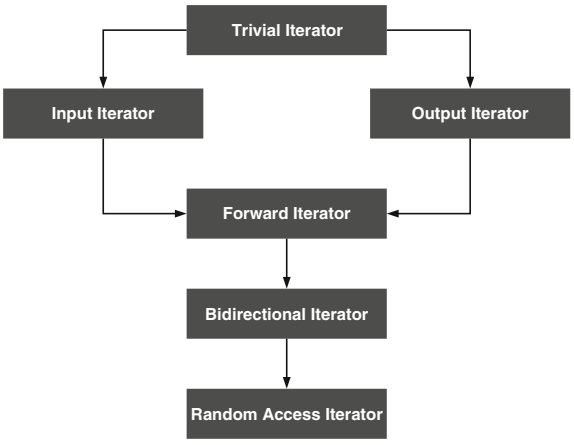

Fig. 2.5 Hierarchy of the different iterator classes used to traverse and access DDTs, as specified by STL

- **Bidirectional Iterator** An iterator that can be both incremented and decremented.
- **Random Access Iterator** An iterator that provides both increment and decrement and that also provides constant-time methods for moving forward and backward in arbitrary-sized steps. The best example of this is a pointer into an array.

The final way that these DDTs can be classified is by how they are used in the application. Here, we discern different ways a DDT can be used in an application. The usage-patterns are based on how the iterators to the DDT are used. While most operations defined on DDTs do require an iterator, some operations do not. For sequences, we consider the operation `append` as based on a forward iterator, as the location where this adds an element is well-defined. On the other hand, assignment at an index for a sequence or assignment of a value to a key in an associative container is considered as based on a random iterator. As such it is possible to define two different usage patterns:

- Static number of iterators accessing the DDT. This is the case when a DDT is used as an intermediate variable and is built up in a set of phases. Note that the actual number at run-time must not be static, but conditionally static, in that there is a well-defined set of loop nests that *could* modify the DDT. Note that this condition is not overly restrictive, as control-flows are mostly static. As such, as long as there is no top-level loop which constantly creates an iterator to access the DDT, the usage pattern falls in this category.
- Dynamic number of iterators accessing the DDT. This is the case when a DDT is used as a central state, and is being modified over and over in a top-level loop. An example of this is a histogram that is being calculated over an entire input, and thus a random access iterator is used for each value of the input to modify the DDT.

Where DDTs that function as central state can be optimized to have efficient operations, for DDTs that function as intermediate variables, it is possible, given the right optimizations, to remove them completely. This is detailed further in Sect. 2.3.3. For all of the above dynamic data types, it is possible to employ information regarding the application to optimize the operations as well as the block-level mapping to the memory hierarchy. This is detailed in Sect. 2.3.

2.2.1.2 Ad-hoc and Custom Libraries

Besides the Boost C++ Libraries [29], which provide a variety of extra functionality missing in C++ and STL, several tree libraries exist that are publicly available for quite a few years [146]. While Boost is an ad-hoc standard, the other libraries are personal contributions from various authors with no standardization or committee. These libraries focus on classes of DDTs (as well as other functionality) that are not present in STL. Most prominently, they focus on graphs and trees, namely data structures that have more structure in their interface. Trees have the concept of parent and child links as well as siblings, where sequences only have sibling elements. Graphs, add even more flexibility to the links.

It is important to note that the discussion here on trees, is for trees where the actual structure of the tree has semantical relevance to the application. Some of the associative containers, for instance *map*, use trees internally, however from an interface perspective they are presented as an associative container mapping keys to values. Trees presented here, however, are used by applications that *rely on a specific structure in the parent-child relationship beyond those employed for pure performance reasons*. As such, these libraries have a lot less flexibility in terms of the internal data structures used for representing these trees. On the other hand, because there is semantics in the physical ordering elements, this information can be exploited for optimizations.

Similar observations here hold as in the last section regarding potential for optimizations. In principle, intermediate versions of tree data representations can be redundant and hence removable. But due to the different interfaces and larger variety in the way these tree libraries are defined, the actual optimization procedure is not worked out in this book. Only the STL sequences are addressed in Sect. 2.3.3. Extending this to sequence type tree DDTs is left as future work.

2.2.2 Dynamic Memory Manager

The task of the DMM subsystem is to supply the application or DDTs of the application with memory blocks where the actual data can be stored. These blocks are requested through *allocation* and are then returned from the application to the DMM subsystem through *de-allocation*. This occurs through operations that are well-defined [172], and these are detailed below. Since the DMM subsystem is only defined as those two operations as well as the expected behavior of those two operations, it is relatively unconstrained in terms of implementation-freedom. Only a few basic guarantees must be provided: that the allocation of a memory block will return a block of at least the requested size; that a memory block that is allocated will not overlap with another allocated memory block; that the memory block is not moved in memory; and preferably that de-allocated memory is reused for later allocations. The two main operations provided by a typical DMM are:

- `malloc` (called by **`new`** in C++): An operation that takes as input a required size and returns a memory block that is at least as large as the requested size.
- `free` (called by **`delete`** in C++): An operation that frees a previously allocated block, therefore allowing it to be reused for future allocations.

Several non-functional constraints are present that must be taken into account before a DMM can be deemed optimal, or even usable. One of these factors is memory fragmentation. It has been introduced already in Sect. 1.3.1, including the distinction between two types of memory fragmentation namely internal and external. To better illustrate this concept, an example is provided here.

In Fig. 2.6, we have two examples of an allocation/de-allocation pattern occurring, at the top and the bottom. In each pattern, a block of 20 kB is allocated by the

application, subsequently de-allocated by the application, and then a 40 kB block is allocated by the application. In both cases, the DMM subsystem returns the first memory block that it can find that fits the required size. In the first case, the DMM subsystem does not split 40 kB free block it finds. This results in a less free memory being available after the allocation than if it were to return a 20 kB block. This is called internal fragmentation and results in unused space that the application will not be able to use as long the 20 kB block is allocated. The DMM subsystem cannot return the remaining 20 kB block as the entire 40 kB block is considered used, and the application cannot use the extra space since it only knows that it allocated a block that can fit 20 kB. In the second case of Fig. 2.6, the DMM subsystem decides to split the 40 kB free block to deal with internal fragmentation. However, this results in the subsequent request for 40 kB to not find any free blocks available. In such a case the DMM subsystem would have to request more memory from the Operating System to be able to comply with the application's request (resulting in a bigger pool of memory blocks). In the meantime, however, the memory occupied by the 20 kB free memory blocks remains unused and thus wasted. This is referred to as external fragmentation. In this simple example, the DMM subsystem could have picked the 20 kB block for the 20 kB request and would have alleviated both fragmentation issues. In real systems, finding the optimally matching free block can become expensive due to the amount of effort of finding that block. The second DMM subsystem could have coalesced the two 20 kB blocks back together into a 40 kB blocks. This also comes with cost trade-offs in real systems. Additionally, coalescing can result in internal fragmentation or external fragmentation at a later point and thus coalescing at a particular point in time may not be globally optimal.

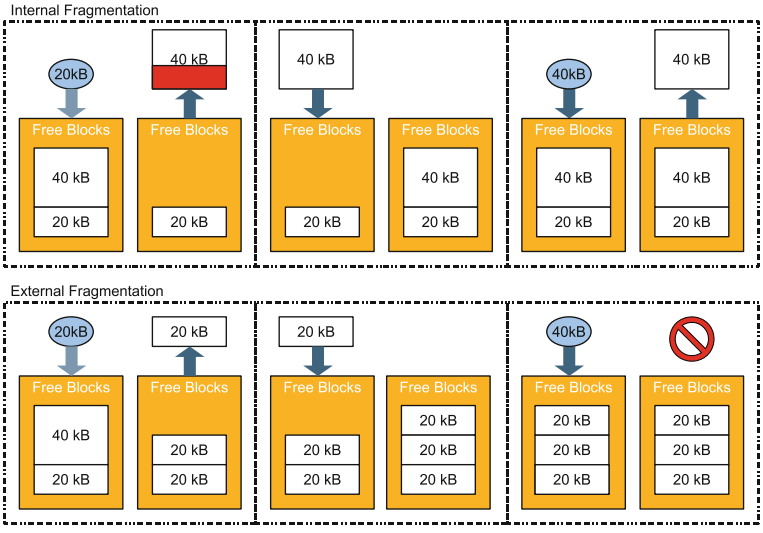

Fig. 2.6 Example of external and internal fragmentation

While most operating systems ship with a standard DMM library [172], these are never optimized for energy consumption. Additionally, they do not take into account the behavior of the application in terms of memory allocations/de-allocations and memory accesses. In the presence of a memory hierarchy such as found in embedded systems, this often results in suboptimal choices for memory allocators [106]. Like the design of DDTs, the design of the DMM subsystem has to take into account several factors before they can be considered useful for a given system:

1. The allocation pattern over time of the application, both directly as well as through the use of the DDTs. It is important that the DMM subsystem is able to efficiently allocate and de-allocate blocks that are frequently allocated/de-allocated. This efficiency is typically measured both in terms of computation as well as the memory accesses that the DMM subsystem performs under the hood for these operations.
2. The memory footprint of the memory allocator in question is not only determined by the dynamic data of the application itself, which obviously is a contributing factor to the dimensioning of the system, but also the memory fragmentation within the dynamic memory manager. If this is not kept in check, the dynamic data can easily consume all the resources of the platform in question, thereby degrading or even completely breaking the expected behavior of the application. As such, memory fragmentation is an important factor for the overall system behavior, that needs to be taken into account.
3. The access pattern of the application to the dynamic data, and hence dynamic memory allocated for it, determines the energy consumption of the application. By placing often accessed blocks onto local and smaller memories it is possible to have a tremendous gain in energy consumption. Thus by taking this access pattern into account, the DMM subsystem can ensure lower energy consumption.

2.3 Proposed Optimization Method

As explained in Sect. 2.2, different optimizations are required to reduce the energy cost incurred by dynamic memory accesses in the application. These optimizations target not only the multimedia applications that need to be improved, but also the libraries as well as the middleware employed by these applications. The method to target such applications is generic, but the steps produce specific results for each multimedia application. By customizing the different aspects like the libraries and the run-time to the multimedia application, it is possible to reduce the energy cost of dynamic memory in these applications significantly. Since the solutions are specific to the application in question, information regarding the application must first be collected though.

The systematic methodology which has been introduced in Sect. 1.4, will now be discussed in more detail. The method that is presented in this book is platform-dependent (i.e., it depends on the instance of the data memory hierarchy organisation

used in the platform, but it is processor architecture independent. Since it deals with the energy consumption of memory behavior, it is dependent on the memory sizes that are present in the system. However, the trade-offs of the different optimization steps will not be dependent on the exact energy cost of each access. As such, only the relative energy costs, and thus the relative sizes of the different memory levels (L1, L2, main memory) of the memory hierarchy are of relevance, and not the exact details of the specific architecture in question. While the results and exact final design choices made in the different optimization steps are platform-dependent, the method itself is generic for all platforms, and only the cost-functions and thus the final design choice for a specific design will be platform-dependent, not the applicability of the method itself.

Where in traditional optimization approaches, the information was about static data, which usually is manifest in the source code in terms of array-sizes and loop-boundaries, the introduction of dynamic data removes this source of information. Although production and consumption loops can still be identified, due to the resizability of the DDTs (that serve as storage for this dynamic data) at run-time, it is no longer possible to identify the quantities of data in question. Therefore, the information regarding the behavior of the application in terms of its dynamic memory consumption must be collected at run-time. As such, the method starts with a metadata collection step before the different optimization steps are applied, resulting in a method flow to tackle dynamic memory [22].

We now look at this method flow, motivating why it is structured and ordered as it is and explaining what the different steps entail. First, the overall method is detailed in Sect. 2.3.1. Then, the step that collects the information required for the different optimization steps is detailed in Sect. 2.3.2. Finally, the different optimization steps are detailed in Sects. 2.3.3, 2.3.4, 2.3.5 and 2.3.6.

2.3.1 Method Overview

In this section we give an overview of the method, explaining the order of the different steps and motivating why this order makes most sense. The explanations of the different steps are then detailed further in the following subsections.

As shown in Fig. 2.7, the first step in the method focuses on the collection of information regarding the behavior of the application. What is not shown in the figure are the high level optimizations that are possible at the object-oriented or higher description levels (e.g., Unified Modeling Language or UML [131]). These need to come first since they can completely alter the design of the application and the existing DDTs. The can be decoupled of the other subsequent steps by introducing high-level estimators [18].

The profiling step information is collected in the form of software metadata, detailing the usage of DDTs and dynamic data in general, both in terms of allocations and de-allocations as well as accesses, and summarized according to a variety of axes.

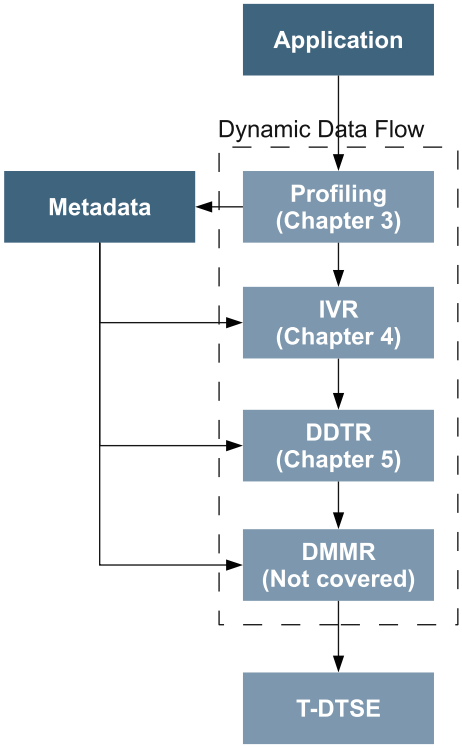

Fig. 2.7 Overview of the proposed methodology

This information is then passed on to the various optimization steps, enabling them. The profiling and analysis step is explained in further detail in Sect. 2.3.2.

After the collection of the information, it is possible to have a holistic optimization with regard to the usage of the DDTs at the application or algorithm level. Typically, DDTs are used in two fashions: As central storage for elements in an algorithm, thus serving as the algorithm's state; or as a buffer to decouple production and consumption phases of an algorithm. In the second case, which is frequently observed in multimedia applications, the use of DDTs is intended to keep algorithms clean and from becoming too complex. In such cases, it is possible to systematically optimize away these DDTs, by a novel optimization step that is termed *Intermediate Variable Removal* (*IVR*). This step enables a trade-off between computation and memory bandwidth. Depending on the platform and the relative costs of memory accesses and computation, this allows for optimizations in term of energy consumption. A small example of IVR is presented here to give the reader some intuition as to what it entails. Further information with regard to IVR is given in Sect. 2.3.3.

In the first code-sample of Fig. 2.8, a DDT is created to which a sequence of elements (generated procedurally) is added. Then, this sequence of elements is consumed in a second loop. Had the two loops been written at once, as shown in the

```
vector<int> a;
for (int i = 0; i < SIZE; i++) {
  if (h(i))
    vector.push_back(f(i));
}
for (vector<int>::iterator i = a.begin(); i != a.end(); ++i) {
  g(*i);
}

for (int i = 0; i < SIZE; i++) {
  if (h(i))
    g(f(i));
}
```

Fig. 2.8 Example of intermediate variable removal

second code-sample of Fig. 2.8, the overhead of first writing all these elements to memory (and the DDT bookkeeping required to make the sequence grow as needed) as well as the reading from memory would not have been necessary. Instead each element could have been written to local memory (or even a register) and then consumed directly, leading to a much lower energy consumption as larger memories require more energy to read and write from. Additionally, the internal DDT bookkeeping would have been completely obviated. IVR focuses on such transformations, ensuring to not break constraints with regard to the ordering of input/output as well as dealing with more complicated production and consumption loops.

Only once it has been decided which DDTs actually remain, it is effective to start optimizing their implementation to a Pareto-optimal design taking into account the platform that the application is being mapped on as well as the resulting behavior of the application in terms of the DDTs. DDT optimizations are not discussed further in this book as they are not part of the core research of this book. Instead I refer the reader to [16, 18]. Again, high-level estimators are used to decouple this step from subsequent steps [18]. Taking the example again in Example 2.8, if the intermediate DDT had not been removed, ideally a list-like DDT would have been used as it leads to less copying when constantly adding elements to the end of it.

Finally, once the implementations of the DDTs are defined, it will be clear what blocks are allocated by these DDTs as well as the application itself. Since the DDT implementations define the internal storage policy on memory blocks, it is required to have the final implementation of the DDTs to see what the allocation behavior of the application looks like. For instance, a simple switch such as moving from a vector style sequence to a linked list style sequence not only fundamentally changes the allocation pattern, but also the types of blocks allocated. In the case of a vector, a contiguous memory block containing the elements will be allocated, resulting in one big allocation (or several as the vector slowly grows and the memory block is expanded and thus reallocated). On the other hand, if a linked list is used, many small blocks will be allocated, each containing a single element and pointers to the previous and next block, with an allocation occurring each time an element is added. Thus, it

is clear that the DDT implementation has a big impact on the allocation pattern. This is why the DMM optimisation step comes after the DDT optimisation step. Further information with regard to the DMM optimisation is given in Sect. 2.3.5.

Note that all the above steps deal with dynamic data and how to map this to memory pools. After these steps, it is then possible to combine these abstract memory pools with the global and stack data. This falls outside of the scope of this book, but it shows how the previous method ties into existing work on static data. More information can be found in Sect. 2.3.6.

2.3.2 Profiling and Metadata Collection

To enable the different optimization steps, as already explained in Sect. 2.3, it is necessary to first understand what the dynamic memory behavior of the application is. To this end, since the target application is dynamic, it is necessary to profile it to get an accurate view of the different demands of the application in terms of memory accesses and memory allocations. Additionally, higher abstraction-level profiling information is also captured, such as the use of the different methods provided by a DDT. By using a standard interface (e.g., the sequence interface from STL [146]) it is possible to study the behavior of the application in terms of this interface and then optimize the DDT implementation based on how the DDTs are used. The profiling library that has been developed captures all this information without requiring a drastic change in the application source code [47].

After the raw profiling information is captured, it is analyzed and summarized into metadata that describes the memory behavior of the application at a high level. Typical information here will be how the different DDTs in the application are used, what the typical allocation behavior of the application is, where the most memory accesses occur, etc. Performing this profiling and analysis for a set of representative inputs, it is possible to get a clearer picture of the application's behavior. With this information, it is then feasible to identify intermediate DDTs that can be removed, optimize the implementation of the remaining DDTs and define an optimal memory allocator for the given application and hardware platform. More information regarding profiling and analysis of the application behavior can be found in Chap. 3.

2.3.3 Intermediate Variable Removal

Prior to deciding what the implementations are for the different DDTs in the application, it is first necessary to determine which DDTs are actually required for the algorithm to run. From a design point of view, it is a good idea to decouple production from consumption statements, as it creates a clear separation with a standard interface, namely the intermediate DDTs between these different steps in the application. Some DDTs are required from a theoretical stand point as they can encode

the central state of a certain part of the application, where the application cycles in a loop and regularly updates the state. On the other hand, other DDTs only exist in specific phases as temporary buffers between two steps of an algorithm.

From the point of view of the modular programming paradigm and the use of abstract data types [5, 32], which are used in all modern programming languages, the functionality of any program should be decoupled and defined independently from the specific implementation of the data structures (dynamic or static) used to store the application data, as long as all the methods and access functions required by the implemented algorithm are available [82]. According to this software programming paradigm, the proposed method relies on the assumption that the DDTs can be identified in the original source code of the application under exploration, and that they can be replaced without collateral effects that require the modifications of the control flow in the application. In other words, the input source code of the application where the presented method is applied must not include the DDT implementations, and they must have been defined in a separate module or standard library of DDT implementations in C++, such as STL. Moreover, all the algorithm using the DDTs implementations must interact with them only through the standard and common set of access methods [6].

It must be noted that the aforementioned restriction does not impose any real limitation with respect to the applicability of the underlying method, as all the studied applications and most of the applications nowadays, specially designed using the object-oriented paradigm, are implemented following this assumption. Moreover, as the object-oriented programming paradigm is being adopted further as the common design standard to exploit the System-on-Chip paradigm at the software level, this previous assumption will need to be enforced further, due to the necessity to define and use reusable libraries, based on standard interfaces.

The IVR optimization step uses information from the profiling step to determine which DDTs are being used in a producer-consumer like fashion. This is one of the reasons why the profiling step not only profiles memory behavior but also behavior of the application in terms of abstract DDT operations as defined by the DDT interface. More information regarding the IVR optimization step can be found in Chap. 5. Only once it is known which DDTs remain, it is possible to move on to the next step, namely the optimization of DDTs, which is not presented in more detail in this book. More information regarding DDT optimizations can be found in [18].

2.3.4 Dynamic Data Type Refinement

Typically a trade-off exists between DDTs that are mostly used in a random access fashion and DDTs that are mostly accessed in a sequential fashion. It is not possible to design DDTs that support both types of operations in O(1). To give the reader a feeling of this trade-off, Table 2.2 gives a few standard implementations provided by most generic libraries. It should be noted that these trade-offs are only at the granularity of complexity (which translates into both memory accesses and computation). However,

Table 2.2 Typical complexity trade-offs between sequence implementations often found in standard libraries

Implementation	Random access	Append	Prepend	Insertion	Removal
`Vector`	O(1)	O(1)	O(N)	O(N)	O(N)
`List`	O(N)	O(1)	O(1)	O(1)	O(1)
`Fingertree`	O(log N)	O(1)	O(1)	O(log N)	O(log N)

once energy consumption is taken into consideration, memory footprint as well as other factors come into play. These factors are detailed further below.

The implementation of the DDT operations depends on the chosen container and each one of them can favor specific data access and storage patterns. Each application can host a number of different containers according to its particular data access and storage pattern in the algorithm. Choosing an improper container implementation for an abstract data type will have significant negative impact on the dynamic memory subsystem of the embedded system [16, 18]. 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. So unnecessary accesses can comprise a very significant portion of the overall power of the system [21, 52]. In fact, energy consumption is one of the most critical limiting factor in the amount of functionality that can be placed in these devices, because portable computers like tablets and notebooks running complex multimedia applications rely on limited battery energy for their operation. As a result, the design of the DDT for a certain nomadic embedded system and multimedia application needs to consider several combined factors:

1. The algorithm implemented in terms of the high level operators that the DDT provides. Since different implementations will have different costs for the variety of operations, this is one of the main factors. While certain implementations may have O(1) access, they will typically require O(N) for insertion, where 'N' is the size of the DDT. A plethora of options exist, but no DDT operation has O(1) for all operations and thus a trade-off exists here between the different operators. This cost is expressed both in terms of computation as well as number of accesses, and thus it heavily affects the overall energy consumption.
2. The storage requirement of the DDT implementations also varies a lot based on the implementation chosen. Certain implementations require extra pointers to enable the management of the underlying memory. Others require no such overhead but do not allow splitting the memory used by the elements into smaller blocks, thereby potentially disabling putting part of the elements onto a smaller (and thus cheaper) memory. These trade-offs have a big influence on both the memory footprint as well as the mapping to the memory hierarchy, and indirectly also to the energy again.

3. The access behavior, or traversals of the elements inside a DDT, specifically consumption of the data, also has a big influence on the design choice of the DDT. As shown, DDTs can be consumed in a variety of ways. This can have a big impact both in terms of complexity in computation and memory accesses required for such traversals, as well as on the caching-behavior of the DDT in question.

2.3.5 Dynamic Memory Manager Optimizations

The final optimization step for dynamic data is the optimization of the dynamic memory allocator. This step should come after the optimization of the implementations of the DDT as these will affect the allocation and access pattern of the application. In the dynamic memory manager refinement (DMMR) step, information about the application in terms of allocation patterns as well as access patterns to the allocated blocks is used from the profiling metadata to determine which memory block types are more heavily used, and which less. With this information, it is possible to determine how to structure the memory allocator in terms of two specific aspects.

First, it helps with the design of the structure of the memory allocator. If many different block sizes are used by the application, then splitting and coalescing is definitely beneficial as otherwise a lot of memory footprint will be lost to fragmentation. Additionally, it helps to determine which memory blocks are often allocated, thereby allowing the decision as to which memory blocks should have their own custom freelists for quick allocation of these blocks.

Second, it helps with the placement of the memory blocks on top of memory pools. Memory pools are typically large blocks of memory that are placed on different elements of the memory hierarchy. The simplest approach is to have a memory pool per memory hierarchy element. Frequently accessed blocks should then be placed onto smaller memory pools that are closer to the CPU, while less frequently accessed blocks should be placed on memory pools further away from the CPU. A good metric is the number of accesses per byte, since it does not make sense to place a frequently accessed block on a smaller memory pool if this block is larger than two smaller pools that combined have more accesses.

To enable this entire optimization step, some exploration is required to find out the exact parameters that give the optimal memory allocator. As such, it is necessary to have a library of combinators that allows one to span the design-space of memory allocators. This set of combinators should be easily composable such that memory allocators of any complexity can be built instead. This book presents such a library of composable blocks for designing memory allocators in Chap. 6.

2.3.6 Task-level Data Transfer and Storage Exploration

After the previous platform-independent steps are done, there is an optimal mapping from the application's dynamic data types to memory pools. Using the memory pools that result from this heap data, along with the memory consumption and accesses due to stack and heap data, it is then possible to map this onto an actual memory hierarchy considering the interaction of different sets of tasks. This final phase, called Task-level Data Transfer and Storage Exploration (T-DTSE), is covered in [36]. However, the part covered in this book related to physical memory allocation and memory assignment (Chap. 7) is complementary to T-DTSE as it enables the application of access ordering, combined with several platform dependent enabling source-code transformations. These steps are possible thanks to the dynamic memory allocation and optimization phases described in the other chapters of this book (Chaps. 2–6).

2.4 Conclusions

In this chapter we have seen how the latest multimedia applications targeting new nomadic embedded systems (e.g., 3D games, video-players) share many features regarding multi-task execution and complex memory management, which makes them ideal candidates for dynamic data optimizations. These complex and dynamic implementations are derived from the initial desktop systems, which have much more memory and capabilities. Therefore, we have analyzed in this chapter the features of dynamic data in this set of new multimedia applications, and proposed a new design flow to optimize and map their dynamic memory management in nomadic embedded systems.

http://www.springer.com/978-3-319-10571-0

Dynamic Memory Management for Embedded Systems

Atienza Alonso, D.; Mamagkakis, S.; Poucet, C.;
Peón-Quirós, M.; Bartzas, A.; Catthoor, F.; Soudris, D.

2015, XIII, 243 p. 86 illus., 53 illus. in color., Hardcover
ISBN: 978-3-319-10571-0