Loosely Coupled Accelerators for Reconfigurable SoC

Jean-Christophe Le Lann and Bernard Pottier
Université de Bretagne Occidentale
LESTER, FRE CNRS 2734
Brest, France
lelann.jean-christophe@neuf.fr
bernard.pottier@univ-brest.fr

Matthieu Godet, Ronan Keryell and The Nhan Luong
Computer Science Department
ENST-Bretagne
Brest, France
MatthieuGodet@hotmail.com
Ronan.Keryell@enstb.org
TheNhan.Luong@enst-bretagne.fr

Abstract

Loosely coupled accelerators allow an intensive "pure software program" (PSP) to share data seamlessly with accelerators. The method involves rewriting the source code, the hardware synthesis of the accelerated parts, and, of course, an architectural support.

At algorithm design time, processing is translated into coarse grain steps. *Steps group a set of data buffers to be consumed and produced by an accelerator, with a function operating on these buffers. The function, with its associated communications, are defined statically and can be understood as very large instructions operating on very large operands. The support hardware can be a reconfigurable unit or a massively parallel processing array inside a* System on Chip *(SoC).*

At run-time, the PSP first produces a queue of buffers as fetch and store descriptions to occur between the main memory and the accelerator local memories. The content of the buffers is then transferred by a DMA communication engine with some stride-enabled scatter-gather capabilities to compact data in the buffers of the accelerators. Execution is then a coarse grain program with loops applying accelerated functions to sets of local buffers. Each step overlaps operations on main memory, communications, and local processing in a predictable way.

This paper describes the principle with an illustration on a simple linear algebra example. It describes a basic set of tools that have been developped to validate the concept: the compiler-assisted production of communication queues and accelerator code that can contains some SIMD parallelism, an example of a communication engine that has been modeled as a System-C program, and the preparation of accelerator code. This code is currently sequential, but is intended to be synthesized on fine or coarse grain reconfigurable units under time related constraints coming from SoC characteristics and level of performances expected for the accelerator service.

1 Designing hardware or writing applications?

System on Chip (SoC) offers a gradation of choices from pure specific design to multiprocessor platforms. Using embedded FPGA to increase flexibility seems to place the reconfigurable SoC (RSoC) at midway from hardware design toward parallel processing platforms.

In this comparison the weight of software is largely increased by taking as reference parallel programming. We propose a new architectural interpretation of the RSoC that can be of great interest for productivity based on software tools, compilers and high level languages, following a seminal approach taken from [5] about stream computing.

A meaningful definition for an RSoC platform is the association of a control processor (CPU), an interface to a main memory (MM), an interconnection network and one or several reconfigurable units (RU). This is a very general organization on which RU can be used to implement dynamically reconfigured accelerators or permanent upgradable functions. Based on this definition, there are several execution modes associated to programming paradigms: sequential, memory-oriented programming, SIMD, MIMD, stream and object-oriented programming…

The paper focus on a sequential execution style (but of course, each sequential part could be seen as a thread of a multi-threaded bigger application), where a coarse grain simple machine executes forever a cycle shown on Table 1.

The three operations are pipelined allowing to overlap communications and computations. The buffer transport is prepared with the assistance of a compiler, according to lo-

LOAD	operand buffers described in a queue prepared at the software level, possibly next kernel configuration
EXECUTE	a coarse grain computation on the buffers. This is the kernel of processes synthesized on the RU
STORE	back result to memory.

Table 1. Basic execution pipeline of an accelerated execution.

cation of data in memory and physical computation requirement on the RU. In our model, the RU can also exploit some SIMD parallelism that the compiler should exploit too.

The paper is organized in four parts describing architectural support for the coarse grain processor in Section 2, compiler aspects in Section 3, synthesis issues in Section 4, and before concluding, the validation of the concept with expected performances illustrated on an example in Section 5.

2 A communication architecture

2.1 Principles

Today SoC hardwares are mostly an assembly of general purpose architectures and dedicated intellectual properties (IP) functions. While the IPs are certainly desirable due to their local efficiency, it is difficult to guarantee their performances inside the global system. The economic interest of one particular IP can also be the source of headaches considering the variability of application standards.

The proposed alternative to speed up computation preserving hardware generality, is based on a small set of points taking into account the platform organization and language supports:

1. the data organization in memory should remain the same for the processor and the accelerator(s) to avoid major remodeling of the application,

2. level of interactions between the software part and the accelerated part should remain small, with little overhead on system activity,

3. the reconfigurable unit is isolated by local memories on which a simple synthesized circuit operates.

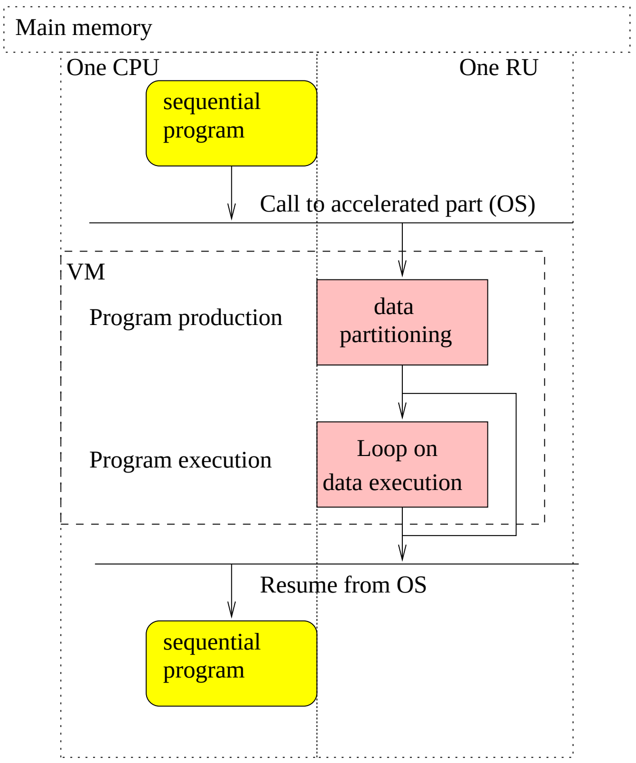

Figure 1. Execution flow with a call to the accelerator.

This machine has basically shared memory characteristics, but a serious gap exists between the accelerator capabilities and those of a common processor. A first issue is data reorganization, fetching, and storing back to memory and, unfortunately, cache are not effective due to poor temporal locality of data for streaming data. A second issue is partitioning computations to deal with the local memory constraints. And there is more, with the control of the accelerator, elementary types from the high level language. . .

Our proposition is to integrate support for these issues at the system level, for sake of performances and simplification of compiler/synthesizer tasks. The idea is to embed the accelerator into a virtual machine (VM) filling the gap with the software tools. This denomination suits, because VM are known to rise the level of the hardware services, reducing the load on compiler tools, which is exactly the case here. Furthermore, the execution model has synchronous properties that allow to fit it as a virtual processor in coarse grain parallel paradigms, such as SPMD computing.

Figure 1 shows the synoptic of a call to an accelerated piece of code. An application program (possibly a thread) suspends itself on a procedure call or an instruction.

The first part of the acceleration is the data partitioning

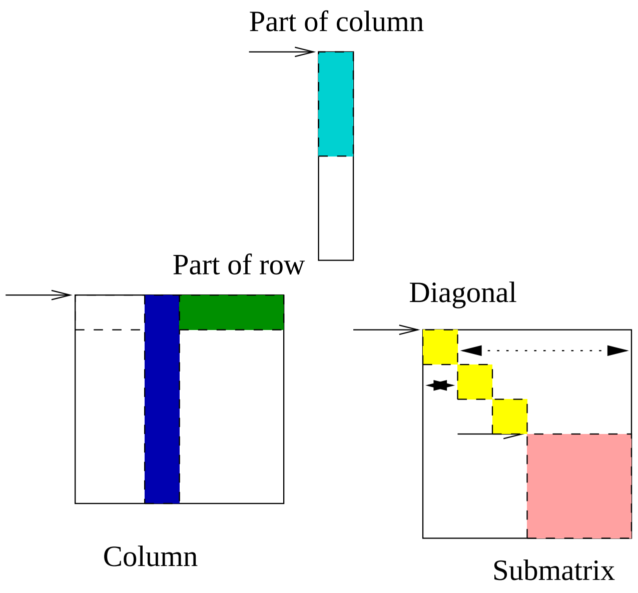

Figure 2. Parts of a main memory data structure are relocated to local memories (or inversely). The figure shows some simple patterns on 1D and 2D arrays.

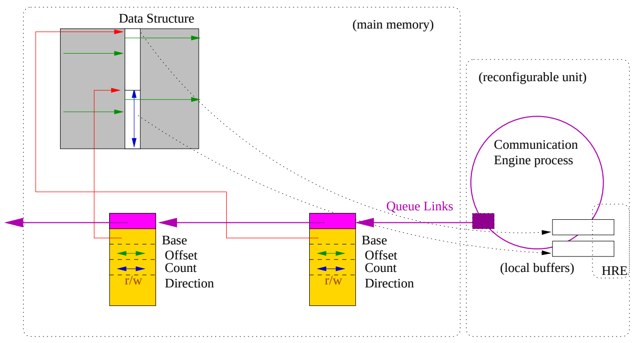

Figure 3. Relations between load/store specifications (queues), main memory data structures and local memories.

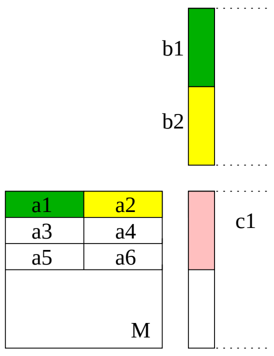

Figure 4. Data partitioning in a matrix product.

operation. It consists of splitting the data into chunks that will be transported to the accelerator by a Communication Engine (CE). To be efficient, this operation will not actually copy data, but just specify how the data are to be accessed as address, strides, or more complex operations such as flattened data structures called Memory to Memory Communication Descriptors (MMCD). We have chosen a strong coupling between the CE and the accelerator because it leads to the highest throuhput [11]. Figure 2 shows some chunks of data mapped on simple data structures. Figure 3 shows the relation between the data in main memory and the accelerator local memories. Here, access to data chunks is organized as a queue that will be read by the CE.

The data can be considered as operands for a program in our VM. During one cycle of the VM, several load and store operations must take place. The processing is specific to the application, and is divided in cycles, even in case where several hardware based processes retaining a state are involved.

Thus the VM can be seen as a processor operating by cycles where a set of load/store and execution will take place. The VM is pipelined, allowing load, execute and write back to overlap. In this machine, the data are normally not word operands, but chunks of data on which the execution circuit is applied, possibly with a very high degree of fine grain parallelism, such as SIMD.

The operating system (OS) manage calls and returns, checking for RU availability and resource allocation, including the completion of a possible reconfiguration (horizontal lines shown on Figure 1).

Our machine model is more easily explained on a simple program such as a matrix product.

2.2 A matrix multiplication example

Let us suppose that we need to produce a product of matrix on an RU.

We suppose that the accelerator has support to achieve efficiently sums of products. The way in which this computation is achieved, nor the data types involved will not be discussed here. We just need to divide the data according to buffer capacities or operators availability.

Figure 4 shows a partition where two matrices A and B are divided in sub-lines and sub-rows. The resulting matrix C will follow the same rule.

An execution is displayed in Table 2, each row being

Table 2. Pipelined execution showing machine cycles and communication identifiers associated with load or store operation for each cycle. The first cycle does not activate execution. The first store operation will occur after the completion of 6 cycles involving an execution.

machine step	xFerId	stage load	stage compute	stage store
1	1	load a1		
	0	load b1		
2	1	load a2	a1 × b1	
	0	load b2	(continued)	
3	0	load a3	a2 × b2	
4	0	load a4	a3 × b1	
5	0	load a5	a4 × b2	
6	0	load a6	a5 × b1	
7	0	load a7	a6 × b2	
8	1		a7 × b1	store c1
	0	load a8	(continued)	
9	0	⋮	⋮	⋮

associated to one load and store, several of these operations possibly occurring during a machine logical cycle. Such a table can be computed at compile time by scheduling input, output, and execution operations.

After building a schedule, it is possible to write a program where transfers and execution are grouped into steps which boundaries can be found at runtime by the communication engine. One of the CE responsibilities is to take decisions on the pipeline progressions based on signals produced by the compute part.

The program associated with the matrix product could be depicted as a sequence of load-store-execute instructions, where the execution corresponds to a particular function applied to the local memories.

3 Compiler and language aspects

To ease the investigation of applications on the target architecture, a compiler add-on prototype has been written. A high level application written in C or Fortran can automatically be transformed into a control code and the Heterogeneous Reconfigurable Engine (HRE) codes. The programmer choose the parts she wants to implement as an HRE in our architecture by embracing it by compiler directives in pseudo-comments. The idea for the programmer is to avoid explicit programming in other stream or kernel-specific languages that are used in other projects such as [7].

The control code contains the scalar part of the application, generates the MMCDs lists for each pipeline step, launches the CE on these lists and synchronizes with it when some data computed by some HREs are needed by the control code or some other HREs.

Each HRE code is the C code of a marked part of the original application code in a suitable form to be synthesizable in a HRE. This C code can be directly used by a synthesis tool to generate the HRE hardware description or pretty-printed as SmallTalk for the Madeo synthesis tool from UBO.

To implement our compiler, many high level analysis and transformation phases must be applied to the code. Furthermore, if some transformations can automatically be applied, some architectural knowledge is useful to transform the code in such a way more suitable for this architecture. This is why we advocate that an interactive tool is interesting since it allows architectural exploration on the target architecture. We have chosen the PIPS compiler framework [2, 1] that is interactive, interprocedural to deal with big applications and comes with many high level analysis (polyhedron based abstract interpretation...) and transformations (vectorizers...) and in which we have a strong experience.

Even if the architecture can execute any code, it is more efficient if the application code contains compute intensive loop nests. So is our compiler: it will generate more efficient code on rather parallel regular loop nests because it gives the opportunity to use regular MMCDs with fixed strides. The target architecture can also have some SIMD operators to exploit sub-word parallelism) and we've added a phase to extract and generate code for these kind of instructions. Our compilation method is divided in 2 main parts.

3.1 Code and data distribution

In order to program our accelerators or more generally some massively parallel WPPAs (weakly programmable processing arrays) with some standard high level programs, we need to split these programs into 2 parts, the one that will execute on the control host computer and the one that will execute on the accelerator elements. Once this choice done, the memory communication descriptors (MMCD) are generated to orchestrate communications between the host computer and the accelerator elements but also between the accelerators and the memory.

The partitioning is chosen by the programmer by adding directives to the source code to specify which parts will be sped up on the accelerators. Afterward, phases have been added in PIPS [2] to extract the accelerator code from the original program, and to add interfaces at host program and accelerators level to transfer data with the help of the MMCDs.

This compilation phases are divided in 3 main parts [12].

3.1.1 Analysis phase

First the program is analyzed with its partitioning directives to

- detect the data to transfer between the host program and the accelerators;
- analyze data dependencies in the code to know if computations will be efficiently pipelined in the accelerators;
- optimize the data transfers to group communications with a constant stride in a MMCD that can directly express these communications.

3.1.2 MMCD generation phase

The host program must generate the MMCDs and launches them at the good time for the good computation orchestration. So the aim of this phase is to transform the original program and to automatically add these generation and orchestration functions.

The MMCD generator adds in the host program some code to build the MMCDs on the fly to command and interact with the accelerators, that is to transfer operands and results but also execution requests. In this first part of the project, we've chosen to use a robust dynamic method inspired from the concepts of inspector-executors [18, 4, 8]: during the execution of the host program, we compute the MMCDs that will be needed later. For each loop nest, the MMCDs to transfer the operands are generated, then the MMCDs to start the computations and at last the MMCDs to send some results back to memory. At the end of each accelerated part, some synchronization code is added to be sure no results is lacking for the following of the program.

3.1.3 Code generation for the hardware accelerators

The code generator for the accelerators extracts the partitioned code from the main program into separate fonctions that will run on the accelerators and will be fed by the MMCDs. Interfacing with the external data are done in this function codes by the call to the hardware macros `READ_BUFFER` and `WRITE_BUFFER` that allow to peek and poke into the hardware queues manipulated by the MMCDs.

3.2 SIMD code generation

In PIPS we have also developed some phases to generate SIMD code that are useful to generate HRE code with this kind of parallelism if we want to improve the throughput of the operators [12]. This phases can also be used to optimize the control code if a SIMD instruction set is available.

In order to have more portable output, we generate some SIMD macro-code that can be retarget on different processors, even general purpose processors to debug our system or to emulate the target code on a PC for example.

In order to exploit a maximum of parallelism, we use a technique able to find parallelism in different loop-nest iterations but also between different unrelated expressions in basic blocs. We began by combining an automatic vectorization based on loop-unrolling [14] and superword parallelism extraction [17]. These transformations have been implemented in PIPS and integrated with some other already existing transformations to reach the expected result [12] :

1. **if-conversion** : first, all the tests (traditionally compiled into conditional jumps incompatibles with SIMD) are replaced by predicate evaluation and predicated move compatible with SIMD. Since it is a sensitive choice in term of performance, some pragmas can be added to precise which tests are converted or to precise the instructions cost below to which any test is automatically converted;
2. the **expression atomizer** transforms complex expressions and instructions into simpler ones close to the instructions executed by the accelerators;
3. **loop unrolling** generates instructions groups with many isomorph instructions that may be grouped together into SIMD operations if allowed also by the dependence analysis;
4. **reduction detection** (also already in PIPS) is used to remove some dependencies prohibitive for parallelization [13];
5. in order to remove even more parasitic dependencies, the source basic blocs are rewritten in **static single assignment** (SSA) form;
6. and of course is the **vectorization** phase itself:
    (a) isomorph instructions are reordered and grouped according to a formal description of the available operators on the target architecture with respect to the dependencies on the code;
    (b) the instruction groups are vectorized by selecting the most parallel SIMD instructions that deal with operands big enough;
    (c) SIMD code generation produces operand load sequences, compute and result writing sequences. Some redundant operand loads are discarded if several instructions use the same data or are useless, by using the dead-code elimination phases from PIPS;

(d) at last, some loop invariant memory accesses are hoisted outside the loop to improve the performance.

3.3 Conclusion

More generally, PIPS can also be used to optimize the code at the system level, for example by parallelizing the code with several threads to run on several processors.

We've chose to use a rather dynamic method based on inspector-executor compared to other teams that use more static methods [7, 11] because we can deal with less regular application without loosing too much performance on regular applications since we can use some partial evaluation in this case.

Many improvement can be still done in our compiler to get more performance on our architecture for more applications. We could deal directly with data remapping to have more compact data more suitable for the MMCD operations. Furthermore, the MMCD operations are right now dynamically generated before each HRE use but this MMCD lists could be cached for successive identical use or even statically generated by the compiler with techniques such as in [3] for more general high performance distributed computing or as sketched in [10] for a high performance decoupled parallel vector machine.

The return experience on the code distribution part is that it has been quite interesting to design the architecture and the compiler at the same time because it allowed many interactions and allowed us to refine some architectural points to have real support for the compiler and no semantics clash with the architecture.

Furthermore, since PIPS can prettyprint its internal representations in different language, we can generate free functional simulators in C instead of generate some SmallTalk HDL hardware description for the Madeo tool [16].

4 Architecture and synthesis for application processes

The target architecture, with the organization of the execution as a set of processes are the entry points in the mapping problem.

Architectures are possibly fine grain architectures, coarse grain reconfigurable data paths or mesh connected small processors.

Organization methods can be applied by hand, compilers, or code transformation tools. Depending on the architectures, the mapping will first produce set of processes, then apply architecture synthesis tools or code production.

An intermediate layer is being defined and developed to handle the problem. The bottom part in this layer extends a previous work achieved in the Madeo project [15, 9]:

- the description of the architecture is achieved on the top of a set of classes enumerating coarse and fine grain devices appearing in reconfigurable architectures. A language and object model allows to define connectivity and hierarchy levels in the target architecture organization;

- the input for a mapping is an application description defined as a set of cooperating processes. Each process is in turn defined as a low level control data flow graph (CDFG-LL) connecting devices. These devices exist in the support architecture and have known semantics, due to the first point;

- mapping algorithms are in charge of resource allocation for processing, memory, and connectivity. The result is a mapped application that still needs to be translated in physical descriptions (bitstream configurations or microcodes, initial values for memories, etc.).

Just above this layer, there will be transformation tools using as input a high level CDFG corresponding to a process oriented application description close to high level languages, and producing the output as a CDFG-LL in view of a given architecture.

5 Validation & status

5.1 Cosimulation and performance analysis

It is expected that variants of the proposed architecture will be implemented on actual architectures in medium term. To validate the functionalities and collect performance analysis informations, a co-simulator has been developed.

The co-simulator is executed by three basic threads as shown on Figure 5:

- a first thread represents the behaviour of the software part of the program, described in C, or automatically generated by a compiler, such as PIPS in Section 3;

- a second thread is produced by the compilation of the systemC representation of the communication engine;

- the third thread represents the behaviour of the accelerated part of the program working on local memories. This thread is to be synthesized on larger RU architectures as a graph of operators and, possibly, some control. It is also expected that this part is to be extracted by a compiler such as described in Section 3.

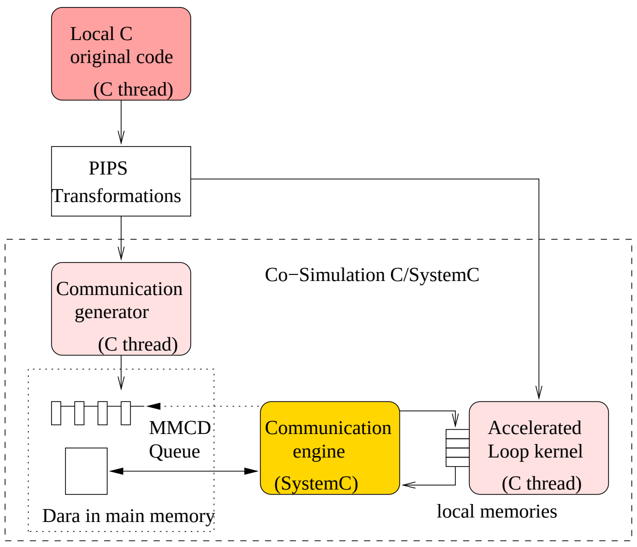

Figure 5. Cosimulation, showing the communication generator that produces data descriptor in memory, the systemC model for the communication engine, and C thread(s) operating on local memories.

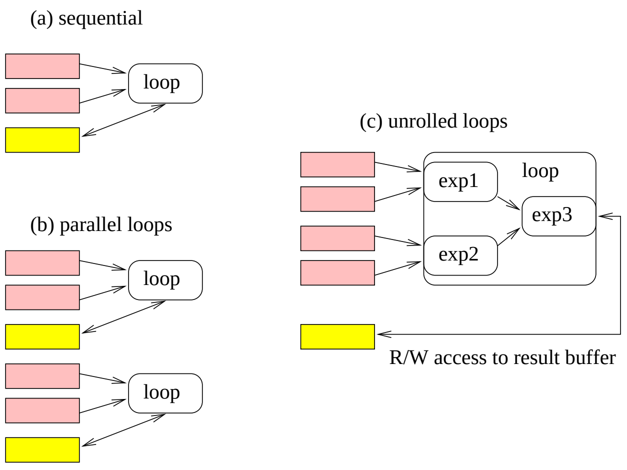

Figure 6. Possible mapping of computations with impact on data partitioning and distribution. First case has affinity with sequential processing, case (b) is globally parallel locally sequential, case (c) has affinity with globally sequential an locally parallel, with difficulties on data distribution outside the HRE, and opportunities for expression optimization during synthesis.

5.2 Performance analysis

5.2.1 On-going investigations

To compare with a pure software solution, we take parameterized measures of the cost of data partitioning, data communication delays and execution delays appearing as the pipeline cycle time. Estimated times (measured in clock cycles) will be taken for the execution and network performances.

The design flow used for the experiments is exposed on Figure 7. PIPS compiler transforms an initial C code (taken as specification) into two different C codes : one for the generation of descriptors in memory, the other for the accelerated function.

The numbering on this figure explains the behavior of the reconfigurable system at runtime : mark labeled “1” refers to the generation of memory descriptors (MMCDs), placed in memory, while “2” and “3” refer to the parallel processes reading and writing data from/to memory.

The marks “A” and “B” refer to the two compilation flow options that are monitored during our experiments:

- “A” refers to a pure-software evaluation, answering the question “how much time would it cost to run the initial function, as is, on the embedded processor *only* ?” (this means that the reconfigurable part appearing on the drawing can be forgotten for this part, as it is not used in this case);
- “B” refers to the evaluation of our solution, based on an embedded processor preparing MMCDs queues for the reconfigurable accelerator. Note that in our methodology, we also compile the accelerated function as an ARM function, in order to have a rough estimation of the complexity of this function, that can be compared with a sequential implementation. As such, the figures given for this part are over-pessimistic, as the inherent parallelism of the function is not taken into account here. In the next section, we provide formulae to ponder and explore this measures by introducing parallelism.

In both cases, the `gcc` toochain is retargeted for StrongARM. The instruction simulator used is Simit-ARM.

5.2.2 Results

The experimental results are presented on Figure 8. They represent the evaluation made on a simple application written in C, by evaluating different instances of the execution, based on the varying SIZE parameter. Despite its apparent simplicity, the application (just as any other linear algebra operating over big data domain) imposes severe data transfers.

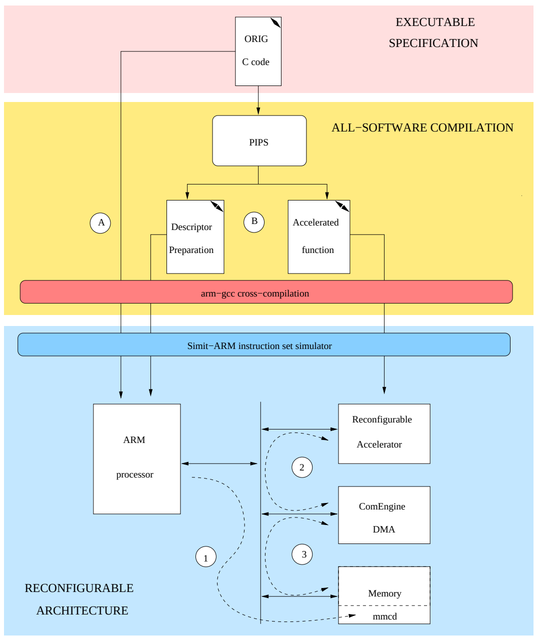

Figure 7. Design flow used for the experiments.

Algorithm 1 Original C code.

```
for(i = 0; i < SIZE; i++)
{
  c[i] = c[i] + d[i];

  for(j = 0; j < SIZE; j++)
  {
    c[i] = c[i] + a[i][j] * b[j];
  }
}
```

The results exposed in the previous section show that the computation time of a pure-software application (ORIG in the sequel) is, for each sample measured, greater than the sum of the two execution of MMCD production and accelerated function. In fact, these results should be interpreted with care : they simply mean that the communication have not been fully taken into account in this sum. More precisely, the gap between the two upper curves reflects the time that *can* be allocated to communication : if the final estimated communication time takes more time than this difference, then the reconfigurable approach becomes uninteresting with respect to a pure software solution [1]. This

[1]Please note that cache effects are not taken into account in this evaluation.

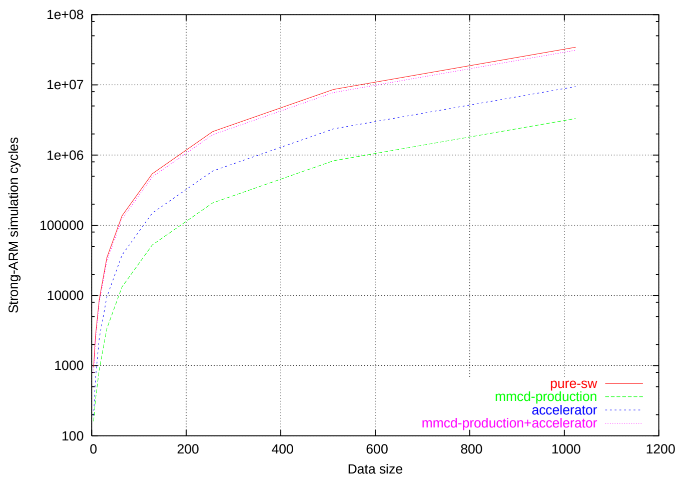

Figure 8. Results of ARM simulations: the x-axis carries evolving matrix sizes (SIZE parameter). Four curves are drawn for (bottom-up) dynamic production of MMCD, execution of the acceleration by an ARM sequential processor, cumulated execution and communication, original pure software execution. Additional possible latencies implied by a slow Network on Chip (NoC) are not figured by this diagram. It is noticeable that the communication+execution transformed method carries lot of opportunities compared to the pure software execution.

is what we will estimate in the next section.

5.2.3 Extrapolation w.r.t Architecture characterization

The previous results are brute-force results obtained just through the ARM ISS simulation. As explained, they do not represent the communication overhead introduced in the architecture. In this section, we introduce new parameters, that better characterize the SoC architecture initially considered. They are used to enhance the whole estimation of the potential gains resulting from the use of MMCDs mechanism.

The parameters that we introduce here are:

1. the target clock frequency f. We suppose that the architecture has a single clock;
2. the bandwith B of the communication media (checked between 2 and 16 GBits/s;
3. the average number n_{op} of elementary operations performed by cycle on the accelerator: for this purpose,

we take as basis the number of instructions that would be required on an ARM processor to execute the accelerated function. We then makes the assumption that each instruction can be executed by one elementary operation. Then, to get the final execution time of the accelerated function, by applying an increasing multiplication factor accounting for the potential parallelism of the application. This factor is checked between 2 and 30 operations/s.

The main objective is to show in which conditions the ComEngine mecanism should be used instead of a pure embedded software implementation, that is when the number of software cycles T_{SW} is less than the number of cycles found when running a reconfigurable solution T_{RC}. We define the number of cycles executed as follows:

$$T_{\mathrm{RC}} = T_{\mathrm{MMCD}} + S \max(C, K)$$

where S represents the number of macro-steps involved in the computation C represents the communication cycles on the network on chip, while K is the number of cycles executed on the coarse-grained accelerator. The use of $\max$ function is justified by the fact that, *by construction*, our compilation techniques overlapps computation and communications.

Let us now refine these definitions:

$$C = \frac{(N_{\mathrm{MMCD}} S_{\mathrm{MMCD}} + N_{\mathrm{data}} S_{\mathrm{data}}) f}{BS}$$

$$K = \frac{I_{\mathrm{ARM}}}{n_{\mathrm{op}} S}$$

Finally the gain in percent is defined as:

$$G = 100 \frac{T_{\mathrm{SW}} - T_{\mathrm{RC}}}{T_{\mathrm{SW}}}$$

Using the preceeding experimental results for a single instance of the application, and making B, f and n_{op} vary, we obtain a set of data that are depicted on Figure 9. Of course, the instance has been choosen as data intensive (for big SIZE parameter). We have chosen to represent on Figures 9 the gain obtained with respect to bandwidth and accelerator parallelism: we clearly show when our ComEngine mecanism becomes interesting w.r.t. these two parameters.

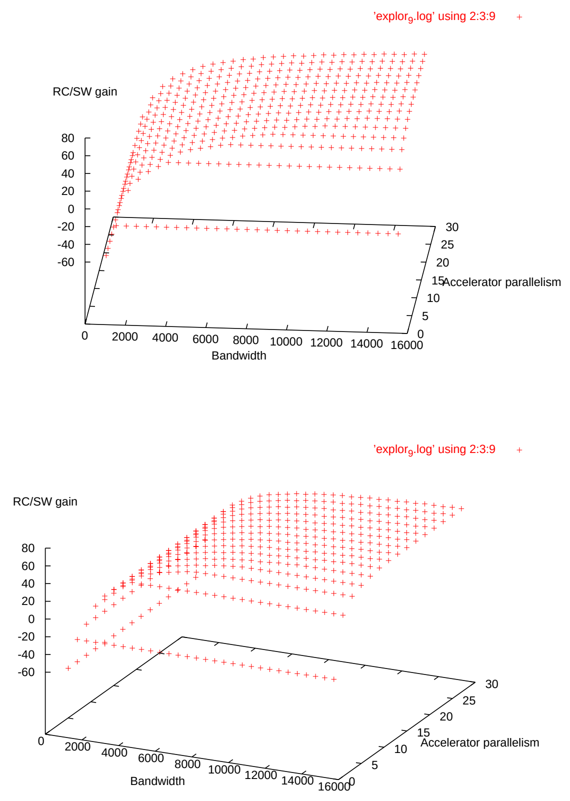

Figure 9. Two different views of the RC versus SW gain (percentage) obtained, w.r.t the required bandwidth (in Mbits/s) and accelerator internal parallelism (expressed in operations by clock cycle achieved).

6 Conclusion

We have presented a new RSoC (Reconfigurable System-on-Chip) architecture based on some loosely coupled reconfigurable hardware accelerators that process compute-intensive parts of a program executed by one or more processors. This allow to map different parts of a global program on the same flexible hardware in an efficient way in the perspective of lower cost, still at high performance and low consumption. The coupling with the global memory is done by a communication engine that execute memory transfer and execution descriptors precomputed by the main processor(s). This computing model avoid reorganizing too much the application to run on the SoC. Operating the reconfigurable hardware by itself is under control of a virtual machine that fills the gap between the software tools and the hardware.

The programming is simplified by the development of a compiler that take a C or Fortran program with some annotations, telling which parts to execute on the accelerator(s), into a global sequential code that controls the data transfers with the accelerators and the synchronizations. The compiler generates the code that will generate the transfer descriptors, even for non static regular loop nests, by the use of an inspector-executor-like method. Since stream processing applications often express SIMD parallelism, we've added automatic SIMD parallelization into the compiler to exploit massive SIMD accelerators.

To validate our methodology, we've implemented a simulator in SystemC of our RSoC with a NoC and an ARM processor that is able to execute our generated code. We've extended the results with a small algebraic model that shows that the methodology is interesting even with small parallelism and low memory bandwidth.

We've designed the compiler at the same time as the architecture and it was a rich co-design experience with positive interactions and feedback on the both sides. We've decided for example to abandon double-buffer hardware because the compiler was not able to use it in the general case. We implemented instead in the compiler a more flexible way to deal with the accelerator memory.

Of course, the project is not finished and there are still many improvements and refinements to be done at the hardware (such as improvement that can be done with more expressive transfer descriptors [6]) and the software (to exploit the polyhedral model present in PIPS to generate more efficient code in the regular case).

Acknowledgements

This work has been funded by several ways:

- the European Morpheus project that targets the general purpose use of reconfigurable logic inside SoC, and develops an hardware/software platform;

- the CoMap French-German P2R project that aims at developing software techniques and model for massively parallel mapping on accelerators made of weakly programmable processor arrays;

- the SAFESCALE project from the French National Research Agency with contract ANR-05-SSIA-005-03 whose goal is to build a secure distributed execution infrastructure and which some parts of the code distribution are reused.

References

[1] C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, P. Jouvelot, and R. Keryell. Pips: a workbench for program parallelization and optimization. In *European Parallel Tool Meeting'96*, ONERA, Oct. 1996.

[2] C. Ancourt, F. Coelho, B. Creusillet, and R. Keryell. How to add a new phase in pips: the case of dead code elimination. In *Proceedings of the Sixth Workshop on Compilers for Parallel Computers (CPC'96)*, pages 19–30, Aachen, Germany, Dec. 1996.

[3] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. *Scientific Programming*, 6(1):3–27, Spring 1997. Special Issue — High Performance Fortran Comes of Age.

[4] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptative scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[5] E. Caspi, M. Chu, and R. H. et al. Stream computations organized for reconfigurable execution (SCORE): Introduction and tutorial. In *Proceedings, Field-Programmable Logic and Applications*, Villach, Austria, August 2000.

[6] S. M. Chai, N. Bellas, M. Dwyer, and D. Linzmeier. Stream memory subsystem in reconfigurable platforms. In *2nd Workshop on Architecture Research using FPGA Platform (WARF'2006)*, 2006.

[7] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Parallel Architectures and Compilation Techniques (PACT'06)*, pages 33–42, 2006.

[8] K. Eswar, P. Sadayappan, and C.-H. Huang. Compile-time characterization of recurrent patterns in irregular computations. In *1993 International Conference on Parallel Processing*, pages II–148–II–155. CRC Press, Inc., Aug. 1993.

[9] E. Fabiani, C. Gouyen, and B. Pottier. Intermediate level components for reconfigurable platforms. In S. Vassiliadis and A. Pimentel, editors, *Synthesis, Architectures and Modeling of Systems (SAMOS 3)*, Samos, Grece, 2004. Springer-Verlag, LNCS.

[10] P. Fiorini, F. Irigoin, and R. Keryell. Modèle de compilation d'HPF pour la machine MIMD à bancs mémoire et réseau distribué programmable phénix. In *RenPar'8*, May 1996.

[11] A. Fraboulet and T. Risset. Master interface for on-chip hardware accelerator burst communications. *Journal of VLSI Signal Processing*, 2007. To appear.

[12] M. Godet. Compilation for an heterogeneous architecture with hardware accelerators including simd instructions and reconfigurable operators with embedded dma engines. Master's thesis, ENSTBr, Sept. 2006. `https://comap.enstb.org/publications/msc-of-matthieu-godet`.

[13] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *ICS*, pages 186–194, 1989.

[14] A. Krall and S. Lelait. Compilation techniques for multimedia processors, 2000.

[15] L. Lagadec, B. Pottier, and O. Villellas-Guillen. An lut-based high level synthesis framework for reconfigurable architectures (25 pages). In S. Batttacharyya, E. Deprettere, and J. Teich, editors, *Domain-Specific Processors : Systems, Architectures, Modeling, and Simulation*. Marcel Dekker, N-Y., Nov. 2003.

[16] L. Lagadec, B. Pottier, O. VillellasGuillen, E. Fabiani, and C. Dezan. A lut based approach for high level synthesis on fpga. In *IWLAS workshop*, New Orleans, June 2002.

[17] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices*, 35(5):145–156, 2000.

[18] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message

passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, Apr. 1990.