Methodology for Memory Analysis and Optimization in Embedded Systems

Shenglin Yang
UCLA Dept of EE
Los Angeles, CA 90095
+1-310-267-4940
shengliny@ee.ucla.edu

Ingrid M. Verbauwhede
UCLA Dept of EE
Los Angeles, CA 90095
+1-310-794-5209
ingrid@ee.ucla.edu

Abstract

This paper describes a methodology for memory analysis and optimization of embedded system design with the goal of reducing memory usage. It acts as a guide to optimize the memory module of the embedded systems in an efficiently way, by which the design-time optimization can be achieve. Two typical embedded signal-processing applications are implemented using the proposed method, gaining 67% and 31% reduction of system memory requirement, respectively.

1. Introduction

Memory behavior including both the size and the number of access is playing a more and more important role in the embedded system design. As new processors continuously improve the performance of embedded systems, the processor-memory gap widens and memory represents a major bottleneck in terms of speed, area and power for many applications [1].

When designing an embedded system, memory analysis at the system-level is critical since the decisions at this level have the largest impact on the final result. In this paper we propose a method of memory estimation for embedded applications based on a C/C++ design environment. Using the proposed methodology, the total required memory size, as well as the memory usage change along with the execution time is estimated. Based on the estimation results, algorithm level optimization can be performed with the target of reducing the memory requirements for the embedded system. We use the LEON-2 Sparc processor embedded platforms for the demonstrated applications, and all the simulations are performed with the TSIM SPARC simulator [11].

The paper is organized as follows. Section 2 briefly reviews some previous work in the system-level memory estimation. Section 3 describes our methodology for memory analysis. In sections 4 and section 5, the proposed methodology is implemented in two embedded signal processing applications. The estimation results and optimization strategies are also discussed.

2. Related Work

Memory estimation techniques at the system-level are used to guide the embedded system designer in choosing the best solution. In data dominated applications, such as image or speech signal processing applications, summing up the sizes of all the arrays is the most straightforward way to get an upper bound of the memory requirement. However "in-place" problem [2] introduces a huge overestimate. In [3], the internal in-place mapping is taken into consideration and the total storage requirement is the sum of the requirements for each array. In [4], the data dependency relations in the code are used to find the number of array elements produced or consumed by each assignment, from which a memory trace of upper and lower bounding rectangle as a function of time is found. In [5], a methodology based on live variable analysis and integer point counting is described. However, this method is not feasible for large multi-dimensional loop nest because of the heavy computation. Unlike the above techniques, [6], instead of assuming an execution ordering, starts with an extended data dependency analysis resulting in a number of non-overlapping basic sets of array elements and the dependencies between them. The methodology described in [7] takes into account partially fixed execution ordering, achieved by an array data flow analysis preprocessing. The method introduced in this paper takes both the program size and the data size into consideration and provides an efficient way to reduce the memory requirements for embedded systems at the system level using the information gathered from run-time simulation.

3. Methodology

3.1 Basic idea

Before describing the methodology, we first give the basic idea of memory usage for any program. When a program is running, the memory module is divided into two parts: program segment and data segment, which includes heap and stack (Fig. 1).

Once the program is loaded into the processor, the size of the memory for program segment is fixed. The rest of the memory is used as heap and stack during the program running time. Heap starts from the bottom of the program segment. Whenever there is dynamic memory allocation, a block of memory is reserved for later use. When a memory free happens, the specific memory block is returned to the memory pool. On the other hand, the stack pointer position changes when a function call is made or finished. Generally, the stack and the heap grow and shrink in opposite direction.

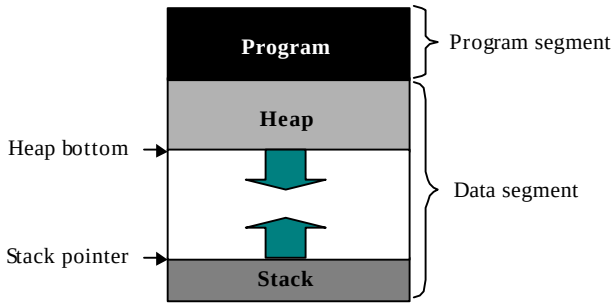

Fig. 1. Memory Partitioning

A collision of stack and heap implies that a fatal error state has been reached. At any particular moment, the memory usage of the system is determined by the sum of the size of program, heap and stack as shown in Equation 1:

$$M_{Total} = M_{program} + M_{Heap} + M_{stack} \quad (1)$$

In another word, for any program, the memory usage changes as the result of the changing of the bottom of the heap as well as the position of the stack pointer. In our proposed method, we take use of this characteristic, getting the position of the heap bottom and the stack pointer dynamically during the program running time. Taking program size into consideration, a dynamic memory usage trace map is generated. From this trace map, we can get information about the overall memory requirement as well as the memory bottleneck of the application.

3.2 Implementation of the method

To get all the reliable and detail information of memory usage during the program running time, we use the change-driven method to implement our methodology. All places in which memory usage changing could happen are traced. For programs written in C/C++ design environment, there are several types of trace points:

1. Where dynamic memory allocation functions are called (*malloc*(), *alloc*(), *realloc*(), etc.), a block of memory is allocated according to the current memory usage situation. Assume that the current heap bottom address is H_{bottom}. When a memory allocation function is called,
   a. If the required memory is small enough to fit into some hole in the current heap, H_{bottom} does not change.
   b. Otherwise, the system has to allocate a new block for it, which means H_{bottom} will increase and the heap size will in turn be enlarged.
   c. Also the *free*() function might effect the heap bottom address. If the memory block closest to the bottom of the heap is freed, the heap size will decrease.
   d. If the freed memory block is in the middle of the heap, then H_{bottom} does not change.
2. Another type of trace point is the entering point of any function call, where the stack pointer changes according to the local variable definition size.

In the following sections, two typical embedded signal-processing applications are implemented using this methodology and the results of memory analysis are discussed. In addition, optimizations are performed based on the memory trace map.

4. Embedded ThumbPod System

4.1 Overview

ThumbPod [8][9] is a secure embedded fingerprint verification system built on the LEON-2 processor. The major computational bottleneck is the embedded fingerprints matching algorithm. For matching two fingerprints, we adopt the minutiae-based matching algorithm. Therefore, a minutiae detection procedure needs to be implemented on the embedded device. Like many other typical embedded multimedia signal processing algorithm, the minutiae detection procedure is array dominated. Hence, memory management for it is very important.

The baseline program we used to extract the minutiae set is taken from NIST Fingerprint Image Software [10]. In the following subsections, the memory usage analysis is carried out using this NIST software as the starting point. In addition, memory optimizations oriented by the analysis results are described.

4.2 Baseline Result

Implementing the methodology described in section 3 to the baseline program taken from NIST Fingerprint Image Software. Results are the following (Fig. 2). During the simulation, 4096K total RAM is chosen. Fig. 2(a) presents the change of the heap bottom address during the running time. Fig. 2(b) is the change of the position of the stack pointer. Putting these results into equation (1), the total memory usage trace map is shown in Fig. 2(c). Fig. 2(d) describes the maximum memory distribution. According to Fig. 2(d), we know that the peak memory usage of the system is 1,572Kbytes, including 325Kbytes program segment memory and 1,247Kbytes data segment memory.

4.2 Memory optimization

For most embedded systems, memory size beyond 1M is too expensive. In order to reduce the memory requirement for this application, we try to shrink the program size as well as the running time memory usage based on the knowledge got from the memory trace map step by step.

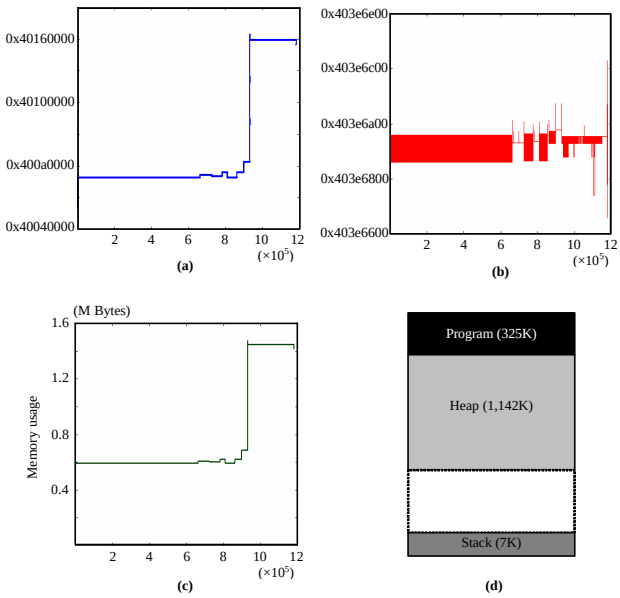

Fig. 2. Memory analysis results for baseline program

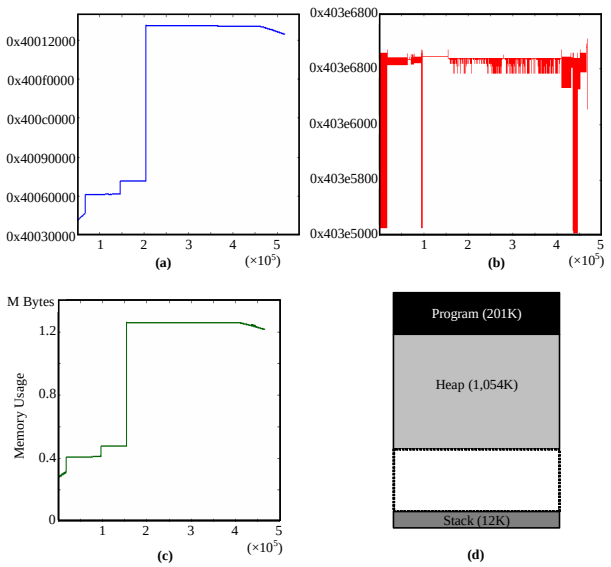

Fig. 3. Memory analysis results for fix-refined program

Step 1: architecture optimization

Since the NIST starting point program is floating-point based, while the LEON-2 processor only supports fixed-point computation, fixed-pointed refinement is necessary for speeding up the program. In this paper, we study the influence in terms of memory of doing fixed-point refinement. Fig. 3 shows the memory analysis results for fixed-point optimized program. From the memory trace map for the program after fixed-point refinement, we notice that both the program segment size and data segment size decrease. This is because that, in one hand, fixed-point refinement remove many floating point calculation related libraries, and on the other hand, the size of the elements of some arrays are modified from 8bytes "double" type to 4bytes "int" type, which reduces the storage memory by half. In total, the memory requirement for fixed-refined program is 1267Kbytes

Step 2: "in-place" optimization

The memory trace map from the previous step shows that there is a major jump, which introduces most of the memory usage in a very short period. Our idea for reducing the data segment memory is first finding out where the jump(s) happen(s), then studying the algorithm to figure out the reason for the big memory use. Finally, we implement memory management techniques to remove or lower the jump(s).

Detailed study of the minutiae detection algorithm of the ThumbPod embedded system shows that the biggest jump happens when a routine named "*pixelize_map*" is called. The diagram of this routine is shown in Fig. 4.

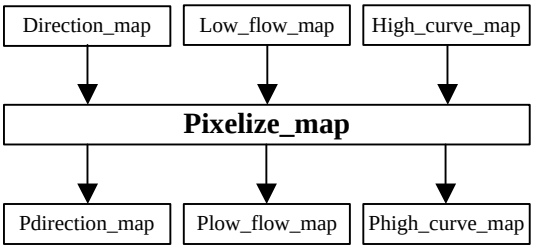

Fig. 4. Diagram for pixelize_map ()

The functionality of this routine is to convert the block-based maps for direction, low flow flag, and high curve flag into pixel-based ones. For each pixelized map, 262,144 (256×256×4) bytes of memory are required since for each pixel, one 32-bits integer is needed to present the value of the map. This is the bottleneck of the memory usage.

There are two ways to reduce the memory requirement. One is to implement "in-place" technique in this routine. The numbers in the *direction_map* are within the range 0–16, and for the *low_flow_map* and *high_curve_map*, simple 0/1 value is used as flags. In addition, the dimensions for the three maps are exactly the same. Therefore taking one corresponding element from each map, the sum of the valid bits number is 6 (four bits for *direction_map*, one for *low_flow_map*, and one for *high_curve_map*), which is smaller than 32. Therefore it is possible to compress these three different maps into one since we can combine three elements (one from each maps) in one 32-bit integer. By doing this, instead of three separate arrays, only one array called "*Ptotal_map*" is used to represent the three pixelized maps (Fig.5).

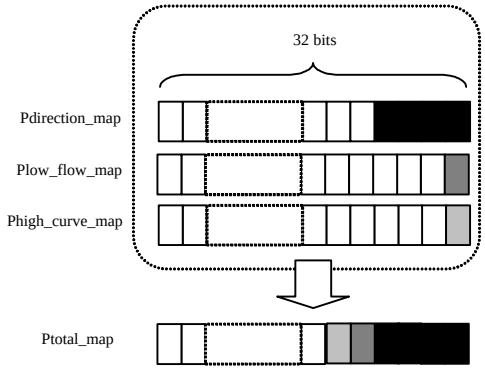

Fig. 5. "in-place" technique for memory reduction

Fig. 6 is the results of this in-place memory management. According to the above results, peak memory requirement becomes 744Kbytes. The data segment memory decreases by 590Kbytes compared to the previous result, while the program segment size increases by 47Kbytes. The reason for program size increasing is that additional calculation is needed for the compression and decompression of the pixelized maps. Another observation needed to mention is that as a side effect of those additional calculations, the execution time of the whole system increases by 0.25sec.

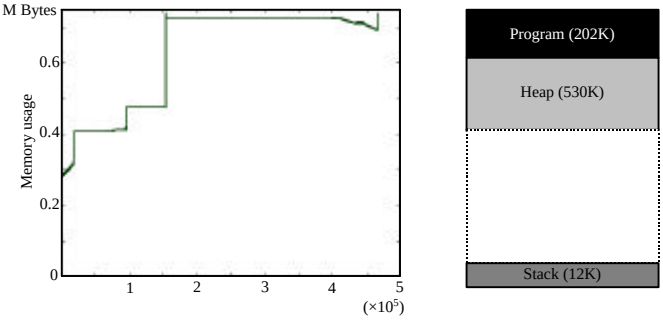

Fig. 6. Memory analysis results of "in-place" technique

Step 3: Skipping bottleneck

From the memory trace map shown in Fig. 6, the memory requirement bottleneck is still the *pixelize_map* routine. Further optimization can be implemented by eliminating this routine. Instead of generating whole pixlized maps, we calculate the map value for each pixel only when it is referred in the program. This technique removes the big memory usage jump in the memory trace map. The drawback of it is that the pixel index needs to be calculated each time it is referred. The result of this method is shown in Fig. 7.

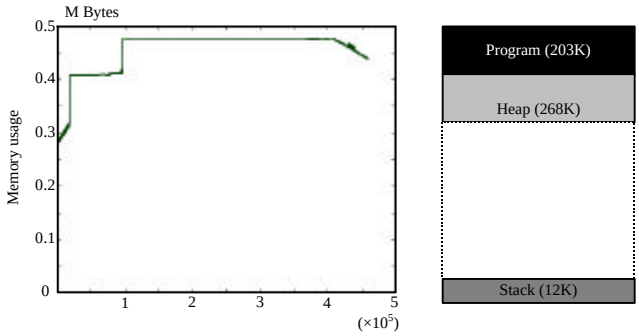

Fig. 7. Memory analysis optimized results

Comparison of the last two results (Fig. 6 and Fig. 7) shows that both the program segment size and the data segment size decrease. The total memory requirement is 483K Bytes. Keeping an eye on the execution time, system speeds up to 5.05sec. This means that the memory optimization techniques gain some memory size with no cost of speed. This is because that the *pixelize_map* routine is skipped in the program, some unnecessary computation is avoided.

5. Embedded Speech Recognition Front-End

5.1 Overview

Speech recognition is an increasingly popular embedded real-time multimedia application. In the speech recognition module of the "Poly-sensing Environment" project [13], for reducing the communication overhead and the energy of the system, the signal processing front-end feature extraction procedure needs to be done in the embedded devices, which are usually small in size and batteries-powered. As a result, memory analysis of such a system is very valuable for system design.

The acoustic feature used in our system is called Mel-Frequency Ceptrum Coefficient (MFCC) [14]. The diagram of this algorithm is shown in Fig. 8.

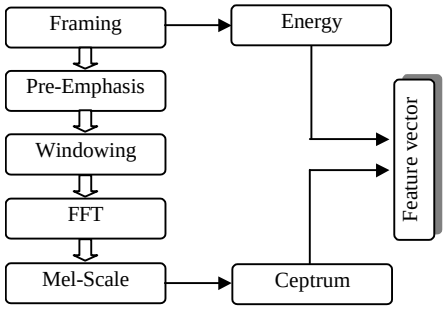

Fig. 8. MFCC Front-End Processing

The floating-point baseline program is taken from ETSI [12]. Using the proposed memory estimation methodology,

the memory analysis results are described in the following subsections.

5.2 Baseline result

Fig. 9 shows the memory trace map for the baseline implementation. The program segment size of this baseline implementation is 69Kbytes and the data segment memory required is 14Kbytes. In total, 83Kbytes memory is needed.

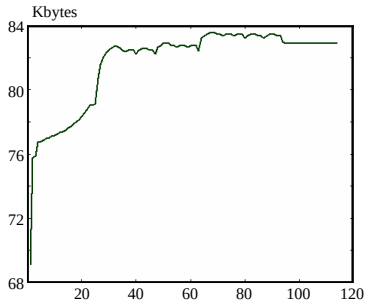

Fig. 9. Memory analysis result for baseline program

5.3 Memory optimization

Since the front-end processing runs on fixed-point embedded devices, the first step we need to do is the architecture optimization, This step of refinement results in 19Kbytes memory save in program segment size and 5Kbytes save for data segment size. Totally, 59Kbytes memory is required (Fig. 10).

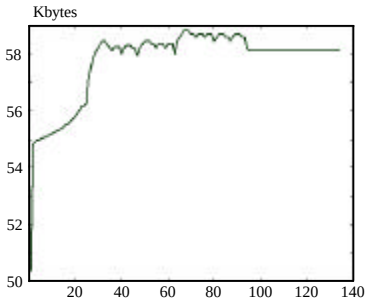

Fig. 10. Memory analysis result for fix-refined program

Following a similar methodology for memory optimization, analysis of the memory trace map shows that the largest jump in the map is caused by memory allocation for array "*vector*", which is used to store the Mel-scale triangle filter bank. According to the algorithm, each filter in the filter bank is used independently when calculating the ceptrums. Therefore storing the whole filter bank at the same time is not memory-efficient. Instead, the filter coefficient could be computed before being used and freed after that. Another jump in the memory trace map comes from the array "*rsrec*", which is used in the recursive part of FFT calculation. The values of it in one recurrence are not used anymore in the following. Therefore, the memory block of this array could be reused in each recurrence. After implementing these optimizations techniques to the front-end processing, the memory trace map is shown in Fig. 11.

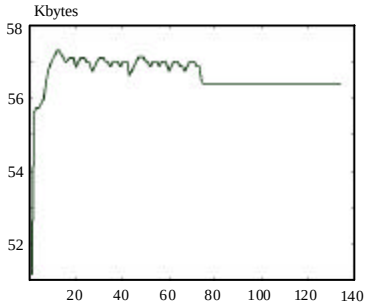

Fig. 11. Memory analysis result for optimized program

Result shows that for this step 57Kbytes memory is needed. Comparing with previous step, program segment size slightly increases by 2%, while the data segment memory decreases by 30%.

6. Conclusion

In this paper a memory analysis method to estimate, analysis, and optimize the memory usage for embedded system is presented. It guides designers to get bigger benefit first by optimize the larger jump in the memory trace map. Two typical embedded applications, ThumbPod embedded secure fingerprint verification system and embedded speech recognition front-end system, are implemented using the proposed methodology. Fig. 12 shows the overall optimization results for both of them. Since program segment memory and data segment memory could be ported into different memory locations, memory analysis for program segment and data segment is shown separately here. Fig. 12(a), (b) show 67% and 31% total memory reduction for ThumbPod secure embedded fingerprint verification system and embedded speech recognition front-end module, respectively.

Acknowledgements

The authors would like to acknowledge the funding of NSF account no CCR-0098361 and the Langlois foundation.

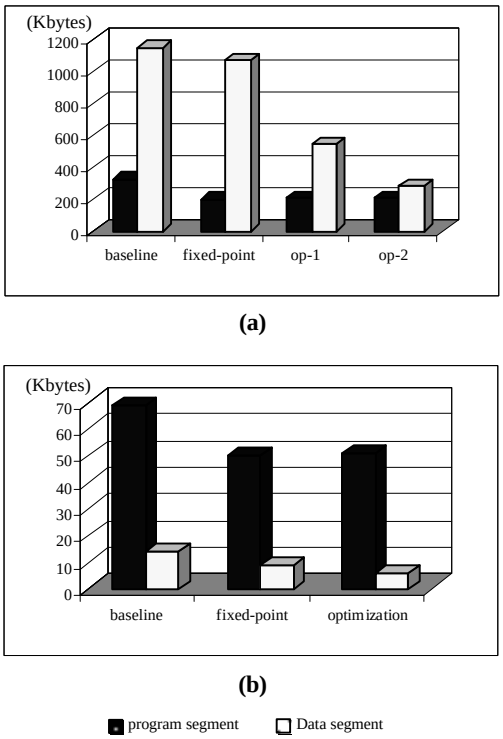

Fig. 12. Memory reduction for two typical applications

Reference:

[1] P. Panda, F. Catthoor, N. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, P.G. Kjeldsberg, "Data and memory optimization techniques for embedded systems", *ACM Transactions on Design Automation of Electronic systems*, vol.6, no.2, pp.149-206., April 2001.

[2] I. Verbauwhede, F. Catthoor, J. Vandewalle, H. De Man, "Background memory management for the synthesis of algebraic algorithms on multi-processor DSP chips", *Proc. VLSI'89, Int. Conf. on VLSI*, Munich, Germany, pp.209-218, Augest 1989.

[3] I.Verbauwhede, C. Scheers, J. Rabaey, "Memory estimation for high-level synthesis", *proceedings of 31st ACM/IEEE Design Automation Conference*, San Diego, CA, pp.143-148, June 1994.

[4] P. Grun, F. Balasa, N. Dutt, "Memory size estimation for multimedia applications", *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, Seattle WA, pp.145-149, March 1998.

[5] Y. Zhao, S. Malik, "Exact memory size estimation for array computations without loop unrolling", *Proceedings of 36th ACM/IEEE Design Automation Conference*, New Orleans LA, pp811-816, June 1999.

[6] F. Balasa, F. Catthoor, H. Deman, "Background memory area estimation for multidimensional signal processing systems", *IEEE Transactions on Very Large Scale Integration systems*, vol. 3, no. 2, pp.157-172, June 1995.

[7] P. G. Kjeldsberg, F. Catthoor, E. J. Aas, "Storage requirement estimation for data intensive applications with partially fixed execution ordering", *Proceedings of 8th International Workshop on Hardware/Software Codesign*, San Diego, pp56-60, May 3-5, 2000.

[8] D. Hwang, P. Schaumont, Y. Fan, A. Hodjat, B. Lai, K. Sakiyama, S. Yang, I. Verbauwhede, "Design flow for HW/SW acceleration transparency in the ThumbPod secure embedded system", *Proceedings of 40th ACM/IEEE Design Automation Conference*, Anaheim, CA, pp60-65, June 2003.

[9] www.thumbpod.com

[10] User's Guide to NIST Fingerprint Image Software (NFIS). NISTIR 6813, National Institute of Standards and Technology.

[11] www.gaisler.com

[12] "Speech processing, transmission and quality aspects; distributed speech recognition; front-end feature extraction algorithm; compression algorithms", ETSI Standard: ETSI ES 201 108 v1.1.2.

[13] http://users.design.ucla.edu/~fwinkler/PSE/descr.html

[14] J.R. Deller, J.H.L. Hansen, J.G. Proakis, *Discret-Time Processing of Speech Signal*, Prentice Hall, Upper Saddle River, NJ, 1987.