Early Phase Memory Leak Detection in Embedded Software Designs with Virtual Memory Management Model

Mabel Mary Joy[1] Wolfgang Müller[2] Franz J. Rammig[3]

[1,2]C-Lab, University of Paderborn, Germany, {mabeljoy,wolfgang}@c-lab.de
[3]Heinz Nixdorf Institute, University of Paderborn, Germany, franz@uni-paderborn.de

Abstract

Virtual platforms are gaining significant importance in early design tests of embedded software as it helps to redesign or optimize the system well advance in time and keeps flaws minimal in the production stage. As embedded system's size gets smaller, expensive resources like memory are limited. Hence memory needs to be managed efficiently and optimally. Memory leak is a serious issue that leads to wastage of expensive memory. We propose a novel approach to detect memory leaks in early design stages of soft real-time systems with no garbage collection. Our approach utilizes a virtual platform modeled in SystemC at an abstract level using Transaction Level Modeling. The software under test is run on top of this model. Potential memory leaks in the software are detected by applying a novel hybrid method combining both static and dynamic approaches. In early design stages where a real execution environment and complete executable software are unavailable, a simulation environment and a virtual platform are necessary. Virtual platforms provide flexibility to change the target architecture to be tested. Our proposed approach runs on the virtual platform we implement. This makes our approach faster and provides early results.

Keywords memory leak, soft real time system, virtual prototype, memory modelling

1. Introduction

With increasing design complexity, the early design tests of software become a crucial factor in electronic systems design. It helps to redesign or optimize the system at early design stages. This keeps flaws minimal in the production stage of embedded systems.

As embedded systems size gets smaller and smaller, memory becomes a crucial resource which needs to be managed efficiently and optimally. Inefficient memory management leads to severe memory problems which may result in system failures. Such flaws if detected early will lead to better designs with better performance.

Memory leak is the main memory related problem in soft real time systems and are considered as "hidden" problems that are hard to detect [23]. A memory leak is a memory location which is not freed after its use by the program, and hence unavailable for other components to re-use, as it is still reserved. A program causing leak allocate more and more memory over time leading the system to eventually run out of memory and fail. However, the code at the point of failure often has nothing to do with the leak and hence they are hard to detect. In soft real-time systems, where dynamic memory management is common, memory leaks have higher probability, especially for non-garbage collecting environments. Manual detection of leaks is tedious as they are hidden and hard to reproduce without any immediate symptoms. Hence an automated approach detecting leaks that is efficient and fast is needed.

Today, embedded systems software development is mainly conducted by virtual system prototypes, in combination with simulation-based approaches at different abstraction and refinement levels [20]. Memory leak, which is an important aspect to be tested at early design stages, could be effectively analyzed with virtual platforms. In this paper we introduce a novel approach to detect memory leaks with the help of virtual platforms. Existing approaches for memory leak detection are carried out at source code level or at actual run time [9], [12], whereas our approach is carried out at simulated abstract levels which makes early design tests feasible and are much faster than actual runs. We focus on memory leak detection in soft real-time systems with non-garbage collecting environments.

The remaining part of this paper is structured as follows. Section 2 describes related work in memory leak detection. Section 3 explains our approach for leak detection. Section 4 summarizes the paper and mentions the key challenges and future extensions of our work.

2. Related Work

In literature various methodologies are available to detect memory leaks, which could be classified into two main categories: static and dynamic.

The 4th Analytic Virtual Integration of Cyber-Physical Systems Workshop
December 3, 2013, Vancouver, Canada.

 Proceedings are available at:
http://dx.doi.org/10.3384/ecp13090
AVICPS website:
http://www.analyticintegration.org/

2.1 Static Methods

Static methods assume the availability of source code or any other static form of the target software. These methods involve leak detection without actual execution. They have the advantage that they do not necessitate the availability of the execution environment and are much faster.

Approaches using Shape Graph [7], pointer analysis [25], escape analysis [30], shape analysis [11], contradiction analysis [24], liveness analysis [27], ownership model [14,15], procedural summaries [6], bi-abductive inferences [18], and value flow [4,28] are some of the prominent ones. LCLInt is a static leak detection tool which annotates the source code with formal specifications [9].

Although these approaches have above mentioned advantages, there are certain disadvantages too. These methods detect leaks to a certain extent, and are not capable of detecting all kinds of leaks, especially the ones that arise during actual run time such as leaks due to dynamic references [17]. Static methods are useful only to a certain extent and do not provide enough accuracy or guarantee to prevent crashes arising from memory anomalies during execution.

2.2 Dynamic Methods

Most of the memory anomalies appear in dynamic scenarios. Dynamic methods involve the actual execution of the tasks, and hence are much more accurate than the static methods in detecting leaks. Reference counting, reachability analysis and liveness are some of the prominent methods used [1,2].

Purify [12], Cork [17], Leakbot [19], Sleigh [1] are the most prominent state-of-the-art dynamic approaches. Other approaches include Hound [22] and SWAT [13]. [31] introduces object ownership profiling at runtime to detect leaks. Leakpoint [5] is yet another dynamic approach to pinpoint the leak and its location. Leakpruning [2], Plug [21], and Leaksurvivor [29] are other approaches to minimize the damage caused by leaks at runtime. [26] is a leak detection approach for android systems via PCB hooking. Apart from these there are several leak detection tools available commercially and non-commercially such as mtrace and valgrind [8].

A major limitation of these approaches and tools are that they require the program tested to be actually executed, which mandates the execution environment and its dependencies. Leak detection at execution also implies more overhead and this affects the overall performance.

The approach we propose is a hybrid approach which combines the advantages of both static and dynamic methodologies.

3. Virtual Memory Modeling

3.1 Simulation Framework

Early design tests are important in scenarios where software is developed in modules independently and integrated later. Hence a final test is possible only after all the modules are ready. Moreover, the real target environment may not be available at hand or it may be too expensive to afford just for testing. Hence there should be some sort of early design environments available to test these modules independent of the availability of the whole project. The simulation environment gains importance here. Simulations have the advantage that they are fast, re-usable and could be customized for each individual case and at the same time not expensive as the real hardware or the platform. It provides almost the same accuracy in terms of performance and other tests and is flexible and portable.

In order to efficiently explore the design space, designers need models of the embedded software running in its execution environment, providing rapid and early feedback about effects of design decisions, such as the chosen scheduling strategy. Models at higher levels of abstraction with fast simulation speeds with enough accuracy are needed [20]. Moreover each design targets a different architecture, and hence simulation environment provides flexibility to adapt different architecture without great changes to the execution environment.

The ARTOS (Abstract Real-Time Operating System) is a simulation framework developed by C-LAB , in order to simulate and analyze the schedulabilty of real time tasks [32]. This framework currently estimates the performance in terms of time consumed for the tasks and thereby determines the failures or design alternatives, which could improve the time. In this framework there is already a platform with various APIs to abstract the software and to run it for determining the run time of the tasks [32]. Currently ARTOS does not consider the memory transactions of the simulated software. Therefore we propose to integrate our memory model into ARTOS, and thereby provide a common platform to carry out timing analysis and detect memory leaks simultaneously.

3.2 Memory Management Model

We follow an abstract implementation of memory management, where transactions involving memory requests and allocations are of major focus. The memory model is basically a transaction level model, where the emphasis is given to the transactions occurring between memory requesting object and the memory object allocated. We implement the model using TLM library in SystemC [3, 10]. Transaction-level modeling (TLM) is a high-level approach for modeling digital systems where details of communication among modules are separated from the details of the implementation of functional units or of the communication architecture. Communication mechanisms are modeled as channels and are presented to modules using SystemC interface classes. Transaction requests take place by calling interface functions of these channel models which encapsulate low-level details of the information exchange [3]. This makes it easier for the system-level designer to conduct experiments on the required abstraction level, without the need to consider other dependencies required for the actual execution and behavioral correctness. Our model in TLM will initially include the basic request-allocate process and later a paging/segmentation model. The available memory in our model is considered as fixed-sized blocks for the respective target architecture.

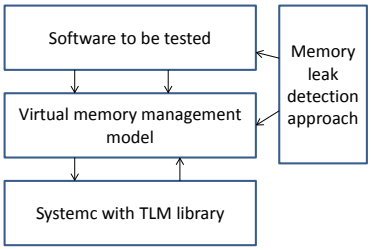

Figure 1: Virtual memory model for leak detection

A memory arbiter would be responsible for servicing the requests arriving from the tested software. The requests generated by the design under test is encapsulated and abstracted and passed to the memory arbiter. The arbiter checks for the available memory in the free memory area. The best-fit algorithm is used to allocate the most suitable sized memory block [23]. The acknowledgement and the allocated block with size and address are passed back to the requested entity. If there is no available free memory, the request is not served, and a request denial is sent back instead of acknowledgement by the arbiter. All these transactions are finally SystemC calls. An overview of our approach is shown in Figure 1.

3.3 Memory Leak Detection

Our approach of leak detection follows a hybrid model. The software under test undergoes static analysis before it is executed on the virtual model. The static analysis involves an automated control flow graph generation technique adopted from our previous work [16] as shown in Figure 2. The source code of the software to be tested is first disassembled. A graph based approach with automated labeling of basic blocks of the program is developed and the graph is then compacted for efficient detection of leaks.

Memory *request-allocate* paths are plotted on the graph with the help of disassembled code analysis. Each of such paths is checked for the corresponding *free* method for any *allocate* method. A one-to-one mapping is generated from the graph for *allocate* and *free* methods. If such a mapping is missing for any allocate function, a leak is notified and the corresponding location and object is marked.

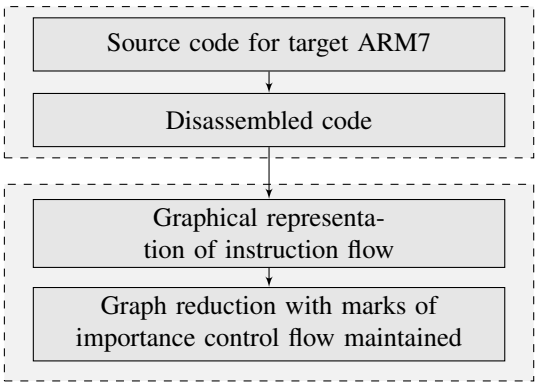

Figure 2: Control flow graph generation technique

Next step is the dynamic analysis, where the software to test is run at an abstract level on the virtual memory model. The memory request calls (e.g. malloc), are encapsulated, and such function calls are redirected to the respective calls to the arbiter in the memory model. The arbiter according to the availability of the memory responds with an acknowledgement or request denial message back to the requested object. A memory control block is generated for each block of memory in the model. The memory control block is responsible to keep track of used and free memory blocks. After each allocation process in the model, the status of simulated memory in the virtual model is tested with the help of all memory control blocks. In *liveness* analysis method [2] the object requesting memory is checked if it still exists, otherwise the memory allocated is reclaimed. Similarly the methods of *reference counting* [1] is applied, where in the pointer references are tracked down to check if any unused memory is producing a leak. The *reachability* [2] approach is applied to check if any of the still live memory requesting and allocated objects are reachable via any pointer references at the end of the execution; and if unreachable it is considered a leak. At the end of each execution, the memory allocation pattern in the model is verified and the probability of various leaks reported through the above three methods are analyzed with various number of executions. The leak and the probability of leak occurrence are generated for further optimizations in the software.

Currently we consider C and C++ programs as our test cases along with gcc compiler. The target architecture we consider is ARM9, with virtual ARM integrator CP development board.

4. Conclusion and Future work

Memory leaks are serious problems in resource constrained embedded environments and need to be fixed at early stages of design, especially in non-garbage collecting environments. A fast and efficient leak detection system at early stages of development is required. We proposed a novel hybrid memory leak detection method in a simulation environment that can support different target architectures. The memory model is implemented at an abstract level using Transaction Level Modeling. The software under test is run at an abstract level on top of this model. Our proposed approach is aimed to be faster, since the analysis is done on top of a virtual platform with simulation. The main challenges include the optimization of simulation time, and we aim to make the simulation as fast as possible, along with precision. Our approach is intended for non-garbage collecting environments as the probability of leaks are higher in such environments.

The future work on this methodology includes the extension of the memory model to detect other memory problems such as fragmentation and corruptions. Moreover we need to have support for other languages and compilers, to make our model more generic. Another extension in this direction would be to incorporate this model to the ARTOS framework, so that users can have both memory and process related checks under one common framework.

References

[1] Michael D. Bond and Kathryn S. McKinley. Bell: bit-encoding online memory leak detection. *SIGPLAN Not.*, 41(11):61–72, October 2006.

[2] Michael D. Bond and Kathryn S. McKinley. Leak pruning. *SIGARCH Computer Architecture News*, 37(1):277–288, March 2009.

[3] L. Cai and D. Gajski. Transaction level modeling: an overview. In *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pages 19–24, 2003.

[4] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. *SIGPLAN Not.*, 42(6):480–491, June 2007.

[5] James Clause and Alessandro Orso. Leakpoint: pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 515–524, New York, NY, USA, 2010. ACM.

[6] Dino Distefano and Ivana FilipoviÄĞ. Memory leaks detection in java by bi-abductive inference. In DavidS. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 278–292. Springer Berlin Heidelberg, 2010.

[7] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Checking cleanness in linked lists. In Jens Palsberg, editor, *Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 115–134. Springer Berlin Heidelberg, 2000.

[8] Cal Erickson. Memory leak detection in embedded systems. *Linux Journal*, August 2002.

[9] David Evans, John Guttag, James Horning, and Yang Meng Tan. Lclint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes*, 19(5):87–96, December 1994.

[10] Thorsten Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[11] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. *SIGPLAN Not.*, 40(1):310–323, January 2005.

[12] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[13] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. *SIGPLAN Not.*, 39(11):156–164, October 2004.

[14] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. *SIGPLAN Not.*, 38(5):168–181, May 2003.

[15] David L. Heine and Monica S. Lam. Static detection of leaks in polymorphic containers. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 252–261, New York, NY, USA, 2006. ACM.

[16] M.M. Joy, M. Becker, W. Mueller, and E. Mathews. Automated source code annotation for timing analysis of embedded software. In *Advanced Computing and Communications (ADCOM), 2012 18th Annual International Conference on*, pages 12–18, 2012.

[17] Maria Jump and Kathryn S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. *SIGPLAN Not.*, 42(1):31–38, January 2007.

[18] Yungbum Jung and Kwangkeun Yi. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 131–140, New York, NY, USA, 2008. ACM.

[19] Nick Mitchell and Gary Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In Luca Cardelli, editor, *ECOOP 2003 âĂŞ Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 351–377. Springer Berlin Heidelberg, 2003.

[20] W. Mueller, M. Becker, A. Elfeky, and A. DiPasquale. Virtual prototyping of cyber-physical systems. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 219–226, 2012.

[21] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Plug: Automatically tolerating memory leaks in c and c++ applications. Technical report, Technical Report UM-CS-2008-009, University of Massachusetts, 2008.

[22] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. *SIGPLAN Not.*, 44(6):397–407, June 2009.

[23] Gary J. Nutt. *Operating systems - a modern perspective (3. ed.)*. Addison-Wesley-Longman, 2004.

[24] Maksim Orlovich and Radu Rugina. Memory leak analysis by contradiction. In *Proceedings of the 13th international conference on Static Analysis*, SAS'06, pages 405–424, Berlin, Heidelberg, 2006. Springer-Verlag.

[25] Berhard Scholz, Johann Blieberger, and Thomas Fahringer. Symbolic pointer analysis for detecting memory leaks. *SIGPLAN Not.*, 34(11):104–113, November 1999.

[26] Jooyoung Seo, Byoungju Choi, and Suengwan Yang. A profiling method by pcb hooking and its application for memory fault detection in embedded system operational test. *Inf. Softw. Technol.*, 53(1):106–119, January 2011.

[27] Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. Automatic removal of array memory leaks in java. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 50–66, London, UK, UK, 2000. Springer-Verlag.

[28] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 254–264, New York, NY, USA, 2012. ACM.

[29] Yan Tang, Yan Tang, Qi Gao, Qi Gao, Feng Qin, and Feng Qin. Leaksurvivor: towards safely tolerating memory leaks for garbage-collected languages. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 307–320, Berkeley, CA, USA, 2008. USENIX Association.

[30] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *In Proceedings of ESEC/FSE 2005*, pages 115–125. ACM Press, 2005.

[31] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. Leakchaser: helping programmers narrow down causes of memory leaks. *SIGPLAN Not.*, 46(6):270–282, June 2011.

[32] Henning Zabel, Wolfgang Müller, and A. Gerstlauer. Accurate rtos modelling and analysis with systemc, January 2009.