The Journal of Systems and Software 91 (2014) 109–123

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A methodology to automatically optimize dynamic memory managers applying grammatical evolution

José L. Risco-Martín[a,*], J. Manuel Colmenar[b], J. Ignacio Hidalgo[a], Juan Lanchares[a], Josefa Díaz[c]

[a] Department of Computer Architecture and Automation, Universidad Complutense de Madrid, 28040 Madrid, Spain
[b] C.E.S. Felipe II, Universidad Complutense de Madrid, 28300 Aranjuez, Spain
[c] C.U. Mérida, Universidad de Extremadura, 06800 Mérida, Spain

ARTICLE INFO

Article history:
Received 11 February 2013
Received in revised form 17 July 2013
Accepted 14 December 2013
Available online 8 January 2014

Keywords:
Genetic programming
Dynamic memory manager
Multi-objective optimization

ABSTRACT

Modern consumer devices must execute multimedia applications that exhibit high resource utilization. In order to efficiently execute these applications, the dynamic memory subsystem needs to be optimized. This complex task can be tackled in two complementary ways: optimizing the application source code or designing custom dynamic memory management mechanisms. Currently, the first approach has been well established, and several automatic methodologies have been proposed. Regarding the second approach, software engineers often write custom dynamic memory managers from scratch, which is a difficult and error-prone work. This paper presents a novel way to automatically generate custom dynamic memory managers optimizing both performance and memory usage of the target application. The design space is pruned using grammatical evolution converging to the best dynamic memory manager implementation for the target application. Our methodology achieves important improvements (62.55% and 30.62% better on average in performance and memory usage, respectively) when its results are compared to five different general-purpose dynamic memory managers.

1. Introduction

Nowadays, multimedia applications are mostly developed using C++. This kind of software programs tends to make intensive use of dynamic memory due to their inherent data management. However, in C++, dynamic memory is allocated via the operator `new()` and deallocated by the operator `delete()`, which are mapped directly to the `malloc()` and `free()` functions of the standard C library in most compilers. Therefore, the creation and destruction of objects is managed by a general-purpose memory allocator, which may provide good runtime and low memory usage for a wide range of applications (Johnstone and Wilson, 1999; Lea, 2010).

However, using specialized *Dynamic Memory Managers (DMMs)* that take advantage of application-specific behavior can dramatically improve application performance (Barrett and Zorn, 1993; Grunwald and Zorn, 1993). In this regard, three out of the twelve integer benchmarks included in SPEC (`parser`, `gcc`, and `vpr` (SPEC, 2013)) and several server applications, use one or more custom DMMs (Berger et al., 2001).

On the one hand, studies have shown that dynamic memory management can consume up to 38% of the execution time in C++applications (Calder et al., 1995). Thus, the performance of dynamic memory management can have a substantial effect on the overall performance of C++applications. On the other hand, new multimedia devices must rely on dynamic memory for a very significant part of their functionality due to the inherent unpredictability of the input data. These devices also integrate multiple services such as multimedia and wireless network communications, which also compete for memory space. Then, the dynamic memory management influences the global memory usage of the system (Atienza et al., 2006b). Finally, energy consumption has become a real issue in overall system design due to circuit reliability and packaging costs (Vijaykrishnan et al., 2003). However, it has been recently proved that the DMM consumes only a 1% of the total enery consumption by the memory subsystem usually in the execution of a given application (Díaz et al., 2011). Thus, the energy consumption by the DMM is not relevant on this case and the optimization of the dynamic memory subsystem has two goals that cannot be seen independently: performance and memory usage. There cannot exist a memory allocator that delivers the best performance and least memory usage for all programs. However, a custom memory allocator that works best for a particular program can be developed using grammatical evolution (Risco-Martin et al., 2009).

To reach higher performance, programmers often write their own *ad hoc* custom memory allocators as macros or monolithic

* Corresponding author. Tel.: +34 3947603.
E-mail address: jlrisco@dacya.ucm.es (J.L. Risco-Martín).

0164-1212/$ – see front matter © 2014 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.12.044

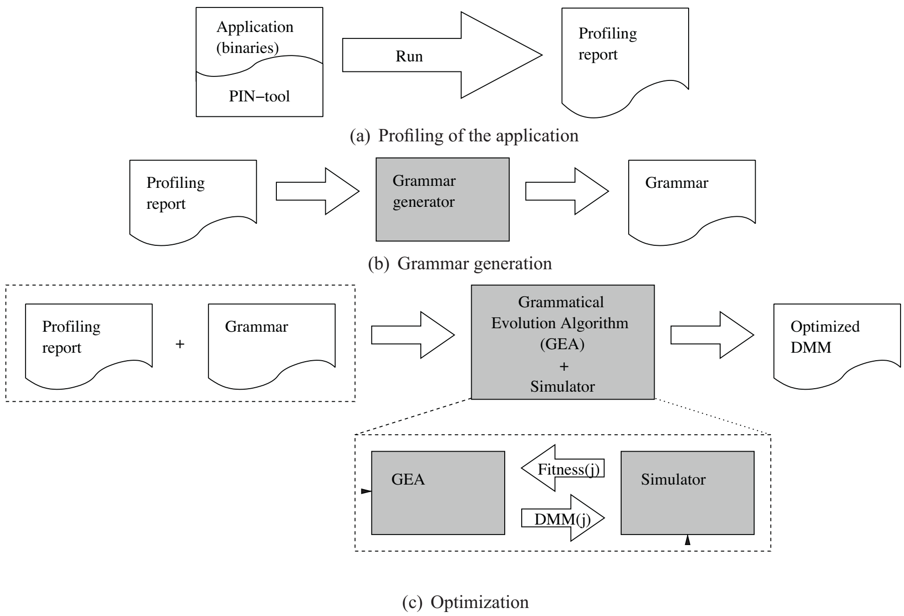

Fig. 1. DMMs optimization flow. In the first phase, we generate an initial profiling of the de/allocation pattern. In the second phase, we automatically analyze the profiling report to generate the final grammar. Finally, in the third phase an exploration of the design space of DMMs implementation is performed using GE.

functions in order to avoid function call overhead. This approach, implemented to improve application performance, is enshrined in the best practices of skilled computer programmers (Meyers, 1995). Nonetheless, this kind of code is brittle and hard to maintain or reuse, and as the application evolves, it can be difficult to adapt the memory allocator as the application requirements vary. Moreover, writing these memory allocators is both error-prone and difficult. Indeed custom and efficient memory allocators are complicated pieces of software that require a substantial engineering effort.

In this work, we have developed a framework based on grammatical evolution to automatically design optimized DMMs for a target application, minimizing memory usage and maximizing performance. Fig. 1 depicts the optimization process. First, as Fig. 1(a) shows, we run the application under study together with an instrumentation tool, which logs all the required information into an external file: identification of the object created/deleted, operation (allocation or deallocation) object size in bytes and memory address. Since all the DMM exploration process is performed simulating the generated DMMs with the profiling report, this task must be done just once. In the following phase, as Fig. 1(b) shows, we automatically examine all the information contained in the profiling report, obtaining a specialized grammar for the target system. As a result, some incomplete rules in the original grammar (see Section 5), such as the different block sizes, are automatically defined according to the obtained profiling. To this end, we have developed a tool called *Grammar Generator*. The last phase is the optimization process. As Fig. 1(c) depicts, this phase consists of a *Grammatical Evolution Algorithm (GEA)* that takes the grammar generated in the previous phase and the profiling report of the application as inputs. GEA is supported by a DMM simulator that tests the behavior of every DMM generated by the grammar applied to the application. Our GEA is constantly generating different DMM implementations from the grammar file. When a DMM is generated (*DMM*(*j*) in Fig. 1(c)), it is received by the DMM simulator. Then, the simulator emulates the behavior of the application, debugging every line in the profiling report. Such emulation does not de/allocate memory from the computer like the real application, but maintains useful information about how the structure of the selected DMM evolves in time. After the profiling report has been simulated, the DMM simulator returns back the fitness of the current DMM to the GEA. The fitness is computed as a weighted sum of the performance and memory usage by the proposed DMM for the target device and application under study. Finally, the DMM with lowest fitness is returned as solution (optimized DMM).

The rest of the paper is organized as follows. First, Section 2 describes some recent advances in the area of DMMs. Next, Section 3 defines the design space of memory allocators. Then, Section 4 details the design and implementation of the DMM simulator, as well as some configuration examples. Section 5 details how grammatical evolution is applied to the DMM optimization. Section 6 shows our experimental methodology, presenting the six benchmarks selected, whereas Section 7 shows the results for these benchmarks. Finally, Section 8 draws conclusions and future work.

2. Related work

Several approaches have been presented in the last decade to design flexible and efficient infrastructures for building custom and general-purpose memory allocators (Berger et al., 2001; Atienza et al., 2006b,a). All the proposed methodologies are based on high-level programming where C++templates and object-oriented programming techniques are used. They allow the software engineer to compose both general-purpose and custom memory allocator mechanisms. The aforementioned methodologies enable the implementation of custom DMMs from their basic parts (e.g., de/allocation strategies, order within pools, splitting, coalescing, etc.). In addition, Atienza et al. (2006b) and Atienza et al. (2006a) provided a way to evaluate the memory usage and energy consumption, but at system-level. However, all the previously mentioned approaches require the execution of the target application to evaluate every candidate custom DMM, which is a very time-consuming task, especially if the target application requires human inputs (like video games). In this regard, Lo et al. (2004) and Teng et al. (2008) presented two DMM design frameworks that allow the definition of multiple memory regions with different disciplines. However, these approaches are limited to a small set of user-defined functions for memory de/allocation. Furthermore, the selection of the "best" DMM is based on a set of predefined rules and mono-objective search, respectively. Thus, new multi-objective