Deterministic OpenMP and the LBP Parallelizing Manycore Processor

Bernard Goossens, Kenelm Louetsi, David Parello

Deterministic OpenMP and the LBP Parallelizing Manycore Processor

Bernard Goossens[a,*], Kenelm Louetsi[a], David Parello[a]

[a] *LIRMM UMR 5506, CC477, 161 rue Ada, 34095 Montpellier Cedex 5 - France*
UPVD, 52 avenue Paul Alduy, 66860 Perpignan Cedex - France

Abstract

This paper presents Deterministic OpenMP, a new runtime for OpenMP programs, and the Little Big Processor (LBP) manycore processor design. LBP isolates the application it runs, using its multiple cores to distribute a non interruptible single parallelized computation. When run on LBP, a Deterministic OpenMP code produces cycle by cycle deterministic computations. LBP and Deterministic OpenMP are particularly suited to safely accelerate real time embedded applications through their parallel execution. The paper reports experimental results to measure the LBP performance on an FPGA based simulator and compares LBP with a Xeon Phi 2. The instruction retired and Instruction Per Cycle (IPC) measured in the experiment indicate that LBP performance does not suffer from the elimination of all the high performance features incompatible with an embedded processor (predictors, caches). LBP performance comes from its hardware parallelizing capability, its capacity to capture distant Instruction Level Parallelism (ILP) and from our proposed Deterministic OpenMP runtime favoring local memory accesses.

Keywords: OpenMP, Determinism, Parallel Programming, Repeatability, Distant ILP, Isolation

1. Introduction

OpenMP developers must correctly synchronize the parallel parts of their applications to avoid non-determinism or unnecessary serializations.

Non-determinism makes parallel programs in general and OpenMP ones in particular hard to debug as a bug may be non repeatable and the action of a debugger may alter the run in a way which eliminates the emergence of the bug.

It also makes the timing of a run non repeatable. Measuring a speedup is a complex and far from scientific process. A lot of precautions, varying from one

*Corresponding author
Email addresses: goossens@univ-perp.fr (Bernard Goossens), kenelm.louetsi@univ-perp.fr (Kenelm Louetsi), david.parello@univ-perp.fr (David Parello)

platform to another, must be taken to isolate the run and eliminate as much perturbations as possible, either from external parasites or from the measure itself. Moreover, many runs must be done with a new measure for each. The final measure can be the lowest achieved time (understood as a smallest parasitism), but usually people prefer the average of all the measures (understood as an average parasitism).

One side effect is that parallelization can hardly benefit to real time critical applications[1] as a precise timing cannot be ensured.

In this paper, we introduce Deterministic OpenMP (section 3), Parallel Instruction Set Computer (PISC) (section 4) and the Little Big Processor (LBP) (section 5). They were designed to offer the performance of parallelism to safety-critical real time embedded applications. In this domain, safety is more important than performance and up to now, industrials have preferred to keep away from parallelism and favor older single core in-order processors. But they need more and more performance and the available CPUs are almost all multicore ones today.

We show from experimental results that: (1) all the internal timings of a Deterministic OpenMP program run on an LBP processor are repeatable, (2) the overhead to parallelize a run is low and (3) the 64 cores and 256 threads LBP processor, even though it has no high performance unit (to keep it aimed to embedded applications), sustains the high demand of a matrix multiplication[2] without saturating its hierarchical bus-based interconnect. In other words, LBP aims to be efficient, safe and low-power.

Deterministic OpenMP is an on-going implementation of the OpenMP standard. The goal of Deterministic OpenMP is to enforce determinism through hardware. The control of the synchronizations is no more a matter of locks, barriers and critical sections inserted by the programmer, properly or not, but is handled automatically by the hardware.

Some standard OpenMP programs can be run on LBP simply by replacing the OpenMP header file by our Deterministic OpenMP one. A new runtime is added by a preprocessor to launch the team of threads on the processor *harts*[1].

In this paper, we define PISC, which is an extension of the RISC Instruction Set Architecture (ISA). PISC adds a few machine instructions to fork, join and send/receive registers directly in the hardware. We have extended the RISCV ISA with a PISC ISA named X_PAR.

LBP is a 64-core processor we have designed, to be compared with industrial manycores like the 72-core Xeon Phi2 [3] or the 80-core Kalray Coolidge MPPA3 [4] but aiming different applications. LBP is based on a much simpler core hardware and interconnect, implementing a new hardware-based parallelization paradigm. As the parallelization of a run can be handled through PISC machine instructions, it may be deployed on an LBP manycore processor without the help of an Operating System (OS). Parallelism is available on bare-metal hardware. Simpler versions of LBP (4 harts in 1 core or 16 harts in 4 cores) can be also

[1] A hart is a hardware thread, as defined in the RISCV specification document[5].

used as high performance microcontrollers for automotive industry applications such as autopilot helpers.

The LBP design aims to keep a rate of one Instruction Per Cycle (IPC) per core with no branch predictor, no cache-coherent memory hierarchy, no Network-on-Chip (NoC) and no memory access ordering unit (e.g. a load/store queue). The single core throughput relies on multithreading with four harts. When they are all active, latencies are fully hidden as the experiment reported in section 7 indicates.

By combining Deterministic OpenMP with an LBP processor, a developer can benefit from a 64-core 256-hart processor to safely parallelize his applications. A Deterministic OpenMP program applied many times to the same input on LBP has an invariant number of retired instructions and most of the time, has an invariant number of cycles and produces an unchanging set of events at every cycle. The LBP processor can achieve *cycle determinism*, i.e. the cycle by cycle ordering of events in the processor is invariant for a given program and data. This is obtained through a perfect isolation of the application run[6]. With cycle determinism in LBP, a run can be described by invariant statements as "*at cycle 467171, core 55, hart 2 sends a memory request to load address 106688 from memory bank 13; at cycle 467183, it writes back the received data*"[2]. The statement holds for any run of the same program applied to the same data on the same LBP processor.

Cycle determinism may be observed on a LBP run only when the program has no interaction with the outside world. For example, a program having an alerting temperature sensor input would depend on a non-deterministic external event and its run would not exhibit cycle determinism (the number of cycles of the run would not even be constant). However, even in such a situation, the LBP run still ensures proper ordering in accordance with the referential sequential order[3] (or logical clock as defined by Lamport [7]) (a computation depending on the non-deterministic event occurs after it, which is ensured by the LBP hardware).

Moreover, LBP does not handle external events though interrupts. LBP is a non interruptible processor. A computation which has to wait for an input, being it produced by a timer as in real time applications, is not stopped and resumed after an interrupt coming from the input device. This classic behaviour breaks the isolation, which impacts the order of the instructions run. On LBP, the input code position in the static code fixes a semantic which the hardware realizes. The instructions are run in the partial order of their dependencies, with no software synchronization like context switch and interrupts. This is illustrated in section 6.

[2]taken from the tiled matrix multiplication run as reported in section 7

[3]the sequential order is informally defined as the one observed when the code is run sequentially

2. Related works

There are previous works on building deterministic hardware for safety critical applications [8][9]. None achieves cycle by cycle internal determinism as LBP does through the concept of hardware parallelization. A recent paper [10] has evaluated the capacity for OpenMP to be suited to safety critical applications, highlighting functional safety problems of non-conforming programs, unspecified behaviors, deadlocks, race conditions and cancellations. This work is complementary to ours which focuses on timing safety.

LBP does not use speculation to reduce latencies. Instead, LBP favours latency hiding through multithreading and distant Instruction Level Parallelism (ILP) exploitation to fill the core pipelines. It is worth to mention the dangers of speculative hardware for privacy (another aspect of the secutity level required for safety critical applications) as Spectre[12], Meltdown[11] and more recently Spoiler[13] have shown. As LBP does not speculate, it does not speculatively modify any internal resource (branch predictor or cache) which, in speculative processor, can be observed by a spying application, indirectly revealing any stored secret.

Even though LBP does not speculate, it is able to fill each core pipeline, thanks to multithreading. Multithreading is a secure substitute to speculation as soon as the multiple threads are all parts of the same parallelized application. The pipeline bubbles are filled by instructions of another thread belonging to the same computation, as a compiler would fill the delays by inserting independent instructions in the holes. However, LBP can insert instructions from an arbitrarily distant hart, which the compiler cannot. The OpenMP paradigm exhibits distant ILP by its *parallel* pragmas, which is used by LBP to keep close to the peak 1 IPC per cycle.

If local ILP is known to be severely bounded (with an average of 6 when full register renaming is applied), distant ILP is also known to be unbounded, increasing with the size of the computation. The stack used to allocated function frames [14] and the control flow [15] are the two main obstacles to ILP. The OpenMP parallelization eliminates the dependencies between the stack frames and between the control flow paths for independent threads. In [16], the authors show that when removing the stack and control dependencies (i.e. as if the function evaluations would be done in full parallelism), the global ILP is the sum of the functions individual ILPs, unboundly increasing with the size of the data, i.e. the number of function calls.

3. Deterministic OpenMP

A Deterministic OpenMP program is quite not distinguishable from a classic OpenMP one[17]. Figure 1 shows an example of a Deterministic OpenMP code to distribute and parallelize a *thread* function on a set of eight harts. The difference with a pure OpenMP version lies in the header file inclusion (*det_omp.h* instead of *omp.h*, in red on the figure).

```
#include <det_omp.h>
#define NUM_HART 8
void thread(/*...*/){
 /*... (1);*/
}
void main(){
 int t;
 omp_set_num_threads(NUM_HART);
 #pragma omp parallel for
  for (t=0; t<NUM_HART; t++)
   thread(/*...*/);
 /*... (2);*/
}
```

Figure 1: A Deterministic OpenMP program.

In a classic OpenMP implementation, the *parallel for* pragma builds a team of OMP_NUM_THREADS threads which the OS maps on the available harts, optionally balancing their loads. In the Gnu implementation, this is done through the *GOMP_parallel* function (OMP API[18]). In Deterministic OpenMP, a team of harts -not threads- is created, each matching a unique and constant placement on the processor. One drawback is that on LBP, load balancing is the problem of the programmer. It is his responsability to evenly divide his job into parallel tasks. If properly done, the efficiency is improved compared to a dynamic load balancing handled by the OS because dynamic load balancing implies costly thread migrations.

The Deterministic OpenMP code in figure 1 is translated into the code in figure 2. The text in black on the figure comes from the original OpenMP source code on figure 1. The green text is added by the translator.

The *LBP_parallel_start* function creates and starts the team of harts. It organizes the distribution of the copies of function *thread* on the harts. It calls *fork_on_current* which creates a new hart on the current core or *fork_on_next* which creates a new hart on the next core (LBP cores are ordered). The machine code for *fork_on_current* is given in section 4. The *LBP_parallel_start* function fills the harts available in a core before expanding to the next one.

Functions *fork_on_current* and *fork_on_next* do not interact with the OS by calling a forking or cloning system call. Instead, they directly use the hardware capability to fork the fetch point, running the *thread* function locally and the continuation remotely (on the same core or on the next one). Figure 3 shows how the different copies of function *thread* are distributed on a 4-core and 16-hart LBP processor (only two cores are filled by the eight iterations of the *parallel for* loop in the example).

The hardware fork mechanism has two advantages over the classic OS one:

```
#define NUM_HART 8
#define HART_PER_CORE 4
unsigned omp_num_threads;
typedef struct type_s{int t; /*...*/} type_t;
type_t st;
void thread ( void *arg ){
   type_t *pt=(type_t *)arg;
   /*...(1);*/
}
static inline void
fork_on_current(void(*f)(void*), void *data){/*p_fc(data);*/}
static inline void
fork_on_next(void(*f)(void*), void *data){/*p_fn(data);*/}
void LBP_parallel_start(void(*f)(void*), void *data){
 type_t *pt=(type_t *)data;
 unsigned nt=omp_num_threads, h, t;
 for (t=0; t<nt-1; t++){
  h=t%HART_PER_CORE;
  pt->t=t;
  if (h<HART_PER_CORE-1) fork_on_current(f, data);
  else fork_on_next(f, data);
 }
 pt->t=nt-1;
 f(data);
}
void main(){
 omp_num_threads=NUM_HART;
 LBP_parallel_start( thread , (void *)&st);
 /*...(2);*/
}
```

Figure 2: The Deterministic OpenMP translator transformation.

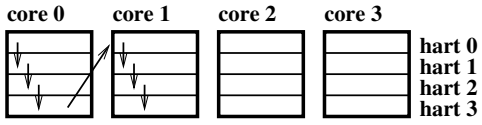

Figure 3: Distributing a team of threads on harts.

- it concatenates the continuation thread to the creating one in the sequential referential order, on which the hardware is able to synchronize and connect producers and consumers,
- it places the continuation thread on a fixed hart, in the same or next core.

In a classic OpenMP run of the code on figure 1, all the function *thread* copies would become non-ordered and independent threads. In contrast in Deterministic OpenMP, the created harts are ordered (in the iterations order) which simplifies communications: a creating hart sends continuation values to

the created one through direct core-to-core links.

LBP offers the programmer the possibility to map his code and data on the computing resources according to the application structure. A producing function can be parallelized on the same set of cores and harts than the consuming one, eliminating any non local memory access. The OS is not able to do the same for OpenMP runs as it has no knowledge of which thread produces and which thread consumes. The OS can only act on load balancing. The task of good mapping in classic OpenMP is the programmer's duty. To do his job properly, he has to deal with his application, but also with the OS and the computing and memory resources (e.g. load balancing, pagination). This leads to difficult decisions, with a complexity proportional (if not worse) to the number of cores.

```
#include <det_omp.h>
#define NUM_HART 16
#define SIZE (1<<16)
int v[SIZE];
void thread_set(int v[], int t){
 /*init chunk t of v*/
}
void thread_get(int v[], int t){
 /*use chunk t of v*/
}
```

```
void main(){
 int t;
 omp_set_num_threads(NUM_HART);
#pragma omp parallel for
 for (t=0; t<NUM_HART; t++)
  thread_set(v,t);
#pragma omp parallel for
 for (t=0; t<NUM_HART; t++)
  thread_get(v,t);
 /*sequential part*/
}
```

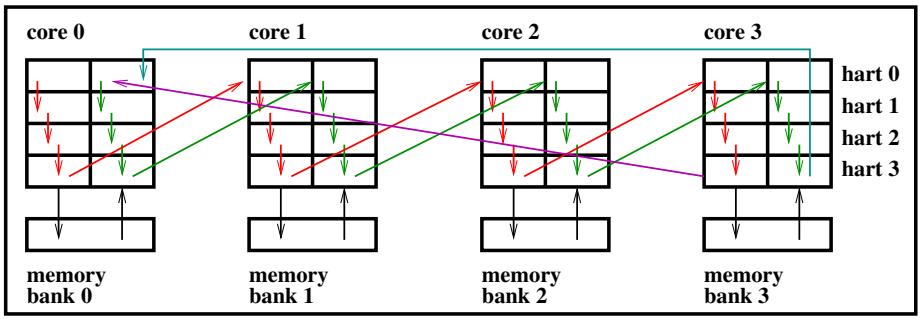

Figure 4: A synchronized and placed execution of a Deterministic OpenMP code.

Figure 4 shows a code with two successive parallel parts embedded in function *main* (upper right). The first one initializes global vector v through function *thread_set* which the second part uses through function *thread_get*. Thanks to the sequential referential order, the second part is serialized by LBP after the first one with no OS intervention. Thanks to the placement of the harts created by the two parts on an aligned set of cores, all the memory accesses reference a local memory bank. Each *thread_get* call reads the vector elements it consumes

from its local memory bank where the *thread_set* call has written them. The producing hart, the consuming hart and the data chunk they access all reside in the same LBP core/local memory.

The consuming phase is perfectly separated from the producing one by the hardware barrier inserted between the two parallelized loops. The hardware ensures that the *thread_get* loads occur after the *thread_set* ones with no added OS synchronizing primitive nor hardware memory coherence mechanism.

The bottom part of the figure represents 4 cores, i.e. 16 harts. Each red arrow (on the left half of each hart representing an iteration of the first *parallel for* loop) stands for a LBP fork (hart creation) of a *thread_set* function. The last hart in a core forks on the next core (raising diagonal red arrow).

The green arrows match the *thread_get* function calls (second *parallel for* loop). The magenta arrow (right to left raising diagonal) is the join separating the set phase from the get one (barrier between the two *parallel for*). The join sends the continuation address to the initial hart (i.e. hart 0 core 0 on the example). The cyan arrow (multiple segments line from bottom right to top left) is the join separating the get phase from the last sequential part of *main*. The sequential referential order is materialized when following successively the red, magenta, green and cyan arrows. The hardware memorizes the necessary links to manage this order and uses them to synchronize harts and make them communicate.

4. The PISC ISA

The PISC ISA extension is a set of 12 new machine instructions. A RISCV extension named X_PAR has been defined. It is summarized on figure 5.

The *p_swcv* and *p_lwcv* instructions (*cv* stands for *continuation value*) serve to achieve a hardware synchronized communication between a producer and a consumer of the same team, for example to transmit an input argument from member to member (e.g. the iteration loop index). The consumer should be the hart next after the producer (same or next core).

The *p_swre* and *p_lwre* instructions (*re* stands for *result*) serve to achieve a hardware synchronized communication between a producer and a consumer of different teams, with the consumer physically preceding the producer (same or preceding core; the connection used to transmit the value is the intercore backward link). They allow for example a team to produce a reduction value and have its last member send it to the join hart.

The *p_jal* instruction is a parallelized call. Instead of pushing the return address on the stack, it sends it to an allocated continuation hart. The *p_jalr* instruction is the indirect variant of the *p_jal* one. It can also be used as a return from a parallelized hart (pseudo instruction *p_ret* standing for *p_jalr zero, rs1, rs2*).

The *p_fc* and *p_fn* instructions serve to allocate a new hart, either on the same core or on the next one.

syntax	semantic
p_lwcv rd, offset	restore rd from local stack at offset
p_swcv rs1, rs2, offset	save rs2 on rs1 hart stack at offset (allocated hart)
p_lwre rd, offset	restore rd from local result buffer number offset
p_swre rs1, rs2, offset	save rs2 to rs1 hart (any prior hart) result buffer number offset
p_jal rd, rs1, offset	send pc+4 to rs1 hart (allocated hart) clear rd goto pc+offset
p_jalr zero, rs1, rs2 (p_ret)	if rs1==0 && rs2==−1: exit if rs1==0 && rs2!=current hart: end current hart send ending hart signal to next hart if rs1==0 && rs2==current hart: keep current hart waiting send ending hart signal to next hart if rs1!=0: send rs1 to rs2 hart (join hart)
p_jalr rd, rs1, rs2	send pc+4 to rs2 hart (allocated hart) clear rd goto rs1
p_merge rd, rs1, rs2	rd=(rs1&0x7fff0000) | (rs2&0x0000ffff)
p_fc rd	allocate a free pc on current core (fork) rd=(4*c+allocated hart)
p_fn rd	allocate a free pc on next core (fork) rd=(4*(c+1)+allocated hart)
p_syncm	stop fetch until all decoded memory accesses in local hart are run
p_set rd, rs1	rd=(rs1&0x0000ffff) | ((4*core+hart)<<16) | 0x80000000

Figure 5: The X_PAR RISCV PISC ISA extension.

The *p_merge* and *p_set* are instructions used to manipulate hart identities. They are used to prepare and propagate the first team member identity to allow the join from the last team member back to the first.

The *p_syncm* instruction serves to synchronize memory accesses within a hart. In a hart, loads and stores are unordered. The hardware provides no control on the out-of-order behaviour of loads and stores. For example, to ensure a load depending on a store is run after it, a *p_syncm* instruction should be inserted between them. The *p_syncm* acts by blocking the fetch (as soon as it is decoded) until all the in flight memory accesses of the hart are done.

More details on PISC can be found at URL [19].

Figures 6, 7 and 8 show the machine instructions compiled for the Deterministic OpenMP code on figure 2. The target processor is assumed to be bare-metal (no OS). The LBP processor implemented in the FPGA directly starts running the *main* function and stops when the *p_ret* instruction is met (with register *ra*=0 and register *t0*=-1, meaning exit).

Figure 6 shows the compiled PISC RISCV code for the *main* function. Registers *ra* and *t0* play a special role. Register *ra* has its normal usage: it holds

```
main: li     t0,-1      #t0 = exit code
      addi   sp,sp,-8   #allocate two words on local stack
      sw     ra,0(sp)   #save reg. ra on local stack, offset 0
      sw     t0,4(sp)   #save reg. t0 on local stack, offset 4
      p_set  t0         #t0 = 4*core+hart (current hart identity)
      li     a0,thread  #a0 = thread function pointer
      li     a1,data    #a1 = pointer on data structure
      jal    LBP_parallel_start
rp:   /*...(2)*/
      lw     ra,0(sp)   #restore ra from local stack, offset 0
      lw     t0,4(sp)   #restore t0 from local stack, offset 4
      addi   sp,sp,8    #free two words on local stack
      p_ret             #ra==0 && t0==-1 => exit
```

Figure 6: The PISC RISCV code for the *main* function.

the return address. When *LBP_parallel_start* is called, *ra* receives the future team join address (labeled *rp* on the figure). Register *t0* holds the current hart number set with the *p_set* instruction and propagated along the team members through register transmission (*t0* contains a value combining the hosting core and the current hart identity in the core). The *LBP_parallel_start* function creates the team of harts to run the parallelized loop. It returns to *rp* label when a *p_ret*[4] instruction in the last created team member is reached. The ending hart sends *ra* to hart *t0* (i.e. core 0, hart 0 in the example), which resumes the run at *rp* label.

There are four types of team member endings (continuation after a *p_ret* instruction) according to the received *ra* and *t0*:

1. *ra* is null and *t0* is not the current hart: the hart ends,
2. *ra* is null and *t0* is the current hart: the hart waits for a join,
3. *ra* is null and *t0* is -1: the process exits,
4. *ra* is not null: the hart ends and sends *ra* to the *t0* hart which restarts fetch (the parallel section ends and is continued by a sequential one).

The *p_ret* instructions are committed in-order (in the sequential referential order) to implement a hardware separation barrier between a team of concurrent harts and the following sequential OpenMP section. The barrier is implemented as a hardware signal transmitted from hart to hart along the team members. Hence, the harts are released in order (each hart commits its *p_ret* only when it has received the *ending hart* signal from its predecessor; it sends its own *ending hart* signal to its successor after this commit).

Figure 7 shows how function *LBP_parallel_start* calls the last occurrence of function *thread* with a RISCV *jalr a0* indirect call instruction. This last call is run by the last created team member on the last allocated hart (i.e. no fork). The same hart runs the code after the return point at *rp2* label. The ending *p_ret* instruction joins with the following sequential ending part of *main* (team member ending type 4 with *ra* being not null).

[4] *p_ret* is a pseudo-instruction to end a hart (*p_jalr zero,ra,t0*)

```
        addi   sp,sp,-8   #allocate two words on local stack
        sw     ra,0(sp)   #save reg. ra on stack, offset 0
        sw     t0,4(sp)   #save reg. t0 on stack, offset 4
        p_set  t0         #t0 = 4*core+hart (current hart identity)
        jalr   a0         #a0 is the pointer on function thread
rp2:    lw     ra,0(sp)   #restore reg. ra from stack, offset 0
        lw     t0,4(sp)   #restore reg. t0 from stack, offset 4
        addi   sp,sp,8    #free two words on local stack
        p_ret             #ra!=0 => end and send ra to t0 hart
```

Figure 7: The PISC RISCV code for the end of the *LBP_parallel_start* function.

The *fork_on_current* function called in *LBP_parallel_start* (figure 2) is a fork protocol composed of the machine instructions presented on figure 8. The code allocates a new hart on the current core (*p_fc* X_PAR machine instruction; the allocated hart identity is saved to the destination register *t6*), sends registers to the allocated hart (three *p_swcv* X_PAR instructions; *a1* holds a pointer on the *data* argument), starts the new hart (*p_jalr* X_PAR instruction; *a0* holds the *thread* address) which receives the transmitted registers (three matching *p_lwcv* X_PAR instructions; the join address is restored from stack to *ra*, the join core/hart to *t0* and the data pointer to *a1*).

```
        p_fc     t6       #t6 = allocated hart number (4*core+hart)
        p_swcv   ra,t6,0  #save ra on t6 hart stack, off. 0
        p_swcv   t0,t6,4  #save t0 on t6 hart stack, off. 4
        p_swcv   a1,t6,8  #save a1 on t6 hart stack, off. 8 (data)
        p_merge  t0,t0,t6 #merge reg. t0 and t6 into t0
        p_syncm           #block fetch until mem. accesses are done
        p_jalr   ra,t0,a0 #call thread locally, start pc+4 remotely
        p_lwcv   ra,0     #restore ra from stack, off. 0
        p_lwcv   t0,4     #restore t0 from stack, off. 4
        p_lwcv   a1,8     #restore a1 from stack, off. 8 (data)
```

Figure 8: The PISC RISCV forking protocol.

The seven first instructions (in red; down to the *p_jalr*) are run by the forking hart and the three last ones (in blue) are run by the forked hart. The *p_jalr* instruction calls the *thread* function on the local hart and sends the return address[5] to the allocated hart, which starts fetching at the next cycle. After the *p_jalr* instruction has been issued, the function called and the code after return are run in parallel by two different harts.

The *p_syncm* X_PAR instruction synchronizes the send/receive transmission protocol (*p_swcv* and *p_lwcv* pairs). The sending hart is blocked until all memory writes are done (*ra*, *t0* and *a1* registers saved on the allocated hart stack). The allocated hart starts only when its arguments have been copied on its stack by the allocating hart.

[5] *pc+4* points on the *p_lwcv* instruction just following the *p_jalr* one

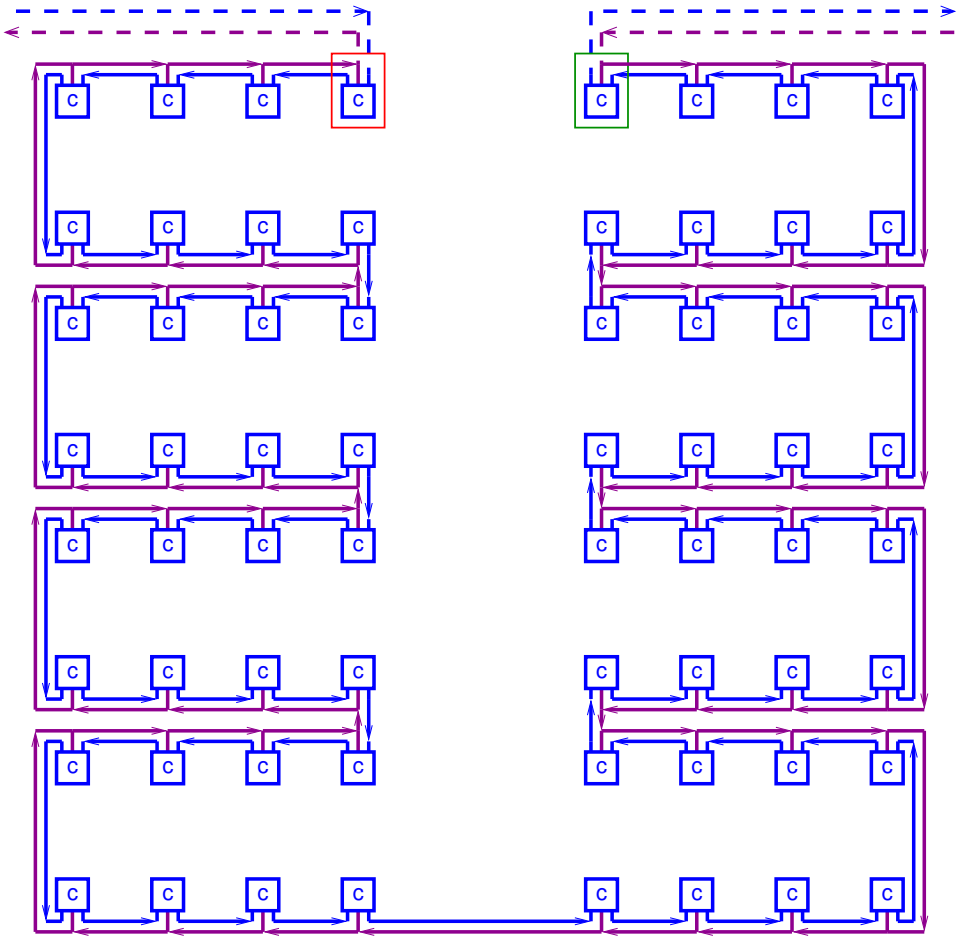

Figure 9: The 64 cores of the LBP processor.

5. The LBP parallelizing processor

5.1. The cores

Figure 9 shows the general structure of the 64-core LBP processor as it is implemented on the FPGA (the global shared memory is presented on figure 13)[6]. Cores are represented by blue squares labeled *c*. Links between cores are represented by magenta and blue arrows. Dashed lines represent optional extensions, either to have a larger manycore or to connect multiple LBP chips. There are 64 ordered cores. The first core in order (core 0) is surrounded by a

[6]The FPGA implementation uses a small FPGA which limits the processor to 8 cores

red rectangle (top of the figure). Its successor is just aside, on the left. The last core (core 63) is surrounded by a green rectangle (at the right of core 0). The line of cores has a serpentine shape. The last core is not connected to the first one. Hence, teams may only expand along successive cores until the last one, no further (a line of cores in a chip can be extended by connecting a second chip; a line of chips can be built to unboundingly extend the line of cores; the first core is the first one of the first chip and the last core is the last one of the last chip as on figure 15).

Each core is directly connected to its successor (blue arrow). Each core is indirectly connected to any predecessor through a unidirectional line (magenta arrows). The direct connections (blue arrows) are used to allocate harts (fork with *p_fc* or *p_fn*), send continuation values (*p_swcv*) and propagate *ending hart* signals (*p_ret*). The backward line (magenta arrows) is used to send join addresses (*p_ret*), function results and reduction values (send a result with *p_swre*).

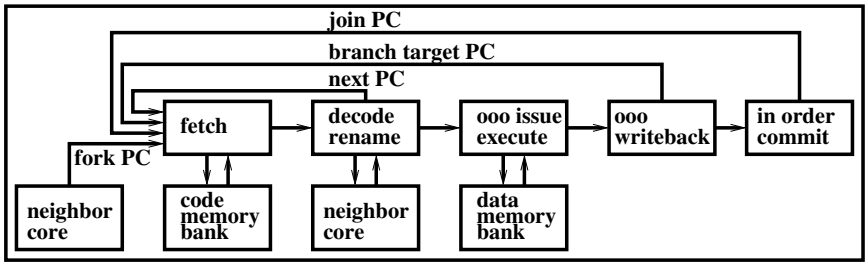

Figure 10: The core pipeline.

5.2. The pipeline

Figure 10 shows the LBP core pipeline. It has a classic five stages out-of-order organization. Each stage selects one active hart at every cycle as shown on figures 11 and 12. In one cycle, a core fetches one instruction for the selected fetching hart, renames one instruction of the selected renaming hart, issues one instruction of the selected issuing hart, writes back one result to the register file of the selected writing hart and commits one instruction for the selected committing hart. The five selections are independent from each other.

A hart may be selected to fetch if its *pc* is full (a thread is running), if it has not been suspended and if the fetched instruction may be saved in the decode stage instruction buffer (labeled *ib* on figure 11)(the buffer remains full until the hart is selected for decode/rename).

In particular, a hart is suspended after fetch until the next *pc* is known, at best after the decoding which produces $nextPC$ as shown on figure 10 ($pc + 1$ or the target of an unconditional direct branch). During the suspension, other active harts on the core are selected. LBP hides branch latency through multithreading instead of eliminating it through prediction (because every hart is suspended after fetch, at least two full harts are necessary to fill the pipeline).

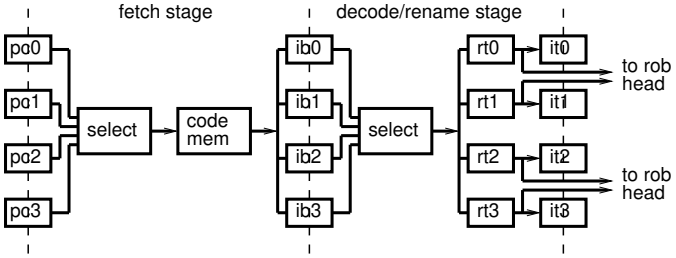

Figure 11: Fetch and decode/rename stages.

A hart may be selected for renaming if its instruction buffer is full with a fetched instruction, if there are available resources to do the renaming (renaming table labeled *rt* on figure 11, decode/rename stage) and there is a free entry in the hart reorder buffer (labeled *rob* in the commit stage on figure 12). Once renamed an instruction is saved in the hart instruction table (labeled *it* in the issue stage) and in the hart reorder buffer.

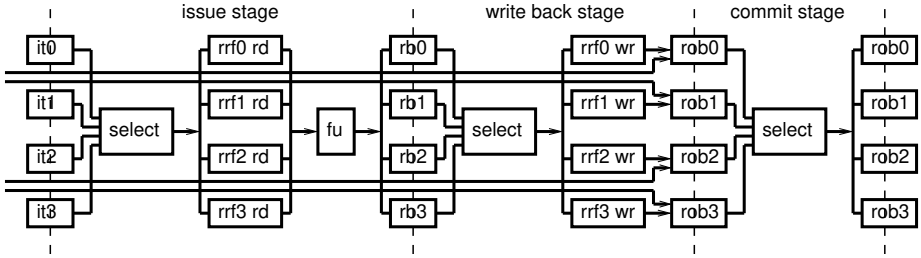

Figure 12: Issue, write back and commit stages.

A hart may be selected for issue if it has at least one ready instruction in its instruction table (renamed instructions wait in the hart instruction table until the sources are ready; there is one table per hart) and if the result buffer of the hart in the write back stage is empty (labeled *rb* in the write back stage; hence, a multicycle computation blocks the hart for issue until the result has been written back, releasing the result buffer). Once issued, the renamed instruction reads its renamed sources in the renaming register file (labeled rrf), crosses a single or multiple cycle functional unit (labeled fu) and saves the result in the hart result buffer.

A hart may be selected for write back if its result buffer is full and the commit buffer of the hart is empty (labeled *cb*). The selected result is written to the renaming register file of the hart. The written back instruction is notified as terminated in the hart reorder buffer.

A hart may be selected for commit if its reorder buffer tail entry is terminated. If the instruction is a hart ending *p_ret*, the *ending hart* signal must have

been received from the preceding hart.

The pipeline has the minimum hardware to make the out-of-order engine work (to keep each core as simple as possible, which allows either to maximize the number of cores on the die for a high performance manycore or to build a very low-cost parallelizing microcontroller with one core and 4 harts).

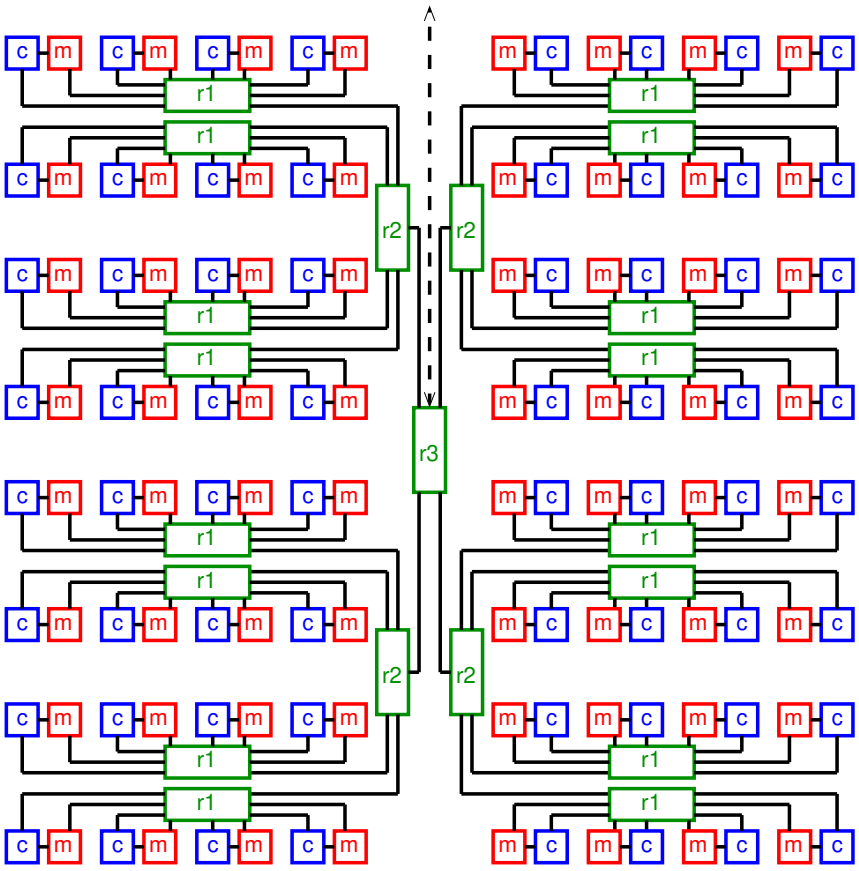

Figure 13: The memory of the LBP processor.

A consequence is that a hart may have to wait in multiple situations: to fetch because the *pc* is unknown (after a branch; this is frequent), to decode because there is no renaming register available to rename the destination (this is rare) or to issue because the result buffer is occupied (waiting for a computation in progress or waiting to be selected for write back; this is frequent in programs with a lot of memory accesses and/or a lot of multiplications/divisions). However, our experiments have shown that when the 4 harts are active, the core pipeline achieves a rate close to the peak of one instruction per cycle.

There is a situation in which an instruction can wait for issue for millions of

cycles. When a *p_lwre* instruction in a consuming hart *ch* has to load a result which is to be sent by an asynchronous producing hart *ph* (e.g. *ph* inputs a data to be sent to *ch* by a matching *p_swre* instruction). Hart *ch* *p_lwre* instruction waits in its instruction table until hart *ph* issues its *p_swre* instruction which sends the input data on the intercore backward link. Once the data has reached the destination hart *ch*, it fills the *ch* result buffer, allowing the consuming *p_lwre* instruction to issue.

This synchronization of asynchronous harts through producing and consuming instructions having a read-after-write dependency (on the implicit result buffer) is the main reason why LBP cores are moved by an out-of-order engine. Traditionally, out-of-order issue serves to capture the local ILP within a hart. In LBP, the out-of-order mechanic mainly serves to synchronize producers and consumers from different harts.

Even though harts are interleaved in the pipeline on a cycle by cycle basis, this interleaving keeps deterministic as it only involves harts belonging to the same application.

5.3. The memory

Figure 13 shows the LBP memory organization. Each core is associated to a set of memory banks (red square labeled m). There are three banks per core. One bank holds the code, another holds local data (a stack) and the last one is used as a shared global memory.

The shared banks have two access ports. One port is used for a local access and the other port is used for distant accesses through a hierarchy of routers which interconnect the banks[7].

Each core has a bidirectional access to a level one router (green rectangle labeled $r1$ and shared by four cores). Each $r1$ router is connected to a level two $r2$ router (shared by four $r1$). Eventually, $r2$ routers are connected through a level three router $r3$. The pattern is extensible (for example to extend the shared memory out of the LBP chip or for future extensions of an LBP manycore).

Each $r1$ router is able to handle one access per link per cycle (i.e. 8 transactions with the connected cores plus 4 transactions with the connected memory banks; the router has the necessary internal buffers to pipeline the transactions from core to memory and back to core). Every cycle, each $r2$ router is able to receive 4 incoming requests from the 4 connected $r1$, send 4 outgoing request results to the 4 $r1$, propagate one request to $r3$ and receive one request result from $r3$. Eventually, every cycle the $r3$ router is able to propagate 4 requests and 4 results to/from the 4 connected $r2$.

Figure 14 shows the superposition of figures 9 and 13 which makes the LBP manycore design (a LBP processor can be built from any subset of the 64 core design; for example, a 16-core LBP has no $r3$ router, a 4-core has no $r2$ nor $r3$ and a single core LBP has no router at all). Figure 15 shows four interconnected

[7]The routers are not yet implemented on the FPGA but simulated for the reported experiment

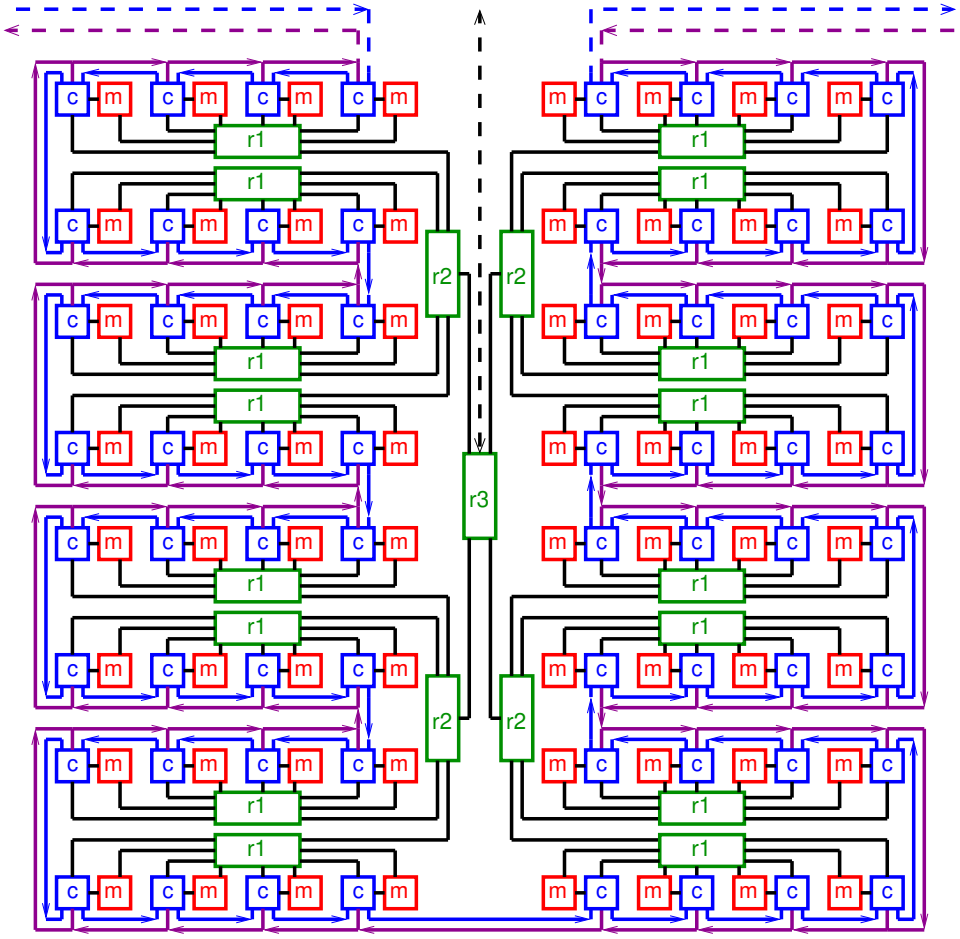

Figure 14: The LBP processor.

LBP chips with their pinout and a shared last level memory (e.g. DRAM controller and DDR4 DIMM).

6. LBP is non interruptible

LBP cores are not interruptible. Each hart *pc* has no external input. Figure 16 shows a Deterministic OpenMP application to be run on a microcontroller connected to 4 sensors and an actuator. A fusion of the 4 values obtained from the sensors is sent to the actuator. The sensors are assumed to respond in any non-deterministic order. However, the ordering of the input values in the fusion computation static code (e.g. $(s[0] + s[1] + s[2] + s[3])/4)$) fixes the ordering of the evaluation (left to right in C) which ensures a deterministic result.

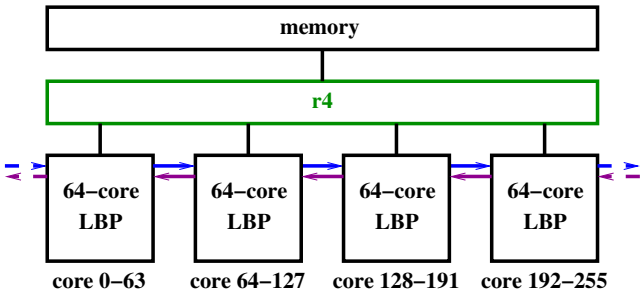

Figure 15: Four interconnected LBP chips.

Deterministic OpenMP source

```
#include <det_omp.h>
int s[4], f;
void main(){
  while(1){
#pragma omp parallel sections
    {
#pragma omp section/*section_0*/
      set_input(0); get_sensor0(&s[0]);
#pragma omp section/*section_1*/
      set_input(1); get_sensor1(&s[1]);
#pragma omp section/*section_2*/
      set_input(2); get_sensor2(&s[2]);
#pragma omp section/*section_3*/
      set_input(3); get_sensor3(&s[3]);
    }/*end parallel sections*/
    f=fusion(s);
    set_output(); set_actuator(f);
  }/*end while*/
}/*end main*/
```

C source after translation

```
void fork_on_current(
  void(*f)(void*), void *data){
  /*p_fc(f, data);*/
}
int s[4], f;
void main(){                                  h0
  while(1){
    set_input(0);                             h0
    fork_on_current(get_sensor0,&s[0]);
    set_input(1);                             h1
    fork_on_current(get_sensor1,&s[1]);
    set_input(2);                             h2
    fork_on_current(get_sensor2,&s[2]);
    set_input(3);                             h3
    get_sensor3(&s[3]);
    f=fusion(s);                              h0
    set_output(); set_actuator(f);
  }
}
```

Figure 16: An example of a non-deterministic application run deterministically on LBP.

The run is distributed on four harts with the *fork_on_current* function of the *det_omp.h* runtime (labeled $h0$ to $h3$ on the right part of the figure). Hence, the four possible inputs are simultaneously monitored. The monitoring is an active wait of each input machine instruction on the input controller. The team of four harts joins back to $h0$. The last team member sends the fused value f to $h0$ which is output to the actuator. Hart $h0$ loops and starts a new team of harts to input.

Figure 17 shows an example of an I/O system connected to a 4-core LBP.

Two of the 16 harts are used as I/O controllers. The input controller is hart 3 of core 3 (rightmost core, lower hart, labeled *in cont*). The output controller is hart 0 of core 0 (leftmost core, upper hart, labeled *out cont*).

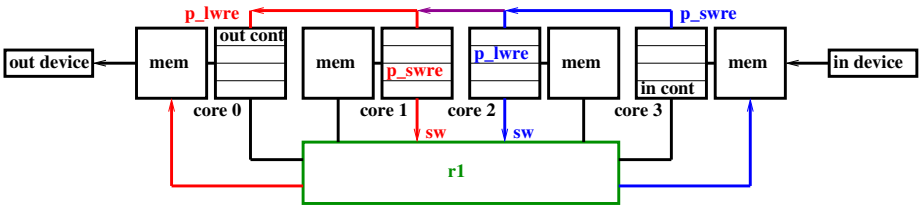

Figure 17: The I/O system connected to a 4-core/16-hart LBP processor.

The input controller is connected to the input devices (e.g. the four sensors). It polls the input ports mapped on the shared memory bank. Once a value is present, the input controller sends it to the requesting core, using the intercore backward link (X_PAR instruction *p_swre*; the input value travels from the input controller to the hart requesting an input).

To request an input (as do the *set_input* function calls in the figure 16 example), a hart (hart 1 in core 2 on figure 17) writes a request in the input controller memory through the $r1$ router (RISCV standard *sw* store word instruction; the written word identifies the requesting hart). To input (as the *get_sensor* function calls), it runs a *p_lwre* instruction which matches the controller *p_swre* one. The *p_swre* and *p_lwre* pair are asynchronously fetched and renamed but orderly issued. The *p_swre* instruction is issued after the input data has been received and loaded in the register to be sent. The *p_lwre* instruction is issued when the hart result buffer is full, i.e. when the travelling input data has been written into the destination hart. This synchronization is done by the cores out-of-order engines from the register or the result buffer read-after-write dependencies.

The output controller works the same way. To request an output (as the *set_output* function call), a hart (hart 2 in core 1 on figure 17) writes a request in the output controller memory. To output (as the *set_actuator* function call), the hart sends the data to be output with a *p_swre* instruction. The data is saved in the output controller result buffer. A matching *p_lwre* instruction is issued by the output controller to receive the data, which is written to the output device by a store word instruction.

This is very different from the interrupt based classic I/O implementation in which the I/O response time is very hard to bound (interrupt handler + thread wake up + thread running, where the interruption may be interrupted by another I/O). On LBP, once the data is available to the input controller, within a few cycles it is received by the requesting hart. The response time is very short (a few cycles) and easy to bound.

Among the input devices can be timers. On classical hardware/OS, external timers are not highly reliable because of the imprecise lag between the very precise periodic signal and its impact on the piece of software which it clocks.

Internal timers are prefered if times close to the internal clock are to be measured (e.g. the exact latency in cycles of a run composed of a few tens or a few hundreds of instructions). The LBP I/O system, based on the producer to consumer dependency and their automatic synchronization by the out-of-order engine, reduces to a few cycles the internal reaction delay after an input, making external timing systems adapted to be used as very precise external clocks for real time softwares.

The LBP I/O pattern is suited to distributed I/Os. A team of harts can collaborate to input pieces of a structured data from an input controller or to output a structured result chunk by chunk through an output controller. The intercore backward link connecting the cores acts as a stream either filling the team of harts or draining them.

The LBP I/O pattern is also suited to build a Direct Memory Access (DMA) unit, using one hart as an input controller to fill all the shared memory banks with a structured data distributed to the computing harts. The synchronization of the DMA with the using harts is done through *p_swre* and *p_lwre* pairs of X_PAR instructions rather than though interrupts.

7. A matrix multiplication program example experiment

Figure 18 shows a Deterministic OpenMP program to multiply integer matrices. Except for the *det_omp.h* reference in red, the remaining of the text is standard OpenMP code and can be compiled with *gcc -fopenmp*.

This program (the *base*) has been run on three sizes of a *vivado_HLS* simulation (Xilinx High Level Synthesis tool, version 2019.2) of the LBP processor (4, 16 and 64 cores). Four other versions have also been implemented and run on the simulated LBP: *copy*, *distributed*, *d+c* and *tiled*. The different codes are shown at URL [19].

The aim of the experience is to show that the LBP design is able to fill the harts pipelines with instructions all along the run, thanks to the high level of distant ILP exhibited by the Deterministic OpenMP parallelization, despite the multiple latencies each hart has to wait for. A second goal is to verify that the shared memory interconnection is dimensioned proportionally to the number of harts. As the number of cores is increased in LBP, the distant memory access requests are more frequent and have a longer latency. The experience should check that the hardware is able to sustain a high proportion of distant accesses without stalling the harts, i.e. keeping the IPC as close as possible to its peak.

Each run multiplies a matrix X with h lines and $h/2$ columns and a matrix Y with $h/2$ line and h columns, where h is the number of harts (i.e. 16, resp. 64 and resp. 256 for a 4, resp. 16 and resp. 64 core LBP processor).

The *copy* code copies a line of matrix X in the local stack to avoid its multiple accesses in the shared memory. The *distributed* code distributes and interleaves the three matrices evenly on the memory banks (four lines of X, two lines of Y and four lines of Z in each bank), to avoid the concentration of memory accesses on the same banks (which happens if matrix Y is not distributed). The *c+d*

```
#include <stdio.h>
#include <det_omp.h>
#define LINE_X        16
#define COLUMN_X      8
#define LINE_Y        COLUMN_X
#define COLUMN_Y      16
#define LINE_Z        LINE_X
#define COLUMN_Z      COLUMN_Y
#define NUM_HART      16
int X[LINE_X*COLUMN_X]={[0...LINE_X*COLUMN_X-1]=1};
int Y[LINE_Y*COLUMN_Y]={[0...LINE_Y*COLUMN_Y-1]=1};
int Z[LINE_Z*COLUMN_Z];
void thread(int t){
 int i, j, k, l, tmp;
 for (l=0, i=t*LINE_Z/NUM_HART; l<LINE_Z/NUM_HART; l++, i++)
  for (j=0; j<COLUMN_Z; j++)
   tmp=0;
   for (k=0; k<COLUMN_X; k++)
    tmp+=*(X+(i*COLUMN_X+k)) * *(Y+(k*COLUMN_Y+j));
   *(Z+(i*COLUMN_Z+j))=tmp;
  }
}
void main(){
 int t;
 omp_set_num_threads(NUM_HART);
#pragma omp parallel for
 for (t=0; t<NUM_HART; t++) thread(t);
}
```

Figure 18: A Deterministic OpenMP matrix multiplication program.

version copies and distributes. The *tiled* version is the classic five nested loops tiled matrix multiplication algorithm. Each tile has $h/2$ elements for matrices X and Y ($\sqrt{h} * \sqrt{h}/2$) and h for the result matrix Z ($\sqrt{h} * \sqrt{h}$).

Figures 19, 20 and 21 show nine histograms (number of cycles, IPC and number of retired instructions) for the five codes on the three sizes of LBP. These values are reproducible thanks to cycle determinism. The three bottom histograms also include the best measures done on a Xeon Phi2 for the tiled version (MCDRAM configured in flat mode and all-to-all cluster mode; OMP_NUM_THREADS = 256, OMP_PLACES = threads, OMP_PROC_BIND = close). The measures are the minimum ones after 1000 runs. They were obtained with a PAPI instrumentation of the original tiled version.

What matters is the number of cycles, i.e. the duration of the run. The IPC is an indication whether the parallelization is effective. However, a high IPC does not mean that useful work is done. The number of retired instructions is important to see the overcost of parallelization.

On a 4-core LBP (figure 19), even though the tiled version has the highest IPC (3.67 for a peak at 4), the base version is better as it is twice faster. The innermost loop has seven instructions (two loads, one multiplication, one

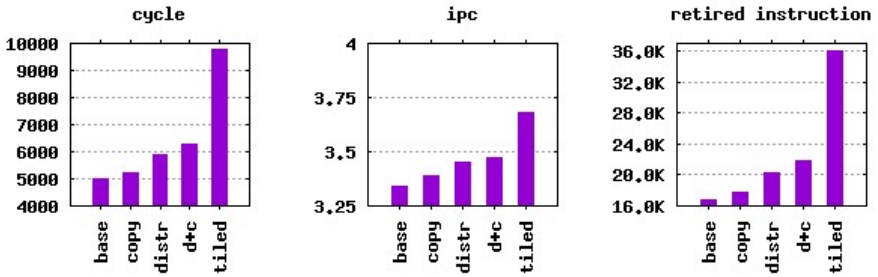

Figure 19: Number of cycles, IPC and retired instructions for the matrix multiplication five versions on a 4-core LBP (16 harts).

addition, two address incrementations and a conditional branch), which are repeated $h^3/2$ times, i.e. 14336 instructions when $h = 16$. The base version has 16722 retired instructions, which leaves 2386 instructions for the two outer loops, the parallelization and its control (creation of 16 threads and their join).

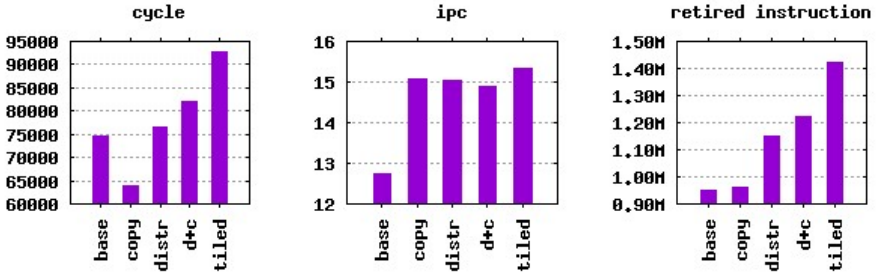

Figure 20: Number of cycles, IPC and retired instructions for the matrix multiplication five versions on a 16-core LBP (64 harts).

On a 16-core LBP (figure 20), the fastest is the copy version. The base version achieves a poor 12.7 IPC when the copy version IPC is over 15 (for a peak of 16), saving more than 10000 cycles (16% faster). The overhead is moderate (14500 instructions, i.e. 1.5%).

On the 64-core LBP (figure 21), the tiled version is the best because it saves many long distance communications and because it distributes the remaining ones more evenly over time and space. It is twice faster than the distributed version and four times faster than the base version (1.18M cycles vs 2.08M and 4.14M). The IPC is 61.7 (for a peak of 64), showing that the LBP interconnect is strong enough to handle the high demand. The tiling overhead is not negligible (73M instructions versus 59M for the base version, i.e. +23%).

The 64-core LBP is not as fast as the Xeon Phi2 (1.18M cycles vs 391K, 3 times more). Firstly, there is no vector unit in LBP, which explains that the

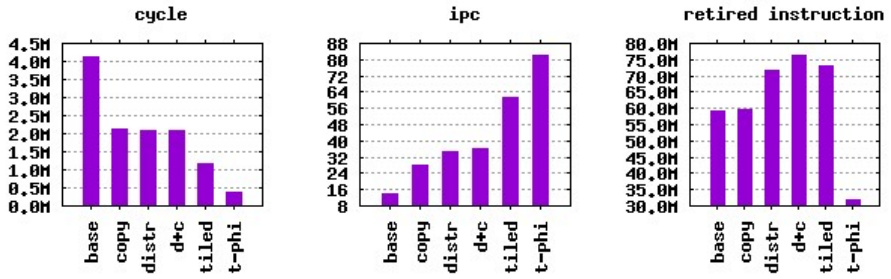

Figure 21: Number of cycles, IPC and retired instructions for the matrix multiplication five versions on a 64-core LBP (256 harts).

Xeon runs 32M instructions and LBP runs 73M, i.e. 2.28 times more. Secondly, LBP peak performance is 1 IPC per core when the Xeon peak is 6 (2 int, 2 mem and 2 vector ops per cycle). Hence, LBP reaches 0.96 IPC per core (96% of 1 IPC peak) and the Xeon reaches 1.28 IPC per core (81.86/64 ; 21% of 6 IPC peak). LBP is aiming embedded applications and should keep low-power and energy efficient, which the Xeon Phi2 is not.

8. Conclusion and Perspectives

Safety critical real time applications can benefit from parallel manycore processors, if a high level of determinism is ensured to guarantee repeatable timings, as on the LBP processor. Moreover, the reported experiment shows that a low-power manycore processor can be built for the embedded high performance computations. The design of the LBP processor is suited to either offer parallelism to microcontrollers or to safely accelerate computations through their parallelization and capture the distant ILP by hundreds of distributed harts.

Deterministic OpenMP is standard OpenMP with a new runtime. For the programmer, the difference resides in the new *det_omp.h* header file and the hardware placement of code and data according to the program structure. The main difference between OpenMP classic runtime and Deterministic OpenMP new one comes from the ordering of harts in a parallel team. This ordering is optional in standard OpenMP but mandatory in Deterministic OpenMP because the hardware synchronization which ensures safety relies on the referential sequential order. As an example, a producing hart has to precede a consuming one in the referential sequential order to exhibit the read-after-write dependency linking the producer to the consumer. In Deterministic OpenMP, a later hart cannot send anything to a prior one (a data cannot go back in time).

In a future work, we will extend the actual 8-core FPGA implementation of LBP to fit a 16 core and two levels of routers on the Xilinx ZCU106 development board. We will also complete the Deterministic OpenMP translator to

automatize the translation of standard OpenMP codes into our LBP specific machine code.

It is also interesting to study an adaptation of LBP to the particular context of High Performance Computing (HPC). The forking mechanism which creates teams of harts on a line topology could be slightly modified to allow the start of a new team on a new LBP chip, taking advantage of the incompletely used fields in the X_PAR instruction set. The links connecting the cores could be duplicated or super links connecting chips could be added to allow more direct communications between distant harts. A deterministic version of MPI [20] could even be proposed, built around ordered communicators where a sender always precedes its receiver(s) (i.e. the sender rank is lower than all its receivers ranks).

References

[1] Lee, E., "What Is Real Time Computing? A Personal View", IEEE Design and Test PP(99):1-1 (October 2017).

[2] Kirk, D., Wu, W., "Programming Massively Parallel Processors, a hands-on approach", 3rd edition, Morgan Kaufmann, Chapter 4 (Memory and data locality), 2017.

[3] Sodani, A. et al., "Knights Landing: Second-Generation Intel Xeon Phi Product", in IEEE Micro, vol. 36, no. 2, pp. 34-46, Mar.-Apr. 2016.

[4] de Dinechin, B., "Consolidating High-Integrity, High-Performance, and Cyber-Security Functions on a Manycore", 56th ACM/IEEE Design Automation Conference (DAC), pp 1-4, 2019.

[5] Waterman, A., Asanovic, K., "The RISC-V Instruction Set Manual", https://riscv.org/specifications, 2017.

[6] Kotaba, O., Nowotsch, J., Paulitsch, M., Petters, M. Theiling, H., "Multicore In Real-Time Systems Temporal Isolation Challenges Due To Shared Resources", DATE 2013.

[7] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System", CACM, Vol. 21, no. 7, pp. 558-565, July 1978.

[8] Zimmer, M., Broman, D., Shaver, C., Lee, E., "FlexPRET: A processor platform for mixed-criticality systems", IEEE 19th RTAS (2014).

[9] Ungerer, T., et al., "Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability", IEEE Micro, vol. 30, no. 5 (2010).

[10] Serrano, M., Royuela, S., Quinones, E., "Towards an OpenMP Specification for Critical Real-Time Systems", 14th IWOMP (2018).

[11] Lipp, M. et al., "Meltdown: Reading Kernel Memory from User Space", 27th USENIX Security Symposium, 2018.

[12] Kocher, P. et al., "Spectre attacks: Exploiting speculative execution", 40th IEEE Symposium on Security and Privacy, 2019.

[13] Islam, S. et al., "SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks", ArXiv e-prints, http://arxiv.org/abs/1903.00446, march 2019.

[14] Austin, T. M., Sohi, G. S., "Dynamic Dependency Analysis of Ordinary Programs.", ISCA'92, pp 342-351, 1992.

[15] Lam, M. S., Wilson, R. P., "Limits of Control Flow on Parallelism.", ISCA'92, pp 46-57, 1992.

[16] Goossens, B., Parello, D., "Limits of Instruction-Level Parallelism Capture.", Elsevier Procedia Computer Science, Volume 18, pp 1664-1673, 2013.

[17] OpenMP, "OpenMP Application Programming Interface", version 5.0, https://www.openmp.org/specifications/, november 2018.

[18] GOMP on-line documentation, https://gcc.gnu.org/onlinedocs/libgomp

[19] https://gite.lirmm.fr/lbp-group/lbp-projects/-/wikis/Little-Big-Processor-project

[20] Lyndon, C., Glendinning, I., Hempel, R., Decker, K. M., Rehmann, R. M., "The MPI Message Passing Interface Standard.", Programming Environments for Massively Parallel Distributed Systems, Birkhuser, Basel, 1994.