Fast Prototyping and Refinement of Complex Dynamic Data Types in Multimedia Applications for Consumer Embedded Devices

David Atienza*[‡], Marc Leeman[†], F. Catthoor[‡], G. Deconinck[†], J. M. Mendias*, V. De Florio[†], R. Lauwereins[†]
* DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain.
[†] ESAT/KULEUVEN, Kasteelpark Arenberg 10, B3001 Leuven, Belgium.
[‡] IMEC vzw, Kapeldreef 71, 3001 Heverlee, Belgium.

Abstract— **Portable consumer devices are increasing more and more their capabilities and can now implement new multimedia algorithms that were reserved only for powerful workstations few years ago. Unfortunately, the original design characteristics of such algorithms do not often allow to port them directly to current embedded devices. These algorithms share complex and intensive dynamic memory use and actual embedded systems cannot provide efficient general-purpose memory management as it is needed. As a result, dynamic memory optimizations are a requirement when porting these applications. Within these optimizations, the refinement of the dynamically (de)allocated abstract data type implementations in the complex multimedia applications involved is one of the most important and difficult parts for an efficient mapping of the algorithms on low-power and high-speed embedded consumer devices. In this paper, we describe a high-level approach for modeling and refining complex data types using abstract derived classes in C++. This approach enables the multimedia developer to compose, evaluate and refine complex data types in a conceptually straightforward way, without a time-consuming programming effort.**

I. Introduction

Multimedia applications have experienced recently a very fast growth in their variety, complexity and functionality. This implies a high demand of memory and performance, which results in high cost and power consumption systems. These new algorithms (e.g. MPEG4) depend, with few exceptions, on Dynamic Memory (DM from now on) for their operations due to the inherent unpredictability of the input data, which heavily influences global performance and memory usage of these systems. In addition, energy has become a real issue in overall system design (both embedded and general-purpose) due to circuit reliability and packaging costs [12]. Thus, optimizations include three goals that cannot be seen independently: memory usage, power consumptions and performance.

Since the DM subsystem heavily influences performance and is a very important source of power consumption and memory usage, system-level exploration mechanisms must be available at an early stage of the design flow for embedded systems. Unfortunately, general approaches do not exist presently at this level for the dynamic implementations of the Abstract Data Types (ADTs) involved, which are sets of data values and associated operations that are specified independently of any particular implementation [4]. This definition stresses more on the effects of operations than the language-specific implementation of the ADTs (or data type implementation). In the following, we will focus on the implementation of these dynamically (de)allocated abstract data types (or DDTs for short from now on). When DDTs are used in programs, they can be implemented by the developer in the most naive form (because the developer is more focussed on the algorithm itself) or in a manually optimized implementation where the number of implementation alternatives is defined by the experience and inspiration of the developer. Adding new implementations of (complex) DDTs often proves to be programming intensive. Even when standardized languages (if used at all) offer considerable support, the developer still has to define the access pattern on a case per case basis.

In this paper we propose a programming methodology based on template and abstract derived classes or mixins [10] that can be used to build and refine complex layered DDTs from basic ones in a modular way. This allows to cover a huge part of the DDT search space with a minimal code base, which allows to obtain early design-flow estimates on implementation trade-offs to refine the system design.

II. Related Work

Regarding DDTs refinement work, general-purpose libraries are available (e.g. in C++) to help designers to develop new algorithms [3] without being worried about complex DDT implementation issues. These libraries usually provide interfaces to simple DDT implementations and the construction of complex ones is responsibility of the developer. Furthermore, these libraries focus exclusively on performance and while they can be considered as acceptable general-purpose solutions, they are not suitable for new generation embedded devices, where performance, power consumption and memory footprint must be optimized. Presently, suitable access methods and power-aware DDT transformations have started to be proposed for multimedia systems [5].

Also, according to the characteristics of certain parts of multimedia applications, several transformations for DDTs [13] and design methodologies [9] have been proposed for static data profiling and optimization considering static memory access patterns to physical memories.

New methods for a modular construction of high-level components in DDT refinement can be envisaged with *abstract derived classes* or mixins [10]. This programming technique has been inherited by functional programming languages (e.g.

0-7803-8603-5/04/$20.00 ©2004 IEEE

Lisp) and has already been used for quite some time in large reusable modules of object-oriented design.

Finally, compiler techniques for code compaction and minimization of energy and power consumption in high-performance systems are also available [6], [1].

III. Dynamic Data Type Characterization and Exploration

When analyzing DDTs, we divide them into two basic types: arrays and graphs. An array contains values (i.e. scalar, complex or even abstract ones when storing pointers to other data). The most important property of an array is that the entire structure is allocated on one shoot, but the memory can be allocated at compile time (static memory) or at run time (dynamic memory). On the other hand, graphs consist of multiple nodes that can store a scalar or complex value each. Furthermore, it contains at least one pointer to another similar memory location (that can be NULL). The combination of all connected nodes forms the complete graph. Contrary to arrays, nodes in a graph are allocated and freed incrementally: when more data needs to be stored, additional nodes are allocated. In most cases, two special cases of graphs are used: trees and lists. In a tree, each node has one *parent* node and at least one *child* node. Exceptions are the *root* node that has no parent and the *leaf* nodes that have no children.

Complex layered data types are combinations of the two aforementioned basic types. In a typical example, the overhead memory in linked lists (i.e. the pointers to the next and previous nodes) are amortized by allocating memory for several elements at once. In this case, the first layer is a linked list and the second one is an array. As a result of the possible combinations of basic types in complex layered structures, the search space of these complex data types grows exponentially and a systematic exploration and construction method becomes a must.

A. Modular Dynamic Data Types

We have developed a flexible and extendible way to explore the search space of complex DDT implementations using a C++ modeling approach based on template C++ classes [11] and mixins [10]. This approach allows easy and flexible modeling and refinement of layered DDTs. In the remainder of the text, we use the definition of mixins as used in [10]: a method of specifying extensions of a class without defining up-front which class exactly it extends. In C++, a subclass is specified and the parameterized parent class is determined during the instantiation of the code later on.

```
template <class SupClass> class Mixin:
    public SupClass{ // mixin definitions };
template <class SupClass> class TemplateClass{
    SupClass* data;
    // template class definitions };
```

Fig. 1. Parametrized Inheritance used with mixins in C++

Figure 1 declares a subclass of SupClass with SupClass itself being a template argument. Since we are using the inclusion method [11], also the subclass is defined. The Mixin is written for one or more types that are not yet specified. A small variant can be seen in the second class definition (i.e. TemplateClass), where the template argument is not used as a parent class, but instead as internal private data members. We use the second approach of template classes to model the memory behavior of DDT implementations, while the mixin approach is used to add, modify and refine functionality of complex DDTs.

As our first approach, we have implemented the typical primitive data structures used in multimedia applications [8], [13] using our mixins-based layered fashion. An example of these simple DDTs are shown in the middle part of Figure 2. Two different sorts of simple lists are implemented, single linked (i.e. SLList) where each node is pointing to the next one only and doubly linked (i.e. DLList) where every node points to the next and previous node. Also, a binary tree (i.e. BTTree or BTT) and array (Array) are implemented. All of them include default interface methods like GetElement, AddElement, DelElement, etc. These simple DDTs are the only DDTs that need to be specified/written to compose more complex ones in most dynamic multimedia applications. Then, to use the template code the developer just needs to specialize them. For example, Figure 2 defines a number of base classes: an array of 256 float elements, a doubly linked list of ints and a binary tree of doubles. In these cases, the last parameter is a combination of two mixins (first line in Figure 2): First, mheap is a thin wrapper for the memory operations malloc() and free(). Second, TypeClass is used as intermediate layer to retrieve profiling information of the DDTs as we show later in Subsection III-B (also define the basic data types of the layers, e.g. float, int).

```
template<type T> class Mem:public TypeClass<T,mheap>{};
template<int Size,type T,class SupClass> class Array{..};
template<type T,class SupClass> class SLList {...};
template<type T,class SupClass> class DLList {...};
template<type T,class SupClass> class BTTree{...};
class F_Array: public Array<256,float,Mem<float> >{};
class I_DLList: public DLList<int, Mem<int> >{};
class D_BTTree: public BTTree<double, Mem<double> >{};
```

Fig. 2. Basic Abstract Data Types definitions and instantiations

Next, these basic DDT layers can be combined in complex multi-layered DDTs as shown in Figure 3. In the first case of Figure 3, a doubly linked list of arrays of 128 integers is declared (i.e. DLLAR). Also, a binary tree of a single linked list is defined (i.e. BTSLL), followed by a multi-layered array structure (i.e. ARARAR) with an array of 2, pointing to an array of 4, pointing to arrays of 2 Point3D elements. These can be typical custom DDTs of a real multimedia application [8].

```
class DLLAR: public \
  DLList<int,ArrayClass<128,int,Mem<int> > >{};
class BTSLL: public \
 BTTree<double,SLList<double,Mem<double> > >{};
class ARARAR: public \
 ArrayClass<2,Point3D,ArrayClass<4, Point3D,\
    ArrayClass<2,Point3D,Mem<Point3D> > > >{};
```

Fig. 3. Examples of multi-layered DDTs

B. Additional Extensions to Distinguish Access Methods and Obtain Accurate Profiling

Then, using our mixins-based DDTs construction approach, the previous modular DDTs (Subsection III-A) can be split in two parts: the memory part of a DDT is put in a template class, while the access part is placed in an abstract derived class. Thus, to define a DDT, the user can specify the memory and access component. This splitting allows to separate the access method (e.g. using keys, sequential access) to a DDT from its internal complex data structure, allowing different access patterns to similar DDTs with minimal changes in the code.

```
template<typename T> class DLList: public \
    DLListDefAccess<T,DLListMem<Mem<T> > >{};
template<typename T, typename Key> \
 class DLListKey: public \
   DLListK1Access<Key,T,DLListMem<Mem<T> > >{};
```

Fig. 4. Double Linked Lists with different access methods

Figure 4 illustrates the aforementioned process. In the first line, the developer specifies that he wants to use doubly linked lists. Furthermore, the DDT will be accessed using a *default* policy. In the second example, again a `DLList` is used as storage primitive, but this time the data will be identified using a `Key`. The specific access method is defined in the *K1* policy (the `DLListK1Access` mixin). In these examples, the *access roles* extend the memory component of the DDT. It is important to remark how the definitions in Figure 4 are the only code the developer has to specify to form a complex DDT. As such, it provides additional abstraction compared to current well-known extendible libraries. For example, an additional layer of abstraction is added with respect to the Standard Template Library (STL) [3], where most of the STL containers (similar to DDTs) include a standard access method and adding a custom one is extremely difficult (or even impossible if the code needs to be changed extensively).

Finally, we can use the same layered mixin-based technique to build DDTs with memory profiling support. This is done by inserting an additional layer between the complex DDT definition and the parametrized class that handles allocations and frees. In this case, the parametrized class `Mem`, serves as a simple allocator class (Figure 5). The basic form is composed of two simple layers: `mallocheap` is a simple wrapper around the `malloc()` function while `TypeClass` provides some basic functionality to tell derived classes if they are operating on memory or on other DDTs (i.e. multi-layered DDTs). The `LogLayer` provides functionality for the standard function calls of `C` for memory (de)allocation and for storing and retrieving data (e.g. `free()`, `malloc()`, `get()`, `add()`). It catches the requests made by lower objects in the hierarchy and prints them together with a timestamp to `stdout`. Thus, the profiling of memory requests is obtained by inserting the `LogLayer` around the `TypeClass` (see definition in the lower part of Figure 5 and also Figure 6).

Figure 6 shows the previously explained process in a graphical way. The `malloc()`, `free()`, `GetElem()` and `AddElem()` methods are inherited by the derived classes or explicitly called. A derived class can explicitly call the methods of the superclass it has overloaded by explicitly specifying the scope (e.g. in the method `GetElem()` of `LogLayer`, it calls `SuperClass::GetElem()`).

```
template <class SupClass> class LogLayer: public SupClass{
 inline void* malloc(size_t sz){
    struct timeval thetime; struct timezone tz;
    gettimeofday(&thetime,&tz);
    EPRINT("%lu alloc. %d bytes\n",
      thetime.tv_sec*1000000+ thetime.tv_usec, sz);
    return SuperClass::malloc(sz); };
};
template<typename T> class Mem : public \
     LogLayer<TypeClass<T,LogLayer<mallocheap> > >{};
```

Fig. 5. Extract of a Simple Memory Usage Profiling Layer

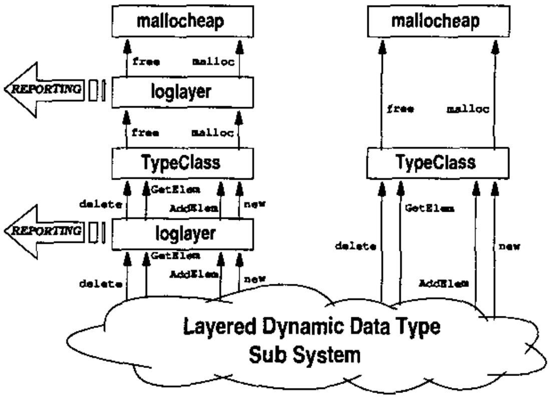

Fig. 6. Graphical representation of the logging process of memory accesses and allocations with our mixins-based layers

These up-going requests are shown with the arrows. As such, the `LogLayers` catch the requests and reissue them. For the original layers nothing has changed, the addition of an additional layer is transparent to the existing ones. It is exactly this behavior which makes abstract derived classes so interesting and flexible.

IV. Method Application and Results

We have applied our profiling and refinement mixin-based approach to the DDTs of a recent and complex 3D multimedia

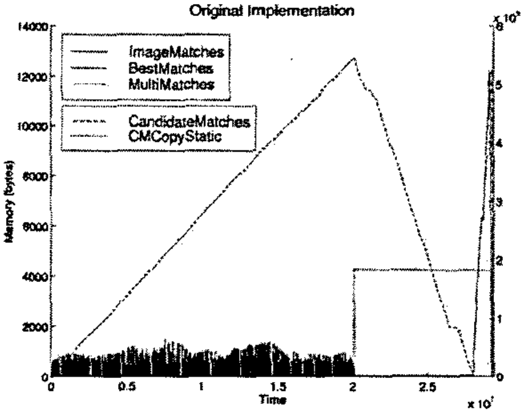

Fig. 7. Memory footprint over time of the DDTs.All plots mapped on the left axis, except `CandidateMatches` and `CMCopyStatic` (right axis)

TABLE I

ORIGINAL DDTS IN THE 3D IMAGE RECONSTRUCTION SYSTEM

Variable	memory accesses	memory footprint (B)	energy .13μm tech.(μJ)
IMatches	1.20×10^{6}	5.14×10^{2}	0.18×10^{3}
CMatches	8.44×10^{5}	2.75×10^{5}	3.03×10^{3}
CMCStatic	6.24×10^{4}	1.08×10^{5}	4.48×10^{4}
MMatches	1.84×10^{4}	3.62×10^{2}	0.02×10^{1}
BMatches	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}

application, i.e. a matching algorithm that forms the corner stone of a 3D image reconstruction algorithm (see [8] for references to the full code of the algorithm with 1.75 million lines of high level C++). It creates the mathematical abstraction from the related frames that is used in the global algorithm by matching corners detected in 2 subsequent frames [8]. In our experiments we have matched a sequence of 101 images (i.e. 100 matchings between 2 images each). The operations done on the images are particularly memory intensive, e.g. each image with a resolution of 640 x 480 uses over 1Mb, and the accesses of the algorithm to the images are randomized. Thus, classic image access optimizations as row-dominated accesses versus column-wise accesses are not relevant.

In the matching algorithm we have replaced the initial DDTs (see Table I) with our mixin-based code. These DDTs were originally implemented using variations of double linked lists and exhibit an unpredictable memory behaviour, typical in many state-of-the-art 3D vision systems [8] since they use some sort of *dynamic candidate selection* followed by *a criterion evaluation*. After using our mixin-based DDTs, the initial implementation of the main DDTs of this application can be easily profiled and a memory use graph is generated (Figure 7). Then, memory accesses, memory footprint and energy dissipation figures are calculated (see Table I). For the energy estimations, we have used a complete energy/delay/area model for embedded SRAMs [2] that can scale to different technology nodes (we use the .13 μ node for the results).

The analysis of the profiling information (Table I and Figure 7) shows how the DDTs affect the system. First, CMatches is the largest DDT. Secondly, IMatches has frequent accesses. Finally, CMCStatic (an "optimized dynamic array" that reduces the accesses to CMatches) consumes an important part of the energy used by the system. Then, using our mixin-based approach we can refine these DDTs observing that part of the elements of the DDTs are accessed almost at the same time and thus an intermediate dynamic-static solution where part of the elements are allocated at one shoot would be better. Thus, the final optimized DDT implementations consist of 2-layered dynamic array structures (pointer-arrays to arrays). First, an external dynamic array of 10 positions; then, each position is another array of 756, 1024 or 16384 Bytes (B) depending on which DDT. With these optimized DDTs, CMatches is now fast enough to interact directly with BMatches and Mmatches, thus CMCStatic is removed. The results obtained are depicted in Table II, which show an improvement of 66.84% in memory footprint, 97.93% in power consumption and eventually up to 95.08% in performance compared to the original version. This proves that the proposed mixin-based method can help designers to efficiently improve their original implementations. Furthermore, to evaluate the design effort with our approach, remark that the optimization of the application took us two weeks. Though no space is available to show them in detail, similar results have been achieved in a 3D game engine (i.e. 65% of improvement in power consumption and 80% in execution time).

TABLE II

FINAL DDTS AFTER OUR MIXIN-BASED REFINEMENT

Variable	memory accesses	memory footprint (B)	energy .13μm tech.(μJ)
IMatches	4.02×10^{5}	6.84×10^{2}	2.91×10^{2}
CMatches	3.89×10^{5}	1.27×10^{5}	7.02×10^{2}
MMatches	7.68×10^{3}	3.81×10^{3}	0.03×10^{1}
BMatches	7.16×10^{3}	3.78×10^{3}	0.02×10^{1}
Total	8.06×10^{5}	1.28×10^{5}	9.98×10^{2}

V. CONCLUSIONS

Presently, embedded devices have increased their capabilities and now complex applications (e.g. multimedia) can be ported to them. Such applications include intensive DM requirements that must be heavily optimized (i.e. memory footprint, power and memory use) for an efficient mapping on current consumer embedded devices. System-level refinement methods are proposed to consistently perform that refinement. Within them, the manual exploration and optimization of the DDT implementations involved are the most time-consuming and programming intensive parts. In this paper we have presented a high-level programming method based on template classes and abstract derived classes (or mixins) that can be used to model complex DDTs from basic ones in a modular way. This method largely simplifies the memory structuring aspect of multi-layered DDTs for the developers and allows them to refine the DDT implementations of their multimedia applications with a minimal effort, thus helping them in the error-prone task of manual characterization of complex DDTs and leading them to important savings in memory footprint, power consumption and performance.

REFERENCES

[1] N. Bellas et al. Architectural and compiler tech. for energy reduc. in high-perf. microproc. *IEEE Trans. on VLSI Systems*, june 2000.

[2] B. S. Amrutur et al. Speed and Power Scaling of SRAM's. *IEEE Trans. on Solid-State Circuits*, 2000.

[3] C++ Standardisation Comittee. Programming languages – C++ – ISO/IEC 14882. Tech. report, USA, 1998.

[4] T. H. Cormen et al. *Introduction to Algorithms*. McGraw-Hill, 1994.

[5] E. G. Daylight et al. Analyzing energy friendly states of dyn. apps. in terms of sparse DTs. In *Proc. of ISLPED*, USA, 2002.

[6] S. Debray et al. Compiler techniques for code compaction. *ACM Trans. on Prog. Lang. and Systems*, March 2000.

[7] National Institute of Standards and Technology. 2003 http://www.nist.gov/dads/HTML/abstractDataType.html.

[8] M. Pollefeys et al. Metric 3D surface reconstruction from uncalibrated image seq. In *LNCS*, 1506: 139 – 153. 1998.

[9] A. Smailagic et al. Benchmarking an interdiscip. concurrent design method. for electronic/mech. systs. In *Proc. of DAC*, USA 1995.

[10] Y. Smaragdakis et al. Mixin-based program. C++. *LNCS*, 2001.

[11] D. Vandevoorde and N. M. Josuttis. *C++ Templates, The Complete Guide*. Addison Wesley, London, UK, 2003.

[12] N. Vijaykrishnan et al. Evaluating integrated HW-SW optim. using a unif. energy estim. framework. *IEEE Trans. on Computers*, 2003.

[13] S. Wuytack et al. Global communication and mem. optim. transformations for low power systems. In *WLPD*, USA, 1994.