Detection of Function-level Parallelism

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

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

ABSTRACT

While the chip multiprocessor (CMP) has quickly become the predominant processor architecture, its continuing success largely depends on the parallelizability of complex programs. We present a framework that is able to extract coarse-grain function-level parallelism that can exploit the parallel resources of the CMP.

The framework uses a profile-driven control and data dependence analysis between large code regions. We target coarse-grain parallelism by finding do-across parallelism in the outer-loops of a program. This parallelism can be exploited in a pipelined fashion. The identification of parallelism reduces the overall loop structure to a preset template by merging inter-dependent code regions. The actual parallelization is guided by the template.

The extracted parallelism results in a significant speedup of factor 5 to 12 on an 8-core Sun UltraSPARC T1 processor.

KEYWORDS: Function-level parallelism, Joint control and data dependence graph

1 Introduction

In the recent past the mainstream processor was one big monolithic processor core. Due to power and performance issues, however, the industry moved to chips filled with multiple simpler cores, the chip multiprocessors (CMP). In order to tap the full processing power of a CMP one needs parallel programs. Since rewriting or developing programs with threads and explicit synchronization is an intricate and time consuming job, a more attractive course is automatic parallelization of sequential code. A lot of research has gone into finding loop-level parallelism operating on array data structures. In contrast, this work focusses on finding parallelism in general-purpose programs which feature complex control-flow and complex data structures.

---

[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 Framework

This section passes through the different stages of our framework [RVB06] that results in finding function-level parallelism.

Defining code regions As we are interested in coarse-grain parallelism, we record dependences only between large regions of code. We consider three types of code regions: *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 different code regions in a program are determined by a static analysis and passed on to the profiler in the next stage.

Profiling dependences We opted for a dynamic dependence analysis allowing a more precise analysis of the dependences compared to a static analysis which is more conservative. If some dependences in the sequential code do not occur during the profiled execution, the parallelized version will neglect them. Without extra precautions, such as manual verification or thread-level speculation support, the detected parallelism is merely *optimistic*.

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 regions has multiple children, the children are sorted in sequential 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).

Data dependences indicate which code region (consumer) is reading a memory location that was last written by a code region (producer). For each data object we record a *memory dependence matrix* which scales dynamically based on the number of producers and consumers for the data object in question. In a matrix the entry at row f and column g records the number of times that code region g consumed a value produced by code region f in that data object. Statically allocated objects are retrieved from the symbol table of the program, while dynamically allocated objects are identified during program execution by monitoring calls to memory allocation routines (`malloc`, `calloc`, `realloc`, `alloca` and `free`). To know the code region that was the last one to produce a value at a particular memory address we use the *last producer table*. Note that the producers are tracked at the smallest writable quantity (i.e. bytes) in order to correctly identify dependences. This is necessary since each field of a data object can have a different producer.

Analyzing dependences Control and data dependences between code regions are recorded in a *joint control data dependence graph* (JCDDG) (Figure 1). This graph contains rectangular nodes which represents the code regions and elliptic nodes which 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 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 an *inter-procedural data flow graph* (IDFG) (Figure 2). The nodes represent code regions and an edge from F to G indicates that code

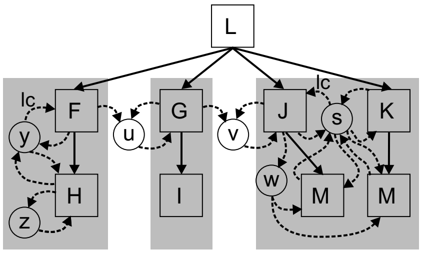

Figure 1: Joint control data dependence graph

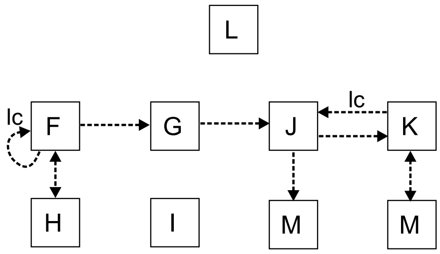

Figure 2: Inter-procedural data flow graph

region G is consuming data produced by code region F. Note that the IDFG contains information that is irreducible from the JCDDG. Based on the JCDDG one could presume that the node J is data dependent on node M through data structure s. The IDFG, however, shows no data stream directed from code region M to code region J.

The format of parallelism we are looking for is depicted in Figure 3. A loop contains multiple code regions that are data dependent on each other, but has no interdependent clusters. If there are loop-carried dependences, then they point from a code region to itself. To detect this code template every loop in the JCDDG containing function calls is analyzed. We match the code template to every loop in the program. Initially, every code region is a separate cluster, so it is unlikely that the template matches, due to control flow or data dependences. To improve this match, we iteratively merge clusters until the template matches. Each cluster is treated as an indivisible unit of work, either because of control dependences or because of data dependences in that cluster.

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 3.

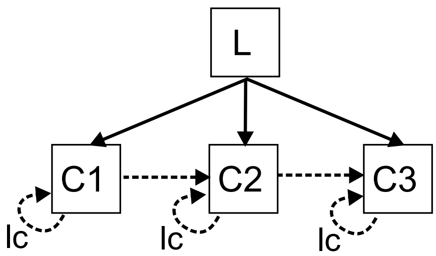

Figure 3: Code template

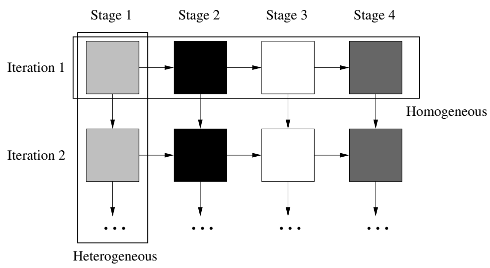

Figure 4: Parallelization of code template

Parallelization The parallelism in a loop of the format of Figure 3 becomes apparent when unrolling the loop (Figure 4). Here each row represents a different iteration, each column represents a cluster of code regions and the edges between clusters are dependences. The program is parallelized by executing instances of the clusters in different threads. A first parallelization consists of a vertical division of dependences. We call this *heterogeneous* parallelization since each thread is running different code. Another possibility is a horizontal division, where one thread handles a complete iteration, so called *homogeneous* parallelization. A third possibility, *composite* parallelization, is only feasible when only the first and last stage have loop-carried dependences. In that case the there is one fetch thread (first stage), several worker threads and one write thread (last stage). The data objects involved in the parallelization can be deduced from the JCDDG.

3 Evaluation

We profile the programs with a modified version of Dynamic SimpleScalar using a small or training input set. For testing the parallelized programs we use a large or reference input set. Speedup results are obtained on a Sun UltraSPARC T1 processor with 8 cores which is able to execute 32 threads in parallel.

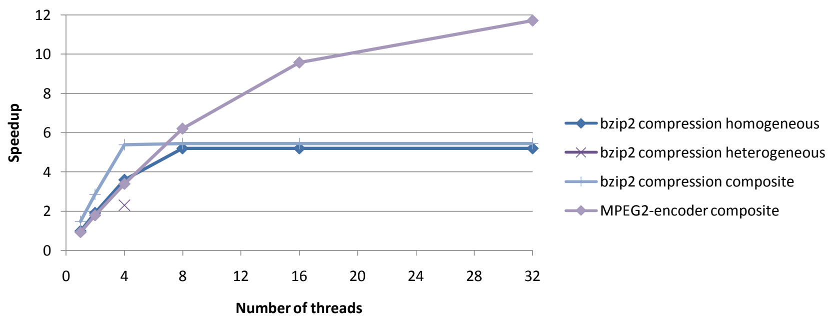

Figure 5: Speedup results for parallelized programs in function of used threads

Figure 5 shows some speedup results for a MPEG2-encoder and the compression part of bzip2. In the case of bzip2 we used the different parallelization schemes. When heterogeneous parallelization is used, the number of threads is determined by the number of clusters. For the composite parallelization the number of worker-threads is represented on the X-axis.

4 Conclusion

We presented a framework for extracting function-level parallelism from a sequential program using dynamic analysis techniques for obtaining precise control and data dependence information. Interdependent code regions in loops are recursively merged until the overall loop structure matches a preset template which can be parallelized.

References

[RVB06] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Function level parallelism lead by data dependencies. In *dasCMP: Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, 2006.