Abstract

Multicore is quickly becoming the norm, even in the embedded world. This trend is thought to be sustained as long as the progress of microelectronics will permit, by regularly doubling the number of available computing cores on a given chip. That means that multicore aware programs are no longer an option, but at the same time, parallel programming is still very hard nowadays despite progresses in parallel languages.

From these observations, we investigated how to use multicore in a fashion that would be as close as possible as single processor programming, at least from the program design and debug point of view, by a kind of generalization of the superscalar design. One of the aim is also to guarantee little latency and synchronization overheads: for that a loose synchronization mechanism called “weak synchronization” is described. It is supported by a hardware/OS co-design. Some first evaluations of the system performance are provided.

Keywords: Multicore processor, execution model, hardware/software codesign, OS for parallel execution

1. Introduction and motivation

1.1. Context

Moore’s law nowadays translates as doubling the number of computing cores on a chip every 18 months rather than doubling the complexity of single processor cores. As a consequence programs must become more and more multicore aware and multicore friendly. Nonetheless, this trend poses several difficulties: parallel programming is generally hard to design and to debug especially with parallel scalability in mind; Amdahl’s law limits the efficiency of parallelism for single tasks; and synchronization issues can further impede performance of a given application.

But from a simple programming point of view, multiscalar processors can be viewed as very good targets: sequential programming is more natural and easier to understand, multitask operating systems for single processors were also quite easy to design, and Out of Order (OoO) execution was a good tool for taking advantage of Instruction Level Parallelism (ILP). However, the performance of multiscalar processors reached a plateau because of the limits of ILP.

So, an intermediate way would be to generalize the old school multiscalar processor approach into a multicore one. In this case, cores would be considered as ordinary execution units, only bigger. The difference would be that

\textit{Email addresses: stephane.louise@cea.fr (Stephane Louise), vincent.david@cea.fr (Vincent David)}

\textsuperscript{1}Corresponding author

\textsuperscript{2}This work was done while this author was PhD student jointly at CEA, LIST and Paris XI University
instead of low complexity instructions, the “cores as execution units” would had much complex processing and, as a consequence, longer execution time. Of course the vision at this level is rather naive, and a lot of care is needed to manage a coherent execution with such an architecture. Nonetheless it is the starting point of our reflexions.

1.2. General idea

Let us start with the architecture principle as shown in figure 1. In this figure, there is an ordinary SuperScalar Processor (SSP) with a standard ISA, in which is added a special execution unit called Allocation and Control Unit (ACU) whose role is to allocate, manage the execution and the synchronization of the other cores (Advanced Processing Units – APU). These cores or APUs can be homogeneous but in the general case, they are heterogeneous. This architecture relies also on “on chip memory (OC memory or OCM)” to store both processing instructions and data to process (to reduce latencies), and the DMA engines that drives the flows of data and micro-programs between OCM and external memory are also driven by the ACU as special cases of APUs.

![Figure 1: Principles of the architecture: a mostly standard superscalar processor (SSP) is modified so that a special unit (Allocation and Control Unit – ACU) is used to allocate and synchronize parallel processing execution on several cores (APUs)](image)

The general idea is to execute the “control intensive” parts (i.e. parts that are hard to parallelize) of a program on the “control core” (or standard core/SSP) and the compute-intensive parts on the APUs. Since APUs are supposed to be specialized units for certain types of code (e.g. vector computing, floating point, etc.) they can be made very power efficient for their domain of specialization, and can be put in reduced power mode when not in use.

1.3. Goals and foreseen issues

The primary long term goal would be to have a programming model close to sequential programming. That probably does not mean, at least in a first time, that automatic parallelization is the preferred way (instead using a parallel extension of a sequential language would be easier). Nonetheless, the programming aspects are out of the scope of this paper. The first goal on this roadmap is to have an execution model closely related to an ordinary Out of Order (OoO) superscalar single-processor: special instructions on the control core (SSP) would prepare, launch and synchronize micro-programs in parallel.

The foreseen issues with such a model is that execution time and latencies for these micro-programs are expected to be much larger than ordinary instructions, therefore allocation and synchronization times can have a huge impact. This impact must be minimized with regards to several criteria:

- out of order termination of parallel processing should be allowed,
- interruptions on the SSP should be allowed without impeding the execution on the APU (asynchronous execution of interruptions),
- even though the overall execution principle is not Von Neumann equivalent, the overall computation must remain deterministic for the results, whatever the outcomes or the order of execution (within allowed bounds [7]), or otherwise generate an exception (which should be rare).
Our conclusion is that it can be done by co-designing some specific parts of the Operating System (OS) on the control core with the automatic allocation and synchronization mechanisms of the ACU.

This paper presents the first steps of optimization and OS co-design of such an architecture. Section 2 presents the general model of execution and the principles for co-designing some parts of the OS with the facilities provided by the ACU, section 3 and 4 respectively explain the specific architecture obtained for the ACU, and how to adapt a single processor OS to this new kind of parallel processor. Section 5 shows an example of how a given multitask program should be executed on this kind of target, and benchmark evaluation is done in section 6. In section 7 we compare to other kind of multicore systems aimed at the same kind of goal before giving some perspectives and hinting at future works in section 8.

2. General principles

2.1. Hardware level

The superscalar processor (SSP) is at the heart of this global processor, and acts as an entry point for the main programs (or tasks). It is modified by adding a special instruction unit called Allocation and Control Unit (ACU). This unit can execute special instructions that enables program executed on the SSP to allocate Auxiliary Processing Units (APUs, i.e. specialized cores or IPs). These special instructions include:

- Allocate a number of (logical) APUs,
- Associate a (typically tiny) program (or “job”) to run on the allocated APUs,
- Associate data to the job and APU,
- Run the program,
- If required, synchronize (and desallocate if no longer required) several APUs before continuing the main program or task execution flow.

Also several instructions are needed to manage the On Chip Memory (OC Memory), but these will not be discussed further. For the sake of the discussion, it suffice to consider the DMA engines and memory allocators as special APUs. APUs will be allocated at a logical level, and the number of logical APUs is more than the number of physical (real) APUs. Hence, logical APUS offers a virtualization of the APU pool and allows for a late allocation and early release of the physical APUs (at SSP level). The way to associate logical and physical APUs can be closely related to the mechanisms of register renaming in OoO execution [10, 4, 9]. Nonetheless, it works backward since the principle is to describe series of jobs thank to the logical APUs: the logical APU pool acts as a job pool for the real APUs.

2.2. Execution model

The easiest way to understand the proposed execution model is probably to look how it works on a simple example. Let us look at the execution scheme in table 1.

The principle is to always allocate early and synchronize late (at the SSP level), with the absolute constraint that whatever the exact order of execution before the synchronization point, the result is valid. So it should be better to avoid strong synchronization before continuing the program execution. That is why we introduced a mechanism called “weak synchronization”. With this mechanism, it is possible to continue work on APUs even when the associated task on the SSP is preempted. So if it is possible to detect whenever the allocation or the synchronization mechanism is stalled (e.g. because the pool of available APUs is exhausted or because the processing on APUs that must be synchronized is not yet finished), the OS can be informed that the current task on the SSP should be commuted until the conditions of the stall disappear (processing finished, or APU freed). This can be achieved by co-designing the ACU with tiny modifications in the support OS.
Table 1: A simple program and its execution on the different parts

<table>
<thead>
<tr>
<th>Original program</th>
<th>Executes on SPP</th>
<th>executes on ACU</th>
<th>executes on APUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a = \det(\text{Mat}))</td>
<td></td>
<td>(\text{APU}1 := \text{allocate}_\text{APU} )</td>
<td>(\det(\text{Mat})[\text{APU}1])</td>
</tr>
<tr>
<td>(b = \text{dotProd}(\text{vect1}, \text{vect2});)</td>
<td></td>
<td>(\text{APU}2 := \text{allocate}_\text{APU} )</td>
<td>(\text{dotProduct}(v_1, v_2)[\text{APU}2])</td>
</tr>
<tr>
<td>(c = \text{dotProd}(\text{vect3}, \text{vect4}))</td>
<td></td>
<td>(\text{APU}3 := \text{allocate}_\text{APU} )</td>
<td>(\text{dotProduct}(v_3, v_4)[\text{APU}3])</td>
</tr>
<tr>
<td>if((a! = 0))</td>
<td>(a := (\text{APU1})_{\text{result}})</td>
<td>if((a! = 0))</td>
<td></td>
</tr>
<tr>
<td>(d = (b \times c)/a;)</td>
<td>(b := (\text{APU2})_{\text{result}})</td>
<td>(\text{APU}2 := \text{run}_\text{APU} )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(c := (\text{APU3})_{\text{result}})</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(d := b \times c/a)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. ACU design: tight coupling the standard core with APUs

As seen previously, the ACU is nearly a standard execution unit in the SSP, and only its goal is not standard: it manages allocation and synchronization of processing cores (APUs). It is a pipelined execution unit for some specialized instructions, therefore it is fed by a standard dispatch unit, outputs to the completion unit, and is in interaction with the global register bank, as an ordinary execution unit in a OoO superscalar processor. For its internal architecture, let us look at figure 2:

- The pipeline executes specific instructions for allocating, running and synchronizing APUs, plus some instructions to help the Operating System. It does not need many stages (we used typically 3 stages).
- The local register file stores the states of the APUs and other values of interest.
- The APU IT decoder sorts and feeds exceptions and events that comes from the APU to the APU manager. These events and exceptions range typically from “end of execution on APU\(i\)” to “an error occurred on APU\(j\)”.
- The APU manager is the core automaton of this unit and is divided into 2 subsystems:
  - The logical APU manager: manages the virtualization of the APUs; it allocates and synchronizes the logical APUs which are the APUs as manipulated by programs running on the SSP (and can be viewed as job descriptors),
  - The physical APU manager: manages the real cores; it makes the correspondence between a logical APU and a physical one when one physical core is available. Its goal is to make late allocation and early release of the real cores so that physical APUs are quickly available to execute another job.

Mechanisms

Here is what happens when the principal instructions are executed on the ACU:
Figure 2: Internal architecture of the Allocation and Control Unit (ACU): as a standard execution unit, it fits between the dispatch unit and the termination unit. It also requires access to the general purpose register bank and summarizes the APU allocation, execution and synchronization states thank to special interruptions or exceptions.

- allocate_APU: takes an available APU in the pool of logical APUs and returns its number to the SSP. If no logical APU is available, a (maskable) exception is raised. If the exception is masked, the SSP stalls until a logical APU is released, otherwise the SSP branches in the OS.
- associate a job to an APU: updates starting context of the logical APU with the address of the program in OCM.
- associate data to APU: updates starting context of the logical APU with address of data (it is pushed on top of its future stack).
- run APU: associates an available physical APU with a job description defined by the number of the logical APU. If no physical APU is available, as in the case of logical APU allocation, either an exception is raised for the OS on the SSP, or the SSP is stalled.
- synchronize APU: takes a mask of several logical APUs to synchronize. If any job of the mask is not finished, an exception is raised for the OS, or the SSP is stalled if the exception is masked,
- loading SSP task context: when the OS commutes to a new task, the task address descriptor must be loaded. The old task context is saved in memory at the previously given address and the new one is loaded from the memory at the given address (the use of OC Memory is encouraged for these partial contexts).

Some other mechanisms are not associated with instructions, but with events from the physical APUs:
- When a physical APU finishes its job:
  - the physical APU is returned to the free physical APU pool unless a job description from a logical APU is already available for running. In this case, the older job available (FIFO style) is attributed to the APU,
  - the logical APU associated with the finished job is released (if the task is not the task running on the SSP, what can be known with a context address given by the OS to the ACU, the mask at the context address is updated, otherwise the mask in the local register file is updated),
  - if a task was preempted for a stall due to the lack of APU, its exception mask is also automatically updated thank to the address given in the context.
- other mechanisms are associated with exceptions on the APUs, including debugging interruptions but will not be discussed further here.

The goal here is to do all the work that can be done, without the help of the OS, automatically. Hence, a lot of potential preemptions on the SSP are avoided, as long as they are not necessary.

These mechanisms require that several registers and tables are integrated in the ACU:
Table 2: Changes to standard task management for the underlying OS are required for a full support of the multicore capabilities. It can run without it but performances are better when several subtypes of running tasks are allowed. This means even though the processor is seen as a sequential one by the OS, several tasks can be virtually running at the same time. Trigger backs are usually prepared by the ACU which updates the task context of its own.

<table>
<thead>
<tr>
<th>New “task stalled” states</th>
<th>state trigger (exception)</th>
<th>trigger back</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual APU Allocation Stall</td>
<td>APU allocation exception (no more log. APU)</td>
<td>Logical APU freed</td>
</tr>
<tr>
<td>OC Memory Allocation Stall</td>
<td>OC Memory allocation exception (no more free OCM)</td>
<td>Memory freed</td>
</tr>
<tr>
<td>APU Synchro stall</td>
<td>synchronization exception (some jobs are not finished)</td>
<td>APU execution ends</td>
</tr>
<tr>
<td>Phys. APU Alloc. partial stall</td>
<td>No more physical APU (optional exception)</td>
<td>APU Execution ends</td>
</tr>
</tbody>
</table>

- A partial context address for the task running on the SSP. It is set by the OS and permits to track the allocation, execution and synchronization status associated with each task running on the SSP which is associated with at least an APU.
- Several mask registers: a virtual APU allocation mask, a virtual APU synchronization mask and a physical APU running mask.
- Correspondence tables between logical APUs and physical APUs (including partial context addresses of tasks that have allocated/non synchronized APUs but are not the running task)

Other tables and registers are used but are not required to understand how the system works.

4. Design principles for operating systems

The principle to extend the functionality of a single processor OS to this multicore system is quite simple: only a new register of the ACU, the task partial context address, must be loaded with the address of the said partial context of the new task, in the context-switch function. By this sole addition, each task receive a stall mask and a synchronization mask which enable the ACU to update the execution state of the tasks (with regards to the APUs) even when they are not the current running task on the SPP. The added states are shown in table 2.

All tasks that use APUs receive an extended context which can be read by the OS (usually 1 to 3 cycles memory read, because it is either in L1 cache or OCM) to dynamically know if a given task is still stalled on the SPP or not. Hence, several tasks can execute at the same time, usually because they have parts running on the APUs, even thought they are not executed at this given time on the SPP.

5. Illustration of execution principles

The best way is to provide a simple example of an execution and how it organizes itself on the chip, between the tasks, the OS, and the micro-programs on the APU. Let us take a simplified version of the architecture with only 5 logical APUs to manage 4 physical APUs. There is 3 tasks in this example as shown in figure 3:

- Task T1 uses 2 logical APUs. It allocates them at time $t_{1/1}$ and $t_{1/2}$, starts execution at time $t_{1/3}$ and $t_{1/4}$, then at time $t_{1/5}$ asks for synchronization. Since the execution on APUs is not finished at time $t_{1/5}$, the OS preempts T1. The execution is restored at time $t_{1/6}$, when both logical APU have their execution finished and the OS has again the opportunity to execute T1 on the SSP. Then at time $t_{1/7}$ task T1 ends.

- Task T2 uses 3 logical APUs: it allocates them at time $t_{2/1}$ to $t_{2/3}$, ask for execution at time $t_{2/4}$ to $t_{2/6}$ and ask for synchronization at time $t_{2/7}$. At this time, T2 is preempted by the OS since synchronization is not yet done (in fact since there is only 4 physical APUs, one of the logical APU was not physically allocated at this time and until time $t_{0/1}$). Like task T1, execution is restored by the OS at time $t_{2/8}$ and finishes at time $t_{2/9}$.

- Task T3 uses 2 logical APUs like T1. But since there is only 5 logical APUs on the system, T3 is preempted by the OS when it tries to allocate its second logical APU (time $t_{3/2}$). It can be restored by the OS at time $t_{3/3}$ since a logical APU is available, and after launching the programs on APUs ($t_{3/4}$ ) it asks for synchronization ($t_{3/5}$). It is only restored at time $t_{3/6}$ by the OS, until its end.
Figure 3: An example of scheduling of 3 tasks on the architecture with the execution model. Task T1 use 2 logical APUs with quite long execution time, task T2 use 3 logical APUs with also long execution time. Task T3 use 2 logical APUs with shorter execution time. The architecture is simple enough to follow the execution principles: 5 logical APUs are managed and 4 physical APUs are available. All time stamps are not displayed for sake of readability (in this case they are numbered by increasing order of the second index, and the task index never changes implicitly).

The instructions executed by the ACU can generate (maskable) exceptions when the conditions for pursuing immediately their execution on the SSP are not met: it authorizes the OS to make pertinent preemptions of staled tasks and only restores their execution when the expected conditions are fulfilled. Therefore the OS can profit as soon as a task is stalled to execute another task on the SSP. This reduce the overall stalling time of the SSP.

Special times $t_{0/1}$ to $t_{0/7}$ are worth noting (plain vertical lines of figure 3): at these times the ACU works alone without interventions from the OS or the tasks to accelerate the execution and perform automatic actions:

- **$t_{0/1}$**: physical APU1 ended job for T1. The ACU updates the execution status of T1’s APUs (see section 3), since physical APU1 is now free, the ACU associates it with logical APU5, and starts the new processing.

- **$t_{0/2}$**: the same for physical APU2. This time the execution status for T1 is cleared so the OS will know that T1 is no longer stalled by simply reading T1 status, next time it is activated. N.B.: it is possible to generate an early interruption on this kind of events to wake up stalled tasks as soon as possible, but this interruption is disabled at this stage on this example.

- **$t_{0/3}$ to $t_{0/5}$, and $t_{0/7}$ are similar to $t_{0/1}$ and $t_{0/2}$**.

- **Between $t_{3/5}$ and $t_{0/6}$**: when synchronization exception for T3 is raised at $t_{3/5}$ the OS as no more active task to run. Therefore it authorizes “wake on end of synchronization” of the APU (as noticed before for $t_{0/2}$ and turn to low energy idle mode (only a part of the ACU is still active). When the last physical APU ends its execution for T2, the ACU raises an interruption to wake the processor from its idle mode.

It is worth noting that the tasks shown here are fine grain tasks: time scale is given by the OS preemptions (black rectangles in the last line of figure 3) that are evaluated under 1µs, even on 500MHz processors. On this example, SSP use is over 95%, 40% for logical APUs, and 60% for physical APU, so there is an added value in providing a logical-physical APU virtualization, for efficiency if for no other reasons.

6. Evaluations

6.1. Silicon implementation of the ACU

Raphaël David and Alexandre Guerre from the LCE laboratory in CEA, LIST, performed for us a (not optimized) silicon synthesis of an older specification of the ACU, but which provided first figures regarding feasibility of these
ideas. They used the Design Compiler tool from Synopsis to generate the synthesis in 65nm technology. With this (not optimized) design, they achieve a 500MHz clock and a surface under 50000µm² which would be less than 1% of the surface of a standard two way superscalar OoO processor like Mips R10k. Power consumption values could not be obtained, though. The typical timing that they had with their design are summarized in the following table:

<table>
<thead>
<tr>
<th>ACU instruction</th>
<th>Number of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>APU allocation</td>
<td>5</td>
</tr>
<tr>
<td>Associate Data or Micro-program</td>
<td>6</td>
</tr>
<tr>
<td>launching job</td>
<td>15</td>
</tr>
<tr>
<td>synchronization</td>
<td>3</td>
</tr>
</tbody>
</table>

It shows that the formalism is achievable, and can have high performance once the design is optimized.

6.2. Benchmark setup and simulation environment

We use 2 types of cores: one Mips R10k-like (OoO) core and a vector (in order execution) core (Altivec-like). Our reference architecture use the R10k-like as SSP and a varying number of vector cores as specialized APU. We compare it with simple R10k-like only multicore and Vector core only multicore, both with dedicated On Chip Memory like our architecture. Hence, all architectures have the same access times to data or programs. Such a comparison, using the same cores and same access time, permits to measure the real advantages that our execution model provides over traditional ones, independently of a particular implementation.

In a first round of evaluation, 3 programs were tested:

- Radar: an embedded HPC application (heavy signal processing, mostly vector computing),
- Mesa GL: as part of Spec’95 benchmark suites,
- An in house simple multitasking application (3 tasks only with periodic activation, quite typical of embedded applications).

We used the Radar benchmark over others such as the Splash benchmark because our laboratory main domain is embedded software, so we already knew this application from other tests. It has a HPC profile (i.e. embarrassingly parallel) and is only tested to check that there is no regression with regards to simple multithreaded applications. This is indeed the case and we have only a marginal gain (about 2%) coming from the hardware based automatic allocation mechanism. So this one will not be developed any further in the following, since the obvious conclusion is that usual HPC applications have little to gain from this architecture.

A modified version of the SimpleScalar OoO simulator[8, 1] that authorizes multitask was used to perform the simulations. Nonetheless, as a full OS was not simulated on SimpleScalar, overestimations of OS costs were used when needed. Finally, the APU themselves were not simulated, only their results and the time required to complete each job was used (the parallel parts were simple jobs with static loop bounds and simple CFG so processing time is constant). If a given time could not be accurately determined, overestimations were chosen instead, and error margins were tracked. So the following evaluations are either accurate within 5 points of percentage, or underestimated.

Adaptation of the programs to take advantage of our architecture was very simply done by inserting ACU instructions (as intrinsic assembly) in the source code before compiling. So, only compute-intensive tiny parts of the programs were optimized. Therefore, the trends of the results are more important than the exact performance shown here.

6.3. Single task case (SPMD, multithread)

As said previously the case of pure HPC shows a linear progression with the number of APU and is not a very good application case (not because it performs badly, but because the gain is marginal). Figure 4 shows the results for mesaGL.

On MesaGL, the vector multicore performs badly on non vector code, especially the context selection of Mesa, so the overall performance is bad. With the R-10k multicore, it performs well on context selection, but poorly on vector computing. With our architecture, we have the best of the two worlds because it performs well on context selection thank to the SSP and also very well on vector parts, thank to the APUs. The interesting point is that a 4 APU architecture performs as well as the best 12 core SMP architecture. This shows that a correct use of heterogeneous computing is the best architecture for efficiency.
6.4. Multitasking case (MPMD, multitasking and multithreading)

The multitasking case is where it is possible to also compare the benefits of weak synchronization over strong synchronization. We tested 2 cases: our in house parallel application on one side and an execution of the Radar program with MesaGL on the other side. The results are the following:

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of cores/APUs</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radar + Mesa</td>
<td>gains over RI0k multicore</td>
<td>31%</td>
<td>34%</td>
<td>35%</td>
</tr>
<tr>
<td></td>
<td>gains over vector multicore</td>
<td>62%</td>
<td>50%</td>
<td>36%</td>
</tr>
<tr>
<td></td>
<td>gains over strong synchronization</td>
<td>5%</td>
<td>6%</td>
<td>4%</td>
</tr>
<tr>
<td>Multitasking</td>
<td>gains over RI0k multicore</td>
<td>92%</td>
<td>94%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>gains over vector multicore</td>
<td>26%</td>
<td>52%</td>
<td>62%</td>
</tr>
<tr>
<td></td>
<td>gains over strong synchronization</td>
<td>18%</td>
<td>47%</td>
<td>56%</td>
</tr>
</tbody>
</table>

The difference between the 2 applications is that Mesa+Radar keeps a HPC-like profile and it spends most of its time to wait for an available APU, whereas the in house application is much less stressful on the APU pool. That is why the weak synchronization mechanism is especially useful when several tasks with a complex CFG are sporadically in need of specialized processing.

It is worth noting that, as in the single task case, this architecture works well and is very efficient with a limited pool of APUs, compared with the SMP multicore case (e.g. the best performances gains are obtained with 8 APUs on these benchmarks). That does not mean that more APUs are not useful but that the return on investment is lower. Again, the case of weak synchronization with 8 APUs is comparable with a 16 core SMP architecture.

7. Comparison with other multicore designs

SMP-like architecture. As was shown in section 6, traditional SMP architecture are not as efficient as our heterogeneous one, unless the application is only composed of embarrassingly parallel kernels and the type of core is well fitted to it. Moreover, it requires a heavy runtime support (w.r.t embedded standards) whereas the ACU permits a very light OS.

Micro-threading. By its philosophy, a close approach to our’s is the Micro-threading concept [5, 2]. Nonetheless this architecture relies on shared registers for communication and only support a split-join execution scheme. More importantly, the micro-threading approach is intrinsically a homogeneous architecture and mostly targeted at embarrassingly parallel problems. Moreover, the micro-threading model does not take multitasking aspects into account contrary to our “weak synchronization” execution model.

Reconfigurable architecture (on-chip FPGA). Some of these architectures can be quite close to our own. Nonetheless the fact that accelerators are usually programed with VHDL or Verilog renders the path difficult if the ease of programing is the final objective.
Cell BE. This architecture is quite close [3] in its concepts. Nonetheless, architecture-wise the interaction of the PPE with the SPE are very loose compared to our architecture. Hence, it is difficult from the PPE to manage task execution on the SPE. Communication take place only through the ring NoC for all the cores. This means large latency. Moreover, DMA engines are associated with SPE because each SPE has its own on chip memory which is not shared with the others. Because of these limitations, we can not implement our principles on this target without an estimated impact of two orders of magnitude on latencies of job allocations.

Carbon architecture. The Carbon architecture [6] is an accelerator for allocation of tasks in CMP designs. It shows a good tolerance to increasing on-die latency. It is nonetheless limited to fork-join models of threads and is not fitted to take advantage of multitasking as in the “weak synchronization” execution model.

8. Conclusion and future works

What was shown in this paper is that the principle of weak synchronization is an easy way to benefit from a heterogeneous multicore architecture. Heterogeneous architecture perform well, as was shown on the MesaGl benchmark, as soon as programs have several aspects (e.g. complex flow graph for a part, and some tiny but often called compute-intensive parts). It is possible to have the same results by using several high-end cores with both OoO execution and vector units. But in this case, such an architecture would lose by a large margin on energy efficiency and silicon surface. It is an asset for embedded applications, but also starts to matter in the server room because of the new trend of awareness about the energy footprint of cloud computing.

The advantages of the architecture is that for heterogeneous applications (sets of different tasks, MPMD principles), it is easier to program at low level, and the OS/runtime glue is lighter since allocation of cores are done at the hardware level, implemented in the ACU. As part of future work, we want to show that the ease to implement heterogeneous computing at low level can be translated as a way to easily implement a parallel compiler for this heterogeneous multicore architecture.

This paper also shows that the weak synchronization principle, more than being a facility which ease HPC job allocations, wins by a large margin on heterogeneous (MPMD) applications where accelerators are moderately used. Such applications are often met in embedded and real-time computing, which explains why our laboratory is interested by such results. As future work, a first simple step is to allow the allocation of several virtual APU at the same time for complex job management. For example, such a virtual “meta-job” would require a DMA-in, one or several accelerators/APUs and a DMA-out for the whole processing. It should be easy to generalize our concept of virtual APUs to these complex jobs.

Last but not least, we would want to implement a real multiprocessor support, hence with several SSP. That means that task migration should be allowed across SSPs, even when jobs on APU are not finished. This requires a mean to share APUs between several SSP, which would not be difficult thank to the task partial context of the ACU. Nonetheless the prime issue is the architecture of the on chip memory in that case.

Thanks. We would like to thank Raphaël David and Alexandre Guerre from CEA, LIST, for experimental results supporting this work.

References