Toward a Model of Computation for Time-constrained Applications on Manycores

Stephane Louise
CEA, LIST, PC172, 91191 Gif-sur-Yvette, France

Keywords: Model of Computation, Embedded Manycore Systems, Time-critical Systems.

Abstract: As computing systems are transitioning from multicores to manycores with the increasing number of computing resources available on modern chips, we can notice a lack of a universal programming model for these new platforms and the challenges they convey. Ideally speaking, a program should be written only once, and making it run on a given target would be the role of the compilation tools. But before addressing this problem, we need a good Model of Computation (MoC) as a base for both programming and compilation tools. In this paper we propose to share our insights on the properties such a MoC should possess. It would take the CycloStatic DataFlow (CSDF) MoC for its good properties, and extend it to overcome its limitations while retaining its good properties.

1 INTRODUCTION

Nowadays, the limits on power usage and dissipation for single-chips is encouraging a trend toward more parallelism in both HPC and embedded systems. This is why off-the-shelf processors went from single-core in the 1990s to the generalization of the multi-core at the beginning of the 2010s. The extended trend shows the happenstance of so-called manycore systems which are already available from several vendors (*e.g.* Tilera Tile-64, Intel Xeon-phi, Kalray MPPA-256, *etc.*). What distinguishes a manycore from a multicore is not only the sheer number of cores but also the communication means between the cores. For a multicore system, the communication medium is either a single bus or an evolution of a single-bus, but for a manycore system, the communication is handled through a Network on Chip (NoC) because approaches based on single buses are no longer sustainable. Another trends is the growing concern about addressing power management issues more dynamically and especially in embedded applications (*e.g.* cell phones), the perceived value is often tied to very specific tasks like decoding high profile video file (*e.g.* H265) or doing elaborate image processing. This phenomenon is generally known as Gustafson's Law (Gustafson, 1988).

So what problems are looming ahead for programmers of future embedded applications? First, parallelism is difficult and automatic parallelization has its limits with the compilation tools of the foreseeable future. Second, lots of the current issues like the memory wall will remain and become even more so accurate. Third, as architectures become more complicated managing time constraints in applications becomes also more and more challenging.

Ideally speaking a programmer would like to write universal code: write once, run anywhere. This goal was arguably reached for single process applications with a C-compiler (or other usual programming language), because C exposes a good generic abstraction of the model of computation of a general purpose processor. This is what is currently lacking for manycore systems. But before heading for compilers, we need to find a sound generic base for a Model of Compilation (MoC), as MoCs provide an abstraction for both programming languages and Intermediate Representations (IR) in a compilation toolchain.

In this paper we will focus on discussing the general directions where a first answer to the problem of a good generic Model of Computation (MoC) for embedded manycores should be found. Section 2 is a quick overview of the state of the art and section 3 provides a discussion on the properties a good MoC should encompass, including timeliness, liveliness, safety and dependability. In the context of this paper, we will focus mostly on embedded devices, but some of the encountered issues are becoming more and more accurate in traditional HPC systems.

Louise S..
Toward a Model of Computation for Time-constrained Applications on Manycores.
DOI: 10.5220/0005467900450050
In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2015), pages 45-50
ISBN: 978-989-758-100-7

2 MODELS OF COMPUTATIONS

2.1 Programming with Manycores

Doubling the number of processing units every two years or so yields an exponential growth of the available processing power. As a consequence, to efficiently exploit this amazing processing power, there is a need of an exponential growth of the expressed parallelism of target applications. Usual programming paradigms do not scale well as was noticed several years ago by Edward Lee (Lee, 2006): They lack determinism and human manageability, even for a limited number of threads. For thousands or millions of threads, unless the application is simple (typically SPMD[1] or at least highly homogeneous) like in computational science and parts of HPC programing, there is no way to resolve a faulty behavior[2].

Another issue is the overwhelming amount of data required to feed this huge source of computing power: Throughputs of memory buses hardly progress with regards to computing power (the so-called memory wall), and single buses are already being phased out for workstations. Networks on Chip (NoCs) scale a bit better but their throughput growth is hardly better than linear for mesh networks. The obvious conclusion is two-folds: First, usual programming models will not scale up well; second, the future is NUMA[3], with local memory serving a primordial role to limit accesses to off-chip memory, and avoid NoC contentions. This can work for the same reason cache memories work, *i.e.* processing often use data elements that are either "recently" used or produced, or "close to" recently used or produced data elements.

This is a reason why usual approaches from HPC, especially OpenMP does not work for time-constrained embedded manycores. Shared data consistency is one of the most expensive feature of a multicore system, in terms of power budget and timing uncertainties. Hence, it makes sense to ponder if it is mandatory in a MoC fitted to manycores. Architecturally speaking some voices claim that cache and memory coherence concepts will not scale well with manycores (Choi et al., 2011). From the HPC world, there is a movement toward using message passing (*e.g.* MPI) to comply with this hypothesis. In the embedded world, this idea gave birth to a renewed interest in dataflow concepts and especially Kahn Process Networks or KPN (Kahn, 1974) and its derivatives. For example this is the case with StreamIt (Amarasinghe et al., 2005), Brook (Buck, 2004) and ΣC ("*sigma-C*") (Aubry et al., 2013).

So, instead of letting a programmer use data coherence and hope for the better, it would be preferable to make programmers utilize a MoC that makes all data usage, reuse and routing explicit for any computation. Moreover, the MoC that drives the proposed programming language should offer properties that simplify the correctness by design of the parallelism, and offer an easy debugging as well as automatic compile-time verifications to help with the programming process.

2.2 Stream Programming and Dataflow

On the front of dataflow paradigms, CUDA (NVIDIA Corp., 2007) and OpenCL (Khronos OpenCl Working Group, 2008) are two ongoing industrial efforts to bring data-driven principles to ordinary (C-like) languages. They are quite well fitted to heterogeneous computing, but in their current form, remain focused on GPGPU-like models of computation and are still too close to the hardware, *i.e.* lacking generality.

The concept of stream programming made a comeback because the underlying MoC offers a path toward an easy and manageable way to express and describe massive parallelism for the programmer. It also renders obvious the data-path and data-dependency all along the processing chain, both to the programmer and to a compiler. Therefore compilation tools for stream programs take an important role: It can place and route processes and data paths, and automatically discard data that are no longer relevant at a given point of processing. Stream programing can offer a deterministic execution model that permits to discriminate between sane parallel programs and unmanageable ones at compile-time. It also offers a manageable memory and data placement with regards to the compilers.

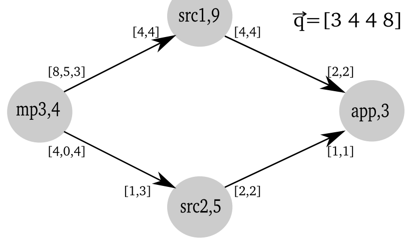


Figure 1: A simple CSDF for MP3 decoding. q is the repetition vectors that exists for well formed applications and provides the base of scheduling.

The bases of stream programing rely on Kahn Process Networks (KPN), more precisely on their derivations, such as Cyclo-Static Data Flows –CSDF,

[1]Single Program Multiple Data

[2]*i.e.* with deadlocks or livelocks or race conditions, *etc.*

[3]Non Uniform Memory Access.

(G. Bilsen and Peperstraete, 1996). Applications are defined as directed graphs whose nodes are processes and edges are communication channels. It can be seen as a formalization of the old-styled "*processes and pipes*" Unix-like way of parallel programming: Any process can possess one or several channels as inputs or as outputs. Input channels are read-only, output channels are write-mostly and data productions and consumptions are modelized as atomic token productions and consumptions, respectfully. Channels are the only communication means between processes and reading is blocking if an insufficient number of tokens is present on any input channel of a process. In that case, the execution of the process is stalled until the condition changes. SDF (Static DataFlow) and CSDF are special cases of KPN: For SDF graphs, all the processes have fixed consumption and production rates; For CSDF graphs, all the processes' production and consumption rates must obey to a fixed cycle (the length of the cycle can differ between processes, and the production and consumption can change from one step of the cycle to the other but not from one cycle to another). KPN and CSDF are locally deterministic for their execution and the possibility to run a CSDF in bounded memory and without deadlocks is a decidable problem (Buck and Lee, 1993b). This means well-formed CSDF applications can be statically determined and they are globally deterministic in that case. A simple example of a CSDF application is show in Figure 1.

2.3 Modern CSDF based MoCs

SDF and CSDF are very simple, and in fact too simplistic to be generic MoCs without adding further semantic. Nonetheless, it is highly desirable to keep the CSDF equivalency in a dataflow MoC because of the good properties, as seen previously. Languages like Brook and StreamIt brought two interesting features to these MoCs:

- The capability to read on a given input channel a fixed maximum number of data tokens without consuming them. This allows for a simple implementation of sliding windows on signals and images which is a classic way to do signal and image processing,
- Some specific predefined generic processes do not perform transformations on data but only reorder the stream of data. The usually defined transformations include:

Splitters are CSDF processes with 1 input and n outputs, and one integer $t_i \in \mathbb{N}$ per output. Each time a data token of order l is received on the input channel, then the data token is put on output channel k such that $\sum_{i=0}^{k} t_i \leq l \mod (\sum_{i=0}^{n-1} t_i) < \sum_{i=0}^{k+1} t_i$. The usual case is when $t_i = 1, \forall i$, so the first received token is put on channel 0, the second on channel 1, *etc.* in a round-robin fashion. This is a usual way to define a Single Program Multiple Data (SPMD) type of parallelism (see Figure 4).

Joiners are the symmetric processes, with n inputs and 1 output. The process waits for t_i tokens on its input line i and outputs the tokens on its output channel, then switches to wait for t_{i+1} tokens on channel $i+1$, and so on in a round-robin fashion. This is useful to close a set of parallel treatments, either SPMD initiated by a splitter or a SDMP initiated by a duplicater.

Duplicaters are simple SDF processes used for copy. They have 1 input line and n output lines. Each time a data token is read on the input line, it is copied on every one of the n output channels. It is useful to implement Multiple Programs Single Data (MPSD) type of parallelism.

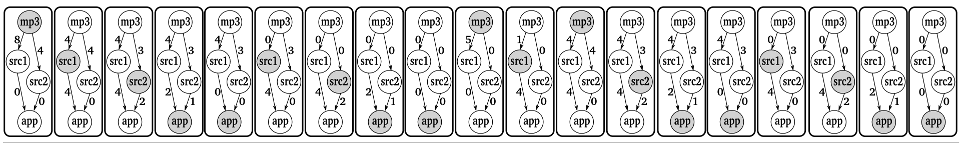
Sources and Sink. Sources are processes without inputs, and Sinks are processes without outputs.

Data distribution processes can be utilized to create any repetitive series of data of a given data-set, as demonstrated in several research works, and if the compiler is aware of the semantic of these processes, it can optimize them to the specificities of both the application and the underlying hardware system on which the application will run (de Oliveira Castro et al., 2010).

One way to extend the semantic of CSDF is to provide data distribution processes which may be outside the scope of SDF or CSDF but which ensure at least a CSDF equivalency when they are combined in valid constructs.

By their nature, stream programming languages offer most of the desired properties for large-scale parallel applications:

- KPN and CSDF provide a clear path of dataflow because channels are the only communication means. Hence, it does not depend on a low-level implementation or the hardware details. These details are a compiler issue, not a programmer one, so portability can be achieved this way.
- Execution determinism is achieved naturally for a subset of KPN, especially CSDF.
- By construction, explicit locks are absent from KPN and CSDF since the execution of tasks is only constrained by data movements,
- The presence of synchronization points in the execution is also obvious when data path converge toward a few processes.



mp3	src1	src2	app	app	src1	src2	app	app	mp3	src1	mp3	src2	app	app	src1	src2	app	app

Figure 2: The base for scheduling the application of Figure 1 with the repetition vector.

- It allows for a natural hierarchical design, as seen in languages like StreamIt or ΣC.

CSDF is a good choice as a base, because it is the most elaborate MoC currently known that preserves the decidability of deadlock freeness and possibility to check properties statically at compilation time (G. Bilsen and Peperstraete, 1996; Stuijk, 2007).

But although it provides a sane programing base, it lacks the versatility required for the relevant applications of the future of embedded computing. Such applications like ADAS and autonomous vehicles (Campbell et al., 2010; Guizzo, 2011), or augmented reality –see *e.g.* (Sato et al., 1998; Feiner et al., 1997; Murillo et al., 2012)– require at the same time a large amount of processing power, a sensitivity to current context (*e.g.* dynamic reconfiguration), and real-time constraints. But what CSDF lacks the most to this end is the ability to cope with real-time constraints and uncertainties in data paths. This is what we want to discuss in the next section.

3 MoCs FOR THE FUTURE

3.1 Time Constraints

The methods for real-time design of applications is usually focused on the concept of tasks. Then, time constraints are given to the set of tasks, and can be applied to any execution step of any task in the set. Lot of real-time systems are reactive, so timing constraints are imposed between the time at which a given event occurs and the end of the execution of a given task. These systems are sometime called "*Event Triggered*". Others are strictly driven by time ticks that provide the rhythm of execution of the application, and allow the system to meet its deadlines. The latter case is called "*Time Triggered*" and yields either a periodic or a pseudo-periodic behavior.

To modelize reactive systems, we can utilize either actual real-time tasks or the hierarchical nature of DataFlow concepts to map the behavior of tasks. As the definition of KPN can be done in a hierarchical way, the time constraints in tasks are reported to some of the input and output communication channels of the CSDF graph associated with the RT task. This can be translated as a set of time constraints on execution and communication times that complements the constraints of a well-formed CSDF application. This way, time constraints can be inserted in SDF and CSDF as long as individual process execution times and communication times between processes are boundable, and the constraints are translated as sets of inequalities. For periodic tasks we can constrain some channels to have a periodical behavior also, based on the so-called "*repetition vectors*" of CSDF (the existence of CSDF is ensured for well-formed applications, as seen in Figure 2). After that we can obtain the activation period of the processes connected to the periodic communication channels with periodic activation which have minimal impact on latency and throughput. One possible model for this and its experimental evaluation was presented in (Dkhil et al., 2015).

3.2 Dynamicity and Determinism

Liveliness and Timeliness are two important properties of real-time systems. Liveliness in a well-formed CSDF is ensured as long as no source (I/O) and no communication channel is faulty. Correct scheduling policies as mentioned in section 3.1 help, but for critical systems this is not enough as they must work even in the case of faulty behavior of any part, including communication. So future MoCs should cope well with this situation. This is not the case of CSDF without adding further semantics (*e.g.* watchdogs within processes). So even without considering dynamic applications, CSDF must evolve to become more reliable.

Any part that may be considered as fragile for the behavior of the application must come with a safety policy to insure a good behavior. Several techniques exists for that: We already mentioned watchdogs, but redundancy is also a key to a dependable system.

For the future, more dynamicity and versatility than with usual CSDF would be required to permit a simpler expression of context-dependent or open and reconfigurable systems (IoT, ADAS, Captor Networks...). However, even one of the simplest dynamic enhancement of SDF which is Bolean

DataFlow or BDF (Buck and Lee, 1993a) do not provide decidable behavior in the general case. So nothing can be said *e.g.* about the absence of deadlocks. This is why keeping the CSDF equivalency as much as possible is so important.

Work in (Louise et al., 2014) shows how to utilize the concept of transaction to extent the expressiveness of CSDF while retaining its properties. As seen in Figure 3, this extension permits to manage redundancy, but also other important aspects of parallel and real-time systems like fault detection, deadline enforcement, speculation…

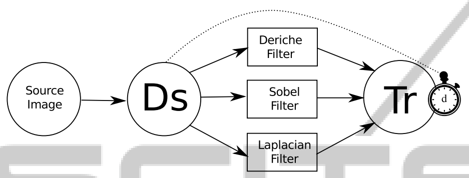
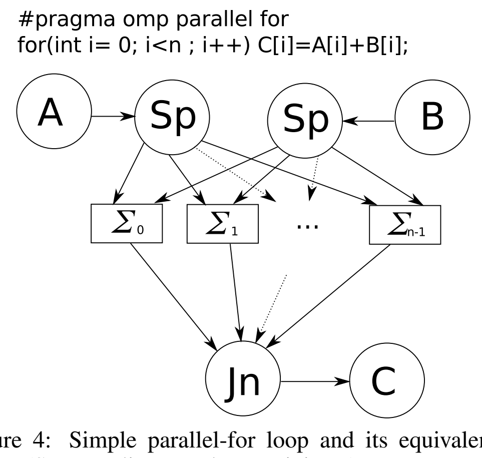


Figure 3: Edge detection in an image using redundant algorithms. Image is fed to 3 processing pipelines using different algorithms through a Duplicator. The result is chosen after a delay *d* by a Transaction.

3.3 Parallelism and Fine Granularity

The finer available parallelism is usually Instruction Level Parallelism (ILP) which is typically handled by (*e.g.*) a C-compiler to achieve the best performance on a given core of the system. But we can find some space in term of granularity of parallelism between dataflow and ILP. Even if it is theoretically possible, no programmer would want to specify all the available parallelism in term of DataFlow, because at the finer levels it becomes counter-productive and boring to specify parallelism this way. The *Forall* or *parallel for* type of parallelism as seen in *e.g.* OpenMP is a very compact way to define parallelism.



```
#pragma omp parallel for
for(int i= 0; i<n ; i++) C[i]=A[i]+B[i];
```

Figure 4: Simple parallel-for loop and its equivalent in CSDF (Sp are splitters and Jn are joiners).

With an evolution of a CSDF MoC as a base for Intermediate Representation, we can use the equivalency between *Parallel-For* constructs (without border effects) and simple CSDF Splitters/Joiners constructs, as seen in Figure 4, to automatically generate fine levels of parallelism. For more complex case *e.g.* parallel loops with border effects or other more elaborate OpenMP constructs we can imagine finding equivalent constructs in Split/Duplicate/Transaction/Join combinations, at least for a subset of usual kernel constructs (Herlihy and Moss, 1993).

3.4 Dependability

Dependability will be a growing concern for upcoming years. First, because more systems will be connected and recent trends in software security, safety and concerns about privacy will bring these themes at the foreground.

Good practices in system design and development with the help of quality assurance as seen in critical systems (*e.g.* DO178 for airplanes) can reduce the problem and minimize the impact on the users, but it requires a huge change in attitude toward designing and programming for mass-market devices. Safety and security can not be well designed and managed without impacts on performance, nonetheless, most of these impacts can often be localized in the tiny parts of the application which handle the interface with the outside world.

Regarding these points, CSDF based MoCs with the addition of watchdogs and transactions can handle parts of the safety design. Other aspects must be handled by the compilation tools and runtime support, by providing segmentation, isolation, and self-monitoring the results and the execution. Safety and security always rely in part on the programmer, but good tools can help a lot.

4 CONCLUSION AND OUTLOOKS

In this paper we try and describe the requirements for the future of Models of Computations (MoCs) for manycore systems. We argued CSDF provides a sweet spot as a base for such a MoC, but it lacks dynamicity, dependability, and time-management support. We showed some important directions of work in our opinion to cope well with these requirements. These aspects can also facilitate the problems of safety and security that arise in connected systems.

Of course deciding on a MoC does not mean the MoC should be transposed *verbatim* in a programming language. After all, even if assembly has no notion of functions or objects, lots of programming languages offer these concepts. But being a good MoC means a programming language can be compiled so that to conform to the implementation of the MoC in a Model of Execution on a given system. A good MoC should scale well from the fine granularities as parallel-for up to the real-time tasks. Using such a MoC would be a good step toward a "*universal*" Intermediate Representation for compiling tools of manycore systems, as it does not require locks or most of the hardware features that render timing so difficult in modern multicore and manycore systems.

The next step will be to verify all the required mathematical properties of the proposed MoC and build a generic Intermediate Representation upon it and then programming tools. Other works will be aimed at taking heterogeneous systems into account.

REFERENCES

Amarasinghe, S., Gordon, M. I., Karczmarek, M., Lin, J., Maze, D., Rabbah, R. M., and Thies, W. (2005). Language and compiler design for streaming applications. *International Journal of Parallel Programming*, 33(2/3):261–278.

Aubry, P., Beaucamps, P.-E., Blanc, F., Bodin, B., Carpov, S., Cudennec, L., David, V., Dore, P., Dubrulle, P., de Dinechin, B. D., Galea, F., Goubier, T., Harrand, M., Jones, S., Lesage, J.-D., Louise, S., Chaisemartin, N. M., Nguyen, T. H., Raynaud, X., and Sirdey, R. (2013). Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. In Alexandrov, V. N., Lees, M., Krzhizhanovskaya, V. V., Dongarra, J., and Sloot, P. M. A., editors, *ICCS*, volume 18 of *Procedia Computer Science*, pages 1624–1633. Elsevier.

Buck, I. (2004). Brook specification v.0.2. Technical report, Stanford University.

Buck, J. and Lee, E. (1993a). Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432 vol.1.

Buck, J. T. and Lee, E. A. (1993b). Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Technical report.

Campbell, M., Egerstedt, M., How, J. P., and Murray, R. M. (2010). Autonomous driving in urban environments: approaches, lessons and challenges. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 368(1928):4649–4672.

Choi, B., Komuravelli, R., Sung, H., Smolinski, R., Honarmand, N., Adve, S., Adve, V., Carter, N., and Chou, C.-T. (2011). Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 155–166.

de Oliveira Castro, P., Louise, S., and Barthou, D. (2010). Reducing memory requirements of stream programs by graph transformations. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 171 –180.

Dkhil, A., Do, X.-K., Louise, S., and Rochange, C. (2015). A hybrid algorithm based on self-timed and periodic scheduling for embedded streaming applications. In *Proceedings of Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (EuroPDP 2015)*.

Feiner, S., MacIntyre, B., Hollerer, T., and Webster, A. (1997). A touring machine: prototyping 3d mobile augmented reality systems for exploring the urban environment. In *Wearable Computers, 1997. Digest of Papers., First International Symposium on*, pages 74 –81.

G. Bilsen, M. Engels, R. L. and Peperstraete, J. A. (1996). Cyclo-static data flow. *IEEE Transactions on Signal Processing*, 44(2):397–408.

Guizzo, E. (2011). How googles self-driving car works. *IEEE Spectrum Online, October*, 18.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533.

Herlihy, M. and Moss, J. E. B. (1993). *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475.

Khronos OpenCl Working Group (2008). The opencl specification. Technical report.

Lee, E. A. (2006). The problem with threads. *Computer*, 39(5):33–42.

Louise, S., Dubrulle, P., and Goubier, T. (2014). A model of computation for real-time applications on embedded manycores. In *Embedded Multicore/Manycore SoCs (MCSoc), 2014 IEEE 8th International Symposium on*, pages 333–340.

Murillo, A., Gutierrez-Gomez, D., Rituerto, A., Puig, L., and Guerrero, J. (2012). Wearable omnidirectional vision system for personal localization and guidance. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2012 IEEE Computer Society Conference on*, pages 8 –14.

NVIDIA Corp. (2007). NVIDIA CUDA: Compute unified device architecture. Technical report.

Sato, Y., Nakamoto, M., Tamaki, Y., Sasama, T., Sakita, I., Nakajima, Y., Monden, M., and Tamura, S. (1998). Image guidance of breast cancer surgery using 3-d ultrasound images and augmented reality visualization. *Medical Imaging, IEEE Transactions on*, 17(5):681 – 693.

Stuijk, S. (2007). Predictable mapping of streaming applications on multiprocessors. In *Phd thesis*.