Dynamic Source Code Analysis for Memory Hierarchy Optimization in Multimedia Applications

Christakis Lezos*, Grigoris Dimitroulakos*, Angeliki Freskou†, Konstantinos Masselos*
*University of Peloponnese, Department of Informatics and Telecommunications
Terma Karaiskaki, 22100 Tripoli, Greece
{*lezos, dhmhgre, kmas*}*@uop.gr*
†Ajax Compilers, Athens, Greece
info@ajaxcompilers.com

***Abstract*—Realizing image and signal processing algorithms in embedded systems is a three step process including algorithmic design, implementation and mapping to a target architecture and memory hierarchy. This paper presents MemAddIn, a dynamic analysis tool for C applications that exposes the critical application's loops which deserve the designer's attention for memory hierarchy optimization. MemAddIn is based on an extension of MEMSCOPT compiler and integrates in the Visual Studio IDE offering a unified environment for the application's implementation and optimization. To conclude on the criticality of the application loops the tool utilizes two metrics which are relevant with the underlying memory architecture cost and performance.**

***Index Terms*—Dynamic source code analysis, memory hierarchy optimization**

I. Introduction

Nowadays the vast amount of digital electronic equipment is based on embedded systems [1]. Their popularity lies in their ability to satisfy many different types of constraints including timing, size, weight, power consumption, reliability and cost. For this reason, the most critical parts of the applications are realized in embedded architectures which exhibit superior performance over general purpose processors. Hardware-Software co-design methodologies [2] facilitate the mapping of applications to this type of systems. For many embedded applications, especially the ones referring to portable multimedia devices, an integral part of these methodologies is the optimization of the embedded systems memory hierarchy. To optimize the application in terms of memory, its high level specification should be inspected to focus on the parts with a higher benefit for optimization. The high level specification is more often expressed in C language while the most critical parts of code are the iterative control flow constructs (for, while etc.) that manipulate large multidimensional array data structures. The identification of the most critical parts of code is a challenge due to the following reasons: 1) embedded applications consists of thousands of lines of code, thus it is difficult to pinpoint the critical parts and 2) usually the designer of the application code and the designer implementing the optimization are different persons.

This paper presents MemAddIn[1], a new tool offering dynamic source code analysis on C language source applications with the purpose of identifying the critical loops in respect to the optimization of memory hierarchy. Such loops are the ones with high iteration count including operations on multidimensional array data structures. Since memory operation can be drastically improved by exploiting the data reuse, MemAddIn quantifies the data reuse of each multidimensional array. In summary, the profiling results reveal the critical loops and rank the multidimensional arrays according to their data reuse in the application context.

The profiling objective is satisfied by the dynamic estimation of two code metrics: 1) the first one called Loop Weight Metric (LWM) weights a loop respecting the number, size and accesses of the arrays it operates on and 2) the other called Array Reuse Factor (ARF) quantifies the data reuse in each array by monitoring the accesses in the whole set of the array elements.

The remainder of this paper is organized as follows: We begin with an overview of the core components of MemAddIn in section II and continue with a description of the LWM and ARF metrics in section III. Section IV introduces the demonstration example and finally, section V summarizes the work and future actions.

II. System Description

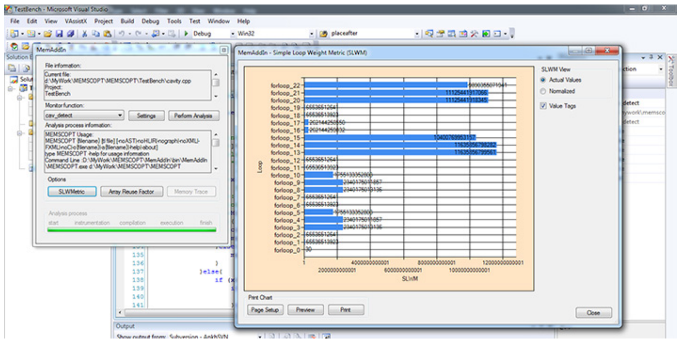
As with the majority of dynamic analysis tools, MemAddIn uses instrumentation to extract the desired information from the code [3]. Instrumentation can be performed at various stages ranging from modifying the source code to rewriting the executable file or even at runtime [4]. In the MemAddIn case the extra code is inserted into the source code of the application.

The tool heavily relies on MEMSCOPT source-to-source compiler [5] for instrumenting the source code with appropriate counters that extract the necessary information. We implemented the tool as a Visual Studio extension in order to provide a seamless and user friendly environment. Thus, making it useful for actual application by developers.

MemAddIn's core component is a Visual Studio addin dll file that arranges other elements, like the MEMSCOPT compiler, in a sequence of phases. It is developed in C# and it makes extensive use of the EnvDTE namespace. EnvDTE is an assembly-wrapped COM library containing the objects

[1]http://www.memaddin.com

and members for Visual Studio core automation. It is the base infrastructure where all Visual Studio addins are developed. Figure 1 shows the interface of the MemAddIn extension.



File information:
Current file:
d:\MyWork\MEMSCOPT\MEMSCOPT\TestBench\cavity.cpp
Project:
TestBench
Monitor function:
cav_detect
Analysis process information:
MEMSCOPT Usage:
MEMSCOPT [filename] [ti file] [-noAST-noHLIR-nograph-noXML-FXML-noC-o [filename] -a [filename] -help -about]
type MEMSCOPT -help for usage information
Command Line :D:\MyWork\MEMSCOPT\MemAddIn\bin\MemAddIn \MEMSCOPT.exe d:\MyWork\MEMSCOPT\MEMSCOPT

Fig. 1. MemAddIn User Interface - LWM window

III. MemAddIn Profiling Metrics

In the following we present in detail the two metrics utilized by MemAddIn:

A. Loop Weight Metric (LWM)

LWM characterizes the criticality of a loop respecting the number, size and accesses of the arrays it operates on. Loops with high LWM are considered heavy and time consuming and could be candidates for loop optimizations. The LWM calculation method for a single loop is described in definition1.

Definition 1.

$$LWM(X) = \sum_{i=1}^{N} (VA_i \times VS_i) \tag{1}$$

where *X* is the name of the loop, *N* is the total number of variables accessed within loop *X*, *VA* is the number of accesses a variable has within that loop and *VS* is the static size of that variable. Scalar variables have *VS* value 1 and are handled the same way as a 1x1 array.

B. Array Reuse Factor (ARF)

It is a fact that smaller memories have lower cost in area, delay and energy [6]. Likewise, an algorithms implementation in an embedded system can significantly benefit from the exploitation of cache or scratch-pad memories in the presence of data reuse. To quantify the data reuse of an algorithm on an array, the Array Reuse Factor (ARF) metric is devised. The ARF measures the mean value of consecutive read accesses that are followed by a write access for every array element. Array elements with high ARF values can benefit from data reuse optimizations [6] that aim to map frequently reused values in smaller memories. The Reuse Factor calculation method for a single array is described in definition 2.

Definition 2.

$$For : A_i \rightarrow 0 < i < n \tag{2}$$

$$ARF_i = \frac{sumr_i}{sumw_i} \tag{3}$$

$$ARF = ARF_0, \ldots, ARF_n \tag{4}$$

where *A* is the name of the examined array, *i* is the index of the array, *sumr* is the number of read accesses to a specific array index and *sumw* is the number of continuous write accesses to a specific array index. ARF focuses on consecutive read operations since consecutive writes in the same memory location has no particular usage in any algorithm implementation [6].

IV. Demonstration

The usage of MemAddIn is demonstrated on a realistic loop dominated image processing algorithm called cavity detector [7]. In this demonstration the LWM and ARF values are calculated for the C code of a specific function within the cavity detector application. The primary objective of this demo is to display the capabilities of the tool. Optimization of the code depends on potential actions taken afterwards by the user.

The cavity detector algorithm is managed within a Visual Studio C/C++ project and the demonstration steps are as follows: 1) the project is opened and MemAddIn is launched, 2) the user indicates the function to be examined from a list of all the available functions in the currently edited C file, 3) a number of required path related settings are set, 4) dynamic analysis is initiated and upon its completion 5) the visualized results are available to the user.

V. Conclusion and Future Work

In this paper, we present a new tool offering dynamic source code analysis on C language source applications with the purpose of identifying the critical loops in respect to the optimization of memory hierarchy. At present, two metrics are defined and utilized for this task. Ongoing work considers the improvement of the current and the development of additional more mature metrics.

Acknowledgment

This work has been partially supported by the FP7 ALMA (FP7 ICT-2011. 287733) project, funded by the European Community.

References

[1] D. Blaza and A. Wolfe. 2013 embedded market study. UBM Tech Electronics. [Online]. Available: http://e.ubmelectronics.com/2013EmbeddedStudy/index.html

[2] G. De Micheli, R. Ernst, and W. Wolf, Eds., *Readings in hardware/software co-design*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[3] D. Binkley, "Source code analysis: A road map," in *Proceedings of Future of Software Engineering 2007*, ser. FOSE '07, Minneapolis, MN, USA, May 2007, pp. 104–119.

[4] S. Shende, "Profiling and tracing in linux," in *Proceedings of the Extreme Linux Workshop 2*, Monterey, CA, USA, June 1999.

[5] G. Dimitroulakos, C. Lezos, and K. Masselos, "Memscopt: A source-to-source compiler for dynamic code analysis and loop transformations," in *Proceedings of the 2012 Conference on Design & Architectures for Signal & Image Processing (DASIP)*, Karlsruhe, Germany, October 2012.

[6] J. Diguet, W. Catthoor, and H. De Man, "Formalized methodology for data reuse exploration in hierarchical memory mappings," in *1997 International Symposium on Low Power Electronics and Design, Proceedings*, Monterey, CA, USA, August 1997, pp. 30–35.

[7] M. Bister, Y. Taeymans, and J. Cornelis, "Automated segmentation of cardiac mr images," in *Computers in Cardiology 1989, Proceedings.*, Jerusalem, September 1989, pp. 215 –218.