Automated Dynamic Memory Data Type Implementation Exploration and Optimization

Marc Leeman[⋆,*], Chantal Ykman[‡], David Atienza[‡,†], Vincenzo De Florio[⋆] Geert Deconinck[⋆]
ESAT K.U.Leuven
Kasteelpark Arenberg 10
B-3001, Leuven (Heverlee), Belgium
[⋆] *firstname.lastname*@esat.kuleuven.ac.be. [‡] *lastname*@imec.be.

Abstract

The behavior of many algorithms is heavily determined by the input data. Furthermore, this often means that multiple and completely different execution paths can be followed, also internal data usage and handling is frequently quite different. Therefore, static compile time memory allocation is not efficient, especially on embedded systems where memory is a scarce resource, and dynamic memory management is the only feasible alternative. Including applications with dynamic memory in embedded systems introduces new challenges as compared to traditional *signal processing applications. In this session, an automated framework is presented to optimize embedded applications with extensive use of dynamic memory management. The proposed methodology automates the exploration and identification of* optimal *data type implementations based on power estimates, memory accesses and normalized memory usage.*

1 Introduction

In order to optimize algorithms, the embedded developer must be aware of a number of constraints: these systems have typically a more constrained memory hierarchy than General Purpose Processors (GPP) where algorithms typically are developed upon. Furthermore, the optimization objectives are not only dominated by performance, but memory accesses and power consumption are (at least) equally important.

When memory usage is deterministic, all memory usage and access patterns can be determined at compile time. The required memory is reserved in one single allocation and freed when no longer needed. Moreover, since this allocated memory is contiguous, the addresses of the elements are known and accessible in a minimum of time. However, this is not longer certain when Dynamic Memory Management (DMM) is required.

MATISSE [4, 3, 1, 2] is a methodology for system-level dynamic memory management. It is intended for applications operating on large and irregular data structures, dynamically allocated and stored in sets, called *dynamic data sets*. It uses a fast, stepwise and cost-driven exploration and specification refinement at the system level of the needed dynamic data sets. As a result, optimized distributed memory architectures can be achieved in embedded systems.

In the remainder, automation and modification of the first sub-step of the MATISSE methodology[1] is discussed in order to handle applications with multiple dynamic data sets which change their behavior during the application: *Abstract Data Type refinement* (ADT). In addition, a comparison of dynamic memory usage between the original version and the optimized one of the main dynamic data type in the application is shown in Figure 2 to illustrate the final results accomplished.

2 Method and Results

In an ADT context, data dependencies can be divided into two distinct types. First, *design or algorithmic dependencies* dependencies built in the program as a result of the control flow of the algorithm, e.g. ordered or unordered

[*]Partially supported by the Fund for Scientific Research - Flanders (Belgium, F.W.O.) through project G.0036.99 and a Postdoctoral Fellowship for Geert Deconinck.

[†]Also at DACYA/UCM. Madrid, Spain. Partially supported by the Spanish Government Research Grant TIC2002/0750 and E.C. Marie Curie Fellowship contract HPMT-CT-2000-00031.

[1]The first sub-step generates for each dynamic data set an optimized implementation. This is called *Abstract Data Type (ADT) Transformation and Refinement*. The second step organizes the dynamic memory space and selects custom memory managers to handle dynamic memory (de)allocation of the needed data structures. This sub-step is call *Dynamic Memory Management Refinement*.

Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)
0-7695-1904-0/03 $17.00 © 2003 IEEE

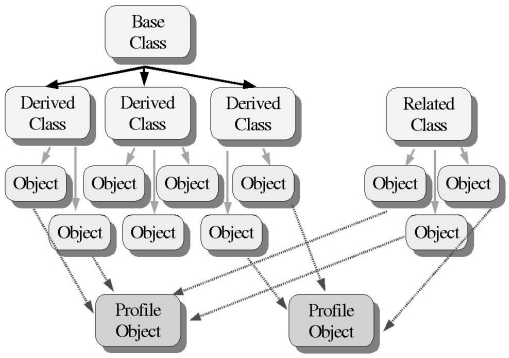

Figure 1. Automatic Profile Framework with application objects reporting their behavior to profile objects.

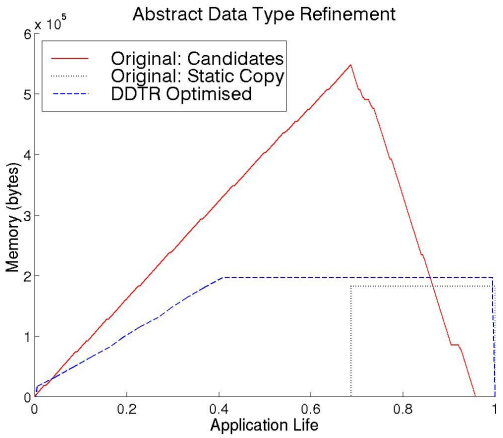

Figure 2. Result of the profiling: the memory usage of dynamic data types is plotted. In this case, the data types of *Candidates* and *Static Copy* are optimized. The DDTR optimized plot shows the result after removal of a crucial *application dependency*.

sets. This kind of dependencies cannot be removed and/or changed without modifying the algorithm itself. Second, data dependencies that are *implementational data dependencies*. These are introduced while moving from specification to implementation due to rigid (immutable) choices that concern e.g. the data type implementation. Once these choices are made, a set of data dependencies will affect the cost of the final implementation, e.g. cache conscious structure reorganizations. These data type induced dependencies are avoidable as long as the final application is not deployed on the target system.

MATISSE employs three basic data types, namely *lists, arrays and trees*, and organizes them in multi-layered structures. All these structures share a number of basic operations like `add`, `insert`, `delete`, etc. In addition, they can have multiple implementations, reflecting different features of the pursued final system, e.g. a different garbage collection system. As a result, this rises exponentially the search space as more alternatives are considered.

This MATISSE methodology provides as one of its key concepts a representative estimate on algorithm and code changes early in the development flow for system developers. Thus, it reduces the number of ulterior costly and time consuming reconfigurations during system design. In order to have these early estimates on the implementation and on important objectives (e.g. power, memory usage and access), MATISSE automatically adds timing information at the source code level.

The driver application that is presented is part of a larger 3D image processing application and has several properties that challenge dynamic memory optimization:

- The application has 4 dynamic data sets with several distinct access patterns.
- The behavior of some data types changes during the application run.

The presence of multiple data sets required a new profiling and instrumentation framework (see Figure 1) both easy to use in an automated code instrumentation tool and flexible enough to handle all kinds of combinations and loose relationships between the data sets. Since MATISSE uses C++ as an abstract data type description language, the profiling information cannot be collected at the object level. In fact, several objects, possibly from different classes can define a multi-layered dynamic data type description. Therefore, the profiling framework decouples this information from the data type class hierarchy. Furthermore, these objects are able to report their memory and timing behavior to *dedicated* profiling objects without the intervention from the developer, enhancing their versatility. Hence, the number of these objects depends on the amount of data type occurrences visible from the application (e.g. if an application has 2 *variables* to be optimized, 2 profiling objects are created during the code transformation). . The code transformation accomplishes several tasks, next to basic code transformations, a reconstruction of the class hierarchy, an instrumentation of the dynamic data type definition code and the creation and and inclusion of the MATISSE specific code. This instrumented source code is used to compose run time information on memory accesses and usage, timing information and function calls. Because the data sets change their behavior at run time, normalized memory usage is used to compute power estimates.

As a matter of fact, in the presence of *multi-objective* optimizations (optimizing for memory accesses, usage and power consumption), a single optimal point is unlikely. Instead, *Pareto* optima are constructed during exploration of

Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)
0-7695-1904-0/03 $17.00 © 2003 IEEE

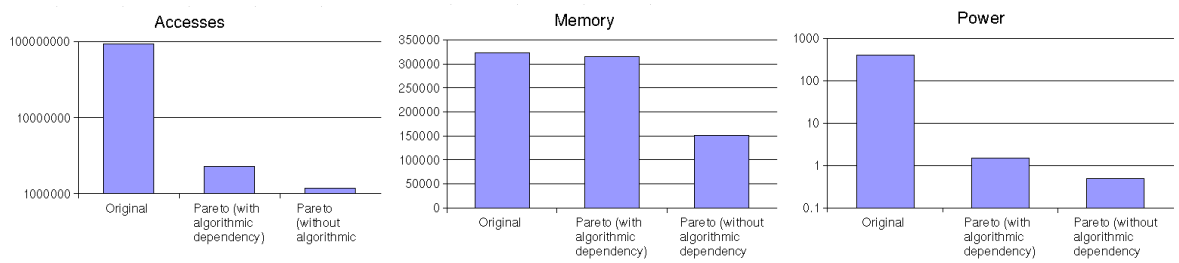

Figure 3. Effect of applying `Matisse` on the driver application in a logarithmical scale.

the search space[2].

The output of the simulation gives feedback to the developer about possible algorithmic bottlenecks, while the implementational ones are optimized automatically:

- Profiling information can indicate clear algorithmic problems that were hidden in the code. In one example, a very high access count in one part of the program was identified while scanning a large data structure. After a closer code inspection, the algorithm assumed that the data was created without order and had to access all the elements in order to construct exhaustive subsets. In fact, the data did contain order and the subsets were located one after another.

- In a different part of the execution of the application, data in one large dynamic data structure was destroyed before the following dynamic data set was constructed. This showed a *hidden* matrix where the data was copied for faster access. As a consequence of this, the original implementation doubled the memory bottleneck. With the combined profiling information, this was removed safely, **without** compromising the performance.

Like many multimedia applications, the driver application has multiple dynamic data types. In figure 2, the result of the Dynamic Data Type Refinement (DDTR) step for one of them is illustrated. As the figure shows, the large `Candidates` list is copied in a dynamically allocated array. From that point onwards, this array is used for fast access to the data that is processed. Even though this obviously speeds up the application to a great extent, it comes at a high price since there is a duplication of the memory bottleneck when the data is copied. In addition, Figure 2 shows the dynamic memory usage for the DDTR optimized version during the application execution. Evidently, an efficient selection of the dynamic data type results in a much lower memory usage and a faster execution (note that while the data is being copied to the fast array, the algorithm cannot make progress).

[2] A point is *Pareto Optimal* when it is no longer possible to improve one of the objectives without worsening another.

As a summary, the first step is to change algorithmic dependencies before examining algorithmic ones. Then, once the optimal hierarchical implementation is selected for each data set under investigation, the code is instantiated (obviously without the profiling information of the framework). In fact, for a number of implementations, `C` counterparts of the `C++` code are available, but their instantiation is currently not included in the tools. These `C` definitions have the advantage that they eliminate much of the `C++` code size overhead.

For the driver application where we have applied our methodology, it allows to improve memory footprint, memory accesses and power by 98.5 %, 51.7 % and 99.9 % respectively, and is illustrated in Figure 3. In the bar charts, the results for memory accesses, memory usage and power are plotted for the original implementation, a DDTR optimized version where the copy to the array (see Figure 2) is kept and the case where both DDTR and the removal of the *algorithmic dependency* are applied . Similar behavior has been identified in several multimedia applications and algorithms, thus, without any doubt, they can be optimized with this methodology as well.

References

[1] D. Verkest, J. da Silva, C. Ykman, K. Croes, M. Miranda, S. Wuytack, G. de Jong, F. Catthoor, and H. De Man. Matisse: A system-on-chip design methodology emphasizing dynamic memory management. *Journal of VLSI Signal Processing*, 21(3):277–291, July 1999.

[2] S. Wuytack, J. da Silva, F. Catthoor, G. de Jong, and C. Ykman. Memory management for embedded network applications. *IEEE Transactions on Computer-Aided Design*, 18(5):533–544, May 1999.

[3] C. Ykman, J. Lambrecht, A. Van der Togt, and F. Catthoor. Multi-objective abstract data type refinement for mapping tables in telecom network applications. In *ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, June 2002.

[4] C. Ykman, J. Lambrecht, D. Verkest, F. Catthoor, B. Svantesson, A. Hemani, and F. Wolf. Dynamic memory management methodology applied to embedded telecom network systems. *Accepted for publication in IEEE Transactions on VLSI Systems*, 2002.

Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'03)
0-7695-1904-0/03 $17.00 © 2003 **IEEE**