Compiler-Directed Data Locality Optimization in MATLAB

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

ABSTRACT

Array programming languages, such as MATLAB, are often used for algorithm development by scientists and engineers without taking into consideration implementation related issues and with limited emphasis on relevant optimizations. Application code optimization, especially in terms of data storage and transfer behavior, is still an important issue and heavily affects implementations' quality in terms of performance, power consumption etc. Efficient approaches for the optimization of high level application code are required to derive high quality implementations while still reducing development time and cost. This paper presents MemAssist, a software tool supporting application developers in detecting parts of the application code in MATLAB that do not exploit efficiently the targeted processor architecture and especially the memory hierarchy. Furthermore, the proposed tool guides application developers in applying code transformations in MATLAB for the optimization of the algorithm's temporal data locality. An image processing algorithm has been optimized using MemAssist as a practical usage scenario. Experimental results prove that the use of MemAssist can heavily reduce cache misses (up to 40%) and improve execution time (up to 30% speedup) on two different processor architectures. Thus, MemAssist can be used for optimized application code development that can lead to efficient implementations while still reducing development time and cost.

CCS Concepts

•**Software and its engineering** → *Compilers;* •**General and reference** → *Performance; Metrics;*

Keywords

Reuse distance analysis; data locality optimization; MATLAB-to-C; source-to-source optimization

SCOPES '16, May 23 - 25, 2016, Sankt Goar, Germany

ISBN 978-1-4503-4320-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2906363.2906378

1. INTRODUCTION

MATLAB is a high level array programming language used broadly for prototyping algorithms in scientific and engineering settings. At this level, developers do not consider code optimization and implementation issues and they focus on concisely evaluating their algorithms at a high level of abstraction. Providing optimization suggestions and applying optimizations at this level can have a big impact on the quality of hardware implementations. This is particularly important in a context where implementation code (C, VHDL) is automatically generated from MATLAB codes targeting embedded systems and devices such as mobile phones. Furthermore, supporting application developers in exploring high level algorithmic space efficiently can lead to significant development time and cost reduction since time consuming design iterations (in case where constraints are not met at low levels) can be avoided.

Most modern approaches on performance optimization of loop dominated algorithms have focused on parallelization, targeting relevant architectures. However to achieve global optimization of an algorithm, optimization for parallelism and locality and the reduction of recomputations should be targeted in a balanced way [14]. In such a context, data locality optimization and the evaluation of memory behavior are very important issues.

In [10], MemAddin, a software tool for data reuse exploration including data reuse distance analysis and optimization, is presented as an extension to Microsoft's Visual Studio IDE. MemAddin supports developers in efficiently applying transformations for the optimization of loop dominated algorithms in C (e.g. image and signal processing applications). The suggested transformations target the optimization of algorithms' data temporal locality and aim at exploiting the target processor's memory hierarchy to reduce cache misses and improve execution time. This paper extends the work presented in [10] and introduces the following innovative contributions:

► *MemAssist*[1], a software tool capable of providing detailed data locality optimization suggestions and supporting relevant optimizations at MATLAB level. At this level users traditionally do not apply implementation oriented optimizations as in lower level descriptions (e.g. in C). Except from being provided as an extension to Microsoft's Visual Studio, MemAssist is currently also offered through the web.

► The required data reuse distance computation is performed at C code level. A MATLAB-to-C compiler has been developed for this purpose (*MAFE*). This compiler can re-

---

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

late the input MATLAB variables with output C code. Then reuse distance analysis is performed in the C code and the relevant optimization suggestions are mapped to the input MATLAB code.

► Methods for inferring proper code transformations have been developed and discussed in detail.

► The optimization of an image processing algorithm from the UTDSP benchmark suite [16] is discussed. This case study proves the effectiveness of MemAssist. Experiments have been conducted on a cache simulator as well as on real systems (a x86 laptop and an ARM smartphone device).

2. BASIC CONCEPTS

2.1 Proposed Flow

Existing design flows for embedded systems development do not take into consideration implementation related issues at MATLAB level. They depend solely on the compile time optimizations performed by MATLAB-to-C/VHDL compilers to achieve good quality C/VHDL code. The effectiveness of optimizations applied at the MATLAB source code level is discussed in the proposed approach. MemAssist exploration tool targets the optimization of MATLAB application code with respect to implementation. The C code generated from the optimized MATLAB code, regardless of the MATLAB-to-C compiler used, leads to better quality C code than the code generated if implementation oriented optimizations are not performed at MATLAB level. The main concept of the proposed approach is presented in Figure 1.

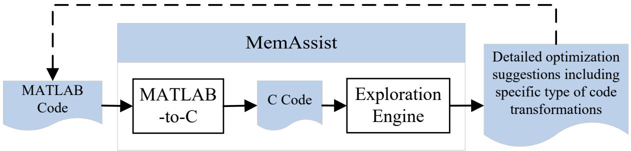

Figure 1: Structure of the MATLAB optimization flow in MemAssist.

2.2 Data Reuse Concepts

A data element e of a source program is a scalar variable or an array index that corresponds to a memory address. a_e is a runtime memory access to data element e and r_e is a reference of e in the source code. In the code fragment

```
x = ...;
y = z*(w/5)+y;
...  = x;
```

the data elements of the program are variables x, y, z and w. The *memory access trace* for this would be the sequence $\{x, z, w, y, y, x\}$, given that the right hand side of an assignment is accessed left to right and before the left hand side. A *data reuse* occurs between an element in the memory access trace and its first reoccurrence. These two occurrences in the stream are called the *reuse pair* and the corresponding references that generated them are called the *reference pair*. The number of accesses that occur between a reuse pair is this pair's *time distance* and the number of distinct data elements accessed between them are their *reuse distance*.

2.3 Relating Reuse Distances to Source Code

The *total reuse distance* of a reference pair is the sum of the reuse distances of all reuse pairs relating to it. For the given example, the time distance between the first and second occurrence of x would be 5 accesses while the reuse distance would be 3 elements (z, w, y). Given the size of a cache memory, the reuse distance of a reuse pair indicates whether a cache miss will occur if the size of the elements accessed between reuses is greater than the size of the cache. Thus, in order to reduce cache misses, the reuse distance must also be reduced by moving the two references that generated the critical reuse pair closer together.

MemAssist uses a *reuse distance histogram (RDH)* to assist the user in deciding upon the importance of each suggested transformation while their type is inferred using the method described in section 3. Loop pairs that encompass reference pairs with high reuse distances are typically of higher priority in the optimization queue. An RDH can be constructed using the total reuse distances of a program's reference pairs. Each bar in this histogram represents a reference pair. A bar's placement on the X-axis signifies the total reuse distances of the pair and the Y-axis value represents the total reuses.

3. OPTIMIZATION APPROACH

The total reuse distances of the reference pairs are obtained through instrumentation and profiling of the C code. During instrumentation, static analysis is also performed in order to get information about the data elements of the program (number of dimensions, dimension sizes etc.). MemAssist suggests either the fusion of a *loop pair* or the implementation of a tiling optimization. Both of these transformations have positive effects on data locality [8]. The following method is used to automatically infer these transformations. Every reference pair is matched to the loop pair where the two references reside in. The data required to apply this association between references and loops are acquired during static analysis/instrumentation step. The whole process described in this section is performed on a structure called *nested loop tree*, where the hierarchy of the application loops is represented. The code block that contains the loops (the body of a function), is the root of the tree and is referred as r. The rest of the nodes represent loops residing in that block and they are identified by a unique incremental positive number for each node. The loop where the uses occur in a reference pair is the *source* loop while the one where the reuses occur is the *sink*. An example loop hierarchy with 8 loops is shown in Figure 2.

From a nested loop tree the distinction between a fusion and a tiling-like optimization can be made: (1) Tiling is inferred if the source is the same with the sink or if one is an ancestor to the other because, in both cases, the reuse occurs between iterations of the same loop, (2) if none is an ancestor to the other it means that use and reuse occur between iterations of different loops and fusion of these loops is required.

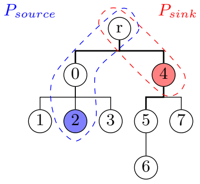

Figure 2: Nested loop tree.

This method could be applied by implementing a tree data structure. In the implementation of MemAssist the usage of a tree data structure and the accompanying traversals over it are avoided. A number of sets equal to the number of loops is initially defined. Each set corresponds to a node in the nested loop tree and it's denoted by P_x. The contents of these sets are the nodes that form a direct path from node x to the root of the tree. This can be formalized using the set-builder notation as follows:

$$T = \{x \mid (x \in \mathbb{Z} \land 0 \leq x < N) \lor x = r\} \tag{1}$$

$$P_x = \{y \mid (y \in T \land y = predecessor(x)) \lor y = x\} \tag{2}$$

T contains all nodes of the tree, both the loops and the block that encompasses them (r). The appropriate decision is inferred according to which of the following statements is true:

$$(T_{source} \cap P_{sink} = \{\varnothing\}) \land (T_{sink} \cap P_{source} = \{\varnothing\}) \tag{3}$$

$$(T_{source} \cap P_{sink} = T_{source}) \land (T_{sink} \cap P_{source} = \{\varnothing\}) \tag{4}$$

$$(T_{source} \cap P_{sink} = \{\varnothing\}) \land (T_{sink} \cap P_{source} = T_{sink}) \tag{5}$$

$$(T_{source} \cap P_{sink} = \{r\}) \land (T_{sink} \cap P_{source} = \{\varnothing\}) \tag{6}$$

$$(T_{source} \cap P_{sink} = \{\varnothing\}) \land (T_{sink} \cap P_{source} = \{r\}) \tag{7}$$

T_{source} is a set that contains only the source loop and T_{sink} contains only the sink. If Equation 3 is true then neither source nor sink is an ancestor to the other and fusion of these loops is required. In case they are the same loop or one is an ancestor to the other, a tiling optimization is inferred (Equations 4, 5). Equations 6 and 7 imply that one of the references in not inside a loop so no transformation is suggested. Consider the example of Figure 2 where a reference pair's use is inside loop 2 and the reuse is in loop 4. The input data would be:

$$\begin{gathered} N = 8, T = \{r, 0, 1, 2, 3, 4, 5, 6, 7\}, T_{source} = \{2\}, \\ T_{sink} = \{4\}, P_r = \{r\}, P_0 = \{0, r\}, P_1 = \{1, 0, r\}, \\ P_2 = \{2, 0, r\}, P_3 = \{3, 0, r\}, P_4 = \{4, r\}, P_5 = \{5, 4, r\}, \\ P_6 = \{6, 5, 4, r\}, P_7 = \{7, 4, r\}, P_{source} = P_2, P_{sink} = P_4 \end{gathered}$$

and the proposed transformation:

$$\left. \begin{array}{r} T_{source} \cap P_{sink} = \{2\} \cap \{4, r\} = \{\varnothing\} \\ T_{sink} \cap P_{source} = \{4\} \cap \{2, 0, r\} = \{\varnothing\} \end{array} \right\} Fusion$$

4. EXPERIMENTAL EVALUATION

An image processing application from the UTDSP benchmark suite [16] has been coded in MATLAB and optimized using MemAssist. C code has been generated for the original and the optimized MATLAB code using both Mathworks MATLAB Coder and MAFE. Execution time has been evaluated for six different versions of the application code (MATLAB original and optimized, MATLAB Coder generated C original and optimized, MAFE generated C original and optimized). All codes have been executed on two different platforms: (1) an Intel Core i5 CPU at 2.50GHz with 7.85GB of usable DDR3-1333 RAM and the following caches: I1 and D1 (32 KB, 8-way associative, 64 byte line size), L2 (256 KB, 8-way associative, 64 byte line size), L3 (3 MB, 12-way associative, 64 byte line size), and (2) an 832 MHz ARM CPU with 512 MB RAM. The C codes have been compiled with: (1) Visual C++ compiler using the /O2 optimization switch on the x86 laptop, and (2) the Android NDK toolset on the ARM smartphone. The MATLAB version of the code has only been executed on the x86 platform. Two raster images have been used as inputs: one with 400x400 pixels size and another with 100x100 pixels.

Figures 3a and 3b present the execution times of the application for 500 executions while Figure 3c shows the average

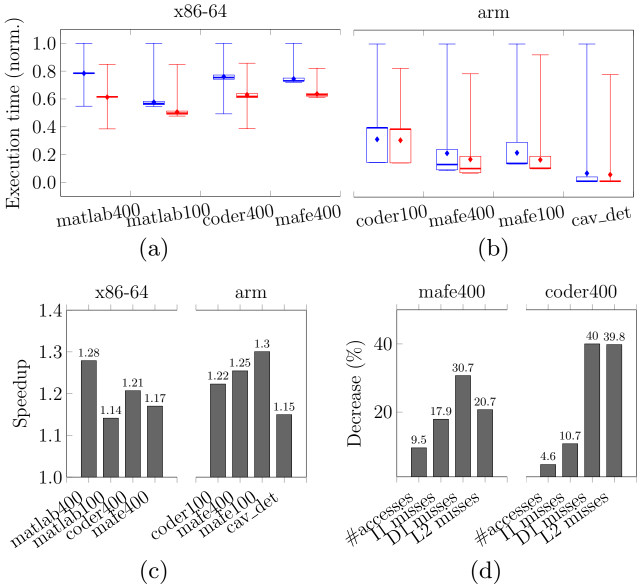

Figure 3: a,b) Comparison of execution time between the original and optimized versions of the code. All values are normalized to the [0,1] range. c) Average speedup. d) Memory performance. Speedup is calculated with *original/optimized* and decrease percentage with *((original-optimized)/original)*100.*

speedups achieved for these executions. The C code generated from the optimized MATLAB code runs up to 1.3 times faster on the ARM device and up to 1.21 times faster on the x86 device than the code generated from the original MATLAB code. A speedup of 1.28 is also achieved on the MATLAB interpreter. Cavity detector [5] is a medical diagnostic application that was optimized in [10] using MemAssist. A speedup of 1.15 was observed for this algorithm on ARM.

Cache performance evaluation has been obtained using the Cachegrind simulator [12]. The following realistic cache configuration has been used for all simulations: I1 = (32KB, 8-way associative, 64byte), D1 = (32KB, 8-way associative, 64byte), L2 = (4MB, 16-way associative, 64byte). Figure 3d presents the results. D1 and L2 cache misses have been decreased by 20.7% to 40% for both MATLAB Coder and MAFE generated C codes while at the same time memory accesses also decreased by 4.6% to 9.5%.

5. COMPARISON TO RELATED WORK

Several tools exist targeting evaluation of memory behavior [9, 11, 15, 17, 7, 19, 1, 13]. Those utilizing data reuse distance analysis are often used to estimate cache miss ratio and to optimize locality. Only some of them are focused on providing suggestions for code transformations that will improve the data locality of an algorithm at a high level of abstraction [10, 4, 3, 18, 2].

In the work presented in [4, 3, 2], SLO, a cache profiling tool is discussed. The tool calculates reuse distances in C programs and suggests code optimizations in a similar way with MemAssist. The following comparative comments can be made between SLO and MemAssist: (1) MemAssist targets MATLAB in addition to C which has already been extensively used as the input specification on most similar systems, including SLO, and (2) SLO is dedicated to optimization suggestions via reuse distance analysis while in MemAssist reuse distance analysis is used only in a part of its features and metrics [10].

In [4, 3] an approach similar to the one followed in this paper is presented for the inference of locality optimizing loop transformations in the SLO tool. The whole process in this approach is performed at the level of basic blocks rather than at the level of loops. A *control flow graph (CFG)* is used in conjunction with a structure, similar to the nested loop tree, called the *nested loop forest*. The CFG is used to infer the pair of *outermost executed loop headers (OELH)* for the basic blocks where the reuse source and sink appear. The OELH of a basic block is defined as its closest to the root ancestor in the nested loop forest that is executed between use and reuse. Given the OELH source and OELH sink: (1) tiling is inferred if they are the same node, as the reuse source and sink occur between iterations of the same loop, and (2) fusion is inferred if they are different loop headers, because the reuse source and sink occur in different loops. The advantage of the set-based approach described in this paper over the corresponding in [4, 3] is that neither CFG nor detailed information about the basic blocks of the program are needed. The only required input data regard the loop hierarchy and information about which loop encloses each memory reference. Thus, the method proposed in this paper is much easier to implement, while predicting the same transformations. In terms of accuracy both methods produce similar results.

Most existing MATLAB compilers target the generation of optimized lower level code for specific architectures. It is the first time that a compiler is used to assist the inference of locality optimizing transformations for MATLAB sources. The only work where some sort of reuse distance analysis is performed on algorithms written in a high level array language is that of Chauhan and Shei [6]. They present an algorithm to estimate reuse distances on MATLAB code using an extended version of dependence graphs. There are two main differences between that approach and the work presented in this paper: (1) in [6] authors perform static analysis on MATLAB code to infer the reuse distances while in this work the actual reuse distances are calculated through instrumentation and profiling, and (2) Chauhan and Shei provide estimations about the cache misses caused by the examined application while MemAssist guides the developer in applying specific transformations that will optimize cache performance by improving locality.

6. CONCLUSIONS

This paper discusses MemAssist, a software tool for the optimization of MATLAB code in terms of temporal data locality. Experimental results prove that the use of MemAssist can lead to the generation of implementation code that achieves shorter execution time and improved cache performance. Furthermore the use of MemAssist can also significantly reduce development time and cost since exploration is moved to higher levels of abstraction thus reducing exploration time. The optimization of MATLAB sources before MATLAB-to-C compilation tools may lead to the generation of better quality implementation code that meets performance requirements and constraints. In this way time consuming iterations are eliminated.

7. REFERENCES

[1] E. Berg and E. Hagersten. Fast Data-locality Profiling of Native Execution. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 169–180, New York, NY, USA, 2005. ACM.

[2] K. Beyls and E. D'Hollander. Refactoring for Data Locality. *Computer*, 42(2):62–71, Feb. 2009.

[3] K. Beyls and E. H. D'Hollander. Intermediately Executed Code is the Key to Find Refactorings That Improve Temporal Data Locality. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 373–382, New York, NY, USA, 2006. ACM.

[4] K. Beyls and E. H. D'Hollander. Refactoring Intermediately Executed Code to Reduce Cache Capacity Misses. *The Journal of Instruction-Level Parallelism*, 10, June 2008.

[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*, 18(3):70–82, May 2001.

[6] A. Chauhan and C.-Y. Shei. Static Reuse Distances for Locality-based Optimizations in MATLAB. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 295–304, New York, NY, USA, 2010. ACM.

[7] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 55–65, Mar. 2010.

[8] M. Kowarschik and C. Weiss. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies*, number 2625, pages 213–232. Springer Berlin Heidelberg, Jan. 2003.

[9] C. Lezos, G. Dimitroulakos, A. Freskou, and K. Masselos. Dynamic source code analysis for memory hierarchy optimization in multimedia applications. In *Proceedings of the 2013 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 343–344, Cagliari, Italy, Oct. 2013.

[10] C. Lezos, G. Dimitroulakos, and K. Masselos. Reuse distance analysis for locality optimization in loop-dominated applications. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1237–1240, Grenoble, France, Mar. 2015.

[11] X. Liu and J. Mellor-Crummey. Pinpointing data locality bottlenecks with low overhead. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–193, Apr. 2013.

[12] N. Nethercote. *Dynamic binary analysis and instrumentation*. Ph.D. Dissertation, University of Cambridge, 2004.

[13] B. Quaing, J. Tao, and W. Karl. YACO: A User Conducted Visualization Tool for Supporting Cache Optimization. In *High Performance Computing and Communications*, number 3726, pages 694–703. Springer Berlin Heidelberg, Sept. 2005.

[14] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[15] A. Rane and J. Browne. Enhancing Performance Optimization of Multicore/Multichip Nodes with Data Structure Metrics. *ACM Trans. Parallel Comput.*, 1(1):3:1–3:20, May 2014.

[16] M. A. R. Saghir. *Application-Specific Instruction-Set Architectures for Embedded DSP Applications*. Ph.D. Dissertation, University of Toronto, Toronto, Canada, 1998.

[17] 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*, SIGMETRICS '13, pages 279–292, New York, NY, USA, 2013. ACM.

[18] O. A. Sopeju, M. Burtscher, A. Rane, and J. Browne. AutoSCOPE: Automatic Suggestions for Code Optimizations Using PerfExpert. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA '11, pages 19–25, Las Vegas, Nevada, USA, July 2011.

[19] Y. Zhong, X. Shen, and C. Ding. Program Locality Analysis Using Reuse Distance. *ACM Trans. Program. Lang. Syst.*, 31(6):20:1–20:39, Aug. 2009.