View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Ghent University Academic Bibliography

A Communication Profiler to Optimize Embedded Resource Usage

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

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

***Abstract*—While 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 C/C++ programs, and constructs a dynamic dataflow 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 dataflow 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 also look at 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.**

***Index Terms*—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 manycore chips are becoming the name of the game: IBM's Cell processor, Intel's 80-core *Polaris* prototype [15] and its Larrabee graphics processor [12], and many others. These examples have set the trend for the continuing integration of many threads of software code onto a single piece of silicon. Also in the embedded domain, Multi-Processor System-on-Chip (MPSoC) now represents the idea of integrating not only traditional instruction set processors, but also hardware accelerators, large blocks of memory and interfaces to the external world into a self-contained System-on-a-Chip.

This new level of integration makes it cheaper – and thus feasible for a much larger number of applications – to have an ever larger number of processor cores. With it comes 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 usually remained hidden inside large servers or supercomputers – the only, niche systems featuring multiple processors up until only a few years ago – but is now prevalent, and must be accounted for during the design of even what used to be small, embedded systems. Indeed, Martin [10] agrees that *communication*, which is a natural result of multiple entities (be they processors, hardware accelerators or even memories) working together on a single problem (or multiple related problems), entails a major MPSoC design challenge.

The up-and-coming solution for solving on-chip communication problems is the Network-on-Chip (NoC) [3]. Compared to the classical solution of dedicated global wiring, a NoC can be a pre-designed and validated IP core, saving a significant amount of design time. At runtime, the NoC allows the expensive long-distance on-chip wiring to be re-used by multiple (often non-concurrent) communication flows, resulting in the required performance but with a much more efficient use of silicon area and power.

Of course, the design of a Network-on-Chip brings with it a whole new range of parameters, which system designers must fix in accordance to their specific requirements for performance given the imposed power and area budgets. Also, several research questions on NoCs remain [9]. One of the most important questions here is the mapping of the communication requirements, i.e. how many bytes of data are exchanged between all pairs of communication partners, at each moment in time; onto the available bandwidth of the network. By extension, the resulting latency and power usage can be seen as either cost functions or boundary conditions (or both) of this problem.

To solve the communication problem, a systems designer will want to do two things. First of all the application, when its parallel form is being constructed, should be laid out such that communication between separate network nodes is minimized. Secondly, once the (remaining) communication pattern is known, all computational nodes must be mapped onto a 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/scratchpad 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 signalling. Finally, once the previous optimizations have been done and the application's (network-visible)

communication pattern is fixed, all computational nodes must be mapped onto a topology. Usually a regular structure is used, mostly a 2-D mesh since, when laid out on a two-dimensional chip surface, this results in connections of equal length between each connected node pair.

For regularly structured programs, a lot of this optimization can be done statically. An important example here is loop transformation, which can be used to optimize memory usage or communication [6], [2].

In a large fraction of important existing and emerging applications, however, this static analysis has proved to be infeasible. These programs have an irregular structure, have a behavior that heavily depends on the input or other external influences, or are even dynamically composed of multiple smaller application components each sharing the same chip. Especially inter-processor communication turns out to be very dependent on these influences. This makes that static optimization – although still useful – can never give the fullest possible benefit. In these cases, dynamic methods of characterization and optimization should be used. However, an approach must still be largely automatic to give the additional benefit that both characterization and optimization can be tailored to a specific (class of) input set, a combination of program components, or even to a specific program phase.

Our profiler, *PinComm*, 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 valuable information can subsequently be used in parallelizing the program (while minimizing communication between threads), in mapping the application's parallel components (optimally matching communication patterns and network topology), or it can be used as input to runtime schedulers in a scenario-based approach such as TCM (see Section IV-D) to make mapping and scheduling decisions in a communication-aware way.

II. PinComm: a communication profiler tool

A. Constructing the dynamic dataflow graph

Our profiler, *PinComm*, constructs a dynamic dataflow graph (DDFG), this is 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, which is a dynamic instrumentation tool [8]. Pin can run any executable program on a Linux/x86 system, independent of the programming language used, and allows modular instrumentation of this program through the use of plugins. Our profiler is such a Pin plugin, which instructs it to intercept all memory accesses and all function calls. Each time one of these instructions is executed, Pin calls back into our profiler. 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 (usually four bytes for the 32-bit x86 architecture).

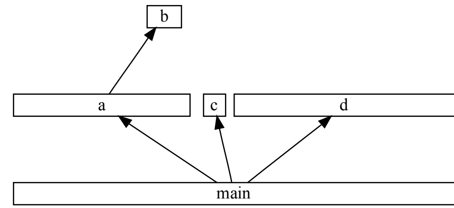

Fig. 1. Call tree for a hypothetical program, as measured by our communication profiler. The `main` function calls three functions, `a`, `c` and `d`, while `a` calls function `b` for some of its functionality. Time (in instruction counts) goes from left to right, node widths are scaled proportional to each function's runtime.

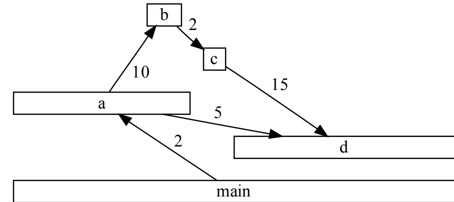

Fig. 2. Communication graph for the hypothetical program from Figure 1, as measured by our communication profiler. Each arrow shows the (inherent) communication between two functions.

Our profiler can thus measure two important program representations: a dynamic call tree (one for each thread), and a communication graph that shows communication streams between all dynamic function calls. An example of the call tree can be seen in Figure 1. Execution time (in instrucion counts) has been visualized by positioning and sizing each function call's node at a horizontal position relative to its starting time and execution time, respectively. Figure 2 shows the corresponding communication graph.

These graphs can be generated completely automatic using our profiler. Function names are extracted from linker information in the executable. When the executable contains debug information, we can also 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. Also, C++ decorates function names which can make the function labels unrecognizable. 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, programmers can split up their programs in functional blocks and use the PinComm profiling tool to view communication between these blocks, rather than between individual functions.

Another way to split up the program, other than by function or by marked region, is by thread. This is interesting for programs that are already parallelized. The communication graph will now show communication between threads. When clustering the communication graph 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). Simulation of a realistic cache subsystem could be added to the profiler (since it intercepts all memory accesses, all information for a cache simulation is available), although existing tools for this also exist (e.g. CPM$im [7], which is also based on Pin).

Finally, markers can also be used to start and stop the measurement halfway 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. Also, 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-E.

B. *Communication through memory regions*

As a first order approximation, the dynamic dataflow graph, when clustering all nodes according to which processor they will run on, gives the communication that will be visible on the on-chip network. This assumes that all memory accesses *internal* to a processor (so 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. So in effect we assume the processor has a perfect cache, or a scratchpad memory large enough to hold all the thread's private and shared-owned data.

For small communication flows, this approximation is usually enough for our applications. Large data structures, however, are often allocated in shared memory blocks which have their own network node, or in off-chip memory (in this case, network traffic flows to/from the network node containing the interface to off-chip memory). 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. This allows one to estimate the resulting network traffic to this node, and also get a quantitative measure of its required memory bandwidth – from which the type of memory, banking/interleaving etc. can be decided.

[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.

III. Case study: 3D wavelet decoder

A. *Application*

3D content made available on the internet, such as X3D/VRML and MPEG-4 content, is transported over and consumed on a lot of different networks (wireless and wired) and terminals (mobile phones, PDA's, PCs, etc.). The difference in the bandwidth of the networks and the performance of the terminals makes it hard for the content provider to choose the right quality, triangle budget and complexity for the content. Moreover, the actual content is rendered heavily according to highly dynamic viewing conditions and hence these rendering decisions are moved to the terminal side. Therefore, multi-resolution frameworks are needed, where the object's quality dynamically adapts to the viewing conditions while respecting constraints such as available hardware platform resources and Quality-of-Experience (QoE).

Our demonstration application is a Wavelet Subdivision of Surfaces (WSS) algorithm, described in detail in [13]. By progressively decoding higher wavelet frequencies, an adaptive quality level can be obtained, depending on the complexity of the input and on external requirements. Such frameworks can broadly be classified under two categories: one maximizes quality under frame rate constraints (suitable for power-plugged systems) while the other minimizes resource/energy consumption under QoE constraints (suitable for portable embedded systems). Our main focus is on the latter, where the best triangle budget and the related Level-of-Detail (LoD) settings for each visible object are decided according to the user requested scene level quality. The LoD settings here indicate not only the discrete LoD parameter but also some decoder specific parameters. In the case of WSS these are continuous LoD parameters.

Clearly, this application is highly adaptive in its resource requirements, but through the input set (the complexity of the rendered scene), input events by the user (setting a camera viewpoint or interacting with the scene, which hides/unhides objects and changes their relative distance to the camera, and thus their required LoD), and environmental influences on the target platform (concurrently running processes, which might be implementing game logic, AI or communication capabilities, and battery or thermal constraints).

B. *Input to 3D-WSS and scenarios*

The objective of these experiments is to show how the information from the application can be used for resource management. We consider as input for the 3D-WSS application a gaming environment with three rooms, with a number of objects (13, 17 or 22) and four different camera positions in each room. For each of the 12 scenarios, resulting from the room the person enters and the camera position, the communication flow incurred by the wavelet decoding will be measured. The 3D-WSS application is yet to be parallelized. We will operate under the assumption that the major functional blocks will be

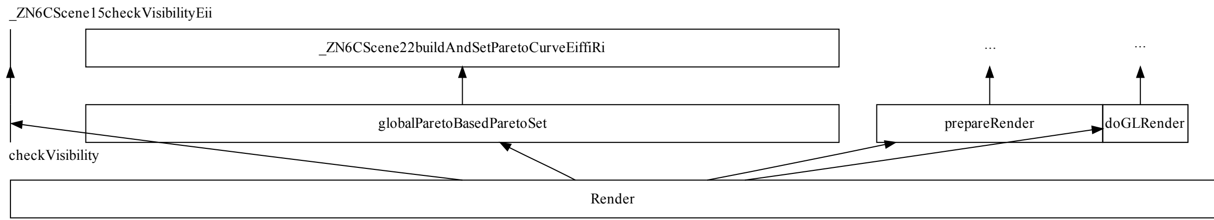

Fig. 3. Partial call tree for WSS with function names (parsed automatically from the debug information) and run lengths (in instruction counts) as made by PinComm (frame #2)

distributed over different processors in a pipelined fashion, and will therefore measure communication between each of these functional blocks.

C. Functional decomposition

Although the call graph generated by PinComm, which includes the runtime of each function, could be used to make a decomposition into the target application's most time-consuming functional blocks, for WSS we already have such a decomposition [13].

First, we ran our profiler on the WSS decoder, without imposing the task graph of WSS that was provided by its developers. This way, the most time-consuming functions should again become visible. We also placed markers at the start of each new frame. The result, when selecting only the second frame, is shown in Figure 3. Each function call is represented by a rectangle on a vertical position corresponding to its stack depth. The horizontal position of each rectangle is determined by the time the function starts and ends, measured in instruction counts. The graph shows that the `Render` function first calls `checkVisibility` which returns very quickly, next `Render` does computation on its own (possibly calling shorter functions, function calls shorter than 10,000 instructions were filtered out), subsequently the functions `globalParetoBasedParetoSet`, `prepareRender` and `doGLRender` are called.

From this call graph one could analyze the application's major building blocks, and arrive at the same functional partitions as those given to us by the application's developers (one region, `WssDecode`, is not visible in Figure 3, which shows the second frame, because `WssDecode` only takes a significant amount of processing time during computation of the first frame). This shows that our profiler can help in identifying the high-level functional blocks of new applications, for which a functional partitioning is not yet known.

Next, we marked the major code regions in the WSS source code and ran the profiler tool again, this time measuring the communication between the regions. This manual marking was done to combine all sub-functions of the major regions of interest of the application, and to clean the graph from uninteresting parts of the program and from cryptic, C++-decorated function names. The result is shown in Figure 4, for both the first and second frame. Again, 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 4, top), a first major stream totaling around 275 kB is from `main` – this region contains all code not explicitly assigned to other regions and includes the function where object data is read from file – 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, these decoded vertices (which are clearly much bigger than the original data fed into `WssDecode`) are used by the `Prepare` code region.

Figure 4 (bottom) shows the results for the second frame. In this frame (and subsequent ones) most of the objects have been decoded by `WssDecode` and the results are cached. The computation time of `WssDecode` is therefore significantly reduced compared to the first frame. Note that Figure 4 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 the communication from `WssDecode` to `BuildPareto`, in Figure 4 (bottom), 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 of the first frame, which is subsequently used by the `Render` region of the next frame.

D. Communication patterns

We can now, for the major communication streams in WSS (those visible in Figure 4), determine their behavior in different iterations of the program. We already noticed from Figure 4 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 (up to several seconds). These decoded objects are stored in local memory; in subsequent frames only newly-visible objects have to be decoded.

Figure 5 shows the runtimes (in instruction counts) for all functional regions. Four input scenarios are shown, in each the

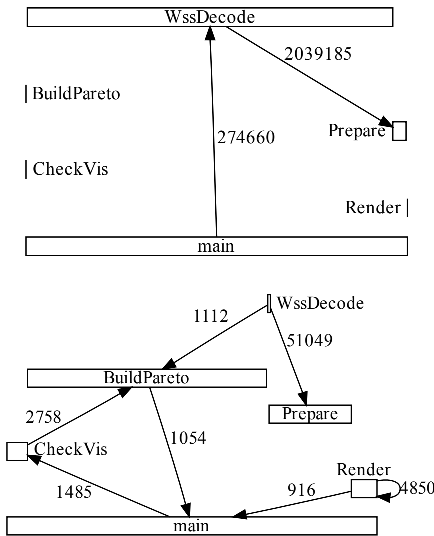

Fig. 4. Communication graph after manual marking of the major code regions `CheckVis[ibility]`, `BuildPareto`, `WssDecode`, `Prepare[Render]` and `Render`. Region lengths (as node widths, proportional to the region's dynamic instruction count) and large inter-region communication flows (marked on each edge, in bytes) are shown, for frames #1 (top) and #2 (bottom).

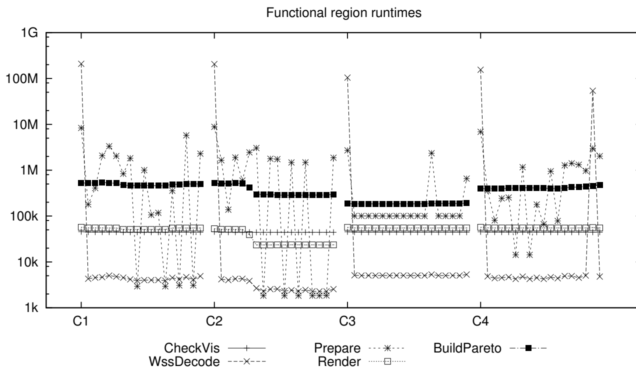

Fig. 5. Runtimes for the functional regions of the WSS application, for all 18 frames of the four input scenarios with a 13-object scene

same scene is rendered but from a different camera viewpoint (C1…C4). In every run, 18 frames are computed while the scene is animated in the same way. The differences in runtime for the `WssDecode` region are immediately apparent: in the first frame it is very long, from the second frame onwards very few additional objects need to be decoded so the runtime falls back significantly. One exception is the 17th frame of C4: here a new object comes into view which needs to be decoded, yielding a longer execution time for `WssDecode` in this case. Note that the four scenarios (camera positions C1…C4) are run separately, objects decoded in the first frame of C1 thus need to be recomputed for scenario C2.

E. Inter-regional communication

Figure 6 shows the magnitudes for each of the major inter-regional communication flows, and its evolution throughout

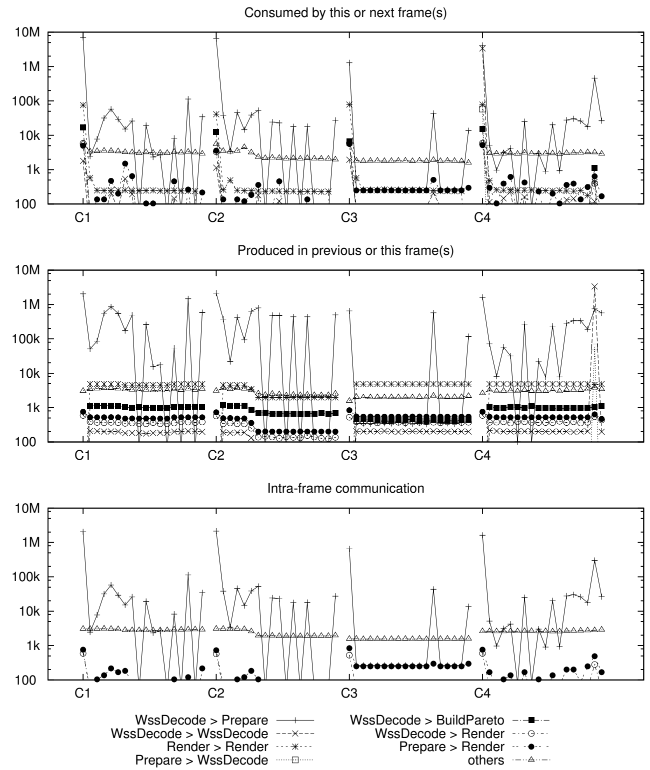

Fig. 6. Inter-regional communication magnitudes, for all camera positions, all frames and a 13-object scene

the program. The points marked with $a > b$ show, for a given frame, the size of information (in bytes) that is produced by region *a* in a previous (or this) frame and consumed by region *b* in the current frame. This was done for a 13-object scene, all frames and the two out of four possible camera viewpoints (C1 and C2). Using the profiler's output data, one can construct similar graphs for visualizing the output of data (produced by the current frame and used in a subsequent one), or for intra-frame communication, all of which can guide optimization in different phases of the mapping of the application's computational pipeline.

By far the largest inter-regional communication stream, for all frames and all camera positions, 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. Subsequent executions of `WssDecode` decode some additional objects, but most of these are only used in the current frame: the magnitude of communication to this and future frames is about as high as the intra-frame communication for this function pair, which is visible on the bottom graph. In the middle graph we see that `Prepare` consumes a relatively constant amount of data, per frame, from a `WssDecode` call in an earlier frame (presumably the first). Also, the bottom graph shows that although `WssDecode` after the first frame generates little data that is to be used

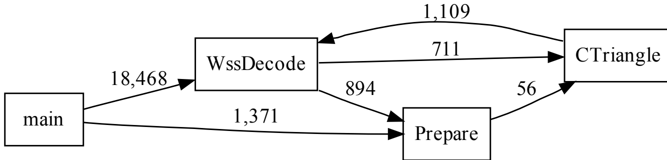

Fig. 7. Communication using shared memory and `CTriangle` classes, in addition to the known streams from Figure 4, for C2, 13 objects, frame #1.

in subsequent frames, it keeps generating (a lower amount of) data that is used in the same frame.

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 it were at all 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 on-chip communication unavoidable. The architecture should thus be dimensioned in such a way that it can accommodate the `WssDecode`-to-`Prepare` communication stream.

F. Communication through shared memory

For Figure 7 we marked the `CTriangle` class for inclusion as a node in the graph. The figure shows, for the first frame camera position C2 and a 13-object scene, the major communication flow (in kB) between all participating nodes (the code regions without large communication streams have been removed from this graph for clarity). In addition to the streams known from Figures 4 and onwards, we now see that `WssDecode` stores a significant amount of local data in `CTriangle` objects (them amount to around 32 MB in total). These objects will therefore most likely be stored in an off-chip memory. The magnitude of the extra communication stream over the on-chip network this would incur is again provided by our profiler's results, and can be read from Figure 7.

IV. Applications

The data gathered by PinComm can be used in mainly two ways: (i) when partitioning the application into threads, and (ii) when mapping the threads onto processors. Although both can be done both on- and offline, usually partitioning will mostly be done at design time (although implementations with a variable number of threads 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 (manual) 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 inner 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. Using communication between functions, as is 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. 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 in some cases even that the introduction of a significant load-*i*balance can even further improve performance.

This technique can be visualized on the communication graph (see Figure 8): communication arrows cut by the partitioning (solid lines) cause inter-processor network traffic, communication denoted by arrows internal to a cluster (dashed lines) can be handled by processor-local caching.

Let's look again at Figure 2 for instance. Assume we want to pipeline the execution of the computationally intensive functions `a` and `d` on a two-processor architecture. Function `a` is slightly shorter than function `d`, to minimize the load imbalance between both processors it would be better to assign function `c` to the processor running `a`. This clustering of functions onto processors can be drawn on the communication graph (Figure 8, top). Each communication arrow that is cut by a cluster's boundary (solid arrows) will result in interprocessor communication at runtime. Arrows internal to a cluster (dashed arrows) result in *intra*processor communication, which – when this data can be allocated to a memory local to this processor, or remain in its cache – will not require use of the on-chip network.

We now see that functions `c` and `d` share a lot of data. It could therefore be more beneficial to tolerate some load imbalance by running `c` on the `d`-processor, a cost that can easily be won back through the reduction in communication cost (Figure 8, bottom). Of course, the final solution will depend on the relative costs of load imbalance and communication, which are implementation-specific.

Finally, note that our profiler does not necessarily see *all* data dependences. Smaller communication streams are removed from the communication graph for clarity but they can still result in data or control dependences which may prohibit parallelization. Moreover, since the graph is constructed based

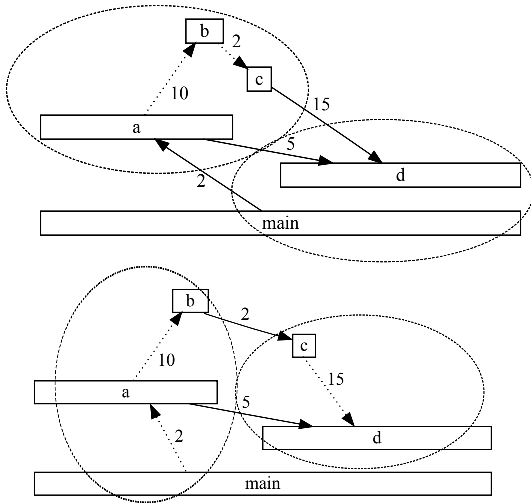

Fig. 8. Possible partitioning of functions `a`, `b`, `c` and `d` onto two processors. Communication arrows cut by the partitioning (solid lines) cause inter-processor network traffic, communication denoted by arrows internal to a cluster (dashed lines) can be handled by processor-local caching. In the top graph, the communication cost (sum of all cut (solid) arrows) is 22. The bottom partitioning, while achieving an almost equally optimal load balance, has a communication cost of only 7.

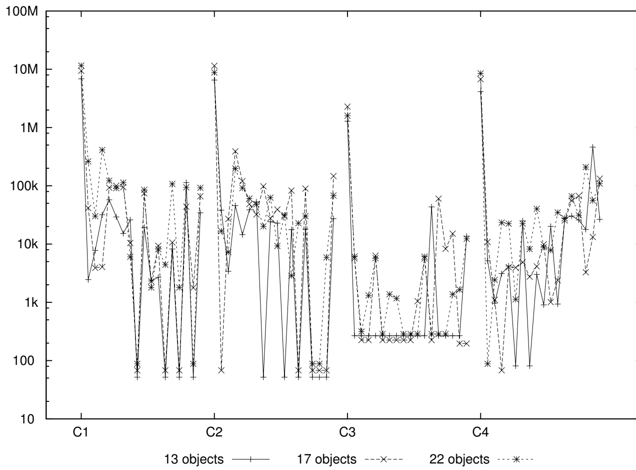

Fig. 9. `WssDecode`-to-`Prepare` communication stream for all input scene sizes

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

B. Optimizing communication behavior

By carefully planning the `WssDecode`-to-`Prepare` communication stream, making use of knowledge provided by Figure 6 and some basic knowledge about the working of the application, we can further minimize on-chip communication. We noticed that there is a large production of data by `WssDecode` in the first frame, which is subsequently consumed by several iterations of `Prepare`. One can now distinguish between two cases, depending on which part of data that is used by the different `Prepare` calls. If all of them used (mostly) the same data, then it would make sense to transmit all data, immediately after production by the processor or accelerator running `WssDecode`, to memory on or near the `Prepare` core. This way the communication cost is paid only once (and in advance), and `Prepare` will spend little time waiting for data. The other case is when the `Prepare` calls in subsequent frames do not use the same data. This is (to an extent) the case in the WSS application, as can be seen on Figure 6: `WssDecode` generates between 1 MB and 10 MB of data, but (much) less than 1 MB is used by each `Prepare` iteration. Based on this observation, one could propose an architecture in which the data is transferred on demand. This way, communication would be spread out through time, and the maximum load for witch the network-on-chip would have to be designed is significantly lowered. The memory requirements for this implementation would be higher, though: since part of the data is re-used by different `Prepare` iterations one would probably provide a local copy near the processor running `Prepare`, containing part of the information contained in the larger memory near the `WssDecode` core. The first scheme needs less memory at the `WssDecode` core, the data can be streamed directly to the `Prepare` core.

C. Communication through shared-memory

As visible in Figure 7, `WssDecode` keeps a large amount of local data, in the form of `CTriangle` objects, in off-chip memory. PinComm can provide the total size and access counts, this can help a designer to implement the memory, its interface, and the paths to it through the on-chip network, accordingly.

D. Making TCM communication-aware

Task Concurrency Management (TCM) is a methodology for runtime management of embedded resources [14]. This methodology uses *scenarios*, which are clusters of (input set, program phase) combinations that have a similar behavior on the target architecture. For each scenario, an optimal thread scheduling is found at design time, while at runtime the specific schedule for the current scenario is used at each time. This way, a large effort can be made at design time to reduce resource and energy requirements, while being adaptive to the specific runtime circumstances but with a very low runtime overhead.

In TCM, program variations can be defined that relate to scheduling (in time) and mapping (in space) of program threads. Mapping can also mean choosing a specific target processor, when multiple types are available in the system. This way the methodology can run a given thread on a DSP or FPGA core when high-performance operation is needed, for instance while decoding a dynamic video fragment, and later reschedule this thread to a general purpose processor

during a more static scene. External events such as lowering the resolution or the system's battery being drained can also be taken into account. In addition, when processors have configurable options such as core voltage, operating frequency or powering on/off parts of the cache, these settings are also part of the scenario. The main advantage of TCM, compared to other techniques, is that it is very automatic: based on profiling data, clusters of similar program phases are identified as scenarios, optimal schedules for all scenarios are computed, and a runtime manager is generated that detects the current scenario and activates the appropriate schedule.

Clearly, 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. In a first phase TCM's scheduling and mapping can be optimized to minimize 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. A second phase can consist of the configuration of the network: just like the processors' voltages, frequencies and cache size are set dynamically depending on the scenario, an on-chip network can be configured for optimal (best performance, lowest power, etc.) support of the expected communication flow. This configuration can take the form of setting buffer sizes, changing clock speeds or bit widths of network links, making changes to the routing protocol, etc. Details on the required adaptations to TCM, its optimization algorithms, and results using our profiler data can be found in [1].

E. Scenario-based runtime management

In a scenario-based methodology, control variables may be used to detect the current scenario. These are application-level variables such as loop counters or state variables, or properties of the input stream, that have shown to exhibit a predictable relationship to the behavior application [5]. This way, the application's properties can be predicted some way in the future, and the system configured accordingly.

In Figure 10, we performed a principal component analysis (PCA) on the relation between the available control variables and the major communication stream (`WssDecode` to `Prepare`). The main contributors to the principal component are the control variables that denote the number of visible triangles (with a normalized weight of 26%), the scaling factor (25%), the total visible area (20%) and the number of objects (11%). Although there is no obvious direct relationship between the principal component and the magnitude of the network traffic for all data points, the *maximum* flow does have a clear, almost linear dependence on the control variables. It is therefore possible to make a good estimate of the maximum expected communication during the next frame, only dependent on variables known at the start of the frame. Since the difference in expected communication can be rather large (more than a factor of five), some good energy savings should be possible when this information can be used to set up the communication network accordingly.

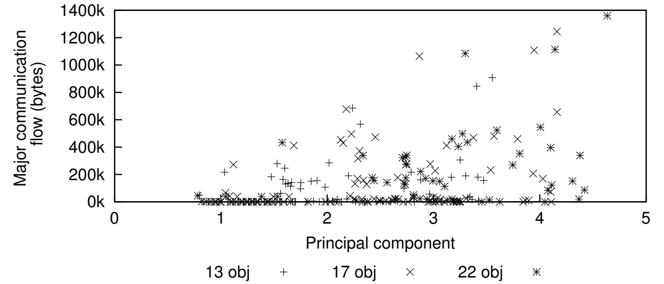

Fig. 10. `WssDecode`-to-`Prepare` communication stream as a function of the optimal combination of control variables

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 [4].

B. Architectural simulation

A common way to extract communication behavior of a program is through simulation on 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 [7], 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. Finally, especially the simulation approach is much more computationally intensive, restricting its use to much smaller data sets.

C. Redux

Redux [11] is also a DDFG profiler, based on the ValGrind instrumentation tool. It builds a very detailed dataflow graph which shows all dependencies down to register level. This results in an exploding complexity of the DDFGs for even the simplest programs, and 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 accounted for dependencies that go through memory and group them on a function call level (or even higher in postprocessing).

VI. Conclusions

On-chip communication is becoming more and more important in 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 dataflow graphs (DDFGs) for C and C++ programs, and can present the 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 which allows validation of a proposed parallel implementation. Using this new source of knowledge, new applications become possible, such as communication-aware parallelization, mapping, and configuration of on-chip network resources.

VII. Acknowledgments

This research is supported by the "Optimization of MPSoC Middleware for Event-driven Applications" (OptiMMA) project, grant 060831 of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

[1] N. R. Miniskar, R. Wuyts, W. Heirman, D. Stroobandt, "Energy efficient resource management for scalable 3D graphics game engine," Tech. Rep., Sep. 2009, submitted to DATE 2010.

[2] Y. Bouchebaba, B. Girodias, G. Nicolescu, E. M. Aboulhamid, B. Lavigueur, and P. Paulin, "MPSoC memory optimization using program transformation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 12, no. 4, p. 43, 2007.

[3] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Design Automation Conference*, Jun. 2001, pp. 684–689.

[4] 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.

[5] S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, and K. D. Bosschere, "System-scenario-based design of dynamic embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–45, Jan. 2009.

[6] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Int. J. Parallel Program.*, vol. 34, no. 3, pp. 261–317, 2006.

[7] 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.

[8] 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 *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. Chicago, Illinois: ACM, Jun. 2005, pp. 190–200.

[9] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. Enright Jerger, and Y. Hoskote, "Outstanding research problems in NoC design: System, microarchitecture, and circuit perspectives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 3–21, Jan. 2009.

[10] G. Martin, "Overview of the MPSoC design challenge," in *Proceedings of the 43rd annual Design Automation Conference (DAC)*. New York, NY, USA: ACM, 2006, pp. 274–279.

[11] 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.

[12] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, Aug. 2008.

[13] 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.

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

[15] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, Jan. 2008.