Classifying Data Dependencies Between Functions

Sean Rul*,[1], Hans Vandierendonck*,[2],
Koen De Bosschere*

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

ABSTRACT

Due to memory dependencies between functions, it is difficult to parallelize a program. In this paper we propose a profiling technique that characterizes the data behavior of functions to minimize the memory dependencies by identifying functions that are operating on the same data.

KEYWORDS: Data dependencies, profiling, parallelizing

1 Introduction

In the recent past the increase of sequential speed (i.e. clock speed) has slowed down, while the parallel throughput keeps increasing. The rise of the multi-core reflects this trend. Unfortunately humans – consequently also programmers – tend to reason sequential, leaving us with programs which cannot utilize all this parallel power. Although writing parallel programs is not impossible, it is prone to bugs which are hard to detect. If we want to spare the software developers from this daunting task and we do not want to change our hardware, it will be up to the compiler to parallelize the program.

While the final goal is to setup a framework that assists the compiler into parallelizing a program (perhaps speculatively), we will focus in this paper on a data-driven profiling technique that provides us with information on how different functions are related to each other. We will only consider dependencies through memory, since registers can be predicted [Tull99] or precomputated [Coll01]. In a following step this information can be used to parallelize a program. Previous profiling techniques [Marc02][Quin05] for parallelizing were mainly control-driven. This technique can also be applied on a Cell processor [Flac06] to divide a single program over several processing units with their own local storage.

[1]E-mail: sean.rul@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).
[2]E-mail: hans.vandierendonck@elis.UGent.be
Hans Vandierendonck is a post-doctoral researcher of the Fund for Scientific Research-Flanders (FWO).

2 Methodology

Figure 1 shows the different ways in which a function can use data. *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). This can be done for each function, leading to a graph where some nodes of data produced by one function, will be consumed by one or more other functions.

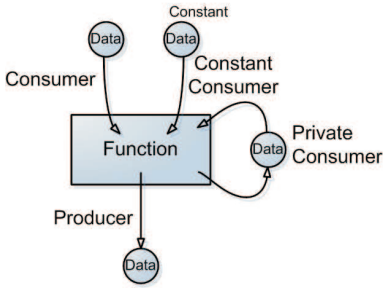


Figure 1: Classification of data behavior.

During the profiling we register all loads and stores, together with the function that issued the instruction. For loads the producer of the loaded value is also recorded. This information allows us to find all the producer-consumer relations between the functions.

Memory instructions that are related to stack operations are ignored, because these are used for passing arguments and local data structures. When we have a read operation of which there is no previous producer, we will assume that it is constant. This situation occurs when for example a program reads data from a constant data section. The same behavior tends to happen with system calls.

Profiling was done with a modified Dynamic SimpleScalar [Huan03], a simulator for the PowerPC architecture. We used the SPEC CPU2000 benchmarks for evaluation.

3 Evaluation

We will mainly discuss the results of the benchmark *bzip2* with reference input *program*, because of its simplicity, keeping the results surveyable in this evaluation.

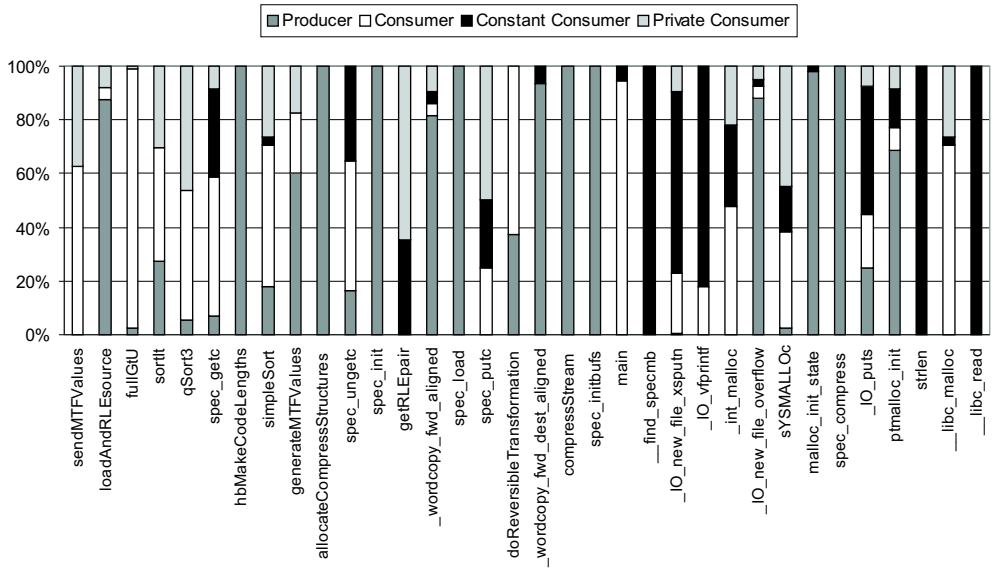
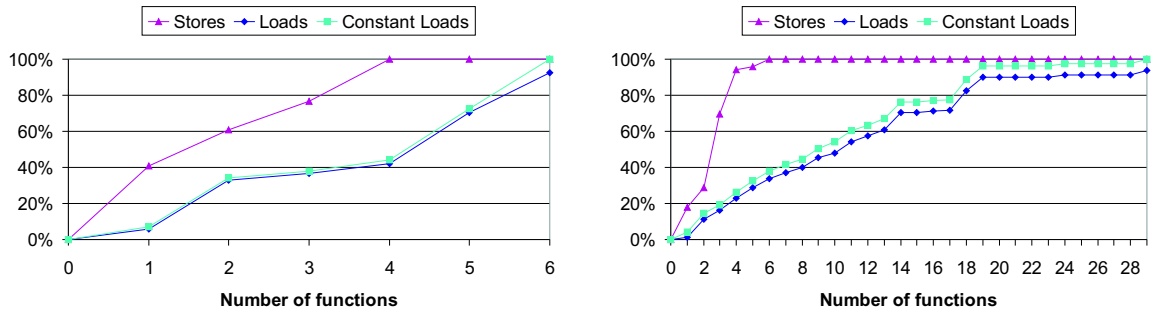


Figure 2: Classification of data behavior of top-ranked functions of *bzip2*.

In a first step we want to classify the behavior of the functions according to the classification introduced in Section 2. We show the functions with the most data dependencies in Figure 2. Only a small fraction of these functions shows homogeneous behavior. In most cases pure producers are initialization functions, while pure constant consumers, such as *strlen* and *__libc_read* are library functions. The majority of the other functions contain a mix of producer, consumer and private consumer data dependencies.

Figure 3 gives the fraction of addresses that are written or read by a number of functions indicated on the X-axis. For load instructions we make a distinction between loads that have a known producer and constant loads that have no preceding store operation, and are considered constant. Figure 3(a) shows that for *bzip2* 7.5% of all loads are constant loads. About 60% of all addresses are written by at most 2 different functions. This indicates that only a few functions are responsible for producing each data structure. In this case any data is written by at most 4 functions. Also data is read by at most 6 functions. Figure 3(b) shows for *crafty* a similar distribution for stores (at most 6), but its data is read by much more functions (50% by more than 10 functions). This points to data structures shared by many functions.



(a) *Bzip2* with *program* reference input

(b) *Crafty* with reference input

Figure 3: Cumulative distribution of accesses by different functions per address

In a following step we build up a data dependency graph (Figure 4), with as nodes the functions and as edges the dependencies between two functions. A node like *f_22* is not connected to any other nodes, meaning that this node reads no data produced by other functions. Communication with this function must happen through arguments on the stack and registers. There are also two separate clusters of functions that do not interact with each other. In the left cluster we have marked a subcluster of three functions. This subcluster is responsible for 30% of the execution time of the whole cluster. The large number of memory dependencies between the functions in the subcluster compared to the limited outgoing dependencies with the other subcluster, makes it possible to pipeline between those two subclusters.

4 Conclusions and Future Work

In this paper we proposed a profiling technique that characterizes the data behavior of functions. Apart from a few functions, most functions did not show a homogeneous behavior.

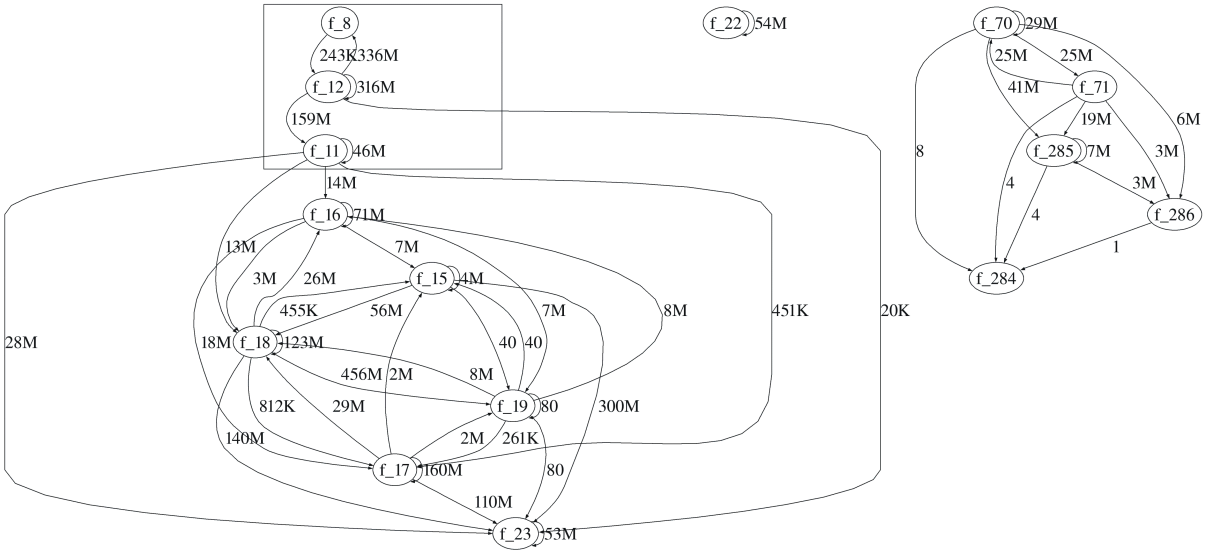


Figure 4: Partial data dependency graph for *bzip2*.

The evaluation shows that functions and data can in principle be clustered with minimal memory dependencies.

Future work consists of developing an algorithm to determine clusters of functions operating on common data, based on the introduced classification.

References

[Coll01] J. COLLINS, H. WANG, D. TULLSEN, C. HUGHES, Y. LEE, D. LAVERY, AND J. SHEN. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25, 2001.

[Flac06] FLACHS, B. AND ASANO, S. AND DHONG, S.H. AND HOFSTEE, H.P. AND GERVAIS, G. AND ROY KIM AND LE, T. AND PEICHUN LIU AND LEENSTRA, J. AND LIBERTY, J. AND MICHAEL, B. AND HWA-JOON OH AND MUELLER, S.M. AND TAKAHASHI, O. AND HATAKEYAMA, A. AND WATANABE, Y. AND YANO, N. AND BROKENSHIRE, D.A. AND PEYRAVIAN, M. AND VANDUNG TO AND IWATA, E.. The microarchitecture of the synergistic processor for a cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):63–70, jan 2006.

[Huan03] X. HUANG, J. MOSS, K. MCKINLEY, S. BLACKBURN, AND D. BURGER. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR-03-03, University of Texas at Austin, Februari 2003.

[Marc02] P. MARCUELLO AND A. GONZÁLEZ. Thread-Spawning Schemes for Speculative Multithreading. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 55–64, 2002.

[Quin05] C. QUINONES, C. MADRILES, J. SÁNCHEZ, P. MARCUELLO, A. GONZÁLEZ, AND D. TULLSEN. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, June 2005.

[Tull99] D. TULLSEN AND J. SENG. Storageless Value Prediction Using Prior Register Values. In *ISCA'99: Proceedings of the 26th International Symposium on Computer Architecture,*, pages 270–279, 1999.