2010 16th International Conference on Parallel and Distributed Systems

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

Wim Heirman, Dirk Stroobandt
Ghent University, ELIS
Sint-Pietersnieuwstraat 41
9000 Gent, Belgium
Email: wim.heirman@ugent.be

Narasinga Rao Miniskar, Roel Wuyts, Francky Catthoor
IMEC
Kapeldreef 75
3001 Leuven, Belgium
Email: miniskar@imec.be

***Abstract*—As the number of cores in both embedded Multi-Processor Systems-on-Chip and general purpose processors 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 or parallel C/C++ programs, and constructs a dynamic data-flow graph at the level of major functional blocks. In contrast to existing methods of measuring inter-program communication, our tool automatically generates the program's data-flow graph and is less demanding for the developer. 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. We will also describe briefly how this information can subsequently be used to guide the effort of parallelizing the application, to co-design the software, memory hierarchy and communication hardware, and to provide new sources of communication-related runtime optimizations.**

***Keywords*-Profiling, dynamic dataflow graph, network-on-chip, communication**

I. Introduction

Due to the recent gap between Moore's law and single-threaded processor performance, multi- and many-core chips are becoming the name of the game. In the embedded domain, traditional instruction set processors are combined with hardware accelerators, large blocks of memory and interfaces to the external world and integrated into a self-contained System-on-a-Chip. This new level of integration brings with it the challenge of designing both the hardware and the software for these complex systems. In this paper, we will focus on the interconnection network. This architectural aspect has long remained hidden inside large servers and supercomputers, but it is now prevalent and must be accounted for during the design of even what used to be small, embedded systems. At the same time, communication busses are being replaced with more scalable topologies such as networks-on-chip. The cost of this scalability is non-uniformity, which makes the communication patterns have a much larger influence on performance.

To solve the communication problem, a systems designer will want to do two things. First of all, when parallelizing the application, computation should be laid out such that communication between network nodes is minimized. Secondly, once the (remaining) communication pattern is known, all computational nodes must be mapped onto a network topology. Clearly, when heavily communicating entities can be mapped onto the same network node, this communication stream will no longer be visible on the network, it will rather be carried by a much more efficient mechanism such as through a processor core's registers or local cache or scratch-pad memory. Between minimizing communication and mapping lies the concept of *shaping* communication, for instance in making it *nearest-neighbor only* which avoids slow, inefficient long-distance signaling. Finally, once the previous optimizations have been done and the application's (network-visible) communication pattern is fixed, a suitable on-chip network can be designed.

While static analysis can be used for some applications, it is infeasible for a large fraction of important existing and emerging applications. These programs have an irregular structure, have a behavior that heavily depends on the input data or on other external influences, or are dynamically composed out of multiple smaller application components sharing the same on-chip resources. Inter-processor communication turns out to be especially dependent on these influences, and can often not be predicted through composition of the traffic streams caused by the various components. 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 easily be tailored towards multiple specific scenarios, which can consist of a (class of) input set(s), a combination of program components, or even specific program phases.

In this work, we introduce the *PinComm* profiler, which allows 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

1521-9097/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPADS.2010.56

visualize communication inside both sequential and parallel programs. Being developed as part of the OptiMMA project, our end goal is to allow efficient runtime management of all system resources. In this project, the communication profile measured by PinComm is used as input to a runtime scheduler such that it can make mapping and scheduling decisions in a communication-aware way. Other uses of this valuable information lie in parallelizing applications (while minimizing communication between threads), in mapping the application's parallel components while optimally matching communication patterns and network topology, or in driving other communication-aware runtime optimizations.

II. PinComm: A communication profiler tool

A. Constructing the dynamic data-flow graph

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

PinComm is based on *Pin*, a dynamic instrumentation tool [1]. 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 together with the memory address that the write instruction referenced. Now, when a read instruction is encountered, we can look up the address in the *last-written-by* table and determine the producer of this piece of data. 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.

For an exact measurement, the last-written-by table should contain an entry identifying the last writer for each memory address, at a one byte granularity. If one such entry is 64 bits large, PinComm would have a memory overhead of $8\times$ the memory used by the application being profiled. While this setting provides the most accuracy, it is usually not needed since most memory accesses are aligned at four or eight byte multiples. Moreover, when one is only interested in the major communication streams, some accuracy can be traded off for a significantly reduced memory overhead by increasing the memory address granularity. This setting is configurable at PinComm's command line. When increasing this setting, less memory is required to keep the last-written-by table, but accuracy decreases when data consumptions can sometimes no longer be attributed to the correct producer. This results in a shift in communication streams, and can also cause an increase in perceived communication, similar to *false sharing* when only part of the memory block (or cache line) actually contains shared data. Measurements showing the effect of this setting will be shown in Section IV.

B. Clustering the data-flow graph

The result of the profiling phase consists of a dynamic call tree (one for each thread) and a communication graph that shows communication streams between all dynamic function calls. Function names are extracted from linker information in the executable, additionally, debug information can be used to find the source file and line number for each function. 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.[1] 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.

When using PinComm on parallel programs, the communication graph can also show communication between threads. After clustering the results according to the processor each thread will run on, the (inherent) communication that can be expected on the on-chip network can be derived. This is done in an architecture-independent way (i.e., assuming perfect caching of private data). Simulation of a realistic cache subsystem can be a next step, using existing tools such as CPM$im [2], which is also based on the Pin instrumentation tool.

Besides the thread number, the approximate point in time at which the memory read or write operation was executed can also be used as an identifier. PinComm counts the number of instructions executed in each thread, and allows this instruction count – with a configurable granularity – to be used to split up the source and destination nodes in the communication graph. This allows one to explore the temporal communication behavior of an application. An example of this will be given in Figure 6 in Section IV.

Finally, markers can also be used to start and stop the measurement at some point during the application. This way, one can select a part of the application to be measured, rather than the complete program which may include uninteresting parts such as initialization. For a streaming application, frame or iteration boundaries can be marked, allowing intra- and inter-frame communication to be visualized separately. An example of this will be given in Section III-D.

[1] These are a specific type of NOP instruction, `xchg bx, bx`, they therefore do not interfere with native execution of the same program binary.

C. Communication through memory regions

As a first order approximation, the dynamic data-flow graph, when clustering all nodes according to which processor they will run on, equals the communication that will be visible on the on-chip network. This assumes that all memory accesses *internal* to a processor (between functions that were mapped to the same processor, or memory writes that are only read by the same function) can always be handled inside the network node this processor is located on. In effect, this implies the assumption of a perfect cache, or a scratch-pad memory large enough to hold each thread's private and shared-owned data.

For small data structures this approximation usually results in a communication profile that is accurate enough. However, large data structures are often allocated in shared memory blocks which have their own network node, or in off-chip memory. In this case, additional network traffic flows will exist between the processor's network node and the node containing the memory block or the off-chip memory interface. To this end, we added the option of marking specific data types or `malloc()` calls with an object type identifier. For each of these object types, a separate node in the DDFG will be added. Edges to and from this node now represent writes to and reads from objects of this type – or traffic to and from a specific memory block or interface. This allows one to estimate the resulting network traffic to this node, and also get a quantitative measure of its required memory bandwidth – on which the type of memory, banking/interleaving and other design parameters can be decided.

III. Case study: 3-D wavelet decoder

A. Application

Our demonstration application is a Wavelet Subdivision of Surfaces (WSS) algorithm, which is a scalable, multi-resolution 3-D decoder and is described in detail in [3]. By progressively decoding higher wavelet frequencies, it can decode objects with an adaptive quality level, depending on the complexity of the input and on external requirements such as the distance of the object to the viewer, required video resolution or available processing power. Clearly, this application is in itself highly adaptive in its resource requirements, both through the input set (the complexity of the rendered scene), input events by the user (which change the viewpoint or the environment) and environmental influences on the target platform (such as concurrently running processes, battery or thermal constraints, etc.).

B. Input to 3D-WSS and scenarios

In this experiment, the input to the 3D-WSS application consists of a gaming environment with three rooms, each containing 13, 17 or 22 objects, and four different camera positions per room. The combination of room number and camera position results in 12 scenarios. In each of these scenarios, the player moves around during a number of image frames. Our profiler measures the communication flow incurred by the wavelet decoding during the rendering of each frame. This experiment was done using a sequential version of the 3D-WSS application, we measured communication between each of the major functional blocks. This should allow us to accurately predict the communication requirements of a future parallel implementation, where these functional blocks will be distributed over different processors and executed in a pipelined fashion.

C. Functional decomposition

We marked the major code regions in the WSS source code, according to a functional decomposition given by the program's author in [3]. We also placed markers at the start of each new frame. While PinComm can just as easily work on unmarked applications, marking code regions of interest already removes uninteresting functions from the communication graphs. This graph is shown in Figure 1, for both the first and second frames. The region's width and horizontal position denote their starting point and length, measured in running instruction counts. Each arrow denotes a major communication stream (containing at least 1% of the total inter-region communication for the frame) of data produced by the origin region (in this frame or a previous frame), and consumed by the target region (in this frame). In the first frame (Figure 1, top), a first major stream totaling some 275 kB originates from `main`[2] and runs to `WssDecode` which reads the object data from memory, performs the Wavelet decoding, and writes decoded vertex data back to memory. In the second communication stream, totaling over 2 MB of data, the decoded vertices (which are clearly much bigger than the original data fed into `WssDecode`) are used by the `Prepare` code region.

Figure 1 (bottom) shows the results for the second frame. In this frame most of the objects have already been decoded by `WssDecode` and the results are cached. The computation time of `WssDecode` is therefore significantly shorter than in the first frame. Note that Figure 1 shows relative durations, the absolute instruction count drops from 8.3M instructions for the first frame to only 180k for the second one. Several other, lower-intensity communication streams now become visible. Also note that, in Figure 1 (bottom), the communication from `WssDecode` to `BuildPareto` is part of a feedback algorithm that crosses frame boundaries – obviously `BuildPareto` in frame two cannot read from frame two's `WssDecode` region since it is executed at a later stage in the render pipeline. Likewise, the arrow from `Render` to itself denotes data generated by the `Render` region which is reused across iterations.

[2]This region contains, in addition to the actual C `main()` function, all code not explicitly assigned to other regions. This includes the function where object data is read from file, which accounts for most of the communication visible originating from the `main` region.

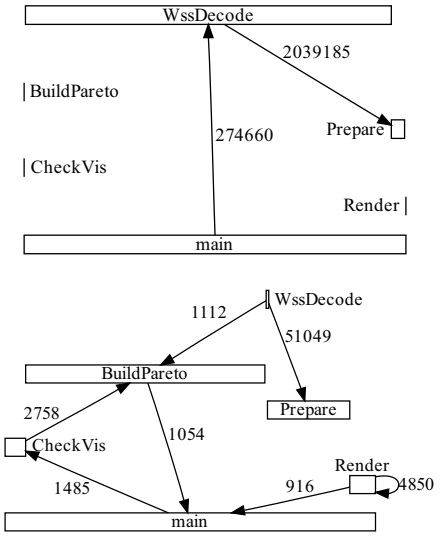

Figure 1. Communication graph after marking of the major code regions `CheckVis[ibility]`, `BuildPareto`, `WssDecode`, `Prepare[Render]` and `Render`. Region lengths (as node widths, proportional to the region's dynamic instruction count) and large inter-region communication flows (marked on each edge, in bytes) are shown, for frames #1 (top) and #2 (bottom).

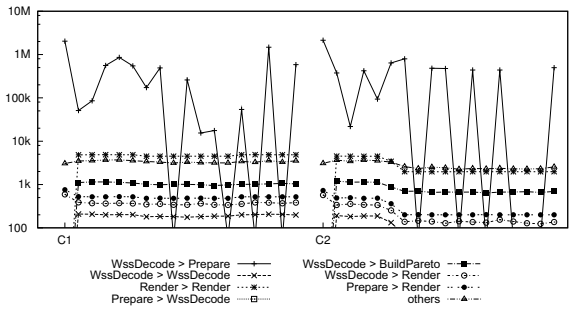

Figure 2. Inter-regional communication magnitudes, for camera positions C1 and C2, all frames and a 13-object scene

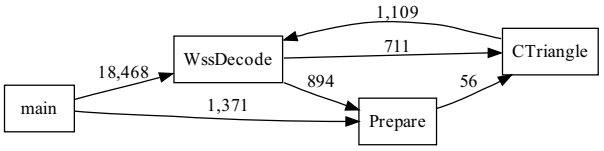

Figure 3. Communication using shared memory and `CTriangle` classes, in addition to the known streams from Figure 1, for C2, 13 objects, frame #1.

D. Communication patterns

We can now, for the major communication streams in WSS (those visible in Figure 1), determine their behavior in different iterations of the program. We already noticed from Figure 1 that both the per-region runtimes and the communication magnitudes are very different in the first frame than they are in subsequent frames. The main difference is the length of the `WssDecode` function. In the first frame, all visible objects have to be decoded, which takes a significant amount of time. These decoded objects are stored in local memory; in subsequent frames only newly-visible objects need to be decoded.

Figure 2 shows the magnitudes for each of the major inter-regional communication flows, and its evolution throughout the program, for a 13-object scene, all frames and two of the camera viewpoints (C1 and C2). By far the largest inter-regional communication stream runs from `WssDecode` into `Prepare`. The graph shows that the `WssDecode` function in the first frame for each camera position generates the bulk of the data (the decoded object data) to be used during the rest of the program. Since this single communication stream by far outweighs all other inter-regional streams, we can conclude that, when optimizing the on-chip communication behavior of the WSS application, the placement of the `WssDecode` and `Prepare` functionality will be the most important parameters. Clearly, both should – if possible from a computational load point of view – be placed on the same processor. Yet, both are also the longest executing functional regions. Therefore, separate cores will almost certainly be used to implement both functions, which makes inter-core communication unavoidable. The architecture should thus be dimensioned in such a way that it can accommodate the `WssDecode`-to-`Prepare` communication stream.

E. Communication through shared memory

Since the communication from `WssDecode` to `Prepare` is not only intense, but involves a relatively large and long-lived data structure, it is conceivable that this data structure will be stored in a separate memory block or in off-chip memory – and thus incur extra on-chip network traffic. To quantify the magnitude of this extra traffic we marked the `CTriangle` class, which is the object type communicated from `WssDecode` to `Prepare`, for inclusion as a separate node in the graph. Figure 3 shows the major communication flows (in kB) between all participating nodes (the code regions without large communication streams have been omitted from this graph for clarity). In addition to the streams known from Figure 2, we now see that `WssDecode` does indeed store a significant amount of local data in `CTriangle` objects (amounting to around 32 MB in total). The `WssDecode` to `Prepare` communication stream visible in Figure 2 will thus incur *two* data flows on the on-chip network, one from `WssDecode` to the external memory controller and one back into the node executing the `Prepare` function. The magnitude of both these communication streams are again provided by our profiler's results, and can be read from Figure 3.

IV. APPLICATIONS

The data gathered by PinComm can be used mainly in two ways: when partitioning the application into threads, and when mapping the threads onto processors. Both can be done both on- and offline, although usually partitioning will be done at design time (implementations with a variable number of threads can move some of the decisions to the runtime scheduler), while mapping and scheduling are more often done at runtime to provide adaptation to different hardware platforms, changing workloads, etc.

A. Communication-aware parallelization

The communication graph can be used to aid in parallelization of an application. The runtime length annotated call tree clearly shows which functions require the most execution time, and are therefore candidates for parallelization – either by assigning (groups of) functions to separate processors in a functional parallelization (such as pipelining), or by parallelizing loops in one or more of the longer functions.

The added value of our profiler comes at the point when, in this otherwise standard way of parallelization, there is a choice in how functions are clustered onto processors. Traditionally, the only metric here is to keep the workload of all processors the same, so that load imbalance and its associated synchronization cost is minimized. But this clustering problem usually has several solutions with similar cost. By considering communication between functions, as visible in our communication graph, a more general cost function can be constructed that also accounts for the estimated delay caused by interprocessor communication. By finding a clustering solution with minimal cost, on-chip network bandwidth and its associated power usage can be minimized, while performance is increased through the avoidance of communication latency. This technique can be visualized on the communication graph (see Figure 4): communication arrows cut by the partitioning (solid lines) cause inter-processor network traffic, whereas communication internal to a cluster (dashed arrows) can be handled by processor-local caching. Using this more detailed cost metric, one can often find that solutions that looked similar from a purely load-balance point of view will perform very differently due to their differing communication loads, and that in some cases the introduction of a significant load-*im*balance can actually improve performance.

Note that our profiler does not necessarily see *all* data dependences. Smaller communication streams are removed from the communication graph but they can still result in data or control dependences which may prohibit parallelization. Moreover, since the graph is constructed based on profiling information, it only contains the communication present during the execution of the input set(s) used. No guarantees can be made for other inputs, which may induce new communication streams and thus more dependences.

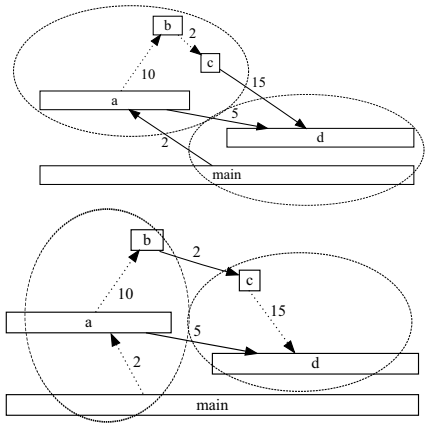

Figure 4. Two possible partitionings of functions `a`, `b`, `c` and `d` onto two processors using a pipelined parallelization model. Communication arrows cut by the partitioning (solid lines) cause inter-processor network traffic, arrows internal to a cluster (dashed lines) denote communication that can be handled by processor-local caching. In the topmost graph, the communication cost (sum of all cut (solid) arrows) is 22. The alternative partitioning of the bottom graph, while resulting in a slight load imbalance, has a communication cost of only 7.

The programmer performing parallelization should therefore still prove that all dependences are honored.

B. Communication-aware scheduling

Task Concurrency Management [4] is an effective methodology for run-time management of embedded resources. This methodology uses *scenarios*, which are clusters of run-time behaviors based on the system cost metrics on the target architecture. For each scenario, an optimal mapping is found at design-time, while at run-time the specific mapping for the current scenario is used. This way, a large effort can be made at design-time to reduce the resource and energy requirements, while being adaptive to the specific run-time circumstances with a very low run-time overhead. The mapping here refers to the scheduling (in time) and assignment (in space) of resources to the tasks.

When a methodology such as TCM is applied to ever larger multi-core architectures, knowledge of on-chip communication is needed to provide additional optimizations. At the design-time phase, the explored TCM mappings can be further optimized to minimize the communication flows through a communication-aware mapping which places highly communicating threads on the same processor, or on a pair of processors with a fast on-chip network connection between them. At run-time, the selected mapping configuration consists of the configuration of the network: just as the processors' voltages, frequencies and cache sizes are set dynamically depending on the scenario, an on-chip network can be configured for optimal (minimizing energy, etc.) support of the expected communication flows.

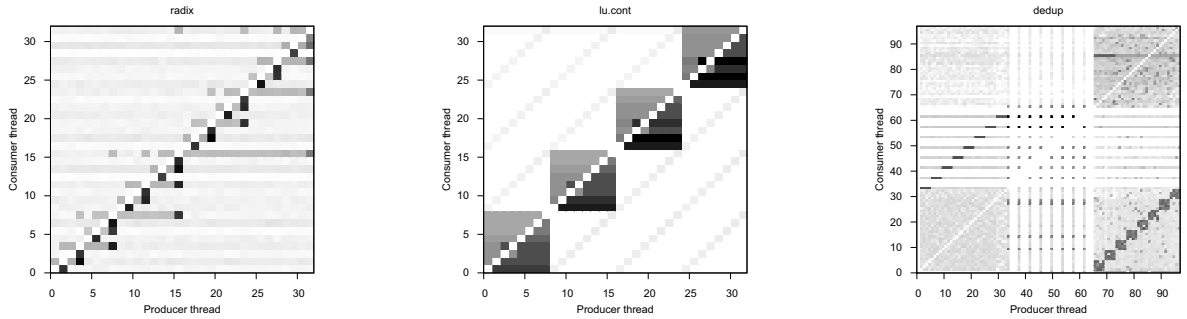

Figure 5. Spatial communication behavior: inter-thread communication during the entire parallel phase of the program for a selection of SPLASH-2 and PARSEC benchmark applications.

We have provided the communication-aware design-time exploration of mapping solutions based on the GECode constraint programming model [5].

To prove the effectiveness of our communication-aware mapping, we applied this methodology on the 3D-WSS application for each of the 12 scenarios identified in it [6]. When compared to a state-of-the-art TCM design-time exploration [7], our approach found better optimal pareto curves of mapping points, which are with $\sim$29% energy consumption savings and $\sim$15% improvement in performance. This exploration of mapping solutions has resulted in over 30% in energy savings at run-time for the 3D-WSS application. More information on these results can be found in [8].

C. Multi-threaded application characterization

Once an application has been parallelized, PinComm can be used to visualize communication between its threads. In this example we will look at a selection of benchmarks from the SPLASH-2 [9] and PARSEC [10] parallel benchmark suites. A communication characterization of these applications has been done in [11]. These results were obtained using Simics, which is a detailed full-system, cycle-accurate simulator. It requires the set-up of a virtual machine and incurs a significant slowdown (up to a factor of 100,000 times slower than running the same application natively). With PinComm on the other hand, virtually the same results can be obtained, but with a lot less set-up work (Pin can directly instrument native binaries on a host machine) and a much more manageable slowdown (although, since all memory read and write operations need to be intercepted, still a factor of about 1,000 slower than real-time). Moreover, PinComm natively supports running on a multi-threaded host, which is at this time not yet possible for most architectural simulators.

Figure 5 shows the communication between threads, for a selection of applications from the SPLASH-2 and PARSEC suites. These can be compared directly to Figures 4 and 5 in [11]. In Figure 6, we plot the temporal communication behavior of data produced by thread 1 (see Figure 6 in [11]).

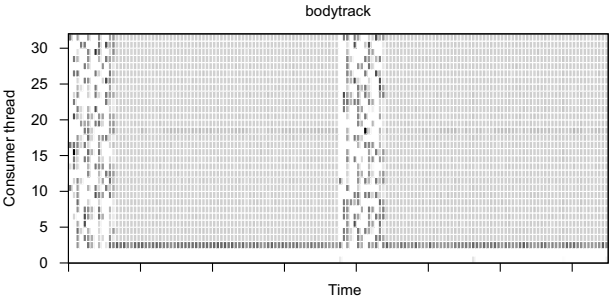

Figure 6. Temporal communication behavior: time and place of consumption of data produced by thread 1.

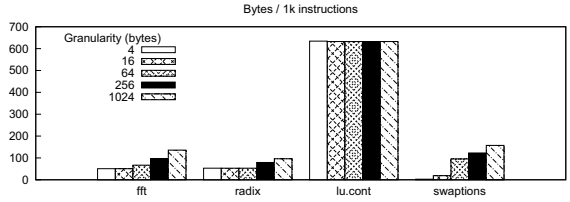

Figure 7. Total inter-thread communication measurement as affected by the memory address granularity setting.

Next, in Figures 7–9 we quantitatively analyze the total amount of inter-thread communication (expressed in bytes per 1000 instructions). Figure 7 shows how the this measurement is affected by the memory granularity setting. Increasing this granularity brings down the memory consumption of the profiler tool significantly, however, depending on the access pattern of the application, an increase in perceived communication can be caused, similar to *false sharing*.

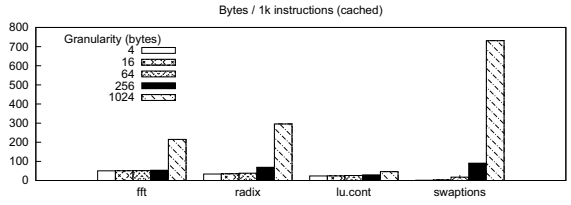

Figure 8. Inter-thread communication characterization when assuming perfect caching.

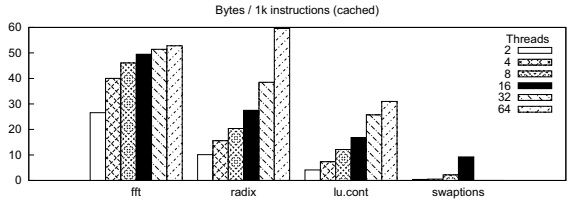

Figure 9. Inter-thread communication characterization (with perfect caching), when changing the number of threads

For Figure 8, in addition to the last-written-by table, we keep track of which data words have already been read by a given thread, and only count the first read of each word as actual communication. This gives a more realistic view of the magnitude of communication to expect when running the program on an architecture that caches remote reads. This extra bookkeeping requires a lot of memory though: the number of items to store is now in the order of *number of data blocks* $\times$ *average number of consumers per data block*. For inter-thread communication the number of possible consumers is relatively low so this extra memory usage is manageable, but it can quickly become excessive when more elaborate visualizations are enabled. When compared to Figure 7, Figure 8 shows a significant reduction in communication, especially for the `lu.cont` benchmark, which would clearly benefit from caching remote read operations.

Finally, Figure 9 shows the variation in the amount of communication when running the benchmarks with the same data set size using an increasing number of threads.[3] PinComm's results here clearly show the increase in communication, which will, on a platform with insufficient communication resources, limit performance long before Amdahl's law comes into effect.

[3]We did not include results for `swaptions` with high thread counts since the amount of work performed by the `medium` input set does not remain constant when running with more than 16 threads.

V. Related approaches

A. Static analysis

As mentioned in Section I, several tools and methodologies exist that can statically predict communication in regularly structured programs. Usually, some form of polyhedral model is imposed, in this case all dependencies – and all communication – can be computed analytically. Several important applications do fall into this class of programs, or at least their most important kernels can be described in this way. Still, other often-used constructs including pointers, such as linked lists, cannot be described in this way [12].

B. Architectural simulation

A common way to extract communication behavior of a program is through simulation of a parallel architecture, and instrumenting the network simulation component to log all network packets. This can be done using a (full-system) simulator, or through the shortcut of running an instrumented binary natively and sending all memory accesses through a simulated cache hierarchy. This latter technique is followed in CMP$im [2], which is, just like PinComm, based on the Pin instrumentation tool.

This approach has two conceptual drawbacks compared to a DDFG profiler. First, it requires a parallel program, and can therefore never help in *constructing* an optimized parallelization, only in *validating* it. Secondly, it is architecture dependent, since the cache hierarchy and coherence protocol have a major influence on the observed communication.[4] Finally, especially the simulation approach is much more computationally intensive. This restricts its use to smaller data sets which can exhibit communication patterns that are not always representative for the behavior of larger data sets.

C. Redux

Redux [13] is also a DDFG profiler, based on the ValGrind instrumentation tool. It builds a very detailed data-flow graph that shows all dependencies down to register level. While this can be very instructive for research or educational purposes, its extreme level of detail results in an exploding complexity of the DDFGs for even the simplest programs which restricts its use to very small programs or small parts of larger programs. In contrast, our approach sacrifices some detail for a much higher speed, which allows us to include the whole program with a realistic input set size. To this end, we only account for dependencies that go through memory and group them on a function call or even higher level.

[4]Note that in [11] most of this architectural dependence was avoided by logging memory accesses rather than network packets, and keeping a last-written-by table just as PinComm does – the results in [11] therefore also represent inherent communication rather than the observed network communication under a specific cache architecture.

D. Profile-driven auto-parallelization

In [14], Tournavitis et al. describe an auto-parallelization methodology based on profiling information. Their profiler constructs a control and data flow graph (CDFG) which is obtained by recompiling a (single-threaded) program using the CoSy compiler and adding instrumentation at the IR level. This metholology focuses on dependency analysis, and thus on the detection of parallelism, whereas the amount of communication is only of secundary importance.

PinComm can, in addition to giving hints about how to parallelize, help to gain insight into the communication behavior of parallel applications and of compositions thereof. This information can be used in off- and online scheduling, as we described in Section IV-B.

VI. Conclusions

On-chip communication is becoming more and more important in CMP and MPSoC settings. To visualize communication inside programs, even before they are parallelized and mapped onto a specific multiprocessor 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, through banks of shared memory, and between threads – the later one allows validation of a proposed parallel implementation. Using this new source of knowledge, new applications become possible, such as communication-aware parallelization, mapping, and adaptive configuration of on-chip network resources.

Acknowledgments

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

References

[1] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '05)*. Chicago, Illinois, Jun. 2005, pp. 190–200.

[2] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "CMP$im: A Pin-based on-the-fly multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA 2008*, Beijing, China, Jun. 2008, pp. 28–36.

[3] N. Tack, G. Lafruit, F. Catthoor, and R. Lauwereins, "Pareto based optimization of multi-resolution geometry for real time rendering," in *Web3D '05: Proceedings of the Tenth International Conference on 3D Web Technology*. New York, NY, USA: ACM, 2005, pp. 19–27.

[4] F. Thoen and F. Catthoor, *Modeling, Verification and Exploration of Task-Level Concurrency in Real-Time Embedded Systems*, 1st ed. Kluwer Academic Publishers, 1999.

[5] Gecode Team, "Gecode: Generic constraint development environment," 2006, available from http://www.gecode.org.

[6] N. R. Miniskar, E. Hammari, and F. Catthoor, "Scenario based mapping of dynamic applications on MPSoC: A 3D graphics case study," in *SAMOS*, 2009, pp. 48–57.

[7] C. Wong, P. Yang, and F. Catthoor, "Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform," in *CODES*, 2001.

[8] N. R. Miniskar, R. Wuyts, W. Heirman, and D. Stroobandt, "Energy efficient resource management for scalable 3D graphics game engine," IMEC, Tech. Rep., Sep. 2009.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, Jun. 1995, pp. 24–36.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, Canada, Oct. 2008.

[11] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterization of SPLASH-2 and Parsec," in *IEEE International Symposium on Workload Characterization*, Austin, Texas, Oct. 2009, pp. 86 – 97.

[12] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, Portland, OR, USA, May 2003, pp. 24–27.

[13] N. Nethercote and A. Mycroft, "Redux: A dynamic dataflow tracer," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 1–22, October 2003.

[14] G. Tournavitis, Z. Wang, B. Franke, and M. F. P. O'Boyle, "Towards a holistic approach to auto-parallelization," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2009.