Procedia Computer Science

Volume 29, 2014, Pages 1100–1112

ICCS 2014. 14th International Conference on Computational Science

τC: C with Process Network Extensions for Embedded Manycores

Thierry Goubier[1], Damien Couroussé[2], and Selma Azaiez[1]

[1] CEA, LIST, DACLE/Lastre, 91191 Gif-sur-Yvette CEDEX, France
thierry.goubier@cea.fr,selma.azaiez@cea.fr
[2] CEA, LIST DACLE/LIALP, Grenoble, France, damien.courousse@cea.fr

Abstract

Current and future embedded manycore systems bring complex and heterogeneous architectures with a large number of processing cores, making both parallel programming at this scale and understanding the architecture itself a daunting task. Process Networks and other dataflow based Models of Computation (MoC) are a good base to present a universal model of the underlying manycore architectures to the programmer. If a language exposes a simple to grasp MoC in a consistent way across architectures, the programmer can concentrate the efforts on optimizing the expression of parallelism in the application instead of porting code to a given system. In this paper, we present a process network extension to C called τC and its mapping to both a POSIX target and the P2012/STHORM platform, and show how the language offers an architecture independent solution of this problem.

Keywords: manycore, programming model, process networks, compiler toolchain

1 Introduction

Embedded manycores, *i.e.* systems-on-chip with over a hundred general purpose cores, are more and more related to full scale computing farms. They typically employ a mix of shared and local memory, often distributed, with also a network on chip (NoC) to enable communication between cores or clusters of cores. Compared to their bigger brethren, they provide very limited memory and are subject to strict dependability and performance constraints (*e.g.* guaranteed performance) most of the time without memory coherence, because of tight power envelops.

As a consequence, developing for those targets suppose handling the following difficulties: meeting performance and dependability requirements within limited resources; running correctly large parallel programs; and exploiting efficiently the underlying parallel architecture, which sports both the lack of hardware abstraction layers (*i.e.* especially operating systems) and hardware support for advanced system support (threading, memory management, *etc.*).

To target with reasonable efficiency such systems, a possible path is to provide a safe, high level parallel Model of Computation (MoC) thanks to a programming language and a compiler

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2014

doi:10.1016/j.procs.2014.05.099

able to target (several) manycore systems. We have chosen process networks as a suitable MoC for the following reasons: its amenability to strict timing and safety requirements when suitably restricted, its simplicity, its ability to provide a high-level abstraction on the hardware, and the feasibility of an efficient compilation process for it.

We show in this paper how such a model is added to C with sound software engineering principles, maintaining compatibility with existing code, and how its compiler and abstraction layer are implemented to target different platforms (homogeneous/heterogeneous) such as a POSIX Threads machine on one hand and the MCAPI communication API for the P2012/STHORM manycore, on the other hand. In this paper, after a short section of related works, we will describe the τC model and syntax, before detailing its compiler implementation for both targets.

2 Related Works

The underlying MoC of τC can be put in perspective with the revival of dataflow models and process network models [15]. It belongs to the class of Process Networks (PN), or Kahn process networks [12]. As a language, τC is an extension to C; alike Cilk [6], OpenCL [13] and XC [19], it adds a parallel programming model to the C language. It does so by adding new constructs to C (similarly to XC [19], and Brook [7]), avoids pragmas (as done the OpenStream effort [3]) and does not defines a new programming language (as done by StreamIt [2]). τC was first described in a technical report for the Ther@ops project [10]. Core components of the τC programming model are similar but a bit less flexible than the core parallel patterns of FastFlow [1]; FastFlow SPSC are certainly faster than the τC POSIX Threads target, however τC is targeting embedded manycores.

ΣC [11] is another dataflow language for manycores developed in collaboration between CEA-LIST and Kalray. Our implementation of τC was done with very little support by a reduced team and thus can hardly be compared to the full fledged implementation of ΣC [4]. Nonetheless, there is no forecast issues in doing the same level of optimizations with τC as was done with ΣC (automatic partitioning, *etc.*). On language design, the expressiveness of τC is more extensive (generic process networks) than ΣC whose MoC is limited to a deterministic superset of CSDF.

3 τC : process network extensions to C

The programming language τC can be seen as an experiment in programming language design: how to add to C the right extensions to be able to build process networks, and with the right abstractions for sound software engineering. The main hypothesis for that approach is that a programming language is a user interface between a machine and a programmer. The ability of the programmer to build complex and efficient parallel programs on an embedded manycore system is dependent on his capacity to construct a mental model of its program out of the source code and the compiler feedback.

This hypothesis may be reflected in a language design, a choice of programming model or an API design. In this paper, we focus on the programming model and the language design.

By construction, a process network model reduces parallelism, synchronization and communication to two concepts: single threaded processes and point to point communications with blocking read/non blocking write. This is a parallel programming model which is surprisingly simple to learn and understand. Additionally, a property of process networks is that, if all processes are deterministic, then the whole process network is also deterministic. A consequence

of that is that any error or incorrect behavior of such a deterministic process network is reproducible given the same input, making understanding faults in this class of parallel programs a lot easier on the programmer. This is therefore a programmer-oriented feature of this class of programming models.

The language aims at two objectives: being familiar and efficient for embedded development, and materialize the MoC concepts correctly. Our choice here is to extend C, hence giving us a familiar and fairly simple syntax and very efficient compilers for all embedded manycores, whatever the degree of complexity of their architecture.

To materialize MoC concepts in the language, our decision was to minimize the changes in the C grammar, while introducing the needed components. The result is a hierarchical component model whose base element is the *task* (a process), an additional derived type called an *interface*, and statements to be able to express behavior sequences inside tasks. The hierarchical construct is also an important point to lay the base for good design and engineering practices when programmers implement applications on manycores. We will describe the model before describing the language extensions in the following subsections.

3.1 Parallel programming model

The basic unit of the programming model is called a task. It is an independent process, with one thread of execution and its own address space. It communicates through point to point, unidirectional, typed links. Communication through links is done in blocking read, non-blocking write fashion, and the link buffers are considered large enough to ensure execution.

An application is a graph of interconnected tasks. The graph is static and is typically embedded in the executable. The model is hierarchical at the language level, with composition handled by modules, a module being composed of tasks. A top-level task or module called main defines the application.

Inside tasks, we make explicit the fact the execution paradigm is an automaton, by introducing state functions and a statement to control which state function is active.

Data distribution and control is typically handled by specific tasks, namely Split, Join, Sink, to ease their implementation using dedicated hardware if appropriate.

Input / Ouput is handled by using the underlying system libraries; making API calls as provided. τC does not require specific agents types for handling I/O and system interaction, unless the target requires, one way or another, those nodes to run on specific parts of the architecture (for example, io-dedicated processors).

An important feature of the τC language, not used yet, is the ability to extract by source code analysis the state machine of a task. Pragmatically, it is feasible to consider a compiler targeting a certain level of safety in design by rejecting code where building this state machine can't be done. Once a state machine for each task in the program is known, it is then possible to compute the class of restricted process network we are dealing with (*e.g.* SDF [14], CSDF) and apply the relevant analysis (static scheduling for SDF, etc...). Literature shows that a class such as CSDF has both tractable analysis and strong expressive power [8] which makes it a convenient target.

3.2 C-language extensions in τC

The extensions are limited to a type, two function definition variants and a statement; this represents about 20 production rules when added to a standard LALR C grammar. Listing 1 illustrates the language elements of τC presented below. It is taken from the real application presented in section 6.

Interfaces and ports The first extension is a type, called interface. It describes the ports of a component, and is written as a sequence of **type : port** where *ports* may be arrays (with or without size), and *types* may be any type except a pointer[1]. The following example defines an interface named `ISplit` with one input port of type `char` named `input`, and an array of output ports of type `char` named `output`. The colon is here to make the difference between a port of type array of `char` and an array of port of type `char`. The chosen syntax is simple and quite common for example in actor languages or stream-oriented languages (StreamIt [2]).

```
typedef ISplit : (char : input) -> (char : output[]);
```

A feature of this syntax is that the size of a ports array may be undefined; the effective number of ports in the array should be defined at the point where the task is instantiated in the source code. This provides for parametric task definitions.

Task functions Tasks and modules are defined by task functions, that is C functions with *type* interface returning respectively task or module. The following example defines a prototype of a task (factory method pattern) with a compatible interface to the `ISplit` defined above. `n` is a parameter to the task function that specifies the array size of the `output` interface.

```
task Split(int n, int k) : (char : input) -> (char : output[n]);
```

State functions A task contains the behavior of the process. To be able to analyze it and to make explicit the fact that processes tend to be state machines, the behavior of a task is written as a set of state functions, that is, functions returning void and having a specific interface type mapping ports to variables with amounts.

The syntax then becomes the following: `i <- input` means that `i` will be defined in the state function, and that it will contain a single element of the type defined for input. For such a variable to read or write multiple elements, then it has to be written `i[n] <- input`, where n can be a parameter of the state function.

The default state function is `main()`; it is called by the task at startup.

A default `exit` state function is also provided. Once called it performs a clean shutdown of the task and propagates application termination over the network of processes.

`next` statement A single statement, `next function_name()`, allows for exiting the state function and selecting a different state function of the automaton.

Modules Modules have the same syntax as tasks, except that they do not contain state functions; instead, they contain topology building code with the following API: creating a task instance by calling the task function, and connecting two ports by an affectation between members of the two tasks (as per their interface). For example, a very simple module would then be:

```
module main() : () -> () {
       Reader()->input = Sender()->output;
}
```

This module creates a `Reader` instance, a `Sender` instance and connects them, by saying the `Reader` instance `input` port is connected to the `Sender` instance `output` port. It is also possible to explicitly allocate variables to hold the tasks. In this case, we would have the following code:

[1]Because we cannot make hypothesis of a uniform memory mapping viewable from any core of the chip in a general embedded manycore

```
typedef iReadFrames : () -> (Pixel : frame, Pixel : prev);
task ReadFrames(char * file, int n, int start, int width, int height) : iReadFrames {
    int i = start;
    void process() : (pFrame[width*height] <- frame, pPrevious[width*height] <- prev) {
        loadBMPData(pFrame, file, i);
        memcpy(pPrevious, pFrame, width*height*sizeof(Pixel));
        ++i;
        next (i < n ? process() : exit());
    }
    void main() : (pFrame[width*height] <- prev) {
        loadBMPData(pFrame, file, i);
        ++i;
        next process();
    }
}
```

Listing 1: τC example of a task implementation; from the target tracking sample application presented in section 6.

```
module main() : () -> () {
       task s : () -> (float : output) = Sender();
       task r : (float : input) -> () = Reader();
       r->input = s->output;
}
```

Modules contain instantiation code and tasks contain execution code. This allows for decoupling the instantiation code and the execution code and enables pruning optimizations for the final executable (removal of all symbols and code used only during instantiation, as in ΣC [11] and Virgil [18]). Input/Output code and system interaction may be used in tasks and modules alike.

Application entry point An application requires a `main` module as a top-level element. Tasks can only be created by code inside modules, and communications may only happen by activating state functions in tasks. State functions can only execute if the data they require is present on the ports buffers, as expected from the blocking read property of the model.

Summary The extensions are simple. There is no send / receive primitives, no specific API for topology building, correct encapsulation by interfaces, no restriction over the topology of process networks thus harnessing their full expressive power (with the ability to build non-deterministic process networks), and no non-obvious non-determinism in task behavior (in τC, the programmer decides which behavior is activated next; to compare with the CAL [9] language where it is the reverse, unless the programmer add guards to emulate the state machine).

3.3 A path from legacy C code

One of the design goals of τC is to make it possible to easily port and tune legacy C code, especially functional prototypes for algorithms that will run on manycores. According to the definition of the language, it can be done by applying the following rules: variables containing data exchanged between tasks are normal C types (arrays) and may be used as parameters

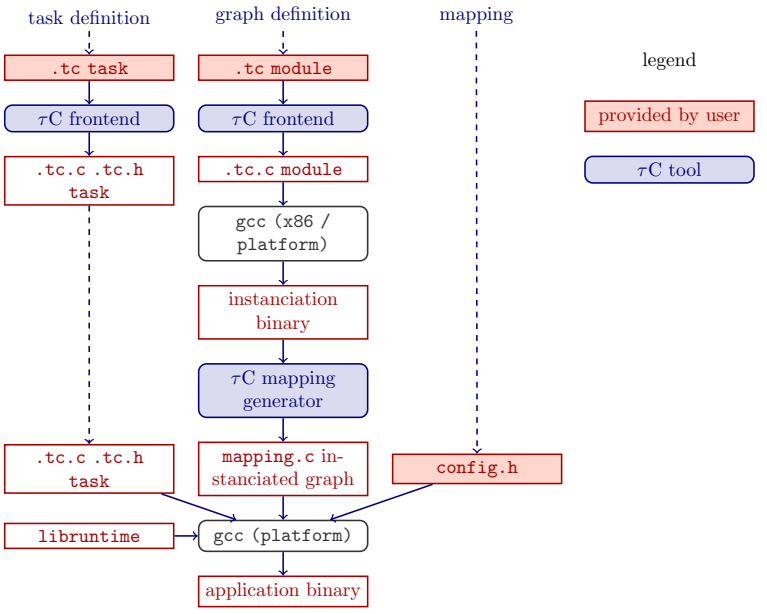

Figure 1: the code generation flow

to C functions. C algorithms run without changes in tasks. As tasks aren't limited in code size, there is no need to build the best parallelization of the application in a first approach: It is possible and desirable to get the low hanging fruits for parallel expression by concentrating porting efforts to the most relevant parts at first.

Tasks and modules are interchangeable: any entity with the same interface can be instantiated in lieu of another. Hence a pipeline of tasks may see one of its stage replaced by an optimized, parallelized combination of tasks in a module. Therefore, the porting efforts can be made in an incremental manner. Moreover, static typing with interfaces in the instantiation code in modules ensures easy verification of the validity of connections.

4 A sketch of the τC Compilation Process

A τC application is composed of three types of items: tasks, modules, and application mapping (figure 1). τC tasks and modules are language elements that are processed in a different manner in the code generation flow.

- τC tasks define the implementation of application processes. Each task implementation is translated to plain C by the τC compiler frontend.
- τC modules define how tasks are connected together in an application. The language elements that are specific to τC must also be translated in plain C, but the main part of the work for modules consists in expressing the instantiated (unfolded) application graph from the module descriptions.

The application mapping is not explicitly present in the implementation of an application. It is built during the code generation process from (1) the instantiated graph of the application,

obtained from the implementation of the modules and (2) a configuration file (provided by the user) that describes the placement of tasks on the target platform. This is not the scope of this work, but thanks to the properties of the τC language allowing for automatic analysis, it is possible to have the configuration file generated from automatic tools.

The τC compiler chain, which is still under development, is built around four passes and uses C as its back-end language. The first pass, the τC front-end, runs on a single compilation unit (one file plus associated headers). Its main purpose is to perform a lexical, syntactic and semantic analysis of the τC code, to perform one level of specification consistency verification as well as to generate preliminary C code for either off-line execution or further refinement.

The second pass, which is part of the τC middle-end, deals with task instantiation and connection, by putting together (possibly in an iterative fashion) the codes generated to that end by the front-end. Once the application graph is complete, a number of parallelism reduction heuristics are applied so as to tailor the application to an abstract specification of the platform resource capacities. The second pass subsequently computes a minimal deadlock-free dimensioning of the buffer and generates additional code related to task and link instances.

The third pass (which also belongs to the middle-end) raison d'être is to perform resource allocation at the system level. In particular, this encompasses real-time constraint-driven buffer dimensioning as well as allocation of tasks to computing resources (cores, clusters, etc. depending on the architecture) and NoC configuration (if appropriate). This pass can be performed in a feedback-directed fashion so as to achieve an appropriate level of performance.

The last pass, the τC back-end, is in charge of generating the final C code as well as the runtime tables which make the link with the target execution support. Using the C back-end tools, the τC back-end is also in charge of link edition as well as loadbuild, a process which usually involves many subtleties on multi and manycore platforms.

5 Runtime support

5.1 Target platforms

We implemented a partial version of the full scale compilation support whose architecture was discussed in section 4. It is above all a working prototype, in which the frontend compiler (pass 1), a fair chunk of the middle end (pass 2), and the runtime generator (pass 4) are implemented.

Our main target was the P2012/STHORM platform [5], and we also targeted standard development stations supporting POSIX Pthreads for easier prototyping and debugging purposes. STHORM is a large scale, scalable multi-core fabric, developped by STMicroelectronics and CEA. It is composed of 4 clusters communicating via an asynchronous Network-on-Chip, allowing each cluster to have its own voltage-frequency domain. Each cluster is featuring 16 processing elements (PEs) and one processor for task control. The platform also contains a fifth cluster only composed of one processor dedicated to task control at the platform level. Clusters each have 256 KB of onchip shared memory. The program memory is located onchip in a shared memory outside of the clusters area. An external memory is also available, offchip, with larger communication delays. This memory is accessible from the platform processors and from the host. All the memories are visible via a global memory map, which eases the possibility of inter-cluster communications. Each cluster also comes with DMA channels for inter-cluster communication and communication with the host.

The work of design and implementation of the code generation toolchain was not dependent of a particular platform architecture. In order to decouple the problems of code generation and the adherence with a particular runtime API, we implemented a simple runtime library designed

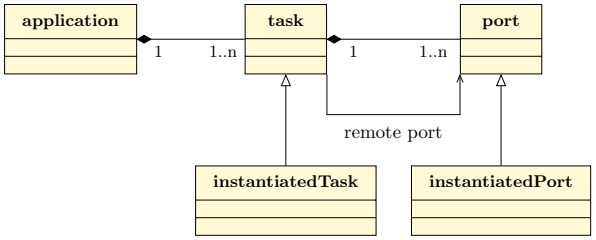

Figure 2: UML overview of the runtime elements

as a wrapper between the generic C code generated by the τC toolchain and the target platform. This runtime library provides a unified interface able to target POSIX Pthreads systems on one hand and MCAPI systems [17] on the other hand. We used the implementation done by CEA for the STHORM platform [16]. In this implementation, MCAPI domains correspond to STHORM clusters; we associate τC tasks to MCAPI nodes.

5.2 General runtime interface

5.2.1 Tasks and ports

The applications produced by our toolchain must be able to run on parallel architectures that do not present a central resource for task management and resource allocation (like STHORM). As a consequence, a primary concern for the code produced by our toolchain was to produce autonomous software entities, that do not require the need to access to a centralized service. In other words, a τC application is only the aggregation of `task` objects, each of them only requiring access to its attributes, and all the application lifecycle, including initialization and termination, is done in a full parallel fashion.

Figure 2 presents a UML overview of the types of objects that are executed at runtime. A τC application is the composition of `task` objects. Each task object instantiated by the toolchain inherits from the generic `task` type. A `task` is composed of `port` objects, and ports are instantiated by the toolchain from the generic `port` type. Task attributes are statically initialized during code generation.

Task and port inheritance also allows us to introduce platform-dependent attributes. For example, MCAPI domains where associated to STHORM clusters. Each task must be associated to a particular MCAPI domain so that it can be correctly instantiated. On POSIX Pthreads platforms, we do no need such information.

5.2.2 Communication channels

Communication channels are built from two instantiated ports that belong to two different tasks (although there are no restrictions in the language for communication channels build from two ports belonging to the same task). Each task is responsible for creating and initializing its own ports. The communication channel is effectively active once the two ports have been associated. In order to self-contain the phase of application startup in tasks, it was necessary to add information about remote ports in tasks. Figure 2 depicts this relation: in addition to the ports associated to a task, a task also has knowledge about the remote ports the task

```
task Sender() : () -> (float : output)
{
  int count = 0;
  void main() : (out <- output) {
    out = (float) count++;
    if(count > 10) next exit();
  }
}
```

(a) τC source code

```
void*__tc_task_Sender(void*arg)
{
  int count=0;
  __tc_func_main: {
    float*out;
    chan_write_lock(&t->output, &out);
    (*out)= (float) count++;

    if (count>10)
      goto __tc_state_exit(0);
    chan_write_release(&t->output);
  }
}
```

(b) The corresponding generated code (simplified)

Listing 2: Exemple of simple `Sender` task: τC source code and generated code

will communicate to. Our internal API for communication channels would typically provide functions similar to:

- `void * chan_read(chan_t * chan, size_t size);` for buffer reading, and
- `size_t chan_write(chan_t * chan, void * buf);` for buffer writing.

Such channel functions lead to memory copy of the buffer contents into or from another memory buffer allocated by the callee. An advantage however of such solution is that the management of shared accesses to the memory buffers is done inside the channel functions rather than in the calling code.

We designed a special API to implement zero-copy accesses to the communication buffers. The main drawback of this API is that if care is not taken to release the read or write lock on the channel, the application could end into a deadlock. In our case, such situation cannot occur because the channel API is only be targeted by the code generation toolchain, and not directly used by the programmer. To illustrate this point, listing 2 exemples the τC code of a `Sender` task (2a) and the corresponding generated code by our toolchain, in a simplified form (2b). The write access to port `output`, aliased `out`, of task `Sender` in listing 2a is split in three steps: (1) requesting a write lock to the port `output`; a copy of the buffer address is copied into `out`, (2) reproducing the pointer operations as found in the original τC source code (3) closing the write access to port `output`.

5.2.3 Application startup

As the compilation steps of the reference compiler (as described in section 4) were not completely implemented, the instantiation of processes is done online with data statically generated during pass 2 of the compiler. Application startup is complete once each task has: (1) created and initialized its ports, (2) established connections with remote ports, (3) opened accesses to its ports to enable communications

Tasks starts their processing job as soon as their initialization process is done, even if some other tasks still have initialization operations pending.

6 Using τC

As an illustration, we used one of our own application, a target tracking video processing algorithm. We used a reference implementation in ΣC and translated it very easily in τC. It exposes a common way of working with τC: all the processing code is held in a C library, and the main application loop is transcribed from C to τC. The process is to take the C code chunks in the top level loop (listing 3), rewrite that function in a task (listing 4) and add the creation of the task in the main graph (listing 5).

The end result is, in the current state of the prototype, compilable to our two targets (MCAPI on STHORM and POSIX Threads), including the offline instanciation phase, resulting in the production of the process graph in figure 3. Note that in that particular case, the width of the graph (the number of parallel processing threads over the video) is a command line parameter of the top-level process in the τC code (and is written as you would read a command line parameter in a C program).

We also use a simulator of the τC toolchain as a framework for students of Master[2] education about working and programming with process networks (lecture and courses parts of the curiculum of University of Bretagne Occidentale –UBO). We have done so for two years (about 40 students), so far, and the feedback of student is excellent both on the illustrative power of using the language as course support, but also as a simple means to start process network programming.

7 Conclusion

We have presented the τC programming language and model, and its implementation on two targets: POSIX Threads and MCAPI on P2012/STHORM embedded manycore.

The language is a limited and simple set of extensions over C, with a parsing and generation of intermediate code in C, helped by an abstraction layer to isolate from the differences between the two targets platform. The parallel programming model is simple, has interesting properties both from the software engineering side and from the possibility of analysis offered by subsets of this model. Finally, this is well suited to platforms such as the P2012/STHORM embedded manycore, where the language and its compiler enables a high degree of isolation from the specifics of the architecture while maintaining a high degree of parallelism and efficiency.

Future directions for the language are to explore some extensions to the model, integrate a level of control flow graph analysis on the network, and improve the code generation and mapping to take in account more information on the target architecture.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: high-level and efficient streaming on multi-core. In Sabri Pllana and Fatos Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, March 2014.

[2] Saman Amarasinghe, Michael I. Gordon, Michal Karczmarek, Jasper Lin, David Maze, Rodric M. Rabbah, and William Thies. Language and compiler design for streaming applications. *International Journal of Parallel Programming*, 33(2/3):261–278, Jun 2005.

[3] Albert Cohen Antoniu Pop. Control-driven data flow. Technical Report 8015, Inria/Project-Team Parkas, 2012.

[2] University's graduate school.

[4] Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Loïc Cudennec, Vincent David, Philippe Dore, Paul Dubrulle, Benot Dupont de Dinechin, François Galea, Thierry Goubier, Michel Harrand, Samuel Jones, Jean-Denis Lesage, Stéphane Louise, Nicolas Morey Chaisemartin, Thanh Hai Nguyen, Xavier Raynaud, and Renaud Sirdey. Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18(0):1624 – 1633, 2013.

[5] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 983 –987, march 2012.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING*, pages 207–216, 1995.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques*, pages 777–786. ACM New York, NY, USA, 2004.

[8] Kristof Denolf, Marco Bekooij, Johan Cockx, Diederik Verkest, and Henk Corporaal. Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *EURASIP Journal on Advances in Signal Processing*, 2007(1):084078, 2007.

[9] Johan Eker and Jrn W. Janneck. CAL language report: Specification of the CAL actor language. Technical Report UCB/ERL M03/48, University of California, Berkeley, CA, 94720, USA, 2012.

[10] Thierry Goubier, Frédéric Blanc, Stéphane Louise, Renaud Sirdey, and Vincent David. τC: Définition. Technical Report DTSI/SARC/08-285/TG, CEA List, 2008.

[11] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. ΣC*: A Programming Model and Language for Embedded Manycores*, volume 7016 of *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin Heidelberg, 2011.

[12] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974.

[13] Khronos OpenCl Working Group. The Opencl specification v1.1. Technical report, 2011.

[14] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.

[15] Edward A. Lee and Thomas Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, 1995.

[16] Julien Mottin and Marius Bozga. Porting applications to multicore platforms: Results from the BIP & MCAPI tool chain for STHORM. In *Platform 2012 / STHORM workshop – DATE conference*, Grenoble, France, March 2013.

[17] The Multicore Association. MCAPI API Specification. Technical Report V2.015, The Multicore Association, Inc, PO Box 4854 El Dorado Hills, CA 95762, March 2011.

[18] Ben L. Titzer. Virgil: Objects on the head of a pin. In *Proceedings of the 21st Annual Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA '06)*, 2006.

[19] D. Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009.

```
void *minVariance(int width,int height,Pixel *pDiff,
               Deviation *pDeviation,int *pMean) {
       computeDeviationMacroBlock(width,height,pDiff,8,pDeviation);
       computeMeanPicture(width,height,pDiff,pMean);
}
```

Listing 3: target tracing sample application: C source code extract for some motion preprocessing code.

```
typedef iMinVarianceStrip : (Pixel : diffStrip)
                            -> (Deviation : deviation, int : mean);

task MinVarianceStrip(int width, int height) : iMinVarianceStrip {
    void main() : ( pDiffStrip[width * height] <- diffStrip,
                    pDeviation <- deviation, pMean <- mean) {
        computeDeviationMacroBlock(width, height, pDiffStrip, 8, &pDeviation);
        computeMeanPicture(width, height, pDiffStrip, &pMean);
    }
}
```

Listing 4: target tracing sample application: τC MinVarianceStrip task as ported from listing 3.

```
...
for(i = 0; i < nb_strips; i++) {
       ...
       task aMinVarianceProcess : iMinVarianceStrip =
              MinVarianceStrip(width, height / nb_strips);
       aGMean->means[i] = aMinVarianceProcess->deviation;
       ...
}
```

Listing 5: target tracing sample application: instanciation of an instance of the MinVarianceStrip task (listing 4) in the main module.

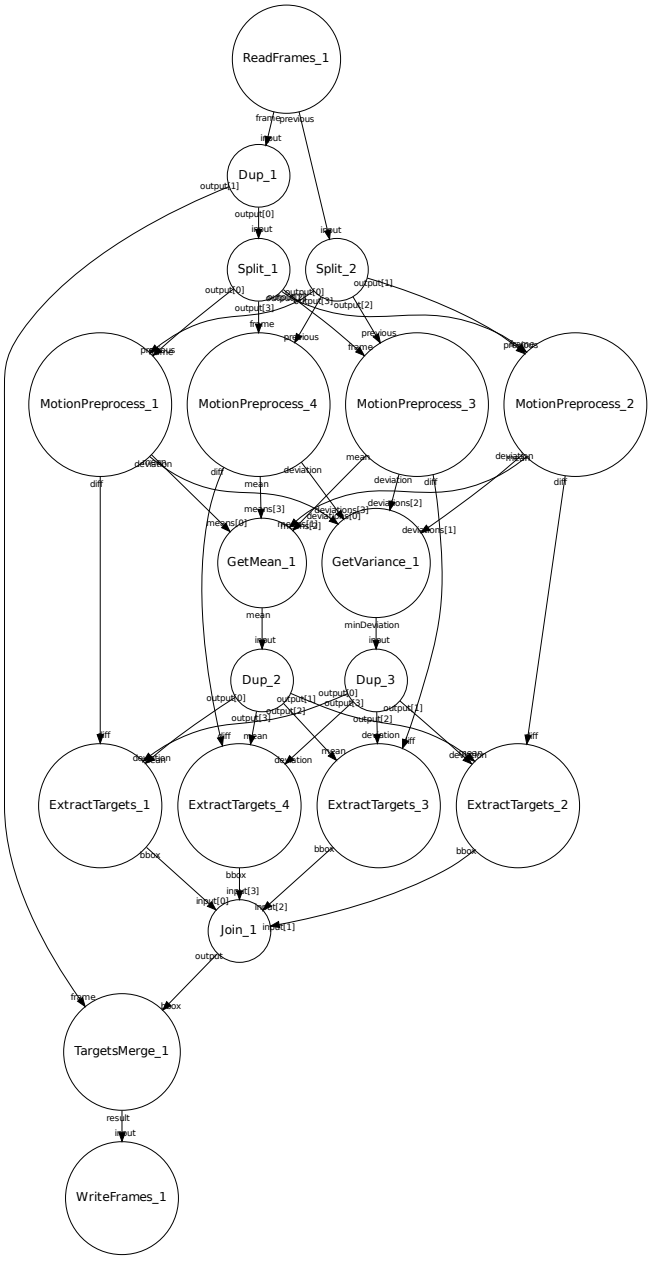

Figure 3: target tracing sample application: the process graph generated for the application with runtime parameter 4 for the number of parallel strips over the video sequence. Note that in this graph, the MinVarianceTask (listing 4) has been merged in a MotionPreprocess task.