Toward a Core Design to Distribute an Execution on a Many-Core Processor

Bernard Goossens, David Parello, Katarzyna Porada, Djallal Rahmoune

▶ **To cite this version:**

Bernard Goossens, David Parello, Katarzyna Porada, Djallal Rahmoune. Toward a Core Design to Distribute an Execution on a Many-Core Processor. PaCT: Parallel Computing Technologies, Aug 2015, Petrozavodsk, Russia. pp.390-404, 10.1007/978-3-319-21909-7_38 . hal-01152664

HAL Id: hal-01152664
https://hal.science/hal-01152664

Submitted on 18 May 2015

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.

Distributed under a Creative Commons Attribution 4.0 International License

Toward a Core Design to Distribute an Execution on a Many-Core Processor

Bernard Goossens, David Parello, Katarzyna Porada, and Djallal Rahmoune

DALI, UPVD 66860 Perpignan Cedex 9 France,
LIRMM, CNRS: UMR 5506 - UM2 34095 Montpellier Cedex 5 France,
{bernard.goossens,david.parello,katarzyna.porada,djallal.rahmoune}@univ-perp.fr

Abstract. This paper presents a parallel execution model and a many-core processor design to run C programs in parallel. The model automatically builds parallel sections of machine instructions from the run trace. It parallelizes instructions fetches, renamings, executions and retirements. Predictor based fetch is replaced by a fetch-decode-and-partly-execute stage able to compute in-order most of the control instructions. Tomasulo's register renaming is extended to memory with a technique to match consumer/producer pairs. The Reorder Buffer is adapted to allow parallel retirement. The model is presented on a *sum* reduction example which is also used to give a short analytical evaluation of the model performance potential.

Keywords: Microarchitecture, Parallelism, Manycore, Automatic parallelization

1 Introduction

Every parallel machine programmer dreams he can run his unchanged C programs on a parallel computer.

Figure 1 shows a C version and a *pthread* version of a sum reduction function.

The difference does not lie in the code text (based on the same algorithm) but in its execution. The C code is run sequentially using a stack and the *pthread* code is run in parallel with the help of the *pthread* system primitives.

This paper aims to show that if we change the execution model, the C code run can have the same behaviour as the *pthread* run, i.e. parallel execution. But of course, the C code is much simpler. Section 2 explains how to run a C program in parallel. Section 3 evaluates the Instruction Level Parallelism (ILP) in benchmarks based on parallel algorithms and lists the main published works on ILP. Section 4 describes the parallel execution model and its core microarchitecture. Section 5 gives an analytical evaluation of the performance potential of the proposed model and core design. It also mentions the on-going developments of simulators and concludes.

```
unsigned long
sum(unsigned long t[], unsigned long n){
 if (n==1) return t[0];
 else if (n==2) return t[0]+t[1];
 else return sum(t,n/2) + sum(&t[n/2],n-n/2);
}
```

(a) C implementation

```
typedef struct{unsigned long *p; unsigned long i;} ST;
void *sum(void *st){
 ST str1,str2; unsigned long s,s1,s2;
 pthread_t tid1, tid2;
 if (((ST *)st)->i>2){
  str1.p=((ST *)st)->p; str1.i=((ST *)st)->i/2;
  pthread_create(&tid1,NULL,sum,(void *)&str1);
  str2.p=((ST *)st)->p + ((ST *)st)->i/2;
  str2.i=((ST *)st)->i - ((ST *)st)->i/2;
  pthread_create(&tid2,NULL,sum,(void *)&str2);
  pthread_join(tid1,(void *)&s1);
  pthread_join(tid2,(void *)&s2);
 }
 else if(((ST *)st)->i==1){s1=((ST *)st)->p[0];s2=0;}
 else {s1=((ST *)st)->p[0];s2=((ST *)st)->p[1];}
 s=s1+s2; pthread_exit((void *)s);
}
```

(b) pthread implementation

Fig. 1: A vector sum reduction: C and pthread implementations

2 Running a C program in parallel.

Figure 2 shows the *sum* function translation into x86 (*gas* syntax ; rightmost operand is the destination). The code is run sequentially because the hardware is unable to *fork* at lines 12 and 20. The control flow travels along the binary tree of calls depth first, leading to a 59 instructions run trace shown on figure 3 (figure 4 left part shows the call tree for *sum(t,5)*).

```
1  sum:                          //sum(t,n)
2        cmpq    $2, %rsi        //n>2
3        ja      .L2             //if (n>2) goto .L2
4        movq    (%rdi), %rax    //rax=t[0]
5        jne     .L1             //if (n!=2) goto .L1
6        addq    8(%rdi), %rax   //rax+=t[1]
7  .L1:  ret                     //return(rax)
8  .L2:  pushq   %rbx            //save rbx
9        pushq   %rdi            //save t
10       pushq   %rsi            //save n
11       shrq    %rsi            //rsi=n/2
12       call    sum             //sum(t,n/2)
13       popq    %rbx            //rbx=n
```

```
14       pushq   %rbx            //save n
15       subq    $8, %rsp        //allocate temp
16       movq    %rax, 0(%rsp)   //temp=sum(t,n/2)
17       leaq    (%rdi,%rsi,8), %rdi //rdi=&t[n/2]
18       subq    %rsi, %rbx      //rbx=n-n/2
19       movq    %rbx, %rsi      //rsi=n-n/2
20       call    sum             //sum(&t[n/2],n-n/2)
21       addq    0(%rsp), %rax   //rax+=temp
22       addq    $8, %rsp        //free temp
23       popq    %rsi            //restore rsi (n)
24       popq    %rdi            //restore rdi (t)
25       popq    %rbx            //restore rbx
26       ret                     //return rax
```

Fig. 2: The *sum* function in X86

Figure 5 shows a modified x86 code for the *sum* function. The hardware is assumed to be able to fork, i.e. start a second instruction flow which occurs on lines 10 and 16 (*fork* instructions replace *call* instructions of the original x86 code). Unlike a *call* instruction, a *fork* instruction does not save a return address.

Non volatile registers (i.e. *rbx*, *rdi* and *rsi* in this example) are copied to the forked path, replacing the stack save/restore pair. Hence *push* and *pop* are removed. The stack pointer itself (*rsp*) is copied to the forked path[1].

[1] the stack in each section keeps its local variables, e.g. *temp* on figure 5.

```
1  sum:   cmpq $2, %rsi  //sum(t,5)
2         ja    .L2
3  .L2:   pushq  %rbx
4         pushq  %rdi
5         pushq  %rsi
6         shrq   %rsi
7         call   sum
8            sum:   cmpq $2, %rsi  //sum(t,2)
9                   ja    .L2
10                  movq   (%rdi), %rax
11                  jne    .L1
12                  addq   8(%rdi), %rax
13           .L1:   ret
14        popq   %rbx
15        pushq  %rbx
16        subq   $8, %rsp
17        movq   %rax, 0(%rsp)
18        leaq   (%rdi,%rsi,8), %rdi
19        subq   %rsi, %rbx
20        movq   %rbx, %rsi
21        call   sum
22           sum:   cmpq $2, %rsi  //sum(&t[2],3)
23                  ja    .L2
24           .L2:   pushq  %rbx
25                  pushq  %rdi
26                  pushq  %rsi
27                  shrq   %rsi
28                  call   sum
29                     sum:   cmpq $2, %rsi  //sum(&t[2],1)
30                     ja    .L2
31                     movq   (%rdi), %rax
32                     jne    .L1
33              .L1:   ret
34           popq   %rbx
35           pushq  %rbx
36           subq   $8, %rsp
37           movq   %rax, 0(%rsp)
38           leaq   (%rdi,%rsi,8), %rdi
39           subq   %rsi, %rbx
40           movq   %rbx, %rsi
41           call   sum
42              sum:   cmpq $2, %rsi  //sum(&t[3],2)
43                     ja    .L2
44                     movq   (%rdi), %rax
45                     jne    .L1
46                     addq   8(%rdi), %rax
47              .L1:   ret
48           addq   0(%rsp), %rax
49           addq   $8, %rsp
50           popq   %rsi
51           popq   %rdi
52           popq   %rbx
53           ret
54 addq   0(%rsp), %rax
55 addq   $8, %rsp
56 popq   %rsi
57 popq   %rdi
58 popq   %rbx
59 ret
```

Fig. 3: The instruction trace for the run of *sum(t,5)*.

The *endfork* instruction ends a flow. Unlike a return, the *endfork* does not give control back to a return address.

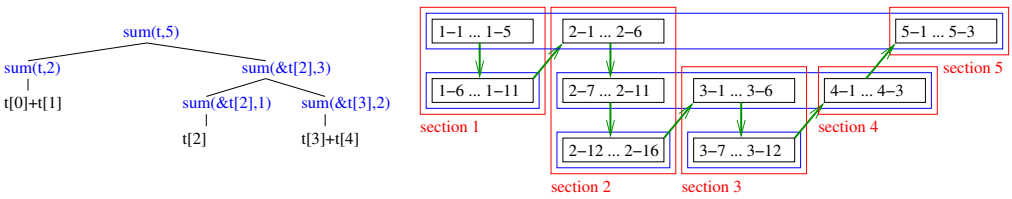


Fig. 4: The call tree (left) for the run of *sum(t,5)* and its sections (right).

Figure 6 shows how the run is parallelized. The run starts on core 1 which fetches and executes instructions 1-1 to 1-5. The *fork* instruction starts a new flow on core 2. The new flow matches the resume path after fork, i.e instruction *subq* on line 11 (figure 5). Core 1 continues its own flow (callee path back to line 1, instruction *cmpq*). Both flows are run in parallel, leading to the fetch and execution of instructions 1-6 to 1-11 (core 1 flow) and 2-1 to 2-16 (core 2 flow).

As core 2 receives valid copies of registers *rdi*, *rsi*, *rbx* and *rsp*, instructions 2-1 and 2-3 to 2-6 can be executed. Only instruction 2-2 must wait until register *rax* is set by core 1 flow. The synchronisation need is easy to detect, thanks to register renaming. Instruction 2-2 consumes a source *rax* produced by the

closest instruction writing to *rax* on the sequential path. As soon as instruction 1-10 writes to *rax*, the written value is forwarded to instruction 2-2.

```
1  sum:                          //sum(t,n)
2        cmpq    $2, %rsi        //n>2
3        ja      .L2             //if (n>2) goto .L2
4        movq    (%rdi), %rax    //rax=t[0]
5        jne     .L1             //if (n!=2) goto .L1
6        addq    8(%rdi), %rax   //rax+=t[1]
7  .L1:  endfork                 //return(rax)
8  .L2:  movq    %rsi, %rbx      //rbx=n
9        shrq    %rsi            //rsi=n/2
10       fork    sum             //sum(t,n/2)
```

```
11       subq    $8, %rsp                //allocate temp
12       movq    %rax, 0(%rsp)           //temp=sum(t,n/2)
13       leaq    (%rdi,%rsi,8), %rdi     //rdi=&t[n/2]
14       subq    %rsi, %rbx              //rbx=n-n/2
15       movq    %rbx, %rsi              //rsi=n-n/2
16       fork    sum                     //sum(&t[n/2],n-n/2)
17       addq    0(%rsp), %rax           //rax+=temp
18       addq    $8, %rsp                //free temp
19       endfork                         //return rax
```

Fig. 5: The *sum* function in X86 modified by *fork* instructions.

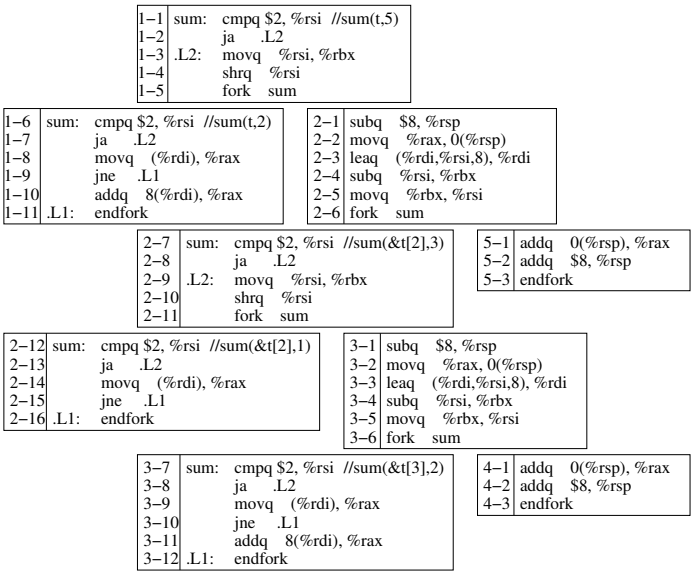


Fig. 6: The instruction trace for the parallel run of *sum(t,5)*.

The full run is divided into 5 flows or *sections* (figure 4, right part). Each section is framed by a red rectangle. The longest section is composed of 16 instructions (section 2, from 2-1 to 2-16). Sections are numbered in execution trace order as indicated by the green arrows. Instructions framed by a blue rectangle belong to the same call level (e.g. instructions 1-1 to 1-5, 2-1 to 2-6

and 5-1 to 5-3 form the same call level). A section is a full recursive descent (e.g. section 1 combines 1-1 to 1-5 for $n = 5$ and 1-6 to 1-11 for $n = 2$).

Out-of-order execution is crucial to parallelize fetch. As instruction 2-2 does not block 2-6, the second call can be run in parallel with the first one.

This example shows that if the hardware is changed, the *sum* function can run in parallel as in the *pthread* implementation. However, any link between threads in the *pthread* or *MPI* models must be explicitely added to the code through OS communication primitives (e.g. MPI_Send and MPI_Recv or socket based communications in *pthread*).

In the parallel model, the sections are totally ordered. New sections are inserted in place in the list of existing sections, possibly in parallel, building the sequential trace of the run. This structure and the renaming process (which assigns a new location to each write of each instruction in the sequential trace) ensure that each read can match the most recent preceding write. In the *pthread* or *MPI* models, this sequential structuration of threads is not available.

For example, if x is local to thread t_x and y is local to thread t_y, to copy y into x thread t_y sends y to thread t_x, which receives it (*MPI* rendezvous). If x is global, t_x and t_y can communicate through x but they must synchronize writes and reads with *pthread* mutex. The OS must be invoked to link the sender and the receiver or to synchronize multiple writers and readers.

In the parallel model, the equivalent of thread t_y is section s_y writing to y and the equivalent of thread t_x is section s_x reading y to copy it into x. Section s_y has instruction i_y writing to y (say i_y is $addq\ \%rbx, \%rax$, with y in rax, i.e. $y = y + z$ for some z in rbx) and section s_x has instruction i_x reading y (say i_x is $movq\ \%rax, \%rcx$, with rcx being x, i.e. $x = y$). Instruction i_y allocates rax_0 to rename rax destination. Instruction i_x renames its source rax. As sections s_y and s_x are ordered and as no instruction updates y between i_y and i_x, the renaming of rax in i_x matches rax_0. Moreover, the hardware synchonizes the reader i_x with the writer i_y until rax_0 is full. Hence, rendezvous or mutex are not necessary and the OS need not be sollicited.

Renaming is the key to synchronization and communication between dependent flows. Renaming should be extended to all hardware locations. For example, instruction 5-1 reads the top of stack word ($0(rsp)$). This memory location is written by instruction 2-2. If instruction 2-2 destination $a = 0(rsp)$ is renamed r, instruction 5-1 renames the same address a with the same name r, exhibiting the dependency with instruction 2-2. Instruction 5-1, which computes the final sum, executes after it has received rax from section 4 (second half of the sum) and a from section 2 (first half of the sum).

3 ILP in programs

Figure 7 displays the ILP of ten benchmarks of the PBBS suite [1]. The PBBS benchmarks implement various classical parallel algorithms (see table 1).

On figure 7, for each of the 10 benchmarks, the 11 leftmost bars (those with numbered keys) match eleven parallel runs of the benchmark with increasing

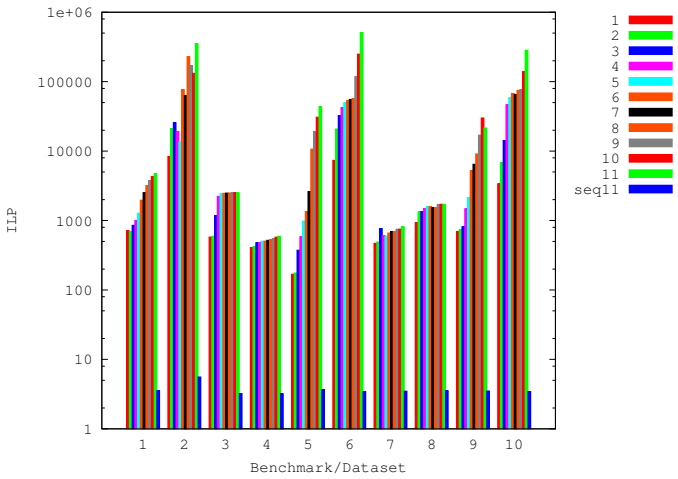


Fig. 7: ILP of ten benchmarks parallel and sequential runs

Benchmark	
01 : breadthFirstSearch/ndBFS	02 : comparisonSort/quickSort
03 : convexHull/quickHull	04 : dictionary/deterministicHash
05 : integerSort/blockRadixSort	06 : maximalIndependentSet/ndMIS
07 : maximalMatching/ndMatching	08 : minSpanningTree/parallelKruskal
09 : nearestNeighbors/octTree2Neighbors	10 : removeDuplicates/deterministicHash

Table 1: Ten benchmarks of the PBBS suite

datasets. The rightmost bar (blue colour, *seq*11 key) matches sequential runs with the same dataset as key 11 parallel runs.

The sequential runs consider all the dependencies excluding the register false ones (Write After Read and Write After Write), assuming an unlimited register renaming capacity, and excluding the control flow ones, assuming perfect branch prediction. The sequential runs ILP measures the ultimate performance of actual out-of-order speculative processors.

The parallel runs assume the trace is available when the run starts (no fetch delay) and in the same time all the destinations (including memory) are renamed. The stack pointer dependencies are not considered. The parallel runs ILP measures the ultimate performance an ideal parallel machine achieves when the run order only depends on the producer to consumer dependencies, excluding the stack pointer. Each instruction on the trace is run at the cycle next to the last source reception. The processor is assumed to run all the ready instructions in the same cycle with a single cycle latency (as in the sequential runs).

All runs are continued until completion. For each benchmark, the 11 parallel runs vary from 1M to 1G instructions (increasing factor 2) and the sequential runs are 1G instructions long.

The figure shows that sequential runs have a very low ILP (ranging from 3.2 to 5.6) and parallel runs have a very high ILP (ranging from 600 to 508K for dataset 11). The difference comes from the dominating distant ILP. Moreover, when a benchmark is data parallel its parallel run ILP increases proportionally to the dataset (e.g. benchmarks 1, 2, 5, 6, 9 and 10).

The sequential run ILP we have measured confirms ILP reported values (such as [6]). Since 50 years many successive research works on ILP were published.

In 1967, Tomasulo [3] presented an algorithm to run floating point instructions out-of-order. He introduced register renaming which is still used in today's speculative cores to parallelize on-the-fly instructions. In 1970, Tjaden and Flynn [4] measured the available parallelism in a 10 instructions window. They ran their test programs at 1.86 instructions per cycle.

In 1984, Nicolau and Fisher [5] measured the available parallelism to feed a VLIW processor. In their experiments, they included a measure of ILP from runs on an ideal machine with infinite resources. They discovered that scientific codes present a high ILP, over 1000.

In 1991, David Wall presented the first study centered on ILP[6]. He measured that the available parallelism a "real" processor finds in 13 benchmarks is 5 on average, ranging from 3 to 45[2]. In an "ideal" processor[3], ILP ranges from 6 to 60 with an average at 25. From this study, we know that there is ILP but it seems impossible to catch more than 5 independent instructions per cycle.

In 1992, Austin and Sohi [7] measured the SPEC89 suite ILP and analyzed its distribution. They showed that ILP is arbitrarily distant from the instruction pointer. They also pointed out the serializing effect of the stack manipulations. The same year, Lam and Wilson [8] studied the impact of control on ILP. Their measures showed that a processor with a perfect branch predictor could dramatically improve its performance. As in Austin and Sohi work, distant ILP was detected. To capture this distant ILP, a processor must be able to speculate on the control flow and use multiple instruction pointers. In 1997, Moshovos and Sohi have proposed memory renaming in [9], using a predictor to find the store renaming a load. In 1999, Postiff, Greene, Tyson and Mudge [10] measured SPEC95 suite ILP. They pointed out that the stack introduces many parasitic dependencies. To capture distant ILP, the application should be multi-threaded.

In 2004, Cristal, Santana and Valero [11] described a kilo-instructions microarchitecture. The authors suggested that to capture more ILP, the processor must have access to instructions far from the fetch point. They gave solutions to allocate later and free sooner the needed resources to optimize their usage and so, take care of more "on-the-fly" instructions with the same resources. In

[2] "good" model with a 2K instructions window size, 64 instructions issued per cycle, 256 renaming registers, a branch predictor based on an infinite number of 2-bits counters and a perfect memory aliasing disambiguation

[3] "perfect" model enhances "good" model: infinite renaming, perfect branch predictor.

2012, Sharafeddine, Jothi and Akkary [12] proposed an architecture to partition a run into parallel threads, forking the leading thread at call. In the sum example this leads to fork on both of the highest levels calls but not on the lower levels, capturing only a small part of the distant ILP. In 2013, Goossens and Parello [13] analyzed distant ILP and showed that ILP could be highly increased when removing stack pointer updates and false memory dependencies.

From these works, we deduce that i) high ILP is available, ii) most of it comes from very distant instructions and iii) sequential fetch and stack are the main obstacles on the ILP capture. Two ideas are suggested to help capture distant ILP : following multiple instruction flows [8] and renaming memory [9].

4 An execution model to run programs in parallel and its core implementation.

In section 3 we assumed the full trace is available at run start and all the destinations are pre-renamed. This is not realistic. However, code fetch and destinations renamings should occur as soon as possible to allow distant ILP capture.

4.1 Parallelizing fetch.

A section is composed of dynamically contiguous instructions. A section starts when a *fork* instruction creates it. It ends when an *endfork* instruction is reached. A control flow instruction (jump, call or branch) does not end a section. The same section continues after the control flow instruction.

When a new section is forked, a message is sent to a hosting core[4]. The message contains the forked Instruction Pointer (IP), the stack pointer and the set of non volatile registers. It also contains the identification of the neighbour sections (e.g. the current creating section). The choosen core queues the message while it fetches another section. When the section creation message is dequeued, it fills the register file local to the fetch pipeline stage. The IP, the stack pointer and the non volatile registers are initialized and other registers are emptied.

For example, when instruction 1-5 forks, a section creation message is sent to core 2 (say), including register rdi value t, register rsi value 2 and register rbx value 5. When instruction 2-6 forks, the stack pointer rsp is transmitted to section 5, pointing on the same stack word as section 2. Hence, sections 2 and 5 share the same stack portion. Thanks to memory renaming, when instruction 5-1 reads stack word 0, it matches with instruction 2-2 write to stack word 0. Both instructions compute the same address $a = rsp + 0$.

The fetch pipeline stage fetches along the section pointed to by IP. The fetch stage has no branch predictor. There are two reasons for such a choice. First, moving fast along the flow is better obtained by a parallel fetch along multiple control-computed sections than by a sequential fetch along a single predicted path, even if the prediction is perfect. Second, a predictor is less cost-effective

[4] hosting core choice to optimize load balancing is out of the scope of this paper.

in a core if the flow is divided into sections and distributed on multiple cores. For these reasons, the fetch stage computes its control rather than predicting it. To keep the stage hardware simple, each cycle fetches and computes a single instruction. As a result, each core fetches more slowly than an actual speculative core but the cores fetch much faster altogether.

Figure 8 shows the fetch-and-decode pipeline stage. The IP addresses the Instruction Memory Hierarchy (IMH, i.e. L1 instruction cache). The fetched instruction addresses the Register File (RF) to read full registers sources. If all the needed sources are full, the instruction is computed in the ALU. Floating point instructions, memory accesses, complex integer instructions and instructions having empty sources are not computed in the fetch stage but later. Computed instructions results are written back to RF, setting the destination register to full. Uncomputed instructions set their destination register(s) to empty.

The fetch stage includes instruction decoding (not shown). When a *fork* instruction is decoded, it generates a section creation message. The created section starts at the next instruction. The current section continues at the *fork* instruction target. It ends when an *endfork* instruction is decoded. The IP register is set to empty and at the next cycle the fifo head message is dequeued and IP and RF are initialized, which starts the fetch of a new section.

As mentioned on figure 8, the stage critical path is longer than in a speculative out-of-order pipeline, including a L1 cache traversal, an instruction decoding, a register file read (2 read ports), an ALU (Arithmetic and Logic Unit) computation and a register file write (1 write port). This leads to a slow frequency processor, such as a GPU. Slowness is to be compensated by parallelism.

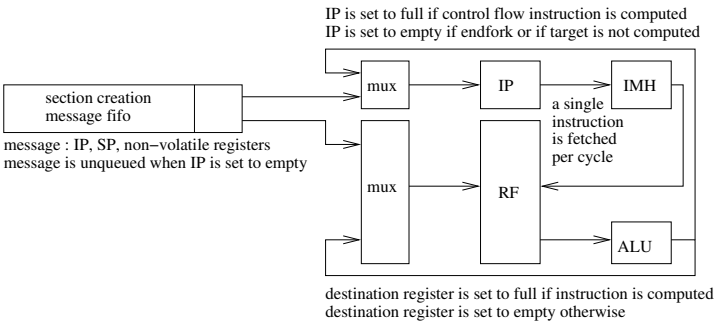


Fig. 8: Fetch-decode pipeline stage

4.2 Core pipeline microarchitecture.

Figure 9 shows the six-stages pipeline building the core microarchitecture. On the bottom part of the design we find a full size rectangle dedicated to commu-

nications with other cores in the processor chip (assumed to be connected by a Network-on-Chip). The forking request unit (FRU) handles the income/outcome of section creation messages. The register renaming request unit (RRRU) handles the income/outcome of source registers renamings. The register exporting request unit (RERU) handles the import/export of renamed registers values. The address renaming request unit (ARRU) handles the income/outcome of source memory addresses renamings. The memory exporting request unit (MERU) handles the import/export of renamed memory values. The instruction exporting request unit (IERU) handles the outcome of retired instructions.

The fetch-decode and register-rename stages follow a single section up to its end. Renamed instructions enter in-order in a Reorder Buffer (ROB in the retire stage) and in the Instruction Queue (IQ in the execute-write-back stage). Load-/store instructions enter in-order in the address renaming queue (ARQ in the address-rename stage). Register-register instructions from multiple sections are mixed in the execute-write-back stage. They read sources in a memory keeping the core renamed registers (register renaming memory or RRM). Load/store instructions compute the access address in the execute-write-back stage and save it in the ARQ. Memory addresses in ARQ are renamed in-order and renamed memory access instructions enter the Load/Store Queue (LSQ). As these instructions are renamed they can be run out-of-order.

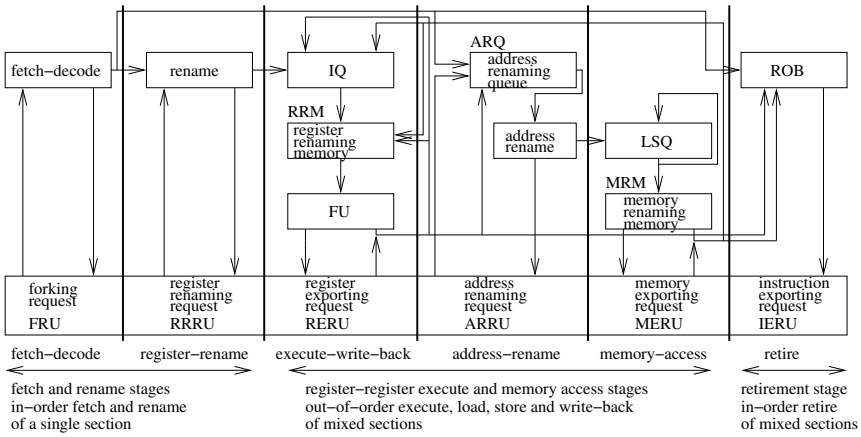


Fig. 9: Six-stages core pipeline

Register renaming. Each instruction in the core can be uniquely identified by its section identifier and its ordinal number in the section. If we assume the number of sections hosted by a core is bounded by *max_section* and the number of instructions in a section is bounded by *max_instruction*, a core can

host at most $max_section * max_instruction$ instructions, i.e. as many renamed destinations. Each renamed destination can be uniquely identified by a pair (#section, #instruction) (or (s,i) in short).

The fetch-decode stage delivers a partially evaluated instruction to the rename stage which renames the empty sources, i.e. either find their local (s,i) renaming or, if not hosted by the local core, look for the producing core.

If a source s may not be locally renamed by an instruction $inst$ (no instruction previously fetched in the same section has written to s), its value is requested to the preceding section, i.e. to another hosting core through the RRRU. In the same time, a destination d is allocated in RRM for the missing register, as if s would be locally written. This destination d serves as a caching of the missing source s. Later references to s in the same section are renamed d.

The renaming request travels from section to section until a producer is found (i.e. an instruction writing to s). In the *sum* function example, for any size of the data, the only register to be renamed is register rax in instructions 12 and 17. In both cases, the producer is the section just preceding the renaming one.

Each core on the travel receives the renaming request in its RRRU. It renames source s. If the renaming misses, the request is propagated through the RRRU. If it hits, an export instruction is added to the IQ which waits for the requested value. When it is received, the export instruction is run, reading the value in RRM and sending it to the requesting section through the RERU.

The value reaches the requesting core through its RERU. It is written in RRM, entry d. The IQ is notified that destination d is ready, which allows the waiting instruction $inst$ to start execution and read d in RRM as source s.

Renaming seems very sequential. To find the producer of source s, the trace of executed instructions must be travelled backward from the consumer down to the first instruction writing to s. However, i) only the portion of code ranging from the producer to the consumer is to be visited and ii) stack pointer based variables with a positive offset (e.g. $0(rsp)$) benefit from a shortcut eliminating instructions belonging to a call level deeper than the consumer. Statement i) implies that if a producer is close from a consumer, the portion of code to consider is short. This is the case for function results used by the resume code (register rax in the *sum* function example). Statement ii) implies that if a consumer and a producer address the same stack frame, the portion of code to consider is also short, excluding in between function calls. This is the case for local variables set at function start and later used (stack location $0(rsp)$ in the *sum* example).

Only for global variables and heap pointers the travel from producer to consumer can represent a long path, as all the in between sections must be visited to make sure they do not contain any more recent producer of the consumed address. However, the caching feature ensures that the high price is rarely paid. Once renamed in an intermediary consuming section, a global or heap variable is cached and it can be consumed by neighbour sections for cheap.

Instruction 2-2 on figure 6 illustrates fast renaming applying statement i). After the renaming of register rax misses in section 2, a request is sent to the

core hosting section 1. The renaming hits in section 1 (instruction 1-10) and the value of rax is sent back to the core hosting section 2.

Instruction 5-1 illustrates fast renaming applying statement ii). After the renaming of stack location $0(rsp)$ at address a misses in section 5, a request is sent to section 2, bypassing sections 3 and 4 which are at a lower call level than instruction 5-1. The renaming hits in section 2 (instruction 2-2) and the value of $0(rsp)$ is sent back to the core hosting section 5.

Instruction 1-8 illustrates high price renaming of global variable $t[0]$. The request travels back to the loader which installs code and global initialized data. The hardware can i) access to full cache lines instead of single words and ii) cache the accessed lines along the return path. From statement i), instruction 1-8 gets its own word $t[0]$ but also instruction 1-10 word $t[1]$. Moreover, from statement ii), core 1 caches the memory line containing $t[0]$ up to $t[4]$ which can be consumed cheaply by sections 2 ($t[2]$) and 3 ($t[3]$ and $t[4]$)[5].

Memory renaming. Memory renaming is done like register renaming. Instead of a Register Alias Table (RAT), the address-rename stage uses a Memory Address Alias Table (MAAT). There is one MAAT per section, each MAAT having one entry per instruction in the section. Each MAAT is a fully associative cache. Renaming address a in section s means looking for a in section s MAAT. If the search misses, it indicates that section s does not write to a and the renaming should be looked for in the section preceding s.

A memory renaming request works like a register one. When renaming address a misses, a memory line is allocated in the MRM to host line l_a containing a (it caches l_a). The renaming request travels along contiguous sections until a producer of l_a is found. Each visited core receives the request in its ARRU. The renaming request is enqueued in the ARQ to avoid bypassing renamings of addresses of the same section not yet done. When the request is dequeued, if the renaming misses, it is propagated to the preceding section. When it hits, an instruction to export l_a is added to the LSQ. The exported memory line travels back to the requesting core where it is received in the MERU. From there, it is written to the MRM and the LSQ is notified that a is ready.

Memory renaming transforms the code at run time into a single assignment form. Synchronisation of consumers with their producers and single assignment ensure sequential consistency without any coherency protocol requirement. Hence, the processor memory distributed in the cores is coherent.

Parallelizing retirement. Sections are created in parallel, as *fork* instructions are fetched and run. To keep the cores loads acceptable (no more than $max_section$ sections hosted in a core), terminated sections should retire at the same speed, i.e. retirement should be parallelized. Retirement frees the sections in the cores to allow new sections in.

[5] stores update full lines. The loader sets a cleared line updated with $t[0]$. This updated line is updated again with $t[1]$. After five such updates, the full line containing $t[0]$ up to $t[4]$ is set and can be exported to consumers, i.e. sections 1, 2 and 3.

Instructions retire in-order (within their section) by exporting their result to the successor section[6]. To be retired, an instruction must be terminated. An instruction is not exported if it holds a result useless for successors, i.e. if i) its destination is updated later in the section or ii) it writes to a non volatile register or iii) it is a control flow instruction or iv) it writes to stack or heap in a location freed later in the section or v) it has exported its computation to a consumer. The successor discards exported instructions if their production may not be consumed anymore. An exported instruction i is discarded by the successor section s' if i writes to a destination renamed or freed in s'.

An exported instruction is accepted by the successor section s' only if its destination is neither the source nor the destination of any instruction in s'.

In the *sum* function example, no instruction is exported. For example, the instruction consuming the final sum to be displayed receives its source from instruction 5-1 which does not need to be exported (statement v).

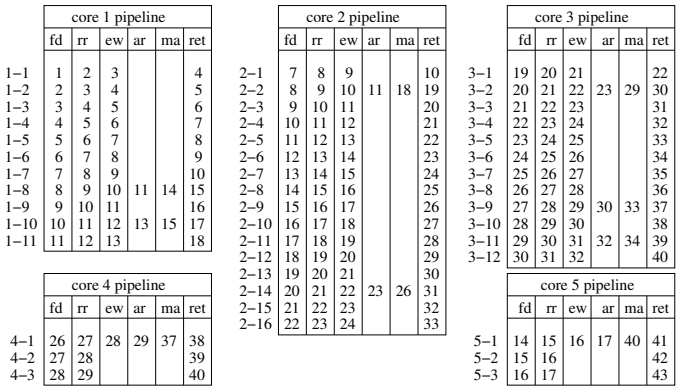
An exported instruction is sent to the successor section through the IERU. It is received in the RRRU (register write) or in the ARRU (memory write). The destination is tentatively renamed and in case of a hit the exported instruction is discarded (already renamed destination). It is also discarded if it writes to stack or heap in a location later freed by the section. Otherwise, the instruction gets a new local renaming and is saved in its new section ROB.

What is new in the proposed core design ? The core shown on figure 9 is much simpler than actual speculative cores. The core is as small as possible to maximize the number of cores on the die. As the core is not speculative, there is no predictor nor renaming repair unit (checkpoints). The LSQ is simpler than the usual Load-Store Queue as loads are not speculative and store-to-load forwarding is not necessary. The data memory hierarchy is kept coherent as only the oldest section can write to and read from it. There is no memory coherency hardware (e.g. MESI protocol handler). There is no vector computing unit (e.g. XMM-like) as vectorization is better obtained through parallelism. Each core implements a single instruction path (no superscalar or VLIW path).

5 Analytical performance evaluation of the parallel execution model on the *sum* example and conclusion.

Figure 10 shows 5 tables (one per core) giving the execution timing of the *sum(t,5)* run. The instructions numbers are given on the left of each table. The columns of a table match the 6 pipeline stages. A value in a column represents the cycle at which the instruction is treated by the corresponding pipeline stage. For example, instruction 1-8 (load) is handled by core 1, fetched at cycle 8, register renamed at cycle 9, load address is computed at cycle 10 and renamed at

[6] The oldest section, i.e. the only one with no predecessor, dumps its renamings to the data memory hierarchy (DMH). When it receives a renaming request which misses, it loads from DMH and exports the loaded line.



	core 1 pipeline					
	fd	rr	ew	ar	ma	ret
1–1	1	2	3			4
1–2	2	3	4			5
1–3	3	4	5			6
1–4	4	5	6			7
1–5	5	6	7			8
1–6	6	7	8			9
1–7	7	8	9			10
1–8	8	9	10	11	14	15
1–9	9	10	11			16
1–10	10	11	12	13	15	17
1–11	11	12	13			18

	core 2 pipeline					
	fd	rr	ew	ar	ma	ret
2–1	7	8	9			10
2–2	8	9	10	11	18	19
2–3	9	10	11			20
2–4	10	11	12			21
2–5	11	12	13			22
2–6	12	13	14			23
2–7	13	14	15			24
2–8	14	15	16			25
2–9	15	16	17			26
2–10	16	17	18			27
2–11	17	18	19			28
2–12	18	19	20			29
2–13	19	20	21			30
2–14	20	21	22	23	26	31
2–15	21	22	23			32
2–16	22	23	24			33

	core 3 pipeline					
	fd	rr	ew	ar	ma	ret
3–1	19	20	21			22
3–2	20	21	22	23	29	30
3–3	21	22	23			31
3–4	22	23	24			32
3–5	23	24	25			33
3–6	24	25	26			34
3–7	25	26	27			35
3–8	26	27	28			36
3–9	27	28	29	30	33	37
3–10	28	29	30			38
3–11	29	30	31	32	34	39
3–12	30	31	32			40

	core 4 pipeline					
	fd	rr	ew	ar	ma	ret
4–1	26	27	28	29	37	38
4–2	27	28				39
4–3	28	29				40

	core 5 pipeline					
	fd	rr	ew	ar	ma	ret
5–1	14	15	16	17	40	41
5–2	15	16				42
5–3	16	17				43

Fig. 10: Execution timing of the *sum(t,5)* run.

cycle 11, renamed memory is accessed at cycle 14 (counting 3 cycles to reach the producer and return the $t[0]$ value) and retired at cycle 15. We assume the 5 sections can be hosted in 5 different cores. We also assume each pipeline stage manipulates a single instruction. Instruction cache L1 is assumed to always hit. From this example we see that the code is fetched in 30 cycles (last fetched instruction is 3-12), i.e. 1.5 instructions per cycle. If the data size is doubled, the fetch time is 42 cycles (104 instructions fetched, i.e. 2.5 instructions per cycle). Only fetch latency (i.e. IL1 miss rate) can impact the fetch time. It is independent of renaming and execute latencies.

The number of instructions is $45 * 2^n + 14 * (2^n - 1)$ for the sum of a $5 * 2^n$ elements array (i.e. 45 for *sum(t,5)*, 104 for *sum(t,10)*). The fetch time is $30 + 12 * n$ (i.e. 30 for *sum(t,5)*, 42 for *sum(t,10)*). For 1280 elements, 15090 instructions are fetched in 126 cycles, i.e. 120 instructions per cycle. This shows that even though one instruction is fetched per cycle per core and the control is computed rather than predicted, fetching in parallel is efficient. Even for modest data sizes, it outperforms any speculative fetching hardware.

Renaming is not penalized by the distribution of the code nor by its extension to memory locations. Most of the sources are provided by register value copy at fork. Function results and stack local variables are obtained from the predecessor section. Global variables are reached quickly from the first section (which starts at the *main* function entrance) and then fastly propagated to other sections, thanks to full memory line caching. Eventually, instruction retirement frees the resources at the same rate they are allocated, avoiding cores saturations. The retirement time is[7] $43 + 15 * n$. For 1280 elements, the 15090 instructions are retired in 163 cycles, i.e. 92 instructions per cycle.

[7] 15 cycles is the fetch time of instructions (figure 5) 2, 3, 8-10 (5 cycles), the creation time of the forked section (2 cycles), the fetch time of instructions 11-16 (5 cycles) and the retirement of instructions 17-19 (3 cycles).

Devil is in the details. Fine simulation of the model is necessary to prove that distant ILP present is captured. Two such simulators are on-going projects: a VHDL implementation of the core pipeline and a *qemu* and *simplescalar* based simulator to quantify the IPC performance of a many-core processor.

The model presented focuses on functions. In the same way, loops can be parallelized. *For* loops can be vectorized, each iteration forming a separate section with no control. It heritates its iteration counter that can be saved in a register and used in the iteration body. *While* loops can be parallelized, launching each iteration in sequence (no speculation) but parallelizing their bodies.

With the introduction of many-core chips as general purpose processors, the time has come to produce parallel programs automatically. This paper suggests that we are not so far from the goal and the hardware can greatly help.

References

1. Shun, J., Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Kyrola, A., Simhadri, H.V., Tangwongsan, K.: Brief announcement: The problem based benchmark suite. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures. SPAA '12 (2012) 68–70
2. Wall, D.W.: Limits of instruction-level parallelism. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS IV (1991) 176–188
3. Tomasulo, R.M.: An efficient algorithm for exploiting multiple arithmetic units. IBM J. Res. Dev. **11**(1) (January 1967) 25–33
4. Tjaden, G.S., Flynn, M.J.: Detection and parallel execution of independent instructions. IEEE Trans. Comput. **19**(10) (October 1970) 889–895
5. Nicolau, A., Fisher, J.: Measuring the parallelism available for very long instruction word architectures. Computers, IEEE Transactions on **C-33**(11) (1984) 968–976
6. Wall, D.W.: Limits of instruction-level parallelism. In: WRL Technical Note TN-15. (1990)
7. Austin, T.M., Sohi, G.S.: Dynamic dependency analysis of ordinary programs. In: Proceedings of the 19th annual international symposium on Computer architecture. ISCA '92 (1992) 342–351
8. Lam, M.S., Wilson, R.P.: Limits of control flow on parallelism. In: Proceedings of the 19th Annual International Symposium on Computer Architecture. ISCA '92 (1992) 46–57
9. Moshovos, A., Breach, S.E., Vijaykumar, T.N., Sohi, G.S.: Dynamic speculation and synchronization of data dependences. In: Proceedings of the 24th Annual International Symposium on Computer Architecture. ISCA '97 (1997) 181–193
10. Postiff, M.A., Greene, D.A., Tyson, G.S., Mudge, T.N.: The limits of instruction level parallelism in spec95 applications. CAN **27**(1) (1999) 31–34
11. Cristal, A., Santana, O.J., Valero, M., Martínez, J.F.: Toward kilo-instruction processors. ACM Trans. Archit. Code Optim. **1**(4) (December 2004) 389–417
12. Sharafeddine, M., Jothi, K., Akkary, H.: Disjoint out-of-order execution processor. Transactions on Architecture and Code Optimization (TACO) **9**(3) (sept 2012) 19:1–19:32
13. Goossens, B., Parello, D.: Limits of instruction-level parallelism capture. Procedia Computer Science **18**(0) (2013) 1664–1673 2013 International Conference on Computational Science.