PinComm: Characterizing Intra-Application Communication for the Many-Core Era

Wim Heirman,*,[1] Dirk Stroobandt*

* *ELIS Department, Ghent University, Belgium*

ABSTRACT

While the number of cores in both general purpose chip-multiprocessors (CMPs) and embedded Multi-Processor Systems-on-Chip (MPSoCs) keeps rising, on-chip communication becomes more and more important. In order to write efficient programs for these architectures, it is therefore necessary to have a good idea of the communication behavior of an application. We present a communication profiler that extracts this behavior from compiled, sequential C/C++ programs, and constructs a dynamic data-flow graph at the level of major functional blocks. It can also be used to view differences between program phases, such as different video frames, which allows both input- and phase-specific optimizations to be made. Finally, PinComm can visualize inter-thread communication in parallel programs, which can help in optimizing communication behavior and spotting communication-related performance bottlenecks.

KEYWORDS: Profiling, dynamic data-flow graph, network-on-chip, multi-core, communication

1 Introduction

Due to the recent gap between Moore's law and single-threaded processor performance, multi- and manycore chips are becoming the name of the game. And while the number of on-chip cores rises, the effects of the communication between these cores rapidly gain in importance from both power and performance points of view.

To keep the effects of on-chip communication in check, first of all the parallelization of an application needs to be done in such a way that communication between network nodes is minimized. Secondly, once the (remaining) communication pattern is known, all nodes must be mapped onto a topology. Most topologies are non-uniform, and benefit when communication can be made *nearest-neighbor only* which avoids slow, inefficient long-distance

[1]E-mail: wim.heirman@elis.UGent.be

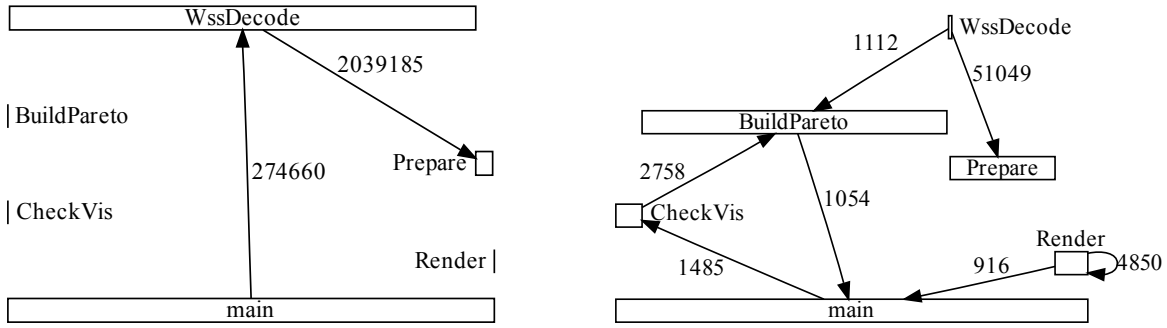

Figure 1: Communication graph between the major code regions of the WSS application. Region lengths (as node widths, proportional to the region's dynamic instruction count) and large inter-region communication flows (marked on each edge, in bytes) are shown, for frames #1 (left) and #2 (right).

signaling. Finally, once the previous optimizations have been done and the application's communication pattern is known, an extra design step can consist of designing a suitable on-chip network, or setting the configuration parameters for an existing NoC.

The first step in doing all this is of course knowing what communication to expect. While static analysis can be used for some applications, it is infeasible in a large fraction of important existing and emerging applications, which often have an irregular structure or a behavior that heavily depends on the input or other external influences. Dynamic methods of characterization and optimization are therefore needed. Yet, an approach must still be largely automatic to give the additional benefit that both characterization and optimization can be tailored to specific scenarios (classes of input sets) or even specific program phases.

In this work, we introduce the *PinComm* profiler, which allows such an automatic measurement of a program's communication patterns. Since it is a runtime profiler, it can be connected to any program (running on a host PC), with any combination of inputs and parameters. It allows a designer to visualize communication inside both sequential and parallel programs. This new insight can be used to make more efficient parallelizations, aid in designing on-chip networks, or serve as input to communication-aware run-time scheduling.

2 PinComm: A communication profiler tool

Our profiler constructs a dynamic data-flow graph (DDFG), this is the communication that flows between parts of the program. These parts can be static functions, dynamic function calls, threads, or specific data structures; each will be represented by a node in the DDFG.

PinComm is based on *Pin*, a dynamic instrumentation tool [L$^+$05]. Pin allows modular instrumentation of executables on several platforms (including IA-32, x86-64 and Xscale) through the use of plug-ins. Our profiler is such a Pin plug-in, which instructs it to intercept all memory accesses and all function calls. For function calls and returns, we keep the call stack and output a call trace, which will later be processed into a call tree. An identifier of the currently executing function is also kept (one for each thread in parallel applications). When memory writes are intercepted, the function ID of the current function (and the thread number, if applicable) is stored in a *last-written-by* table which keeps, for each memory address, the producer of this piece of data. Each time a read instruction is encountered, we can look

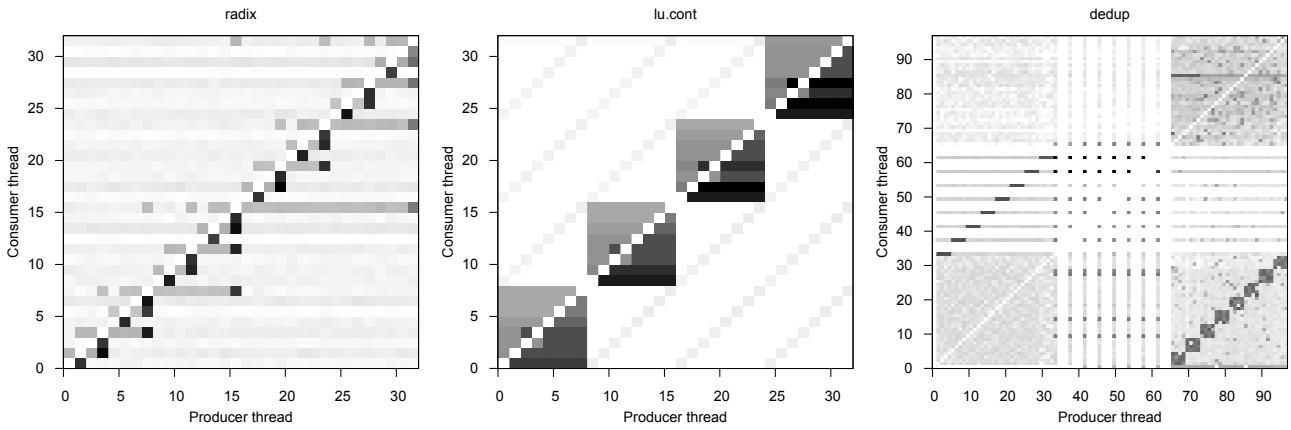

Figure 2: Spatial communication behavior: inter-thread communication intensity for a selection of SPLASH-2 and PARSEC benchmark applications.

up the producer for the data word at the referenced address in the *last-written-by* table. We now know that communication has occurred between two functions, the consumer being the current function and the producer being the function found in the *last-written-by* table. The size of the communication stream is given by the size of the memory read instruction.

PinComm can automatically extract function names from linker and debug information in the executable, and make call graphs based on this information. For complex programs, however, there are usually too many functions, which clutters the graph and makes a visual analysis impossible. To this end, markers can be inserted in the source. These are recognized by the profiler and can signify the start and end points of *code regions*. Each of these regions can be named and will appear on the graph as a single node. This way, a program can be split up in functional blocks, and PinComm shows communication between these blocks, rather than between individual functions.

Also, markers can also be used to start and stop the measurement at some point during the application. This allows one to select a part of the application to be measured, rather than the complete program which may include uninteresting parts such as initialization routines. Also, for streaming applications, frame or iteration boundaries can be marked, allowing intra- and inter-frame communication to be visualized separately. Figure 1 shows the behavior of the WSS application, a 3-D rendering application. The two subgraphs each show the communication behavior (and relative function call durations) pertaining to the rendering of two different video frames.

Alternatively, the communication graph can be clustered by thread, which allows inter-thread communication to be analyzed inside parallel programs. Figure 2 shows the spatial communication distribution between each thread pair, for some of the applications from the SPLASH-2 and PARSEC suites. In Figure 3, we plot the temporal communication behavior of data produced by a given thread. Note that the communication is obtained here in an architecture-independent way (i.e., assuming perfect caching of private data), so only communication inherent to the application is shown. This is in contrast architectural simulation, which measures explicit communication and is affected by a given architecture's cache sizes, coherency protocols and communication infrastructure.

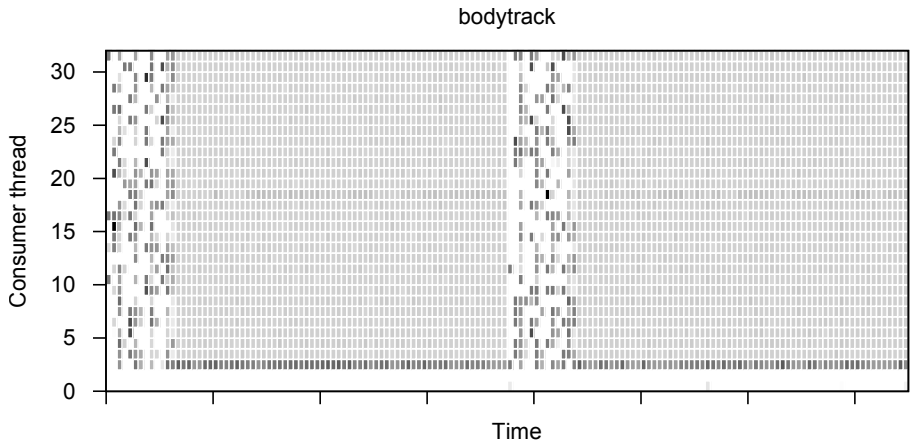

Figure 3: Temporal communication behavior: time and place of consumption of data produced by thread 1 for the PARSEC `bodytrack` benchmark.

3 Conclusions

On-chip communication is gaining importance in both CMP and MPSoC settings. To visualize communication inside programs, even before they are parallelized and mapped onto a specific multicore architecture, we developed PinComm. This is a communication profiler which measures dynamic data-flow graphs (DDFGs) for both sequential and parallel binary programs, and can present results in a way that is meaningful for the developer: communication can be viewed between major code regions or between threads – the latter way allowing for validation of a proposed parallel implementation. Using this new source of knowledge, novel applications become possible such as communication-aware parallelization, mapping, and configuration of on-chip network resources.

4 Acknowledgments

This research is supported by the "Optimization of MP-SoC Middleware for Event-driven Applications" (OptiMMA) project, grant 060831 of the Agency for Innovation by Science and Technology (IWT-Vlaanderen).

References

[L+05] C.-K. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI 2005*, pages 190–200, June 2005.

[H+09] W. Heirman et al. PinComm: A communication profiler to optimize embedded resource usage. In *Proceedings of ProRISC*, November 2009.
`http://www.elis.ugent.be/~wheirman/pincomm/`