EXAMINATION OF ALGORITHMS USING DYNAMIC DATA FLOW GRAPHS

Gábor WACHA
Advisor: Béla FEHÉR

I. Introduction

In some of the algorithmically intensive embedded real time streaming applications, a single CPU is not sufficient to handle the task in real time, but the processing power of more processor cores would be theoretically enough. Algorithms are usually implemented in C/C++ languages which natively do not support parallel implementation and algorithm slicing [1] into multi-core processor systems.

Many of the embedded processing platforms – including FPGA based soft core processor systems, or Cell architecture – do not offer cache coherency among the processors. Standard profiling results – showing runtime information only – are sufficient to slice the algorithm into multiple CPU nodes, thus creating a balanced load distribution, but internal communication bandwidth often remains a bottleneck [2].

This paper relies on the visualization and generation of Aggregated Dynamic Data Flow Graphs (ADDFG) introduced in [3]. This ADDFG can help to visualize these critical data paths of a given software. Also, with the information of the critical data paths, an estimation of a possible slicing of the algorithm can be given. Finding the correct slices of an algorithm can achieve greater performance on multicore systems. Once the slices are found, a multithread program can be generated from a given task graph [4]. Silva, R.E. et al. use Dynamic Data Flow Graphs to measure the network contention of MPI-based applications in [5]. Their methods can be used to measure the performance of an automatically sliced algorithm.

This paper describes a basic communication model between specific functions of a given software. With this model, a clustering of the data flow graph can help to find the data-independent functions of an algorithm. The data-independent functions can be executed as different software threads, thus minimizing the necessary synchronization. An algorithm sliced in this way can perform better on multicore environments.

II. Introduction to program slicing based on run time analysis

As Weiser, M. introduces in [1], program slicing is a method for automatically decomposing programs to slices by analyzing their control and data flow. Each slice does a specific subset of the algorithm's behavior.

Weiser's approach is static in the form that it uses only compile-time analysis to slice algorithms. With this method, all possible data paths can be explored. Unfortunately there is no information about the run-time "importance" of the path. It is possible that the software has a data path which is rarely used for communication during the execution. The program will not be sliced through this data path – altough the path is almost never used.

Fig. 1. shows such a case: communication channels (for example memory or register reads and writes) between nodes `D` and `Q`, `0` and `P` are rarely used in most of the time. A compiler can not slice the algorithm into independent parts (for example `0`, `A`, `B`, `C`, `D` and `P`, `Q`, `R`, `S`) without run-time knowledge.

A run-time generated dynamic data flow graph can help to find these data paths which then can be used for more advanced slicing. Also, a higher level approach is preferred: instead of an instruction or

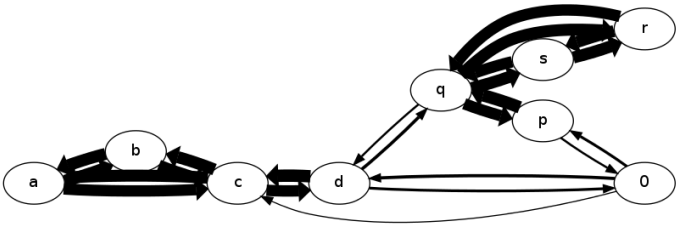

Figure 1: Dynamic data flow graph of an algorithm which can not be sliced using compile-time methods

statement level data flow graph a function level aggregated data flow graph (ADDFG) is used.

As introduced in [3], the ADDFG is a graph generated from run time data flow information. Each vertex of the graph is equivalent to a software function in the algorithm under test. An edge between two vertices means that communication happens between two specific functions during the execution of the program. This communication can be done through a memory or register access. The weight of an edge shows the amount of data transferred between the two functions during the execution.

III. Communication model

To enable the analysis of the algorithm under test a simplified communication model between functions is introduced.

In this basic model the following assumptions are made about the data transfer between functions:

- Control flow (function call, call graph) information separate from the ADDFG can be discarded. Since every function call encapsulates a data transfer, the ADDFG contains the call information. For example: the function arguments are written in the caller function and are read in the callee. A more frequent function call has an edge with greater weight in the ADDFG.
- The direction of the data transfer can be ignored, because the performance hit on each possible communication between multiple threads will usually be the same regardless the direction of the transfer.
- The communication rate is constant. Every function transfers data continuously at an average rate. This means that the actual timing of the data transfer is ignored.
- The slicing algorithm optimizes only for inter-core data transfer, not for a balanced processor load. For each function the execution time equals, the processor requirements of the algorithm are ignored.

The first two assumptions are correct to the effect that a real-world software has these characteristics. The consequence of the third simplification is that the slicing algorithm can not find slices which could be pipelined in the optimized program. An other simplifying consequence is that the weight of an edge in the ADDFG can be treated as the data communication rate between two functions.

The fourth simplification has problems. With this assumption the balanced load distribution on each processor core can not be guaranteed. The ADDFG does not hold the necessary information to enable a more balanced load.

IV. Program slicing based on spectral clustering

With the communication model introduced in Section III., a spectral clustering method on the adjacency matrix of the ADDFG can find a data-independent slicing of the algorithm.

The intent of the clustering is to group the vertices of the ADDFG so that the sum of the weight of the edges in the cut $W(A_i, \overline{A_i})$ between the partitions A_i and $\overline{A_i}$ are minimal. Because the edge weights represent the data communication rate between different functions, finding a minimal cut between these partitions of the ADDFG can be used to slice the program.

To avoid having separate vertices as a partition, each partition is weighted with the number of vertices inside the partition. Such a clustering, which satisfies the criteria in Equation 1., is called the minimum ratio cut [6]. Finding the minimum ratio cut is an NP-hard problem, the following algorithm is an approximation [7].

$$\min \frac{1}{2} \sum \frac{W(A_i, \overline{A_i})}{\|A_i\|} \tag{1}$$

The clustering algorithm is the following [8]. Let G be the weighted adjacency matrix of the ADDFG. Since the direction of the data transfer can be ignored, the adjacency matrix of the undirected ADDFG is $A = G + G^T$. A is a symmetric matrix, on which the spectral clustering method [8] can be used. Let L be the unnormalized Laplacian matrix of A [9]:

$$l_{ij} = \begin{cases} deg(i) & \text{if } i = j.\ deg(i) \text{ is the degree of the } i\text{-th vertex,} \\ w(i,j) & \text{if } i \neq j \text{ and there is an edge between } i \text{ and } j.\ w(i,j) \text{ is the weight of that edge,} \\ 0 & \text{otherwise.} \end{cases}$$

The unnormalized Laplacian matrix can be calculated as $D - A$, where D is the diagonal degree matrix of the weighted symmetric adjacency matrix A.

The eigenvector corresponding to the smallest eigenvalue of L has the coordinates of 1, the eigenvalue is 0. All the other eigenvalues are real [9].

The k-means clustering of the eigenvector corresponding to the second smallest eigenvalue of L can be used as an approximation for the clusterization of A [7]. Comparing the coordinates (which are real numbers) of the eigenvector to a given threshold value can give two clusters of the corresponding graph.

For more accurate results, a recursive approach on the Laplacian of the clusters, or a k-means clustering on all of the eigenvectors of L can be used [8].

V. Results

The clustering method was tested on the ADDFG shown on the left side of Fig. 3. The unclustered and the clustered adjacency matrix of the ADDFG is shown in Fig. 2. The adjacency matrix has two clusters, the algorithm will be sliced to run on two CPU cores. The test software was written so that the assumptions made in Section III. were valid. Each of the functions of the algorithm have about the same processor load.

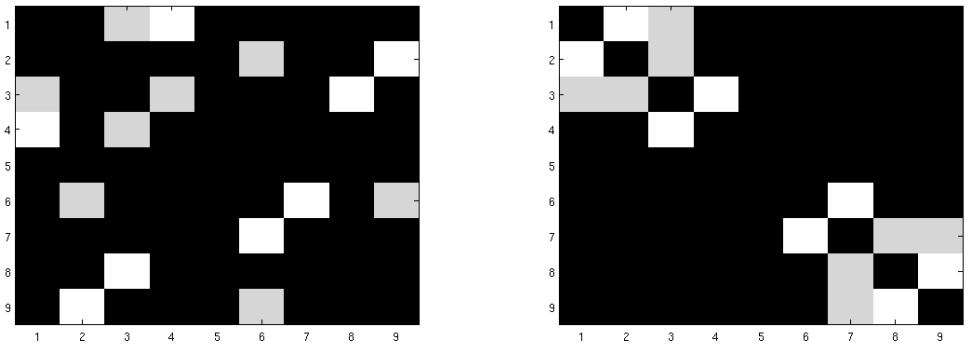

Figure 2: Adjacency matrix of the unclustered and the clustered data flow graph

To examine the performance of the sliced program, the execution time of the sliced and unsliced software was measured.

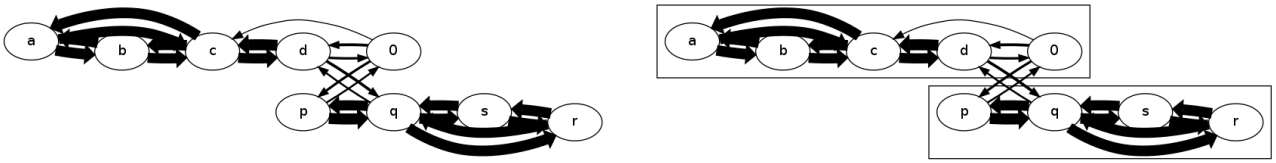

Figure 3: The slices of the program found by clustering the ADDFG

Compared to the single threaded algorithm, the execution time was 47% less when the program was sliced into two threads on a multicore CPU. To slice the algorithm, the slicing shown in Fig. 3 was used.

This measurement only shows that with the found slicing the performance of the software increases, it does not show that the slicing is optimal. A comparison with static slicing tools (for example CodeSurfer or Frama-C) should also be done to verify that a dynamic method can achieve better results.

VI. Further work

In Section III., four assumptions were made for a simpler inter-subroutine communication model. One of the assumptions was that the exact timing of the data transfer is ignored. With this simplification, pipeline structures can not be created. A future version of the ADDFG generation can include timing information in the edge weights, thus enabling the creation of pipelines.

As mentioned, the slicing algorithm does not take the computational requirements into regard, thus making load balancing not possible to attain. With the help of program profiling information this deficiency should be solved.

Usually the actual structure of the ADDFG depends from the input parameters of the algorithm. The research so far does not examine the input dependency, this should be addressed.

References

[1] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pp. 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[2] M. A. Kim and S. A. Edwards, "Computation vs. memory systems: Pinning down accelerator bottlenecks.," in *ISCA Workshops*, A. L. Varbanescu, A. M. Molnos, and R. van Nieuwpoort, Eds., vol. 6161 of *Lecture Notes in Computer Science*, pp. 86–98. Springer, 2010.

[3] I. Szabó, G. Wacha, and J. Lazányi, "Aggregated dynamic dataflow graph generation and visualization," *Carpathian Journal of Electronic and Computer Engineering*, 6(2):50–54, 2013.

[4] E. Jeannot, "Automatic multithreaded parallel program generation for message passing multiprocessors using parameterized task graphs," 2001.

[5] R. Silva, G. Pezzi, N. Maillard, and T. Diverio, "Automatic data-flow graph generation of MPI programs," in *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, pp. 93–100, 2005.

[6] L. W. Hagen and A. B. Kahng, "New spectral methods for ratio cut partitioning and clustering.," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 11(9):1074–1085, 1992.

[7] U. Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, 17(4):395–416, Dec. 2007.

[8] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, pp. 849–856. MIT Press, 2001.

[9] B. Mohar, "The Laplacian spectrum of graphs," in *Graph Theory, Combinatorics, and Applications*, pp. 871–898. Wiley, 1991.