Function Level Parallelism Lead by Data Dependencies

Sean Rul*, Hans Vandierendonck*,
Koen De Bosschere*,[1]

* ELIS, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium

ABSTRACT

Due to the rise of Chip multiprocessors (CMP's) the amount of parallel computing power has increased significantly. This is in contrast with the fact that a lot of programs are sequential and cannot exploit these parallel resources, urging the need of developing new techniques to extract parallelism from sequential programs. For this we present a new profile based technique. It works in a non-speculative way, based on data dependencies between functions and finds large chunks of code to parallelize. To achieve this, we introduce the so called interprocedural data flow graph and the data sharing graph. To test our technique we used the bzip2 program from the SPEC-cpu2000 benchmark suite. Our mechanism could significantly speed up the compression part (3.74 times), with a global speedup of 2.45 on a quad processor system.

KEYWORDS: Interprocedural data flow graph, Data sharing graph, Parallelizing

1 Introduction

Creating parallel programs by hand is an intricate and time consuming job. Nevertheless if we want to utilize all the computing power on the oncoming processors, we need parallel programs. Although there is a group of programs, such as scientific and media applications, that inherently have a lot of easy exploitable parallelism, another majority of programs are inherently sequential. In this abstract we explore how we can parallelize these sequential programs by detecting data dependencies between functions. We form two graph representations that from an abstraction of the profiled dependencies. These will, together with the call graph, help in detecting the chunks of code that can be parallelized. This data-driven approach sets our method apart from previous profiling techniques for parallelizing programs which were mainly control-driven [Wolf91] and/or speculative [Stef98].

[1]E-mail: {sean.rul, hans.vandierendonck, koen.debosschere}@elis.UGent.be
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 researcher of the Fund for Scientific Research-Flanders (FWO). This research is also funded by Ghent University and HiPEAC.

2 Method

Call graph We keep track of how many times the function is called, the number of different functions it calls and the fraction of execution time it consumes. The latter one will be taken into account for balancing the work between different threads. In Figure 1 we give an example of a call graph. These caller/callee relations form a first restriction on program parallelism, since the caller passes arguments and the callee may give a return value back.

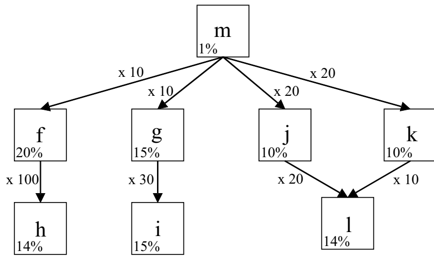

Figure 1: Example of call graph

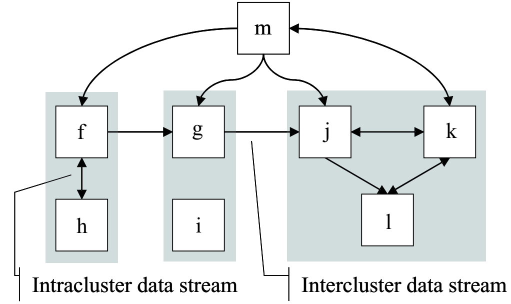

Figure 2: Interprocedural data flow graph

Interprocedural data Flow graph Data dependencies are measured during a profiling phase. During this phase all loads and stores are registered with their corresponding memory address. With this information we can determine which function reads data from other functions. This can be represented in a directed graph where the nodes are the functions and the edges show the data streams (Figure 2). We use the notation $f \xrightarrow{x} g$ to indicate that a function g consumes x bytes of data produced by f. The next step is to search for strongly interconnected functions. That is, functions that share a large amount of data. This will lead to clustering of functions sharing common data structures with each other. The clustering will divide the data streams in two categories: *intercluster data streams* and *intracluster data streams*. In Figure 2 the clustering of functions is shown with grey rectangles. To indicate that a function f belongs to group N we use the notation f_N. Consequently an intercluster data stream will be noted as $f_N \rightarrow g_M$, while an intracluster data stream will have the form $f_N \rightarrow g_N$.

One of the aims of this clustering is to impede the amount of bidirectional intercluster data streams. If data streams between two groups of functions go in both directions, it will be hard to parallelize them further on. Unidirectional intercluster data streams form less of a problem, since these are suitable for certain parallel constructs. Same for bidirectional intracluster data streams, since the functions in that cluster will be executed sequentially.

The clustering of functions is also guided by the call graph, since this graph imposes a certain hierarchy between different functions. For example in the call graph of Figure 1 we see that function h is only called by function f, so it may be a good choice to put these two functions in the same cluster. On the other hand the fraction of execution time of cluster should be taken into account, in consideration of finding a balanced solution. So the second role of the call graph is to find clusters that are balanced in execution time.

Data sharing graph While the previous representation showed the existing data streams between functions, it does not show *how* the data is shared. This requires another abstraction

of the profiled information. The idea is to show both the data dependencies as well as the involved data structures. The resulting graph will have two kinds of nodes: function nodes and data nodes. An edge from a function node to a data node indicates the number of write accesses made by the function. We use the notation $f \xrightarrow{w} ds_1$. The opposite, an edge from a data node to a function node, should be seen as the number of read accesses from that function to the data structure, which will be noted as $ds_1 \xrightarrow{r} f$.

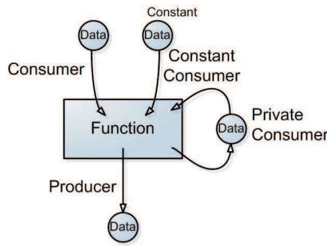

Figure 3: Classification of data dependencies.

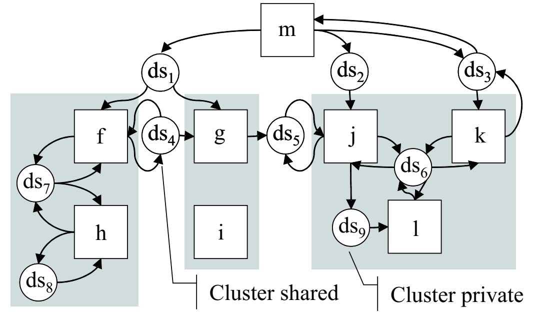

Figure 4: Example of data sharing graph.

If we examine one function and all the data structures it accesses, we can distinguish 4 types of data usage, which are represented in Figure 3. *Producer* (writes data, read by other functions), *Consumer* (reads data, written by other functions), *Constant Consumer* (reads data, without traceable origin) and *Private Consumer* (reads data, self-written). If we do this classification for each function, we will get a graph, the *data sharing graph* (Figure 4), which shows how the data is shared between different functions. Rectangular nodes represent functions, while elliptic nodes are data structures.

If we map the clustering of functions, obtained with the interprocedural data flow graph, we detect which data structures become private within a cluster and which ones are shared between different clusters, respectively called *cluster private* and *cluster shared*. This will prove to be useful when the actual parallelization is performed.

Parallel constructs A pipeline construction, where each pipeline stage produces data for the next and consumes data from the previous one, can easily be detected in our interprocedural data flow graph. The first requirement is that between several clusters of functions the intercluster data streams are unidirectional. Also there are no dependencies from f_m to g_n with $m < n$. This last requirement should be interpreted as a function from group m that in sequential executions comes before all the functions of group n, is not dependent from a previous execution of a function from group n. The shared data between the cluster represents the pipeline registers. Depending on the partitioning of work, we can consider two cases. The first is the *heterogeneous pipeline* in which each stage of the pipeline handles a different group of functions. The second is called the *homogeneous pipeline* where each stage executes the same code. This is similar to the master-slave paradigm, but the synchronization is different.

Parallelization We should detect the data structures that can be private for the different threads. Given the fact that sharing involves communication, which on its turn entails sequentiality, we want to minimize the amount of shared data. This information can be obtained from the data sharing graph. The necessary initialization and startup code for the

threads should be generated in order to preserve a correct execution of the program. As well as some code to complete the work after the threads are finished. New data structures will be introduced to allow passing on shared data between different threads, again the data sharing graph will help in this task. When there exists only one version of a shared data structure, this data has to be locked when it has to be altered. Another possibility is to create multiple instances of the shared data structure, so the data itself needs no locking. In case of a pipeline construct, the pipeline registers (the data that is shared between two pipeline stages) can be duplicated. Each pipeline stage works on its own pipeline register and passes it on to the next stage when ready. To enforce the correct execution the necessary synchronization needs to be added.

3 Results

We profile the program with a modified version of Dynamic SimpleScalar. The program we used as a test case was *bzip2* from the SPEC2000 benchmark suite.The parallelized version of the program was tested on a platform with four Itanium processors.

The analysis of bzip2 revealed that the compression part could be split in 4 parts that could be transformed into a pipeline. The decompression part could only be split in 2 parts.

Figure 5 shows the speedups for both the compression and decompression part, as well as the total speedup. The heterogeneous version has a total speedup of 1.99, while the homogeneous version can run 2.45 times faster than the original version.

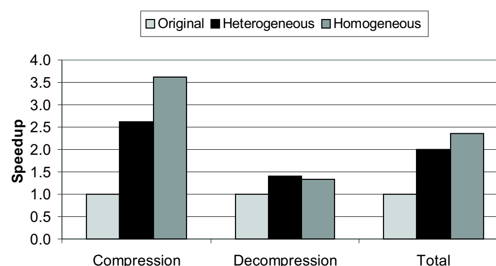

Figure 5: Speedup results of parallelized *bzip2* using 4 threads for compression and 2 for decompression

4 Conclusion

We presented a framework for extracting parallelism from sequential programs, by measuring all data dependencies between functions during a profiling phase. Then we used an interprocedural data flow graph to extract a possible clustering of functions. This in collaboration with the call graph that shows the hierarchy between functions. We also introduced the data sharing graph which reveals the data affinity between functions. These three representations form a reliable base for finding possible parallelism.

References

[Stef98] J. STEFFAN AND T. MOWRY. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[Wolf91] M. WOLF AND M. LAM. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, 1991.