ΣC: A Programming Model and Language for Embedded Manycores

Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David

CEA, LIST, Embedded Real-Time Systems Lab
Mail Box 94, F-91191 Gif-sur-Yvette Cedex, France
name.surname@cea.fr

Abstract. We present ΣC, a programming model and language for high performance embedded manycores. The programming model is based on process networks with non determinism extensions and process behavior specifications. The language itself extends C, with parallelism, composition and process abstractions. It is intended to support architecture independent, high-level parallel programming on embedded manycores, and allows for both low execution overhead and strong execution guarantees. ΣC is being developed as part of an industry-grade tool chain for a high performance embedded manycore architecture.

Keywords: programming model, programming language, embedded manycores, embedded high performance computing.

1 Introduction

Programming embedded systems is a difficult task and so is parallel programming. Embedded manycores, that is systems-on-chip with over a hundred general purpose cores, are full scale parallel machines, typically employing a mix of shared and local memory, distributed global memory or multilevel cache hierarchy, and a network on chip (NoC) to enable communication between cores. Compared to their full scale brethren, they provide a limited amount of memory per core, no guarantee on memory coherency and are subject to strict dependability and performance constraints (e.g., guaranteed performance at peak utilization or close to peak).

As a consequence, developing for those targets suppose handling simultaneously the following three difficulties: meeting performance and dependability requirements subject to limited resources, running correctly large parallel programs, as well as exploiting efficiently the underlying parallel architectures. To render this task manageable and cost-effective, we have identified the following set of requirements for a programming model and language suitable for manycores:

- Ability to handle a variety of algorithms, both data flow (streaming) and control oriented at least from the fields of signal and image processing.
- Familiarity to embedded developers, that is similarity to C and ability to integrate efficiently with existing C code.

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 385–394, 2011.

- A compiler able to prove that the final executable is guaranteed to execute in bounded memory and that any run is reproducible.
- A development tool chain able to support good programming practices, modularity, encapsulation and code reuse.
- A run time and support microkernel fit for embedded manycores.

The main result described in this paper is a well defined model of computation, and the ΣC programming language which implements it. The paper is organized as follows. We first position our model with respect to the embedded models of computation and parallelism. We then detail the programming model, the language and the compilation process, before a short overview of the techniques used to guarantee execution in bounded memory, as well as the points for compilation optimization as currently implemented.

2 Related Work

ΣC as a programming model takes place among the numerous works done in the field of embedded models of computation[1]. The programming model belongs to the class of Process Networks (PN), or Kahn process networks (KPN)[2]. One of the main trade-off made during the design of ΣC, in this space, is the amenability to formal analysis, the field being split between models with restricted expressive power for which interesting properties such as deadlock freeness and bounded memory are decidable (SDF or Synchronous DataFlow[3], CSDF or Cyclo-Static DataFlow[4], HDF or Heterochronous DataFlow[5]) and models with increased expressive power such as the aforementioned process networks, Boolean DataFlow (BDF) and others, which come at the cost of Turing-completeness. The ΣC programming model trade-off consist in increasing the expressive power above SDF and CSDF while being less general than non deterministic process networks or BDF; in that following a similar constructive approach as [6].

The ΣC programming model is a subset of a process network model of computation with non deterministic extensions, of sufficient expressive power for most applications, while maintaining the possibility of tractably performing a formal analysis, for example using the well-known VASS or Petri net formalisms[7].

As a programming language, ΣC relates to StreamIt[8], Brook[9], XC[10], and more recently OpenCL[11], all programming languages, either new or extensions to existing programming languages, able to describe parallel programs in a stream oriented model of computation (CSP for XC).

ΣC differs by extending C without restrictions on the type of C supported or changes in the semantics of C apart from the extensions; the ΣC compiler parses standard C and support computational kernels of any level of complexity and call depth. It is high-level: no mention of the memory hierarchy or chip layout is necessary in the source code. It supports proving guaranteed execution in bounded memory over non-deterministic process network topologies. It supports stand-alone systems, with no host and no operating system, on an embedded manycore target with tens of Kbytes of RAM per core and over a thousand cores.

The implementation as a programming language extension provides strong type checking, scoping, modularity and correctness at the source code level.

3 The ΣC Programming Model

3.1 Components

The basic unit of the programming model is called an agent. It is an independent process, with one thread of execution and its own address space. It communicates through point to point, unidirectional, typed links behaving as fifos.

Communication through links is done in blocking read, non-blocking write fashion, and the link buffers are considered large enough. Agents have an interface, that is a list of typed and oriented ports to which links may connect, and a behavior specification.

An application, for example in figure 1, is a graph of interconnected agents, with an entry point, the `root` agent. The graph is static, and does not evolve through time (no agent creation or destruction, no change to the topology, during the execution of an application).

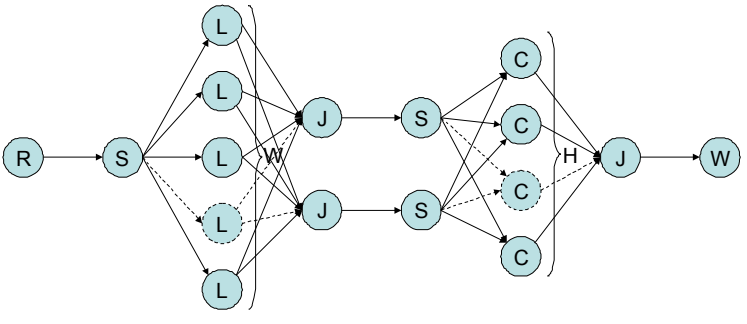

Fig. 1. Process network for a Laplacian computation (Huertas-Médioni operator), showing line (L) and column (C) filter agents

System agents ensure distribution of data and control, as well as interactions with external devices. Data distribution agents are Split, Join (distribute or merge data in round robin fashion over respectively their output ports / their input ports), Dup (duplicate input data over all output ports) and Sink (consume all data).

3.2 Behavior

The communication behavior of each agent is specified. It is a cyclic state machine with variable amounts of data. Each transition specifies a fixed or variable amount of data consumed on in ports, and a fixed or variable amount of data produced on out ports of this agent. All variable amounts are bounded ranges.

An example is a run length decoding agent with input and output ports of type unsigned char, described by : `{{input[2]}; {output[1:256]}}`. This specification means that this agent consumes two tokens on its input in one transition, and then produces 1 to 256 tokens on its output in the second transition, before looping back.

Data-dependent control in the process graph is introduced through `Select` and `Merge` agents which take, in addition to their data links, a control link selecting which input or output link is activated. `Select` and `Merge` are evaluated as simulation of execution of the branch not taken (and execution of the branch taken), so as to ensure correct execution of agents connected to the branch. In this semantic, a branch not taken is seen as being executed for its environment.

3.3 Designing for Execution Guarantees

As already emphasized, the ΣC programming model has been designed with amenability to formal analysis in mind, in particular with respect to properties such as absence of deadlock and memory bounded execution. In essence, formal analysis is performed on the basis of both the topological and behavioral specifications provided by the programmer.

In particular, being a special case of KPN, the ΣC programming model inherits their nice properties with respect to determinism and monotony. This has interesting consequences on the problem of safely dimensioning the statically dimensioned communication buffers: as long as one finds a deadlock-free buffer size solution for all links (preferably small with respect to an adequate objective function), any other solution with non-smaller buffers is also deadlock-free. This thus allows one to decouple the problems of proving deadlock-freeness and of tuning the buffer dimensioning so as to achieve a high level of performance.

At the programming language level (see below), the programming model appears hierarchical, with behavior specifications and interfaces. This aspect is designed to support hierarchical, divide-and-conquer analysis of minimum buffer bounds and deadlock-free solutions.

4 The ΣC Programming Language

The ΣC programming language is designed as an extension to C. It adds to C the ability to express and instantiate agents, links, behavior specifications, communication specifications and an API for topology building, but does not add communication primitives. It defines a component model with composition[12], and enforces strict encapsulation and type checking on components interfaces.

The ΣC language has been designed to allow two levels of execution: off-line and on-line. The static topology of a ΣC application is handled through instantiation of components and topology building at compile-time, by executing off-line parts of the application source code dedicated to that effect. The on-line level is the application execution on the target hardware.

Listing 1. A Line filter prototype

```
1 agent LineFilter(int width) {
2   interface {
3     in<int> in1;
4     out<int> out1, out2;
5     spec {in1[width]; out1[width]; out2[width]};
6   }
7 }
```

4.1 Components

The basic entity in a ΣC program is an agent. It abstracts a programming model process and so corresponds to an execution unit, with its own address space and a single thread of execution. It has an interface, describing its communication ports (direction and type) and the specification of its behavior in the model described above. It is written as a C scoping bloc with an identifier and parameters, containing C unit level terms (functions and declarations), ΣC-tagged sections: `init`, `map`, and `exchange` functions.

The `interface` section contains the communication ports description and the behavior specification. Each port is a direction (`in`, `out` or `inout`), a type (any C except pointers and functions) and an identifier. Ports may be provided with a default value (if `out` or `inout`) and a sliding window size (if `in`). Arrays of ports can be expressed. The behavior specification is a `spec` espression describing the behavior as shown on line 5 of listing 1.

The `map` section contains component instantiation, topology building and initialization code for the agent. As a ΣC agent in the language is an abstraction, it has to be instantiated to be part of an application. This is done off-line by executing the `map` section of agents. Each agent is then responsible for its initialization, the instantiation of the agents it contains (an agent is a composite) and the topology it encapsulates. Two extensions to C are introduced here: a `new` keyword for instantiating an agent, and template-like type parametric instantiation for system agents. A topology building API is used here, with two functions : `connect` to connect two ports, and `preload` to preload data in a link. For an example, see listing 2 for the map code used to build the network of figure 1. All assignments done to agent state variables in the `map` section are saved and integrated in the final executable, allowing for off-line complex initialization sequences on a per-instance basis.

The ΣC language enforces strict encapsulation: the internals of an agent, and all its contents (contained agents, links, local ports, etc...) cannot be accessed from outside that agent.

`Exchange` functions implement the communicating behavior of the agent. An `exchange` function is a C function with an additional `exchange` keyword, followed by a list of parameter declarations enclosed by parenthesis (see line 8 of listing 3). In the parameter declaration, the type is the name of a port and the

Listing 2. Topology building code

```
1  subgraph root(int width, int height)
2  {
3    interface { spec {}; }
4    map
5    {
6      ...
7      agent output = new StreamWriter<int>(ADDROUT, width*height);
8      agent sy1 = new Split<int>(width, 1);
9      agent sy2 = new Split<int>(width, 1);
10     agent jf = new Join<int>(width, 1);
11     ...
12     connect(jf.output, output.input);
13     ...
14     for(i=0; i<width; i++) {
15       agent cf = new ColumnFilter(height);
16       connect(sy1.output[i], cf.in1);
17       connect(sy2.output[i], cf.in2);
18       connect(cf.out1, jf.input[i]);
19     }
20   }
21 }
```

declarator creates an `exchange` variable of the type of that port. They can be used in the code in exactly the same way as function parameters, the direction of the port (in, out or inout) indicating whether the variable resolves to an input or an output buffer. An `exchange` function call is exactly like a standard C function call, the `exchange` parameters being hidden to the caller. `Exchange` variables can be used as parameters to C function calls without overhead or hidden data copy in most cases. Listing 3 is an example of an agent implementation.

The agent behavior is implemented as in C, as an entry function named `start()`, which is able to call other functions as it sees fit, functions which may be `exchange` functions or not. No communication primitives are available or visible at the function or `exchange` function level, and it supports `exchange` functions calling `exchange` functions with, however, possible performance effects.

Subgraphs are similar to agents, except that a `subgraph` implements only composition, without behavior. As such, a `subgraph` has only an `interface` and a `map` scope, and subgraph ports have a slightly different meaning: they are aliases for the ports of internal agents or subgraphs instances.

4.2 System Agents

System agents are special agents implementing data distribution and synchronization, and making it available to the compilation tools for transformation and optimisation purposes. They are handled through stream-like agents: `Split`, `Dup`, `Join`, `Select`, `Merge` and `Sink`. Those agents are type generic and take a C type in parameter when instantiated.

Listing 3. The Column Filter agent used in figure 1

```
1  agent ColumnFilter(int height) {
2    interface {
3      in<int> in1, in2;
4      out<int> out1;
5      spec {in1[height]; in2[height]; out1[height]};
6    }
7
8    void start() exchange (in1 a[height], in2 b[height], out1 c[height]) {
9        static const int
10           g1[11] = { -1, -6, -17, -17, 18, 46, 18, -17, -17, -6, -1},
11           g2[11] = {0, 1, 5, 17, 36, 46, 36, 17, 5, 1, 0};
12       int i, j;
13       for(i=0;i<height;i++) {
14         c[i] = 0;
15         if(i < height - 11)
16           for(j=0; j < 11; j++) {
17             c[i] += g2[j] * a[i+j];
18             c[i] += g1[j] * b[i+j];
19           }
20       }
21    }
22 }
```

4.3 Input / Output

Input / Ouput is handled with a special class of agents, identified by the keyword `ioagent`, and support type parametrization with a C++ template like syntax. They provide a way to write agents handling low level system and input / output tasks as well as drivers or protocol stacks, but interfacing with the ΣC programming model. The tool chain is then open to the possibility of targeting those agents on dedicated hardware such as DMA engines, IO Processors or different execution environments.

A sample of such ioagents are implemented to access external memory: those are the `StreamReader`, `MemReader`, `StreamWriter` and `MemWriter` ioagents, with variants allowing for synchronization of memory transfers. `Timer` ioagents produce a stream of timed events, allowing for time-based synchronization.

4.4 Software Architecture

Agents and subgraphs can represent any granularity: large processes, thread-size entities, fine grain parallelism (SIMD-like). It is designed so that the port of a sequential C program to ΣC may be done by making it a single agent, and to progressively turn it into a massively parallel implementation by repeated decomposition in smaller agents and subgraphs.

ΣC supports libraries of ΣC components and the design of carefully crafted and optimized libraries of algorithms abstracted behind a generic interface, or parallelism patterns.

Standard C code, even non-reentrant, may be reused and compiled as ΣC code. C pointer equivalence in `exchange` functions allow for passing pointers in standard function calls with no loss of performance or generality.

A standard C back-end is needed for completion of the compilation process; as such, ΣC is compatible with vector extensions, attributes, pragmas, inline asm, automatic vectorisation and other specifics of the target ISA and backend compiler.

5 A Sketch of the ΣC Compilation Process

The ΣC compiler chain is architectured around four passes. The first pass, the ΣC front-end, performs a lexical, syntactic and semantic analysis of the ΣC code, and generates preliminary C code for either off-line execution or further refinement by substitution.

The second pass, the ΣC middle-end, deals with agent instantiation and connection, by compiling and executing 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. Most system agents are combined and transformed into shared memory buffers or NoC transfers so as to fit the system communication and memory architecture. The second pass subsequently computes a deadlock-free lowest bound of the buffers size for all links (see [13]).

The third pass performs resource allocation at the system level. This encompasses computing buffer sizes and construction of a folded (hence finitely representable) unbounded partial ordering of tasks occurrences (see [14]) as well as allocation of tasks to computing resources (cores, clusters, etc.) and NoC configuration. Resource allocation can be performed in a feedback-directed fashion so as to achieve an appropriate level of performance.

The last pass, called the ΣC back-end, is in charge of generating the final C code as well as the runtime tables which, based on the partial orderings built by the third pass, make the link with the target execution model. Using the C back-end tools, the ΣC back-end is also in charge of link edition as well as loadbuild.

Optimization Points

ΣC is designed with the following three goals when it comes to optimization. First, rely on a proven, portable, efficient backend compilation toolchain, in practice, a C compiler and associated tools (linker, assembler).

Secondly, at the process network level, optimize through buffer fusion, rewriting cascades of Split and Dup as patterned access to data, specifically if the

architecture has DMA engines. Another level of optimization is adjusting by reduction of the degree of parallelism of the process network graph, by detecting and replacing specific topologies.

The third design goal for optimization was that performance tuning of performance sensitive code on embedded devices is possible in a pragmatic way: ΣC is sufficiently flexible to allow developers to express parametric parallel code, where instances execution cycles, buffer sizes and degree of parallelism can be adjusted with instance parameters, allowing developers to adjust the shape of the process network to a better match for the target architecture.

6 Evaluation

Two teams, one internal to our lab, one within our industrial partner, are collaboratively stress test the language and the model on a first (but wide ranging) round of applications optimized for the target platform amongst which multi camera target tracking applications, augmented reality video processing, H264 video encoding, and 4G/LTE channel coding implementations. Our current results indicate that the level of expressiveness chosen has proven itself so far appropriate, that is target applications have been designed without encountering algorithmic constructions that are either clumsy or (worse) impossible to express.

Potential efficiency concerns are regularly expressed and handled in the course of the language evolution, without major changes so far. The informal usability testing underway has shown that the component model and process model expressed in ΣC has not been considered a barrier and that developers with a background in C or SystemC have few difficulties to adapt to it.

Furthermore, the model has proven interesting for designing parallel solution algorithms to some operational research problems, so we may have a possibility to retarget the implementation on larger scale, non embedded, parallel systems (or a large collection of high performance embedded manycores). It has also proved suitable as a back-end language for higher-level stream processing languages such as [15], and may be used as a target for source code automatic parallelisation tools such as Par4All/PIPS[16].

7 Conclusion

We have presented a well formed programming model and ΣC, a programming language implementing it. This programming model and language have so far a result on two of our criteria: ability to express a variety of algorithms, and familiarity to embedded developpers through C compatibility. It has now evolved through a few iterations, mostly removing unneeded features and adding target integration, and can now be seen as a stable foundation.

For the remaining three criteria set forth in the introduction, we have shown that the programming model support formal analysis and computation of a bounded memory schedule. Implementation details such as the ability to do

in-place data modifications and buffer sharing allow for a strict, memory constrained implementation with a dedicated micro-kernel. And the programming language support type checking, components and reusability.

This language and its compilation tool chain is being industrialized as part of the technology offering for a many-core architecture jointly developed with one of our semi-conductor partners.

References

1. Jantsch, A., Sander, I.: Models of computation and languages for embedded system design. IEE Proc.-Comput. Digit. Tech. 152(2), 114–129 (2005)
2. Kahn, G.: The Semantics of Simple Language for Parallel Programming. In: IFIP Congress, pp. 471–475 (1974)
3. Lee, E., Messerschmitt, D.: Synchronous Data Flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
4. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.: Cycle-Static Dataflow. IEEE Trans. on Signal Processing 44(2), 397–408 (1996)
5. Girault, A., Lee, B., Lee, E.A.: Hierarchical Finite State Machines with Multiple Concurrency Models. IEEE Trans. on Computer-aided Design of IC & S 18(6), 742–760 (1999)
6. Gao, G.R., Govindarajan, R., Panangaden, P.: Well-behaved dataflow programs for DSP computation. In: IEEE ICASSP 1992, pp. 561–564 (March 1992)
7. Reutenauer, C.: Aspects Mathématiques des Réseaux de Petri, Dunod (1997)
8. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. In: Proceedings of CC 2002, Grenoble, France, pp. 179–196 (2002)
9. Buck, I.: Brook Specification v0.2. (2003), `http://merrimac.stanford.edu/brook`
10. Watt, D.: Programming XC on XMOS Devices. XMOS (2009)
11. Khronos OpenCL Working Group: The OpenCL Specification v1.1 (2011)
12. Lau, K., Wang, Z.: Software Component Models. IEEE Trans. on Software Engineering 33(10), 709–724 (2007)
13. Sirdey, R., Aubry, P.: A Linear Programming Approach to General Dataflow Process Network Verification and Dimensionning. Electr. Proceedings in Theorical Computer Science (to appear)
14. Sirdey, R., David, V.: Système d'ordonnancement de l'exécution de taches cadencé par un temps logique vectoriel. Patent pending, filling no 1003963 (2010)
15. De Oliveira Castro, P., Louise, S., Barthou, D.: A Multidimensional Array Slicing DSL for Stream Programming. In: Proceedings of CISIS 2010, pp. 913–918 (2010)
16. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In: Proceedings of ICS 1991, pp. 244–251 (1991)