Aggregated Dynamic Dataflow Graph Generation and Visualization

István Szabó, Gábor Wacha, János Lazányi
Department of Measurement and Information Systems,
Budapest University of Technology and Economics,
Budapest, Hungary
sz.pista@gmail.com, {wacha, lazanyi}@mit.bme.hu

***Abstract*—Aggregated Dynamic Dataflow Graphs can assist programmers to uncover the main data paths of a given algorithm. This information can be useful when scaling a single-threaded program into a multi-core architecture. The amount of data movements is crucial when targeting for cache incoherent and/or heterogeneous platforms. This paper presents two methods for generating function-level *Aggregated Dynamic Dataflow Graphs.* Instruction level trace log was used as a basis, which was generated by Microsoft Giano processor simulator platform. Top-down aggregation strategy and relational database was used to speed up the generation of different views of the aggregated dataflow and call graphs.**

Keywords—Dynamic Dataflow Graph, Function level aggregation, multi-core, program slicing

I. INTRODUCTION

In some of the algorithmic intensive embedded real time streaming applications, a single CPU is not sufficient to handle the task real time, but the processing power of 2-10 cores would be theoretically enough. Algorithms are usually implemented in C/C++ languages which natively do not support parallel implementation and algorithm slicing 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 CPUs, while internal communication bandwidth is often a bottleneck.[1]

This paper describes efficient methods for generating *Aggregated Dynamic Dataflow Graphs (**ADDFG**)* to record internal data flow, and to visualize critical data paths. Our future goal is to develop a method to assist efficient program slicing and mapping in multi-core environment.

II. RELATED WORK

A *call graph* [2] is a graph that represents calling relationships between subroutines in a program, which can be recorded by standard profiling tools. The call graph is a directed graph, where each node represents a procedure and each edge indicates a function call. Static call graphs can be generated based on static code analysis of the application without executing it first. The benefit of the static call graph is that it contains all possible direct function calls, but the drawbacks are that it shows no information about the program execution and calls made through function pointers. *Dynamic call graphs* contain only the trace of a single execution of the software, with given input arguments.

*Dynamic Dataflow Graphs **(DDFG)*** can be used to visualize the data flow of a given algorithm. DDFGs are directed acyclic graphs where nodes represents instructions and edges represents data flow. Figure 1. shows a theoretical DDFG of a four tap FIR filter calculating single output. Both *ADD* and *MUL* are arithmetic instructions, input edges are operands while output edges are results.

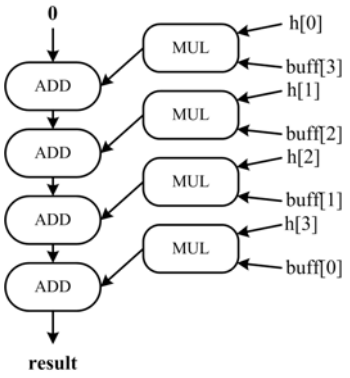

Fig. 1. Theroretical DFG of a four tap FIR filter.

DDFGs can be used during development very efficiently, some examples are:

- *Debugging.* Using standard debuggers, breakpoints can be set on certain data or instruction. After a breakpoint occurs, the programmer faces the problem, but does not know its root cause. Using DDFGs the program can be traced back because the DDFG stores all previous calculations. This way the root cause of the problem can be determined easily.
- *Critical path, data flow visualization*. As it can be seen later, ADDFGs can be used to visualize main data flow of the given algorithm. This can assist the programmer to have a better understanding of the code being analyzed; moreover we believe that it can be used for efficient program slicing in a multi-core environment.

In order to generate DDFGs, code execution must be traced at instruction level. A hardware trace module can track

the code execution. Alternatively Just in Time compilation techniques or processor simulators can be used.

Valgrind [3] is a debug framework for memory debugging, memory leak detection and profiling. Valgrind uses just-in-time compilation techniques, including Dynamic Recompilation and *Dynamic Binary Instrumentation and Analysis* (DBA). Valgrind framework translates each instruction of the code being analyzed into an intermediate representation, instruments it with the selected DBA tool, translates back to the machine code of the host system and executes it.

Many different DBA tools are available. *Cachegrind* [4] is an efficient tool to generate dynamic call graph, to track code execution and cache usage at instruction level, while *Redux* [5][6] can generate the dynamic dataflow graphs.

III. Dynamic Dataflow Graph generation using processor simulator

Processor simulators can be used to track all executed instructions of the CPU. We have modified the Microsoft Giano[7] processor simulator tool. Giano was selected, because it supports multiple processor architectures, multi-core configuration, hardware-software co-simulation and also because it is available in source code.

Figure 2. depicts our implemented analysis tool flow:

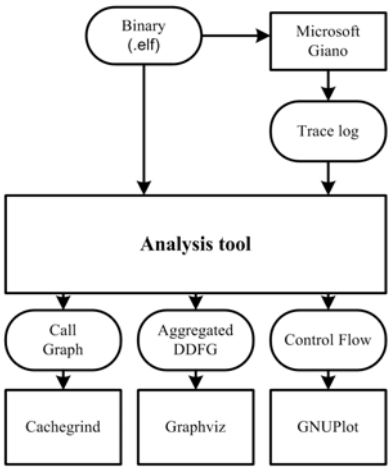

Fig. 2. Analysis tool flow for DDFG generation

- C source was compiled with general GCC compiler. (For ARM7 architecture)
- The binary was executed in the Giano processor simulator system. Giano was modified to create a full trace log of the program execution. Each line of the log contains timestamp, processor opcode and disassembled instruction, the value of all processor registers, and a flag which denote if the register was written and/or read in the current instruction.
- The developed analysis tool takes the trace log and the original binary file to resolve the function names from the debug symbols. The analysis tool generates three different outputs as follows:
- Dynamic call graph is being generated using the Cachegrind file format.
- Memory profiling results are visualized with GNUPlot
- ADDFGs are visualized using Graphviz. Henceforth this paper focuses this output.

IV. Data aggregation using bottom-up strategy

DDFGs visualize data dependency among instructions at a very fine grained level. While traditional call graphs can represent only argument passing, using DDFGs data dependency can be revealed independent from data passing method, e.g. using global variables or pointers. This fine grained analysis is very helpful in some cases, but usually a real-word program results in unmanageable size of DDFGs, with hundred-thousand nodes and edges, therefore some aggregation is needed to reduce graph complexity.

Data aggregation is performed by grouping the DDFG's nodes into a hierarchical structure, and summing the corresponding edges using bottom-up strategy. Figure 3. shows the implemented hierarchy levels. DDFG atomic nodes are representing the lowest hierarchical level. Each of these nodes corresponds to a single assembly instruction, represented by their memory address. Each assembly instruction pertains to a C function, and finally C language files consist of functions.

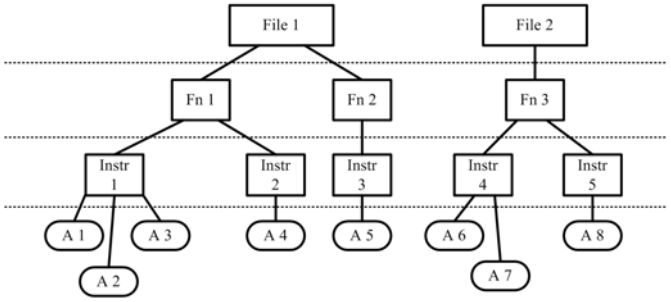

Fig. 3. Aggregation levels of the dataflow graph.

Graph edges are merged in a similar manner, using hierarchical structures. Figure 4. shows edge aggregation method between two hierarchical levels.

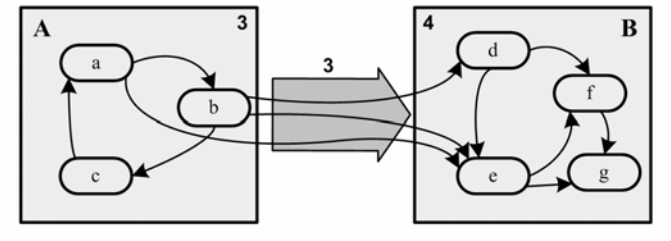

Fig. 4. Edge Aggregation of the dataflow graph.

At the higher hierarchical level *A* and *B* nodes are connected using single *E* edge. Both *A* and *B* nodes consist of multiple lower hierarchical level sub-nodes and some internal edges. Internal edges which connect sub-nodes inside a single node do not appear at higher level representation. Multiple lower hierarchical level edges are merged into a single *E* edge at the higher hierarchical level. Node weight represents total runtime (i.e. instruction count), while aggregated edge weight corresponds to total data being transferred.

Visualized aggregated data graph can be generated at any hierarchical level using depth-first search algorithm of the aggregated graph, where the result of the algorithm (e.g. the output of the program) is selected as a root node. Graphviz program was used to visualize the generated ADDFG. In order to thin-out the output image – to visualize only the critical functions and corresponding data paths – aggregated edges below a selected threshold level, and the corresponding nodes are eliminated.

V. IMPLEMENTATION RESULTS

A real-world application, JPEG decompression was selected to test the aggregated data graph generation, and to visualize the main data paths. The Independent Jpeg Group JPEG reference decompression code was primarily developed for PC platform. The code had to be modified to accommodate the embedded environment by eliminating dynamic memory management and file I/O functions. To lessen the amount of data, the decompression was done on a 64x64 pixel image.

The 64x64 pixel JPEG decompression took 680000 processor clock cycles to compute. The atomic level DDFG had 540000 nodes and 840000 edges in between. It is impossible to visualize such graph. Figure 5. depicts the result of the function level aggregation of the DDFG, with 114 nodes and 409 edges.

Figure 6 is generated automatically from Figure 5. by the elimination of edges which perform less than 2% of the total data movement. This graph can be used to uncover the main data paths of a given algorithm. By following the thick edges, all JPEG specific main functions appear in a logical order. These are: file input, MCU decoding, IDCT, color up-sampling, YCbCr to RGB conversion and file output.

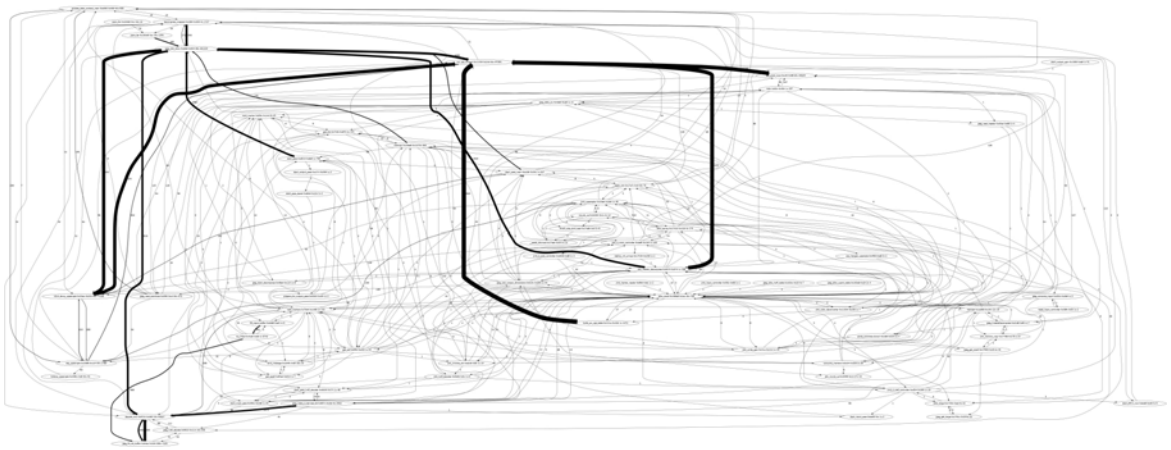

Fig. 5. Function level ADDFG

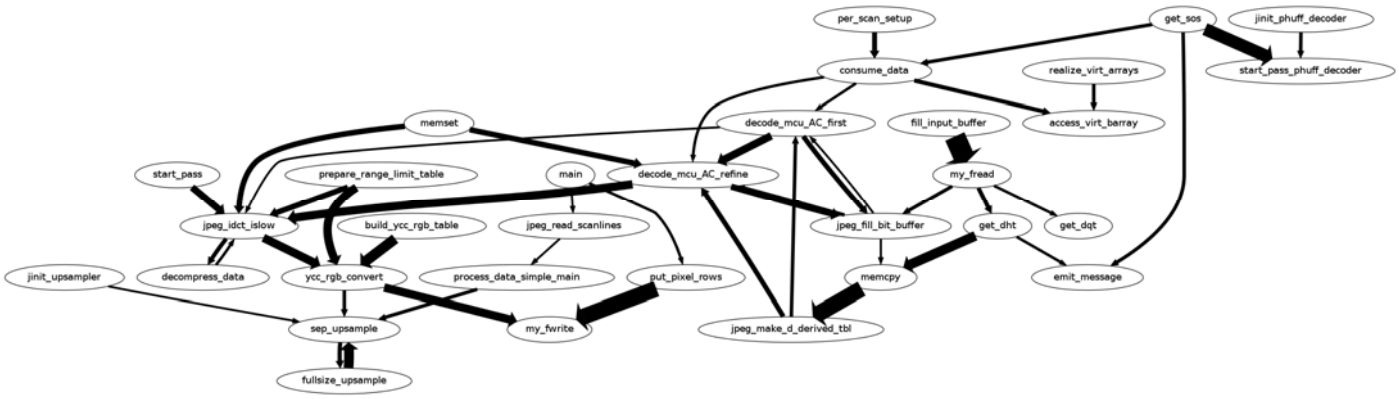

Fig. 6. Function level ADDFG with edges and nodes thinned out.

VI. ADDFG USING A TOP-DOWN STRATEGY

A. The top-down strategy

To generate an ADDFG, the bottom-up strategy can be used, which enables to generate a view of the software's data flow by first generating the instruction level graph. However - as earlier mentioned- with an application of a larger code base this method soon leads to unmanageable amounts of low level graph data, which will be omitted anyway after the aggregation process.

It is preferable to skip the low level graph generation, and to generate the function level aggregated dataflow graph with a top-down method. The top-down strategy collects every high level, inter-function communication (e.g. arguments and return value), without generating the low level, intra-function data flow (e.g. variable assignments), thus decreasing the amount of memory and processing time needed for the dataflow graph generation.

The bottom-up generation of the ADDFG needs sequential data access by parsing each processor instruction. In contrast, the top-down strategy needs a random data access only of values with specific requirements, thus storing the dump information in a relational database is preferable.

The top-down strategy also enables to easily generate different views of the data flow, since SQL queries can be used to access the data.

B. Generating DDFG with top-down strategy

The construction of the ADDFG using the earlier, bottom-up generation strategy analyses each processor instruction line parsed from the trace log sequentially to examine the data flow *every time* a graph is generated.

In contrary, the top-down strategy first searches for every function entry and exit point in the program flow (loaded from the debug symbol information stored in the executable), thus partitioning the run time of the program to intervals. As seen in Figure 7, for each interval a function name and call instance (i.e. the number of the call of this function) is assigned, so the interval stored as function call fragments. If no subroutine call happens from a start clock cycle count to an end clock cycle count every executed processor instruction is part of the specific function. This sequence of intervals hereinafter will be called the function *control flow*.

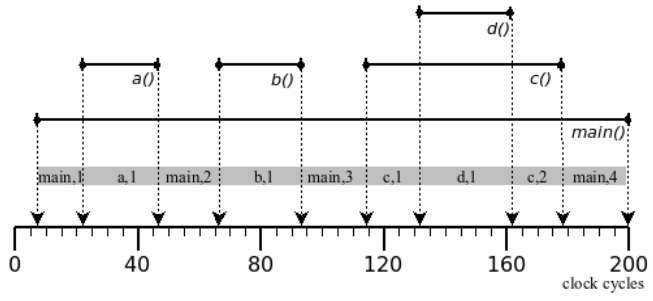

Fig. 7. The control flow

The *control flow* enables to generate the function level ADDFG without parsing each executed processor instruction. To build the DDFG, edges should be added. After the generation of the *control flow*, the algorithm examines each interval, searching for all inter-function communication. When a data flow is detected between the intervals of two functions, a graph edge is added to the DDFG.

C. Organization of the relational database

The database tables are organized into three groups as seen in Figure 8. The first group can be considered as *low level* information imported from a Giano trace log and the examined user elf file. These low level database tables contain all information necessary for DDFG generation (with both the bottom-up and the top-down strategy).

Data access table stores every data access (register and memory data transfer at a given address). *Opcode lines* table contain each executed processor instruction and the processor cycle count the instruction executed in. *Function symbol* table is loaded from the debug information stored in the executable under examination, containing the name of the function and the address of the function entry point.

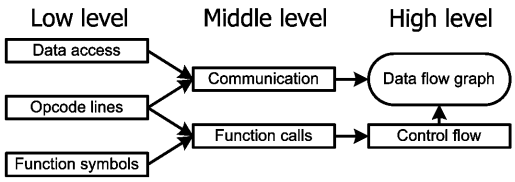

Fig. 8. The three levels of the database

For accelerating the generation of different DDFG views, *middle and high level* tables are introduced. These tables are created from the low level tables using database queries. The middle and high level tables are organized so that the function-level ADDFG view generation does not need to access low level tables.

On the *middle level Communications* table holds every memory and register communication pair. (A write instruction to a data transfer cell paired by all of the subsequent reads before the next write instruction.) *Function calls* table holds every function call done during the execution of the software (the entry and exit point of an instance of a function call).

The *high level* database table holds the *Control flow* information. In a first pass for every executed function an interval is created with the call and return clock cycle count, then later these intervals are merged, and finally stored into the database, thus allowing classifying every clock cycle under a fragment of a function call instance.

D. Different views of the ADDFG

Using the top-down strategy and the relational database storage which enables to query the data flow information, function level ADDFG generation is faster compared to the bottom-up strategy, moreover allowing to combine the call graph and data flow information and also to create different views of the same graph.

Figure 9. shows an example merged call and dataflow graph for function *B*. The solid edges shows that *B* was called by function *A*, then B calls *D* and *E*. The dashed edges show data transfer. Since function *C* does not directly call *B* the edge means communication through pointers or global memory.

With the *control flow* information, the function level aggregation can be extended to function instances, treating the different calls to a function as a separate entity. Figure 10. shows that the current instance of function *B* receives data from *D* and sends data to *E*. Another instance of the function can receive data also from *A* and *C* and can send data also to *F*.

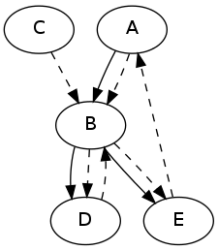

Fig. 9. Data and call edges of function B

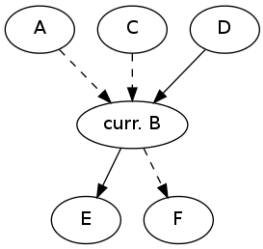

Fig. 10. Potential and current data edges of the instance of function B

The usage of a relational database as storage also allows to quickly map all data interchanges between two given functions, searching for a communication for which the data write happens in the first function and the data read is in the second function.

With the low level database tables, the bottom-up strategy can be used to trace the intra-function dataflow.

VII. Results

The generation time of an ADDFG with both the top-down and bottom-up strategy was measured using the JPEG decompression algoritm. The bottom-up strategy finished in 3 minutes, whereas the top-down strategy in 1.5 minutes. The other advantage of the top-down strategy is lower memory consumption. To generate the ADDFG, the bottom-up strategy's peak memory usage was 3 GBytes of system memory, which is used for storing the low level graph nodes. On the other hand, the top-down strategy used only 6 Mbytes of memory, thus allowing to handle larger graphs.

VIII. Conclusion and future work

In this paper we presented processor simulator based methods for generating Dynamic Dataflow Graphs. Function-level aggregation was applied to help visualizing critical data paths. By using a top-down aggregation strategy and relational database different views can be generated very easily.

In the future we will introduce *loop-level* into the aggregation hierarchy levels (between instruction and function levels) and detect *control dependencies* to efficiently identify and visualize data cycles.

References

[1] Kim, Martha A., and Stephen A. Edwards. "Computation vs. memory systems: pinning down accelerator bottlenecks." *Computer Architecture*. Springer Berlin Heidelberg, 2012.

[2] Graham, Susan L., Peter B. Kessler, and Marshall K. Mckusick. "Gprof: A call graph execution profiler." ACM Sigplan Notices 17.6 (1982): 120-126.

[3] Nethercote, Nicholas, and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." ACM Sigplan Notices 42.6 (2007): 89-100.

[4] Weidendorfer, Josef, Markus Kowarschik, and Carsten Trinitis. "A tool suite for simulation based analysis of memory access behavior." Computational Science-ICCS 2004. Springer Berlin Heidelberg, 2004. 440-447.

[5] Nethercote, Nicholas, and Alan Mycroft. "Redux: A dynamic dataflow tracer." Electronic Notes in Theoretical Computer Science 89.2 (2003): 149-170.

[6] Nethercote, Nicholas, and Julian Seward. "How to shadow every byte of memory used by a program." *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 2007.

[7] Forin, Alessandro, Behnam Neekzad, and Nathaniel L. Lynch. "Giano: The two-headed system simulator." Microsoft Research, Redmond (2006).

[8] Rul, Sean, Hans Vandierendonck, and Koen De Bosschere. "Function level parallelism driven by data dependencies." *ACM SIGARCH Computer Architecture News* 35.1 (2007): 55-62.