Reuse Distance Analysis for Locality Optimization in Loop-Dominated Applications

Christakis Lezos, Grigoris Dimitroulakos, Konstantinos Masselos
University of Peloponnese, Department of Informatics and Telecommunications
Terma Karaiskaki, 22100 Tripoli, Greece
{*lezos, dhmhgre, kmas*}*@uop.gr*

***Abstract*—This paper discusses MemAddIn, a compiler assisted dynamic code analysis tool that analyzes C code and exposes critical parts for memory related optimizations on embedded systems that can heavily affect systems performance, power and cost. The tool includes enhanced features for data reuse distance analysis and source code transformation recommendations for temporal locality optimization. Several of data reuse distance measurement algorithms have been implemented leading to different trade-offs between accuracy and profiling execution time. The proposed tool can be easily and seamlessly integrated into different software development environments offering a unified environment for application development and optimization. The novelties of our work over a similar optimization tool are also discussed. MemAddIn has been applied for the dynamic computation of data reuse distance for a number of different applications. Experimental results prove the effectiveness of the tool through the analysis and optimization of a realistic image processing application.**

***Index Terms*—Data reuse distance analysis, locality optimization, 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 facilitate the mapping of applications to this type of systems. For many embedded applications, especially those referring to portable multimedia devices, an important part of these methodologies is the memory hierarchy optimization. One approach to optimize an application in terms of memory usage is the application of locality optimizing transformations at a high level. A sequential execution of a program can be viewed as a stream of memory data accesses, where the notion of time is defined by the number of accesses rather than clock cycles. The number of distinct data elements accessed between two consecutive uses of the same element is called the *reuse distance*. This notion is equivalent to temporal data reuse but in a machine independent manner. Data *reuse distance analysis (RDA)* [2], [3] is a valuable process that provides quantitative measures of program locality that can be then used to drive the locality optimization process.

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

MemAddIn[1] [4] is a compiler assisted dynamic code analysis tool that supports the optimization of application code in C targeting embedded system implementation. In most cases the most critical parts of the code are the iterative control flow constructs (for, while etc.) that manipulate large multidimensional array data structures.

This paper discusses the implementation of data reuse distance analysis in MemAddIn tool. The tool's capabilities are used to indicate and realize source code optimizations that may improve temporal locality and reduce cache misses on the targeted architecture cache(s). The main contributions of the proposed work are:

- Efficient algorithms for data reuse distance computation. Measuring the reuse distance between two memory accesses is not a straightforward task and algorithms are needed to perform this task efficiently. A consistent realization of data reuse distance computation algorithms developed over the years is provided in MemAddIn. The profiling execution time for different algorithms has been evaluated and the results prove that our algorithms' implementations perform proportionally to their theoretical time complexity.
- The integration of the proposed tool into a popular development environment. MemAddIn is released as an extension for the Visual Studio IDE offering a unified environment for the application's design and optimization. The integration of MemAddIn tool in Linux based environments is planned as well.
- The optimization of a realistic loop-dominated edge detection image processing application [5], using MemAddIn, proving the effectiveness of the proposed tool.

II. Background and Basic Concepts

For a coherent presentation and understanding of the data reuse distance analysis topic the basic concepts used in this work are briefly introduced below:

- *Memory Access* is a runtime access to a specific data element of the memory, and *Memory Reference* is a source code construct that generates memory accesses.

[1]http://www.lezos.gr/tools/memaddin/

In C language it could be a variable or an array index being referenced.

- *Memory Access Trace* is a sequence of all the memory accesses performed during the execution of a program.
- *Data Reuse*. A reuse happens between an element in the memory access trace and its first reoccurrence in the stream.
- *Reuse Pair*. The two occurrences of an element in a reuse are called the Reuse Pair. The first one is the *Reuse Source* and the second the *Reuse Sink*. The equivalent terms for the memory references associated with a reuse are the *Reference Source*, *Reference Sink*, and *Reference Pair*.
- *Time Distance* [6] aka *Reference Distance* [7] or *Absolute Reuse Distance* [8]. The total number of accesses between reuses. The time distance for every access is calculated by subtracting the time of the reuse source from the time of the reuse sink (Figure 1a). The overall time distance of a reference pair is the sum of all distances relating to it.
- *Reuse Distance* [3] aka *LRU Stack Distance* [9] or *Unique Reuse Distance* [8]. Similar to time distance but for the number of distinct data elements accessed between reuse source and sink (Figure 1a).
- *Reuse Distance Histogram (RDH)* aka *Reuse Signature*. The distribution of reuse distances in an execution can be viewed as a histogram (Figure 1b) where each bar represents a source code data element or a reference pair. A bar's placement on the X-axis signifies the total reuse distances of the memory accesses related to this element and the Y-axis value represents the total reuses.

Increased temporal locality is achieved when accesses to the same memory data element occur in a small time interval [10]. Depending on the size of the cache we can say that high temporal locality can offer a smaller cache miss rate. Time and reuse distance have both been proposed as a metric for data locality [2], [3], [6], [7]. Reuse distance though, has a clearer connection with cache behavior. It can actually determine the miss ratio for a fully-associative LRU cache by comparing the reuse distance of a memory element with the cache size. Beyls and D'Hollander [3] show that even for caches with a low associativity level or direct mapped caches, reuse distance can predict the number of cache misses accurately.

Figure 1a presents an example of a program execution with five data elements and eleven memory accesses. Each dashed arrow represents a reuse pair starting from the reuse sink of an element in the access trace and pointing to its previous occurrence: the reuse source. The time and reuse distances of a reuse are shown under its reuse sink. Distances for the first access of each element are set to infinite because there is no reuse source available. The corresponding histogram for this execution can be seen on Figure 1b. Element a has two reuses and a total reuse distance value of 3, while elements b and c are not visible because they have no reuses. On a hypothetical memory with size equal to three elements there would be no misses caused by element a. Though, misses will be caused by elements d and e because their reuse distances are greater

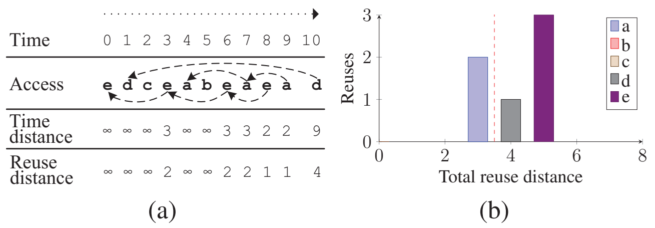


Fig. 1. a) Memory access trace. b) Reuse distance histogram.

than 3. The memory is presented as a dashed vertical line on the histogram. MemAddIn uses a reuse distance histogram as a visual aid to make optimization suggestions for a given memory hierarchy.

III. Data Reuse Distance Computation

A number of data reuse distance analysis algorithms have been implemented in MemAddIn, including: algorithms for measuring the data reuse distance of the reference pairs [9], [12], [13], [14], [11], [2] and algorithms for sampling the memory access trace before any actual measurement is performed [15]. Each implementation obeys the following pattern on every runtime memory access: 1) for simplicity all array indexes are flattened at a first step, and 2) one of the available sampling algorithms is then executed. The sampling algorithm determines the number of accesses to be skipped. Profiling overhead of reuse distance computation can be reduced significantly through sampling of the access trace. MemAddIn implements the set sampling, time sampling, and reservoir sampling methods [15]. 3) If sampling indicates that an access should be evaluated for reuse, one of the data reuse distance measurement algorithms is called. This algorithm updates a structure holding the reuse distance data for each reference pair.

Computing the data reuse distance of reference pairs provides a more accurate approximation metric for temporal locality than time distance. Unfortunately, this is a more complex task, than measuring the time distance. Apart from the simple but time demanding algorithm of naively counting the distinct elements, a number of more efficient algorithms have been proposed over the years [9], [12], [13], [14], [11], [2]. Most of them have been realized in MemAddIn. The data structures used throughout these implementations were as

TABLE I
Reuse Distance Measurement Algorithms.

Proposal	**Method**	**Time**
Time distance [6], [7], [8]	-	$O(N)$
LRU stack (list-based) [9]	From scratch	$O(NM)$
Markers (two lists) [11]	Based on [9]	$O(N\sqrt{M})$
Bit vector / m-ary tree [12]	Based on [9]	$O(NlogN)$
Balanced BST (AVL tree) [13]	Based on [12]	$O(NlogM)$
Balanced BST (splay tree) [14]	Based on [13]	$O(NlogM)$
Tree of holes [11]	Based on [12]	$O(NlogM)$
Preallocated tree of holes [11]	Based on [12]	$O(NlogN)$
Dynamic tree compression [2]	-	$O(Nlog^2M)$

N is the length of the memory access trace and M is the number of data elements used by the program.

consistent as possible. We realized slightly altered versions of the original algorithms, designed to operate on loop-dominated multimedia applications. Hence, they focus specifically on arrays omitting any noncritical scalar elements. A list of the available algorithms is provided in Table I.

IV. MemAddIn Tool Description

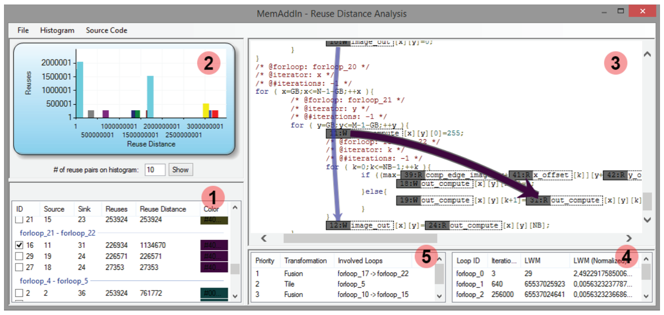
A. Basic Structure

As with the majority of dynamic analysis tools, MemAddIn uses instrumentation and profiling to extract the reuse distance for each reference pair. The extra code is inserted into the source code of the application and the tool heavily relies on MEMSCOPT source-to-source compiler [16] for this task. We implemented MemAddIn as a Visual Studio extension in order to provide a seamless and user friendly environment. Thus, making it useful for application developers. The integration of the tool in cross-platform environments, such us eclipse and NetBeans, is planned as well.

B. User Interface

The input algorithm is handled as a Visual Studio C/C++ project. When launching MemAddIn the user should firstly indicate the function to be examined from a list of all the available functions in the currently edited C file. A number of path related settings are also required to be set, as well as the preferred algorithms for reuse distance measurement and sampling. The second step of the procedure involves the actual computation of each reference pair's reuse distance. Depending on the user's indications, regarding measurement algorithms, the execution times for this phase varies. Finally, the visualized results are available in the optimization environment presented in Figure 2.

All reference pairs are listed in a descending order, according to their reuse distance (Figure 2, Point 1). They are categorized by the loop pair they are enclosed in and a distinct color is assigned for each group. A reuse histogram is also available for the reference pairs (Figure 2, Point 2) as well as a source code representation (Figure 2, Point 3). Reference pairs are portrayed as colored arrows in the source code, starting from the reference source and pointing to the sink. Point 4 in Figure 2 lists the *Loop Weight Metric (LWM)* values for each loop. LWM characterizes the criticality of a loop respecting the number, size and accesses of the arrays



ID	Source	Sink	Reuses	Reuse Distance	Color
☐ 21	15	23	253924	253924	#40...
forloop_21 - forloop_22					
☑ 16	11	31	226934	1134670	#40...
☐ 29	19	24	226571	226571	#40...
☐ 27	18	24	27353	27353	#40...
forloop_4 - forloop_5					
☐ 2	2	36	253924	761772	#00...

```
                10:W image_out [x][y]=0;
        }
}
/* @forloop: forloop_20 */
/* @iterator: x */
/* @#iterations: -1 */
for ( x=GB;x<=N-1-GB;++x ){
        /* @forloop: forloop_21 */
        /* @iterator: y */
        /* @#iterations: -1 */
        for ( y=GB;y<=M-1-GB;++y ){
                11:W [x][y][0]=255;
                /* @forloop: 22 */
                /* @iterator: k */
                /* @#iterations: -1 */
                for ( k=0;k<=NB-1;++k ){
                        if ((max- 39:R comp_edge_imag 41:R x_offset [k]][y+ 42:R y_o
                                18:W out_compute [x][y]
                        }else{
                                19:W out_compute [x][y][k+1]= 31:R out_compute [x][y][k]
                        }
                }
                12:W image_out [x][y]= 24:R out_compute [x][y][NB];
```

Priority	Transformation	Involved Loops
1	Fusion	forloop_17 -> forloop_22
2	Tile	forloop_5
3	Fusion	forloop_10 -> forloop_15

Loop ID	Iteratio...	LWM	LWM (Normalized)
forloop_0	3	29	2,4922917585006...
forloop_1	640	65537025923	0,0056323237787...
forloop_2	256000	65537024641	0,0056323236686...

Fig. 2. MemAddIn user interface - reuse distance analysis window.

it operates on. Loops with high LWM are considered heavy and could be candidates for loop optimizations. The LWM calculation method for a single loop is described in Definition 1. The tool's suggestions are available to the user as an enumerated list (Figure 2, Point 5) where the transformations are sorted ascending according to their importance for locality optimization. The higher reuse distance a reference pair has, the more important it is considered. At present, two types of transformation suggestions are available. Both of them rely on the grounds that reference source should be brought closer to the sink in order to minimize the reuse distance: 1) when the reference source is enclosed in a different loop than the sink, a *fusion* of these loops is proposed, and 2) if they both reside in the same loop some sort of *tiling* is appropriate. The designer consults these results together with the LWM values and decides upon proper transformations for reuse distance reduction and the subsequent locality optimization.

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.

V. Experiments

We optimized a realistic loop-dominated image processing application following MemAddIn's suggestions. Cavity detector [5] is a medical diagnostic application used to analyze MRI images. Five optimization steps have been applied and we count the cache miss rate and overall RAM accesses on each one of them. Two memory hierarchy scenarios are tested using the XMSIM memory simulator [17]: 1) an SDRAM with an underlying cache unit of 8 kb size, direct-mapped, 1 word per block, 32 bits per word, with write-back policy, and 2) the same hierarchy but with 16kb cache size. The input to the algorithm is always a 640x400 pixels raster image. Cavity detector involves 23 loops which comprise 5 top-level loop nests. MemAddIn's recommendations included the gradual fusion of these nests according to the LWM weights of the involved loops and the reuse distances of the array reference pairs relating to them. After the application of the proposed transformations there were 7 loops left, comprising a single loop nest with all algorithmic stages of the original code. RAM accesses reduced down to 61% (Figure 3b) while the cache miss rate dropped from 46% to 38.5% for the 16kb cache architecture (Figure 3c). Further optimization of cavity detector is possible over tiling the remaining loops, as revealed by an additional execution of MemAddIn. These optimizations are not portrayed in Figures 3b and 3c though.

Furthermore, the reuse distance measurement algorithms' implementations of the tool are demonstrated on cavity detector and also on the sobel, roberts, robinson and canny operators. We evaluate the time consumed by each algorithm

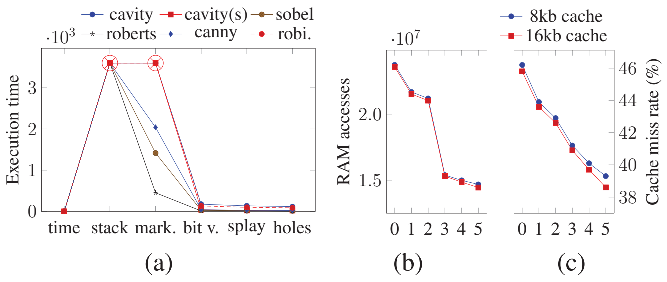


Fig. 3. a) Performance comparison of reuse distance measurement algorithms. ⊗ denotes aborted executions that were taking too long (>1 hour). b) RAM accesses and c) cache miss rate of each optimization step.

for data reuse distance computation for the reference pairs of all arrays. Measurements were obtained without sampling except from *cavity(s)* (Figure 3a) where the time sampling algorithm was used on cavity detector, omitting 80% of the memory access trace. MemAddIn ran on a Mobile DualCore Intel Core i5-2450M CPU at 2.50GHz with 7.85 GB of usable DDR3-1333 RAM. Figure 3a shows that all algorithms performed proportionally to their theoretical time complexity presented in Table I.

VI. Comparison to Related Work

A plethora of tools for memory oriented optimizations have been proposed. Those utilizing data reuse distance analysis are often used to estimate cache miss ratio [8], [18], [19] and for locality optimization [20].

In [20] Beyls and D'Hollander present SLO, a cache profiling tool that measures reuse distances in C programs and suggests code optimizations in a similar way with MemAddIn. The main differences of our work compared to SLO are: 1) MemAddIn is an integrated framework for high level algorithm design and optimization for embedded systems. Reuse distance analysis is only a part of its features. SLO on the other hand is dedicated to optimization suggestions via reuse distance analysis, 2) a selection of reuse distance measurement and sampling algorithms is available to the MemAddIn user for experimentation, while SLO incorporates a single approach for these tasks, and 3) a supplementary metric called LWM is devised in MemAddIn for better user assistance in deciding the proper code transformations. Additionally, work is currently carried out towards automating the application of the optimizations suggested by MemAddIn directly into the source code with no intervention from the developer. To achieve this, a mechanism will be incorporated that uses the transformation engine of the MEMSCOPT compiler [16].

VII. Conclusion and Future Work

In this paper, MemAddIn tool is discussed with emphasis on data reuse distance analysis capabilities. These are brought forth through a mechanism for the generation of locality optimization suggestions on input C applications. To achieve this, we need to compute the reuse distance for all reference pairs of the program. A number of algorithms from the bibliography have been implemented and optimized for this performance demanding task. Experimental results prove a reduction of 39% in overall RAM accesses after optimizing an application respecting the transformation hints proposed by the tool. Future work considers the addition of parallel reuse distance analysis algorithms and concepts. Effort is also devoted on methods to automate the application of the optimizations suggested by MemAddIn.

References

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

[2] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 6, pp. 20:1–20:39, Aug. 2009.

[3] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior." in *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, Anaheim, California, USA, 2001, pp. 617–622.

[4] C. Lezos, G. Dimitroulakos, A. Freskou, and K. Masselos, "Dynamic source code analysis for memory hierarchy optimization in multimedia applications," in *2013 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Oct. 2013, pp. 343–344.

[5] F. Catthoor, K. Danckaert, S. Wuytack, and N. Dutt, "Code transformations for data transfer and storage exploration preprocessing in multimedia processors," *IEEE Design Test of Computers*, vol. 18, no. 3, pp. 70–82, May 2001.

[6] X. Shen, J. Shaw, B. Meeker, and C. Ding, "Locality approximation using time," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 55–61.

[7] C. Pyo, K.-W. Lee, H.-K. Han, and G. Lee, "Reference distance as a metric for data locality," in *High Performance Computing on the Information Superhighway, 1997. HPC Asia '97*, Apr. 1997, pp. 151–156.

[8] R. Sen and D. A. Wood, "Reuse-based online models for caches," in *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '13. New York, NY, USA: ACM, 2013, pp. 279–292.

[9] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970.

[10] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*, 1st ed. Burlington, MA: Morgan Kaufmann, Sep. 2007.

[11] G. Almasi, C. Cascaval, and D. A. Padua, "Calculating stack distances efficiently," in *Proceedings of the 2002 Workshop on Memory System Performance*, ser. MSP '02. New York, NY, USA: ACM, 2002, pp. 37–43.

[12] B. T. Bennett and V. Kruskal, "LRU stack processing," *IBM Journal of Research and Development*, vol. 19, no. 4, pp. 353–357, Jul. 1975.

[13] F. Olken, "Efficient methods for calculating the success function of fixed-space replacement policies," Lawrence Berkeley Lab., CA (USA), Tech. Rep. LBL-12370, May 1981.

[14] R. A. Sugumar, "Multi-configuration simulation algorithms for the evaluation of computer architecture designs." Thesis, 1993, ph.D.

[15] Y. Tille, *Sampling Algorithms*, ser. Springer Series in Statistics. Springer New York, 2006.

[16] G. Dimitroulakos, C. Lezos, and K. Masselos, "MEMSCOPT: A source-to-source compiler for dynamic code analysis and loop transformations," in *2012 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Oct. 2012, pp. 385–386.

[17] G. Dimitroulakos, T. Lioris, C. Lezos, and K. Masselos, "XMSIM: A tool for early memory hierarchy evaluation," in *2012 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Oct. 2012, pp. 405–406.

[18] E. Berg and E. Hagersten, "StatCache: a probabilistic approach to efficient and accurate data locality analysis," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, 2004, pp. 20–27.

[19] Y. Zhong, S. Dropsho, X. Shen, A. Studer, and C. Ding, "Miss rate prediction across program inputs and cache configurations," *IEEE Transactions on Computers*, vol. 56, no. 3, pp. 328–343, Mar. 2007.

[20] K. Beyls and E. D'Hollander, "Refactoring for data locality," *Computer*, vol. 42, no. 2, pp. 62–71, Feb. 2009.