# A Many-core Parallelizing Processor

Katarzyna Porada, Bernard Goossens

▶ To cite this version:

Katarzyna Porada, Bernard Goossens. A Many-core Parallelizing Processor . [Technical Report] Université de Perpignan Via Domita. 2017. hal-01558374

HAL Id: hal-01558374
https://inria.hal.science/hal-01558374

Submitted on 7 Jul 2017

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

# A Many-core Parallelizing Processor

Katarzyna Porada, Bernard Goossens
DALI, UPVD
LIRMM, CNRS, UMR 5506 - UM2
Perpignan, France
Katarzyna.Porada@univ-perp.fr, Goossens@univ-perp.fr
mai 2017

## TECHNICAL REPORT

***Abstract*—This paper presents a new many-core processor design to parallelize by hardware. The parallel run is built from a deterministic parallelization of the sequential trace, hence inheriting its order. The code pieces are distributed according to the sequential order in a way which favors neighbor cores communications. The ordered placement simplifies the processor interconnect and the memory sharing. The paper presents a VHDL implementation of a 64-core version of the processor. The synthesized prototype proves that the automatic parallelization technique works and the speed and size of the synthesis show that the design is scalable.**

***Keywords- many-core processor, out-of-order processor, parallelizing processor, determinism, VHDL prototype***

## I. INTRODUCTION

Actual computers are the resultant of the never ending pursuit for higher computing performance. Fifteen years ago, the constructors started to introduce multiple cores in a chip (the first not-embedded microprocessor with two cores on a single die was the IBM POWER4) [12].

We can classify current CPUs in two main categories: multi-cores embed less than 50 cores and many-cores integrate more than 50 cores. Their power no more comes from an acceleration of the execution of each instruction, but from the parallelism provided by the set of cores on-chip. Actual many-cores have a limited scalability because of the complexity of the interconnection network and the shared memory management. The Intel XeonPhi2 [11] introduced in 2015 has 72 cores when its predecessor Knights Corner had 61. The first Kalray MPPA Andey had 256 cores in 2012 and its successor Bostan [4] offers no increase in 2016.

Programming a many-core is not easy with actual compilers, APIs and libraries. Too often, a code must be updated, if not redesigned when the topology of the processor changes. This is the case for example for codes written for the first Xeon Phi, where the communication between cores relies on a ring. Optimizations based on this may be irrelevant or even disastrous when applied to the Xeon Phi2 grid of cores.

Moreover, all the parallel systems (MPI, OpenMP, pthreads, Cilk) are based on the OS-thread abstraction. Threads are mostly out of the control of the programmer. They are distributed on cores according to load-balancing. Hence, communicating threads may be distant. Even worse, they are created and managed with no order relationship. This leads to non determinism - the main reason of parallel programming complexity. To quote Lee [9]: threads "make programs absurdly nondeterministic and rely on programming style to constrain that nondeterminism to achieve deterministic aims".

The paper is organized as follows. In section 2 we present a novel organization for the many-core architectural paradigm. The described design simplifies the parallelization process which is fully handled in hardware. The code and data are automatically placed on the available cores according to their order derived from the sequential run. The required communications are automatically detected based on producer to consumer dependencies in machine instructions. Section 3 describes the parallelizing processor micro-architecture. In section 4 we report experimental results obtained from a VHDL prototype synthesized with the Xilinx tool. Section 5 mentions some related works and gives some perspectives.

The paper does not deal with the required programming style which assumes that the program is organized around functions to provide the source of parallelism. Nor does it explain how the memory is managed. The proposed hardware includes a hierarchical memory avoiding coherency control but it is out of the scope of the paper.

## II. AUTOMATIC PARALLELIZATION

A sequential run can exploit Instruction Level Parallelism (ILP) through out-of-order computation. However, Wall [16] measured a rather limited potential of 5 Instructions Per Cycle (IPC) on a processor with infinite size branch predictor and a 256 renaming register set. In [10], we analyzed execution traces showing that dependencies have two origins: algorithmic and architectural. Some producer/consumer dependencies come from the algorithm on which the trace is based. But the biggest part of the dependencies is introduced by the compiler through the Instruction Set Architecture (ISA) employed: register false dependencies, stack pointer management true dependencies, stack locations false dependencies ...

Goossens and Parello [5] showed that when the architectural dependencies are removed the ILP is no more bounded. It increases with the data size. However, independent instructions can be separated by a long sub-trace, e.g. billions

of instructions. A major challenge is to be able to exploit such distant parallelism. Based on this work, the Dali team designed a parallelizing many-core processor [7]. The research results we report here is the implementation of a 64-core prototype in VHDL based on their design, excluding the memory hierarchy.

The processor proposed by the Dali team is a fully operational solution to automatic parallelization. The hardware is able to run any machine code issued from a standard gcc-like compiler. The programmer writes a non threaded C program which is translated into any target ISA. The code can be run sequentially on a single core processor or parallelized on a many-core chip. In order to maximize speed-up, the program should be designed to keep the computing resources filled.

The parallelizing many-core proposed by the Dali team relies on a changed semantic of the machine *call* instruction. The return address is sent to the next core instead of being saved. The code after return is fetched, decoded and out-of-order run in parallel with the called function. Each flow is called a *section*. Dividing the run at each call is a way to jump quickly to very distant parts of the trace and makes it possible to catch distant ILP. It also allows to compute multiple independent paths in parallel which is faster than predicting a single sequential path even if the branch predictor is perfect.

As new parallel sections are dynamically created at each function call, the sequential trace of the run can be deployed in parallel. The major advantage is that the parallel execution can be kept deterministic.

The partially ordered parallel run has two particularities:

- any consumer can be related to its producer (any reader depends on the first preceding writer)
- a consumer can be kept close to its producer by controlling the mapping of the sections on the cores (the partial order serves to place communicators on neighbor cores)

The processor parallelizes by locally running a called function and remotely running its continuation on the next core. The function inherits its caller context as both are run on the same core. The continuation gets the calling environment values it needs through register communication. Such values are called *continuation values*.

The transmission uses a direct connection between the calling core, say *c*, and its neighbor *c+1* hosting the continuation. The continuation is fetched and run in parallel with the called function. If the called function returns a result used in the continuation, this result is sent from core *c* to core *c+1*. The dependent part of the continuation is run once the result is received (i.e. out-of-order).

When the parallelization keeps track of the original sequential order, every communication involves two ordered sections: the producing one sends to the consuming one. When the communication concerns a continuation value the consumer is the core next to the producer. When the communication concerns a result, the consumer core *c+1* is the core next to the caller. But the caller may embed other calls which create other sections. The last created one is the producer. It is placed in core *c+k* with *k>0*. Hence, core *c+k* sends the result to core *c+1*. For this purpose, the cores are linked with a unidirectional ring. The unidirectional pattern of all communications is a direct consequence of the ordered deployment of the parallelization.

As a result, the many-core design does not require a NoC interconnect. Such interconnects are widely used in many-core designs (e.g. Intel KNL [11] and Kalray Bostan [4]) but they are not scalable further than 256 cores. Even for such size, it is wiser to cluster the cores to prohibit or limit communications outside a cluster. The problem does not come from the interconnect itself but from the need to communicate from any core to any core. The scalability is ensured only if the communications are localized between close cores.

## III. The Processor Micro-Architecture

The processor is composed of non-speculative OOO (out-of-order) cores. Each core hosts a set of sections (another name for Simultaneous MultiThreading (SMT) threads [15]). There are 4 sections per core. Sections are the main way to hide latencies involved by communications and memory accesses.

Each core has a 3-stage pipeline (see Fig. 1): Fetch, Decode-Rename (DR) and Issue-Execute-Writeback (IEW).

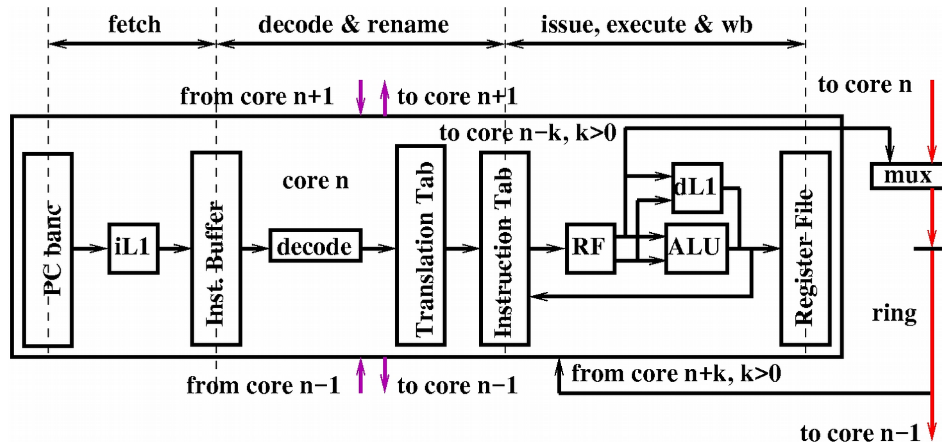


Figure 1. A parallelizing core micro-architecture

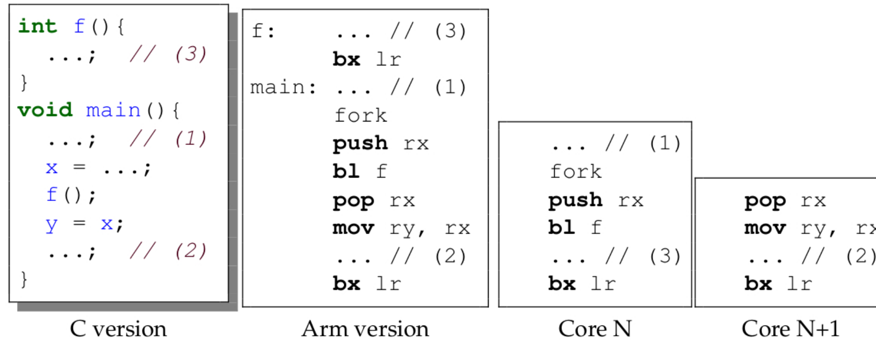


Figure 2. A fork with a continuation value transmission

Each section has its own Program Counter (PC) and a private Translation Table (TT) for renaming.

The fetch stage chooses a ready PC among the section ones. The instruction read in the instruction cache (iL1) is saved in the single place Instruction Buffer (IB) of the same section. The fetching section is suspended until the next PC is known.

The DR stage chooses a ready instruction in one of the IB, which is decoded and renamed through its section TT. The renamed instruction is saved in the section Instruction Table (IT). If the next PC is known (the decoded instruction is not a control instruction or it is an immediate direct branch), the section fetch is resumed.

The IEW stage chooses a ready instruction form IT which is executed (only one instruction issued per cycle).

A branch instruction suspends its section fetch until the target is known. The latency is variable for a conditional branch as the condition may depend on a register received from another section. An instruction cache miss also suspends its section fetch. During these suspension cycles, the three other sections are crucial to keep fetching and filling the instruction tables.

The DR stage monitors machine instructions involved in the parallelization process. They are standard machine instructions (quite no ISA extension is required) with a new semantic. The function call machine instruction (BL/BLX for the ARM ISA) forks the fetch. The function data saving instruction (PUSH) sends a register. The function data restoring instruction (POP) receives a register. The function return instruction (BX LR) stops a section.

The forking protocol (see Fig. 2) reserves a free section on the next core with a special *fork* instruction (Fig. 2 second column; this is the only ISA modification).

The *fork* instruction allocates a free section slot on the next core. If no section is free on the next core when a *fork* instruction is decoded the section switches to *sequential mode*.

After a fork registers can be pushed (i.e. sent) to the allocated section (*push* instruction after *fork* on Fig. 2 second column). The sender part of the communication is prepared and waits until the receiver is ready, i.e. the created section is started.

The new section starts when the call is decoded (*bl* instruction after *push*). The return PC is sent to the allocated section slot in the next core. At the next cycle, both the creator and the created sections fetch in parallel. Hence there is no delay to start a section. This is to be compared to the duration of a thread creation: in *pthread* more than 700 machine instructions are run in the *pthread_create* function.

The created section may pop registers (*pop* instruction after *bl*). The PUSH/POP pairs create hardware rendez-vous implying communications between two adjacent cores.

Column 3 in Fig. 2 shows which instructions are run by the calling core *N*. Column 4 shows which instructions are run by the continuing core *N+1*. Any instruction in core *N+1* depending on register *rx* is kept waiting by the out-of-order machinery relying on standard register renaming (e.g. instruction *mov* after *pop*).

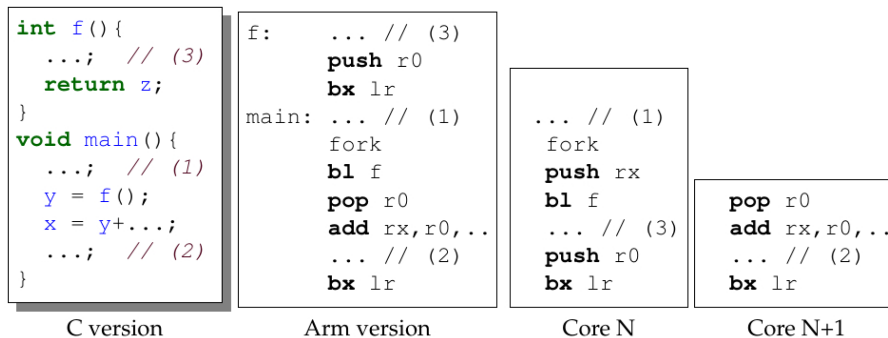


Figure 3. A function returning a value

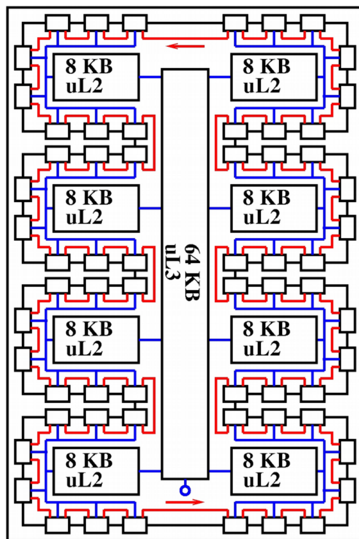

Figure 4. 64-core chip

A function may return a result. Column 2 in Fig. 3 shows how it is achieved (in conformance to the ARM protocol): the result is stored in the *r0* register, and its value is sent before returning (instruction *push r0*). The continuation pops the result with a *pop r0* instruction placed just after the *bl* one. This implies a communication between possibly non adjacent cores, using a unidirectional ring. The sender knows the receiver (traced in a special *Successor* register updated every time a section forks a new section) and sends the pushed value on the ring. The value moves to the receiver (e.g. one cycle between two adjacent cores) and waits there until the *pop* instruction can be issued.

Keeping the original instructions like PUSH/POP with a new semantic allows to run the code in sequential mode. This mode is turned on when a fork fails. The return address is locally saved rather than sent (in the ARM architecture the return address is saved in register *lr*).

The function is called and upon return its return address is restored. The continuation code is sequentially run after the called function. However, if section slots become available in the next core, the continuation code switches back to parallel mode and forks on calls.

In sequential mode the PUSH/POP instructions address a local stack located in the data memory hierarchy. Hence, a continuation is parallelized only when a free section slot is available in the next core. Otherwise, the continuation is serialized.

With a careful design of the program it is easy to write a code which fills all the available section slots in the processor and keep them running sequentially. Each core fetches, decodes and runs one instruction from one of the four available sections per cycle. This execution pattern maximizes the IPC throughput.

Fig. 4 depicts the general organization of the many-core processor on a 64-core chip example. The small rectangles are the cores. Each core is linked to its two neighbors (most external segments). It is also connected to the unidirectional ring (middle oriented segments). A subset of 8 cores share a L2 unified cache (most internal segments). The 64 cores share a L3 Last Level Cache (LLC) which is also connected to the external memory. The size of the caches is kept small because:

- it helps to maximize the number of cores on the die
- temporal and spatial locality is less effective when the run is distributed
- spare sections help to hide cache misses

The L1 size is 1.5KB (1KB for data and 0.5KB for instructions). The L2 size is 8KB. The L3 size is 64KB. Hence, a 64-core chip contains 224KB of cache memory.

Fig. 5 shows a 64-core building block and Fig. 6 a 4096-core chip.

The building block opens the ring between two adjacent cores. The L3 cache is no more a LLC but an intermediate level. Two building blocks can be directly connected with their set of three links (inter-core, ring and L2 bus).

The 4096-core chip on Fig. 6 is made of 64 interconnected building blocks. A subset of 8 building blocks share a L4 cache (512KB) and the 4096 cores share a L5 LLC (4MB) connected to the external memory. The 4096-core chip contains 22.5MB of cache memory (to be compared to the 41MB of the Intel XeonPhi2 KNL).

The building pattern is easily expendable. The scalability is ensured because communications are kept local between neighbor cores. Populating core sections is fast: a new section is allocated in a single cycle (however, it takes at least 16384 cycles to fill the 16384 sections of a 4096-core processor).

The programming model presented in [6] favors local communications. Thus, complex interconnection is no more necessary because distant communications are eliminated.

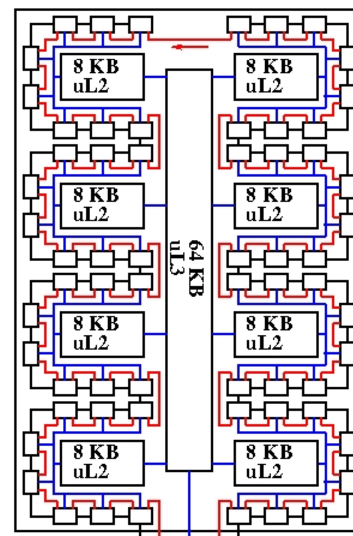

Figure 5. 64 core building block

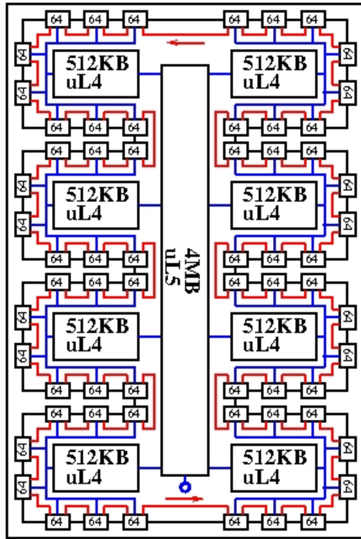


Figure 6. 64*64 core processor chip

## IV. A VHDL PROTOTYPE EVALUATION

The processor has been implemented in VHDL and a prototype has been synthesized with Xilinx tools. It features the register renaming technique used to run out-of-order. The purpose of the implementation is to show that the proposed parallelizing processor works and that it is scalable.

The prototype is based on the ARM v7 ISA [5]. Its actual state allows to create new sections, i.e. parallelize a run. Sections may communicate through PUSH/POP instructions sending and receiving register values (either through a direct link between adjacent cores or via the unidirectional ring to propagate a result).

Fig. 7 shows the implemented 64-core processor. As already mentioned, the memory hierarchy has not yet been implemented, nor the switch to the sequential mode. The big rectangles represent the cores, and are directly linked to their neighbors (lines in-between the cores). The others lines

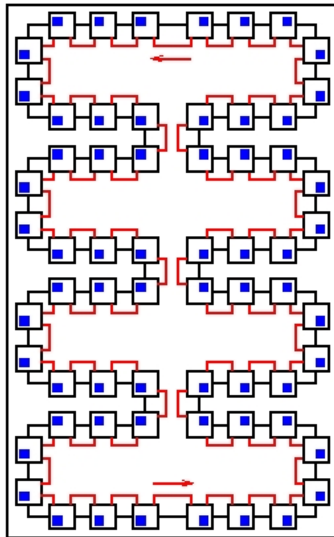

Figure 7. 64-core prototype

(marked with the arrows) represent the ring. The small rectangles inside the cores symbolize asynchronous ROM modules with read-only behavior used as instruction memory. Their content is initialized from an external file (pure binary data). The contents of the files are based on the ARM instruction encoding generated by the cross-compiler with some hand-made modifications: insertion of fork instructions, removal of unnecessary instructions (e.g. pushing the return address of the main function onto the stack, alignment directives, ..), labels translations to instruction-relative addresses.

Fig. 8 illustrates the material implementation of the forking protocol presented on Fig. 2. When the *fork* instruction is decoded on core N (Fig. 8 assumes the fork occurs in section number 0), the core emits the *forkDecoded* signal to the next core to book a new section. If a free section slot is available on core N+1 (which happens to be section 1, chosen in the *FreePCSelection* table), the section 0 successor (*succ0* in the core N *Successor* table) is copied to the newly booked section on core N+1 (*forkingSuccessor* signal).

If no continuation values are sent, the next decoded instruction is a function call (BL) (the other case is described in the next paragraph). Core N sends PC+4 to the next core (*prevCorePCFork* signal). The created section is marked as ready and can henceforth be chosen to fetch in the next cycle. The creating section sets its own PC to the address of the called function to continue along the sequential path.

Fig. 9 depicts the communications between cores when a continuation value is transmitted (cf Fig. 2). The instruction initiating the transfer (*push rx*) is decoded after the new section has been booked on core N+1. It is stored in the Instruction Table (IT) and waits until the *rx* register value is computed and the *popSectNum* signal is received. This signal is sent when a matching *pop* instruction is decoded on core N+1. When both conditions are satisfied, the *push* renamed instruction can be issued from IT. It transmits the local renaming of the *rx* register to the *pop* instruction (*pushSrcRen*), and marks *pop* as ready.

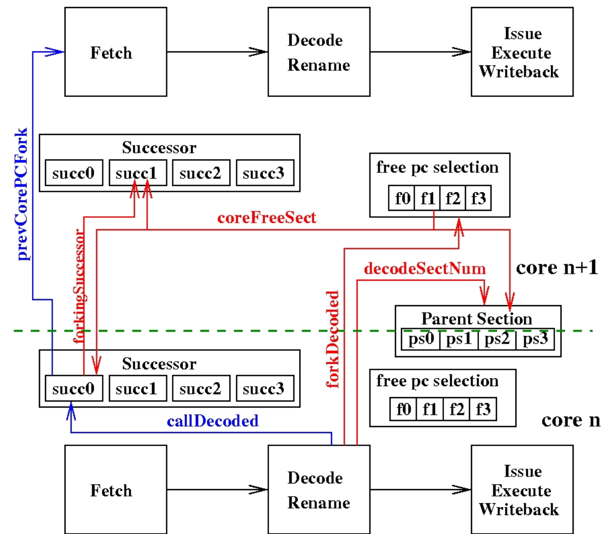


Figure 8. Material implementation of the forking protocol

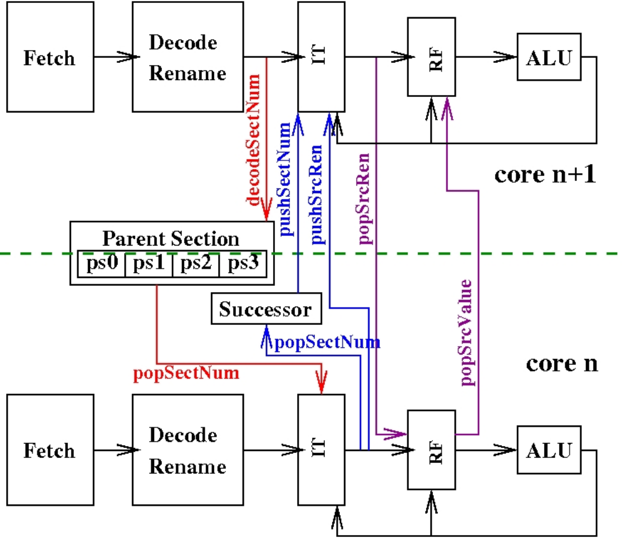


Figure 9. Material implementation of continuation value transmission

When issued (from the next cycle onwards), *pop* reads the register value in the core N Register File (RF) through a dedicated port (*popSrcValue*).

Unlike continuation value transmission, the communication in the case of a result transmission may concern non adjacent cores. This is one of the reasons why we implemented a different synchronization protocol for *push* and *pop* instructions in this case. Another reason is that the *push*/*pop* pair might be decoded out-of-order. The *pop r0* instruction can be read before the matching *push r0* instruction (this order is not possible for the continuation value transmission *push/pop* pair).

A result is transmitted through a direct connection if the destination is the next core, which is the most common situation. If the destination core is a distant one, the result is propagated through the ring.

The renamed *push r0* instruction stored in IT is ready to issue when the transmitted value is computed and when the segment of the ring connecting core N to core N-1 is free. When those conditions are fulfilled, the instruction executes. The value moves one core backward at each cycle until it reaches its destination where it is written into a special register associated to the emitting section.

Fig. 10 shows the result transmission when the destination is a neighbor core (cf Fig. 3). This is checked when the *push r0* instruction is issued by reading the *Successor* table. The register value (*pushValue*) is sent directly to core N+1 *Result Buffer* (RB). When the matching *pop r0* instruction is issued on core N+1, it reads its source value in RB instead of RF.

Table 1 gives the synthesis results on two devices with the following parameters: 4 sections (i.e. 4 PC, 4 IB, 4 TT), 32 micro-architectural general purpose registers (RF) and 8 micro-architectural registers dedicated to the ARM state register renaming (PSR).

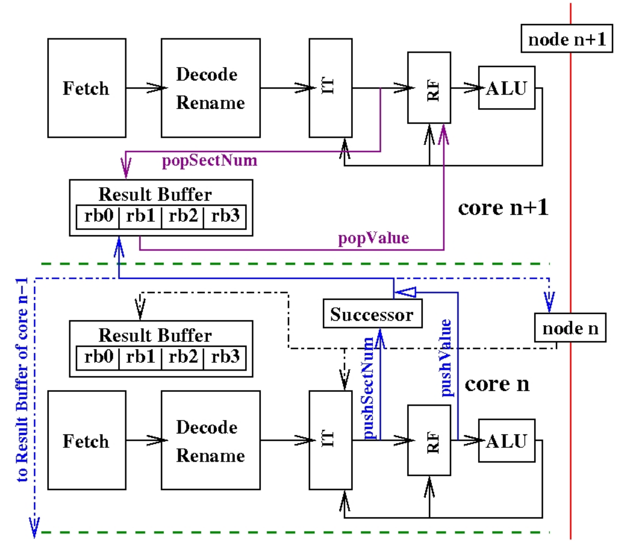


Figure 10. Material implementation of result transmission

The frequency remains stable whatever the number of cores which means that the single core hardware complexity is independent of the number of cores in the processor.

The size of a core (without the caches) is about 8000 LUTs.

The synthesis of the many-core performed with the Xilinx ISE 14.7 tool confirms the scalability of the architecture.

We have run a few tool programs composed of a few instructions to check that the following parts of the many-core work:

- the renaming unit
- the forking unit
- the continuation value transmission
- the result transmission (between adjacent and non adjacent cores)

TABLE I. SYNTHESIS RESULTS

| Cores | Virtex 6, XC6VLX550T | | Virtex 7, XC7VX550T | |
|---|---|---|---|---|
| | *Max. frequency* | *LUT-s* | *Max. frequency* | *LUT-s* |
| 4 | 41.4 MHz | 34012 | 51.2 MHz | 33950 |
| 8 | 41.5 MHz | 68386 | 51.3 MHz | 67634 |
| 16 | 41.5 MHz | 135522 | 51.3 MHz | 135942 |
| 32 | 41.2 MHz | 275017 | 51.4 MHz | 271575 |
| 64 | 41.1 MHz | 530499 | 51.3 MHz | 518096 |

## V. RELATED WORKS AND PERSPECTIVES

The parallelizing processor efficiency mainly depends on the ILP. Instruction Level Parallelism has been thoroughly studied, mainly in the 90's. In 1967, Tomasulo [14] proposed a method to reorder the execution of floating point instructions. He invented the register renaming mechanism. However in the same time Tjaden and Flynn [13] measured on sample programs that their average ILP was 1.86. At this time, we had a way to exploit ILP but as processors only had a scalar data path, renaming was mainly used to hide long operations latencies.

We had to wait for more than 20 years to see a renewed interest in ILP with the introduction of superscalar processors. In 1991, David Wall [16] tried to find ways to increase ILP. He measured that on a set of benchmarks the ILP could be increased up to 5 through register renaming, i.e. false register dependency elimination. He also measured that when the renaming register set was unlimited, the ILP could increase much more, reaching 25 on the average (with an issue width limited to 64 instructions per cycle). By this time, we knew that ILP can be high but difficult to catch.

In 1992, Austin and Sohi [1] analyzed the ILP distribution in the trace of the instructions run. They showed that ILP can be found at any distance from the fetch point. They also revealed that the stack accesses create a serialization all along the run. Simultaneously, Lam and Wilson [8] studied the impact of control instructions on ILP. They suggested that to catch distant ILP, multiple parallel fetches would be efficient.

At the end of the 90's it was commonly admitted that adding resources to increase ILP capture was not enough rewarded on a single core processor. This is known as the ILP wall. For the second time ILP was abandoned as a source of performance.

The multiplication of cores on a chip, starting with the IBM Power4 in 2001 [12], may resurrect ILP as the parallelizing processor proposed in this paper shows. A many-core processor has the capacity to follow multiple paths in parallel as suggested by Lam and Wilson. It also has thousands of registers, increasing the renaming capability as studied by Wall. Eventually, the stack can be considered useless to save and restore registers and return addresses as they can be transmitted instead. The serializing factor shown by Austin and Sohi can be eliminated.

However, the researchers have mainly concentrated their efforts on the development and capture of Thread Level Parallelism (TLP). When threads are independent, TLP can be as high as the number of threads. However, TLP has the following problems:

- non determinism which creates synchronization and communication difficulties, the main one being to keep shared memory coherent
- large management overhead (thread creation, synchronization, termination, mutual exclusion) forbidding fine grain parallelism
- uncontrolled placement implying the necessity to communicate between distant cores

In 2013, Goossens [5] separated algorithmic dependencies from architectural ones. The former are producer to consumer dependencies between program variables and control dependencies. The latter include false register dependencies and stack management and reuse dependencies. When removing the architectural dependencies, it has been measured that the ILP is no more bounded. It increases with the data size. In other words, in this assumption the ILP wall no more exists.

Many-core processors may capture distant ILP. Instead of statically dividing a program into threads to expose its parallelism, the programmer can let the hardware do the job. The processor can better place the parallel pieces of code from the sequential order: communicating parts are settled in adjacent cores. The sequential order simplifies the hardware needs, removing the necessity of both a NoC [3] and a directory-based memory coherency system [2].

To complete the evaluation of the parallelizing processor, a trace driven cycle-accurate simulator is developed which is not described in this paper. This simulator is useful to test the parts of the many-core not included in the prototype, i.e. the memory hierarchy and the sequential execution mode. It also quantifies the parallelization efficiency (IPC measures) when benchmarks are adapted to fill the sections and to avoid distant communications.

The simulator has already been tested on a few kernel benchmark examples. It is from such simulations that the number of sections per core has been fixed to 4. This is enough to hide latencies.

The simulator allows to vary the number of cores and sections. Simulations have shown that the implemented mechanism of automatic parallelization never reaches a deadlock. The kernel benchmarks have been tested on a single core and single section machine (sequential run) as well as on a 4096-core and 4 section processor (parallelized run).

The simulator simulates unmodified ARM code generated by the gnu gcc cross compiler (gcc-arm-linux-gnueabi). To increase the parallelization speed-up, the callee-save protocol generated by gcc can be replaced by a caller-save one (the caller pushes, i.e. sends the saved registers which the continuation pops, i.e. receives).

The simulator has not yet been tested on real benchmarks because it is necessary to rewrite them to eliminate a lot of sequentializing features (from the algorithm or from the data structures), like one has to rewrite an application to thread it. Hence we have no measure on the performance of the processor yet.

The actual VHDL prototype should be soon completed by the implementation of the switch from parallel to sequential mode. It assumes we add a data memory to hold the stacks. Our results should also be complemented with measures of the energy consumption.

# REFERENCES

[1] Austin (T.M.) and Sohi (G.S.). "Dynamic Dependency Analysis of Ordinary Programs." ISCA'92, 342–351, 1992

[2] Censier (L.M.) and Feautrier (P.). "A New Solution to Coherence Problems in Multicache Systems". IEEE Transactions on computers, Vol C-27, n°12, 1112-1118, december 1978

[3] Dally (W..J.) and Towles (B.). "Route Packets, Not Wires: On-Chip Interconnection Network". In *Proc. of the 38th Design Automation Conference (DAC)*, june 2001.

[4] Dinechin (B.D.), Graillat (A.) "Network-on-Chip Service Guarantees on the Kalray MPPA-256 Bostan Processor" AISTECS: 2nd International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems, HiPEAC 2017 Stockholm Conference, January 2017

[5] Goossens (B.) and Parello (D.). "Limits of Instruction-Level Parallelism Capture". Procedia Computer Science, vol. 18, 2013, pp. 1664 – 1673.

[6] Goossens (B.), Parello (D.), Porada (K.) and Rahmoune (D.). "Parallel Locality and Parallelization Quality". In PMAM : Programming Models and Applications for Multicores and Manycores, Barcelona, Spain, march 2016.

[7] Goossens (B.), Parello (D.), Porada (K.) and Rahmoune (D.). "Computing On Many Cores". In Concurrency and Computation : Practice and Experience. Published online in Wiley InterScience DOI : 10.1002/cpe, april 2016.

[8] Lam (M.S.) and Wilson (R.P.). "Limits of Control Flow on Parallelism". ISCA'92, 46–57, 1992.

[9] Lee (E.A.). "The Problem With Threads". Computer, vol. 39, n°5, may 2006, pp. 33–42.

[10] Porada (K.), Parello (D.) and Goossens (B.). "Analyse et réduction du chemin critique dans l'exécution d'une application". (in french) In ComPAS : Conférence en Parallélisme, Architecture et Système, Neuchâtel, Switzerland, april 2014.

[11] Sodani (A.). "Knights Landing (KNL) : 2nd Generation Intel Xeon Phi Processor". In IEEE Hot Chips 27 Symposium (HCS), pp. 1–24, august 2015.

[12] Tendler (J.M.), Dodson (J.S.), Fields (J.S.), Le (H.) and Sinharoy (B.). "POWER4 System Microarchitecture". IBM Journal of Research and Development. 46 (1): 5–26. doi:10.1147/rd.461.0005, 2002.

[13] Tjaden (G.S.) and Flynn (M.). "Detection and Parallel Execution of Independent Instructions". IEEE Trans. on Comp., Vol. c-19-10, 889–895, 1970.

[14] Tomasulo (R.). "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". IBM JRD, vol. 11-1, 25–33, 1967.

[15] Tullsen (D.M.), Eggers (S.J.) and Levy (H.M.). "Simultaneous Multithreading: Maximizing On-chip Parallelism". 25 Years of the International Symposia on Computer Architecture (Selected Papers), ISCA '98, ACM: New York, NY, USA, 1998; 533–544.

[16] Wall (D.W.). "Limits of Instruction-Level Parallelism". SIGOPS Oper. Syst. Rev., vol. 25, Special Issue, april 1991, pp. 176–188.