A Dynamic Analysis Tool for Finding Coarse-Grain Parallelism

Sean Rul Hans Vandierendonck
Koen De Bosschere
Department of Electronics and Information Systems (ELIS),
Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
{sean.rul,hans.vandierendonck,koen.debosschere}@elis.ugent.be

ABSTRACT

While the chip multiprocessor (CMP) has quickly become the predominant processor architecture, its continuing success largely depends on the parallelizability of complex programs. In the early 1990s great successes were obtained to extract parallelism from the inner loops of scientific computations. General-purpose programs, however, stayed out-of-reach due to the complexity of their control flow and data dependences. In this paper we present a tool to extract coarse-grain parallelism in the outer program loops, even in general-purpose programs, and helps the programmer to parallelize it. This coarse-grain parallelism can be exploited efficiently on multi-cores without additional hardware support.

1. INTRODUCTION

In the past Moore's law and microarchitectural improvements enabled an exponential increase in single-thread performance whereas nowadays this is no longer the case. Monolithic processors are exchanged for more power-efficient multi-cores which are able to run an increasing number of threads. Now Moore's law implies a doubling of threads every two years. Consequently in order to continuously increase the performance of a single application it will have to be parallelized to greater and greater extent.

A lot of research has gone into finding loop-level parallelism operating on array data structures. Significant advances have been made for scientific computation, where inner loops exhibit large loop bodies, high loop trip counts and predictable data dependences. In contrast, this work focusses on extracting DO-ACROSS parallelism [1] in the outermost toplevel loops of general-purpose programs, which corresponds closely to pipeline-like operations on a data set [5]. To achieve this we developed a tool based on dynamic dependence analysis that will help the programmer in finding coarse-grain parallelism even in programs with complex data structures and control flow. A related approach, where coarse-grain parallelism is dynamically exploited after the programmer has indicated the parallelism, is presented by Thies et al. [6].

2. TOOL OVERVIEW

In order to model the data flow we first use the symbol table of the program to determine the different static data structures. Dynamic objects are identified during program execution by monitoring calls to memory allocation routines (such as malloc). Using dominator analysis we extract the three types of *code regions* for our next step: functions, loops and snippets. Snippets are code fragments that contain memory references, but do not contain loops or function calls. Thus, snippets are potentially data dependent on other code fragments.

The next step is to dynamically map the data and control flow

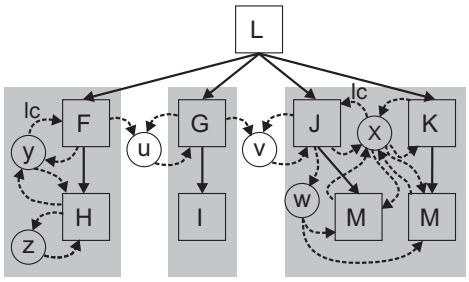

Figure 1: Joint control data dependence graph for loop L calling functions $F-M$. The functions access data structures $u-z$.

of the program. The motivation to use a dynamic analysis over a static one is that the latter must make conservative assumptions, whereby many false data dependences are reported. These false dependences obscure the parallelism in a program. Therefore control flow and data dependences are measured at the assembly code level during a profiling run of the program. A risk of using a dynamic analysis is that not all the potential dependences show up during the profiling phase. However, we observed that coarse-grain dependences between large code regions (e.g. functions) are stable across different inputs: a function typically uses a data structure or it does not use it. In contrast, fine-grain data dependencies between individual statements are much more sensitive to branch behavior and data values.

During the profiling data dependences are monitored when a code region (*consumer*) is reading a memory location that was last written by a code region (*producer*). Dependences through registers are not tracked. The control dependences are tracked in order to respect the sequential semantics when parallelizing a sequential program. Hereto, we structure the program as a tree where each node in the tree corresponds to a previously defined code region. In case a code region has multiple children, the children are sorted in sequential execution order. The actual control flow corresponds to tracing a path through the tree, where control flow can move down in the tree (entering a sub-region of the current region) or it can move up in the tree (exit a region).

After the profiling the tool will analyze the loops of the program for DO-ACROSS parallelism. For this purpose it uses two graph representations of the control and data dependences. The first one is the *joint control data dependence graph* (JCDDG) (Figure 1). This graph contains rectangular nodes that represent the code regions and elliptic nodes that represent the data structures. Code regions are linked by edges according to control flow (solid lines). Data flow edges (dotted lines) indicate which functions access a

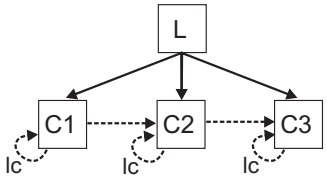

Figure 2: Code template that contains pipeline-like behavior in DO-ACROSS loop

data structure. The data flow edges point from a function to a data structure when the function writes the data structure. The data flow edge points from the data structure to a function if that function reads the data structure. Data flow edges that are marked with lc are loop-carried data dependences.

The data dependences are also represented in a second graph: an *inter-procedural data flow graph* (IDFG). The nodes in this graph represent code regions and an edge from a node A to a node B indicates that code region B is consuming data produced by code region A. Note that the IDFG contains information that is not deducible from the JCDDG. If a data structure has multiple producers, one cannot deduce from the JCDDG which producer a consumer is reading data from.

The analysis itself consists of two steps and tries to find loops that have a pattern as shown in Figure 2. In the first step we cluster code regions based on control dependences.We recurse through each of the child nodes and mark every node recursively as a member of the corresponding cluster. If a node is reached that is already assigned to a cluster, then the clusters are merged. The second step is to cluster code regions by data dependences. Hereto we start by mapping the clustering of the JCDDG on the IDFG. The goal of this step is to remove data dependent loops between clusters by merging the involved clusters. The algorithm to perform the clustering by data dependences uses topological sorting of clusters and detection of connected components to simultaneously sort the clusters by data dependences and merge mutually dependent clusters. The result of the final clustering is depicted by the grey background in Figure 1, resulting in the targeted code template of Figure 2.

The last step is for the programmer. In its current form the tool will not do the parallelization automatically, but provide the programmer with the necessary information to ease the task. This information consists of synchronization points in the code and which data structures are involved in the synchronization. Depending on the occurrence of loop carried dependences, the parallelization is a form of loop unfolding [2,4], software pipelining [3] or a combination of them.

3. EVALUATION

Figure 3 shows for several benchmarks the outer parallel loops that were detected. Each part of the bars correspond to the execution time of the sequential part of the program or one of the parallelizable loops (loop n). We show for each loop its original execution time (S: sequential version) and its estimated reduced execution time (P: parallel version). For DO-ACROSS loops $t_{par} = t_{1iter} + (n-1)t_{lcmax}$, where n is the number of iterations and t_{lcmax} the execution time of the most time-consuming cluster with loop-carried dependencies.

The benchmarks on the left-side contain little DO-ACROSS parallelism in their loops or have a very unbalanced pipeline (one stage responsible for the largest part of the loop). The benchmarks on the right-hand side contain more coarse-grain parallelism and are more suitable for parallelization. On an 8-core Sun UltraSPARC T1 processor we have obtained speedup results of factor 5 to 12 for that

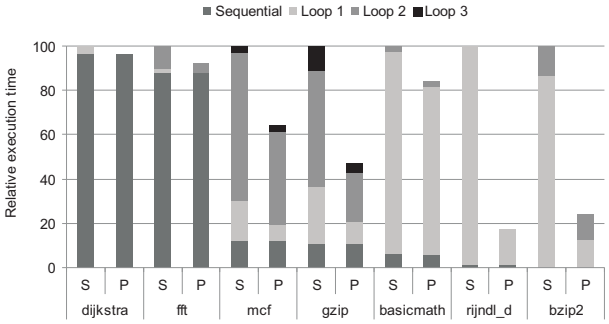

Figure 3: Relative execution time of detected pipelines in sequential execution (S) and in parallel execution (P)

group of programs [5].

4. CONCLUSION

We presented a dynamic analysis tool for extracting coarse-grain parallelism from a sequential program using profiled information for obtaining precise control and data dependence information. To achieve this it looks for DO-ACROSS parallelism which can be easily exploited on a multi-core processor. This tool helps the programmer to create parallel programs in a more efficient way.

Acknowledgments

Sean Rul is supported by a grant from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Hans Vandierendonck is a post-doctoral research fellow with the Fund for Scientific Research-Flanders (FWO). This research is also funded by Ghent University and HiPEAC.

5. REFERENCES

[1] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Int'l Conf. Parallel Processing (ICPP)*, pages 836–845, 1986.

[2] S. Ha and H. Kim. Ku-loop scheme: An efficient loop unfolding scheme for multithreaded computation. *Journal of Information Science and Engineering*, 14(1):223–236, 1998.

[3] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *MICRO 14: Proceedings of the 14th annual workshop on Microprogramming*, pages 183–198, Piscataway, NJ, USA, 1981. IEEE Press.

[4] E. H. Rho, S. H. Ha, S. Y. Han, H. H. Kim, and D. J. Hwang. Compilation of a functional language for the multithreaded architecture: Davrid. In *Int'l Conf. Parallel Processing (ICPP)*, pages 239–242, 1994.

[5] S. Rul, H. Vandierendonck, and K. De Bosschere. Detecting the existence of coarse-grain parallelism in general-purpose programs. In *Proceedings of the First Workshop on Programmability Issues for Multi-Core Computers, MULTIPROG-1*, page 12, 1 2008.

[6] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–369, 2007.