Infra-JVM - Report 1

SCHEDULING IN A PARALLEL ERA

April 9, 2013

Xavier de Rochefort
Laboratoire Bordelais de Recherche Informatique (LaBRI)
Progress team - Software Engineering research theme
xavier.de-rochefort@labri.fr

Abstract

The Infra-JVM project aims to enhance the design of the Java environment to better manage resources [1]. The Progress team of the LaBRI [2] focus on the processor resource and the new challenges raised by the adoption of microprocessor architectures that allows parallel computing (simultaneous multithreading [3], multicore [4]).
The standard libraries of Java supply parallelism abstractions. The classical thread abstraction has been introduced in the earliest release of Java with the class *java.lang.Thead*. More recently, the release 1.5 has introduced the package *java.util.concurrent*. This package is an implementation of the fork/join model [5]. Java developers can thus express parallelism. However, the Java language does not provide ways to express reservation of processor resources. Developers do not have control over the way parallel flows of execution are dispatched on the processor resources. In other words, there is no way to express the scheduling of created tasks. Scheduling is operated by operating systems.
This first report covers the parallel architectures landscape, the related problems, the scheduling algorithm to address them and their implementation.

Contents

1 Introduction

Scheduling problems occur when it is necessary to organise the use of a shared *resource* given a set of constraints. A solution to a given scheduling problem must respect the set of constraints and can target the optimisation of a set of criteria.

Scheduling problems are common. For example, an airport must schedule the take-offs and the landings of the planes depending of the numbers of landing runways available, a college must schedule the courses depending of the teachers and classrooms availability etc. In the former problem, the resources are runways landing. No constraint can be given, however the schedule can be guided by an optimality criterion, for instance the maximisation of the traffic to make the airport cost-effective. In the latter problem, resources are teachers and classrooms and the constraints are that every courses has to be done before a certain date (e.g. the end of semester).

Computer sciences raises scheduling problems too. Modern operating systems [6] makes possible for many users to run many programs in the same time. They are *multitask*. The pieces of hardware that are involved in the execution of a program can not serve every program at the same time. Their use must be scheduled. Operating systems mainly schedule the use of two resources :

- ◇ **Disk**. The physical on-disk locality of addresses targeted by each read/write request has an impact on time spent to proceed a request on the hard-disk. Simply submitting such requests in their arrival order would be critical for performances. A request scheduler will consider locality in order to group requests in the same area. This is globally efficient. On individual bases, a request can be delayed and therefore disadvantaged. However, global performances are improved. It is worth noting that the popularisation of solid state drives, which has no moving parts, may mitigate the need to reorder requests and thus the role of scheduling ;
- ◇ **Core**. The study focus on the organisation of the computations asked by the different programs that can be handled by a multitask operating system. Computation means numerical operations and numerical comparisons. These are processed by a dedicated piece of hardware : *a core*. A core is embedded in an integrated circuit that is commonly called Central Processing Unit (CPU) or *processor*. A core computes operations submitted in the form of instructions. User programs are no more no less a succession of operations, described by instructions, that is intended to be executed by the core in the order of submission

: a *thread* of execution. Moreover, the logic of a program can be described in the form of several threads of execution. Each thread is intended to be processed by a core. This is where an operating system encounters scheduling problems. In a multitask system, many programs that are submitted by one or several users call for the execution of one or several threads. Resources are the available cores that can executes the instruction of each thread. Constraints and optimality criteria may vary depending of the purpose of the programs. A due date or a minimal number of instructions in a given period that has to be met for a subset of thread are examples of possible constraints. In the case of a system that allows lateness of due dates, an optimality criterion could be the reduction of the average lateness.

The writing on this report has been carried out with three main goals in mind :

- ◇ give an overview of existing architectures as the domain of scheduling cannot be considered independently from architectural constraints ;
- ◇ introduce scheduling notions ;
- ◇ make a first state of the art of task scheduling for parallel architectures, intended to be expanded as we go along.

Accordingly, section 2 treats machine design that runs general-purpose operating systems, section 3 introduces general scheduling notions and makes an overview of the state of the art of task scheduling in operating systems. The study only focus on general purpose operating systems, the usage of the term operating systems will use both terms interchangeably.

2 Parallel architectures

2.1 Types of parallelism

Flynn's taxonomy [7] is often mentioned when parallel computer architectures are approached. It is indeed a convenient way to introduce the notion of parallelism in computer science. Flynn defined four categories of computer's architecture based on the number of instruction streams and data streams involved during a processor cycle (Figure 1).

- ◇ Single Instruction Single Data (*SISD*) allows no parrallelism at all. It refers to traditionnal *Von Neumann machines* [8] with a single processor executing a single stream of data (*scalar processor*).

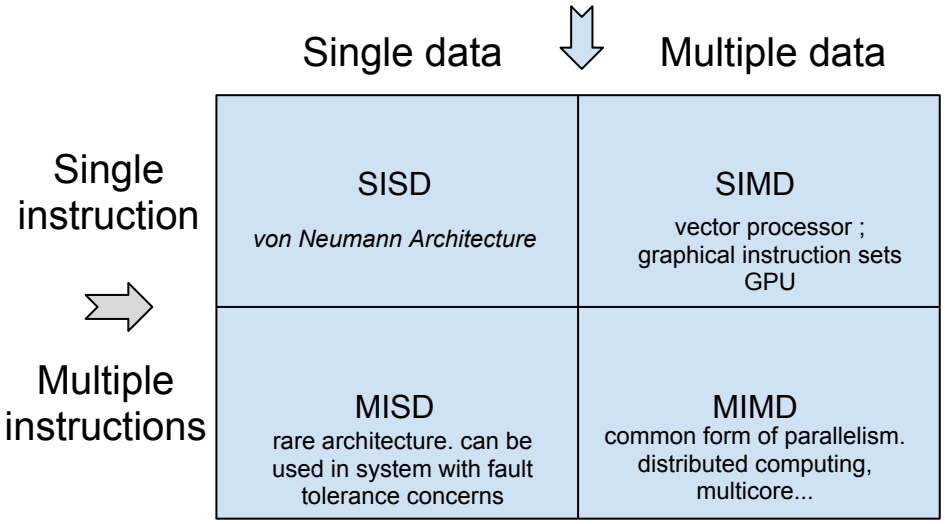

Figure 1: Flynn Taxonomy

- ◇ Single Instruction Multiple Data (*SIMD*) refers to architectures with several processing units executing the same instruction on a data pool. It is also called *data parallelism*. e.g graphical processing units (*GPU*) uses this kind of parallelism to speed image processing : image are arrays of pixels that are most of time processed the same way.

- ◇ Multiple Instructions Single Data (*MISD*) allows multiple instructions simultaneously processing the same piece of data. This type of architecture is rare. e.g embedded devices in fault tolerant critical systems can use it to make redondant check.

- ◇ Multiple Instructions Multiple Data (*MIMD*). This is the most intuitive type of parallelism : different processors can run multiple instructions on different pieces of data, allowing *tasks parallelism* and inherently calling for scheduler. Hence, this category is the one that targets our study

Two main types of MIMD architecture exist. *Distributed architectures* (or *loosely-coupled multiprocessor systems*), and *tightly-coupled systems*. The former considers processing units, called *node*, that does not share any memory space. Nodes are synchronised through high speed communication systems (e.g gigabyte network). This architectures are often called *clusters*. This study does not target such architectures, essentially used to resolve problems with a great need of computing power, for instance physics simulations. The latter, on contrary, considers systems where every

processing units shares a same memory space. It is the common architecture for commodity devices and usual servers, making it a subject of interest in our case of study.

Modern microprocessors allows MIMD parallelism at three levels with instruction level parallelism, simultaneous multithreading and multicore.

2.2 Instruction level parallelism

The Von Neumann computation model has let programmers agnostic to microprocessors parallelism for years, carrying on the illusion of a single processing unit executing a single stream of sequential instructions. Indeed, in a worry of backward compatibility, hardware designers have maintained this model for decades while improving the automatic parallelisation of instructions. This type of parallelism is called *Instruction Level Parallelism* (*ILP*). Two architectures are common :

- ◇ **Pipelined architecture**. Processor instructions take cycle to execute, each cycle using dedicated part of the processor to fetch the instruction from memory, decode it, execute the corresponding operation etc. This can be seem as an assembly-line work : some microprocessor's components being used only once per instruction. In the same way a car factory does not wait for the complete build of a car to begin the construction of another one, *pipelined* processors load multiple instructions in its execution chain, each being treated at different execution stage. The number of stages a processor can pipeline defines its *pipeline depth*. For instance (Figure 2) shows a 4-stages pipeline : each instruction can be fetched, decode, execute, and then its results written back, independently. Four instructions have to be executed. During the first clock cycle the first instruction is loaded into the pipeline for its first stage : fetching. Then, during the second cycle, the same instruction can be decode while it is a new instruction's turn to be pushed into the pipeline for its first stage, and so on. In this ideal exemple, the pipelining technique allows the 4 instructions to be processed in 8 cycles. In the same case, if every instruction had to wait for the former to finish they would have been processed in 16 cycles.

- ◇ **Superscalar architecture**. Pipelining allows stages of different instructions to be processed in parallel. Each stage is treated by independent but exclusive part of the processor. If each of this independent part of the processor is replicated, the same stage can be processed for more than one instruction. The same component is present many times in the same processor. This type

of processor is said *superscalar*. The number of instruction that can be simultaneously processed by a processor defines its number of *way*.

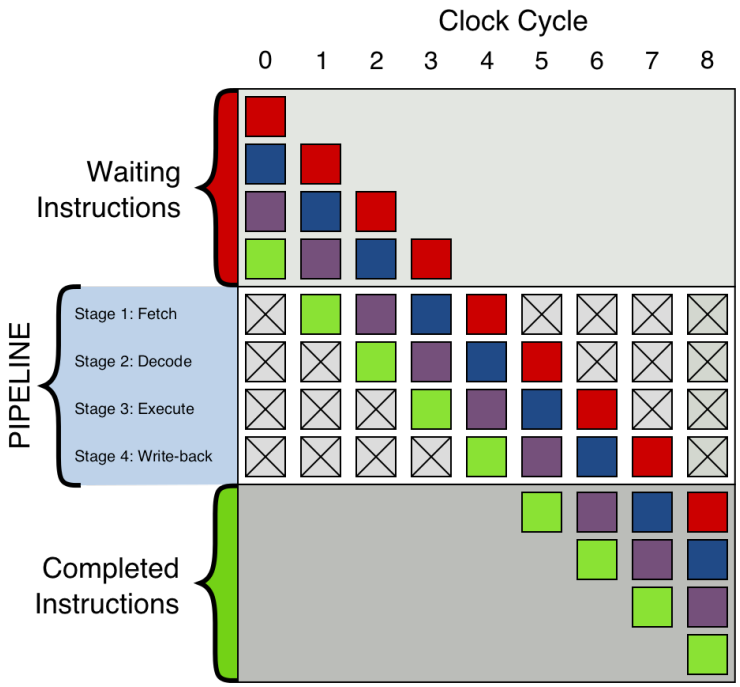

Figure 2: Schematic view of a pipelined execution[1]

Both designs are still present in modern microprocessors. They imply an important degree of hardware complexity in order to avoid as much as possible instructions dependency. Moreover, the parallelism availability of a sequentially designed software is low, and instructions dependency avoids efficient pipelines and redundant hardware full usage. An instruction using the result of an other one can not load the corresponding data until the first instruction has written it. Microprocessors reorder instructions to maximize throughput : program instructions are executed *out-of-order*.

This kind of parallelism is not in the scope of OS scheduling as it is a hardware mechanism impossible to drive by machine users.

[1]image from commons.wikimedia.org

2.3 Thread Level Parallelism

2.3.1 Multithreading

Users of multitasking operating systems can explicitly make the machine handle different applications simultaneously through the notion of *process* and *thread of execution* (simply called *thread*). A process can be seen as a box into which a program evolves once it is loaded for execution. Into this box, a thread is a sequence of instructions following a path in a process. Their existence in the OS can be summed up as the storing of the processor state matching their execution stage (program counter, registers) : the *thread context*. Multitasking is performed by alternating thread contexts. Swapping the thread that is running on the processor by an other one is called a *context-switch*. Figure 3 schematically shows two loaded programs and their two corresponding processes. Thread 1 of process 1 is first scheduled. Then a context-switch happens to let a second thread run, in the same process. Finally a second context-switch allow thread 1 of process 2 to run.

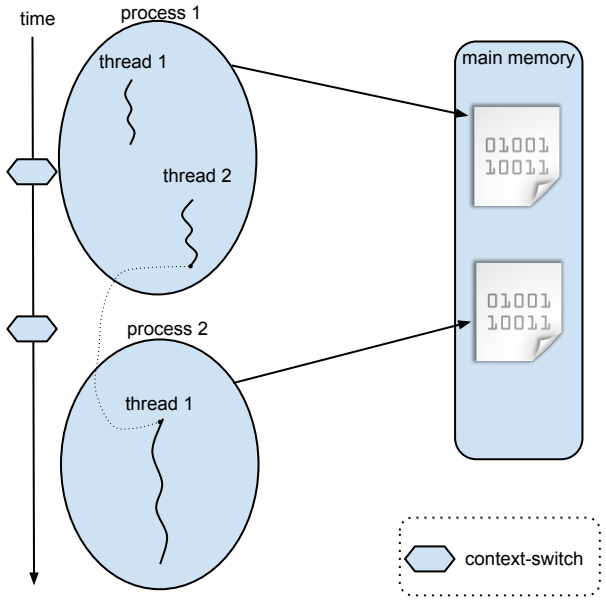

Figure 3: Schematic view of mutual relation of process and thread in an operating system

Context-switch is expensive. It requires processor's cycles during which no effective work is done. If several thread contexts can be handled by the microprocessor (*hardware contexts*), thread switch can be very fast. Processor with this ability are

said *multithreaded*. Each hardware context is seen as a core (*logical core*) by operating systems.

Three kinds of multithreading architectures are available :

- ◇ Coarse Grained Multithreading (*CGMT*). The oldest multithreading design, similar to what is done with a context software switch instead that it takes much less cycles (by orders of magnitude).
- ◇ Fine Grained Multithreading (*FGMT*) or interleaved multithreading. It aims to switch between thread contexts every cycle with no switch delay.
- ◇ Simultaneous Multithreading (*SMT*). More recent works [9, 3] adapted superscalar architectures to the handle of thread context at hardware level, allowing any thread handled by the processor to use any stage of the pipeline at any cycle. At any time, any subpart of the processor in charge of a stage can be used by any thread which needs it.

Simultaneous Multithreading architectures allows parallelism each time two threads that share the processor (*co-runners*) use different resources. It has been implemented by Intel under the commercial name *Hyper Threading Technology* (*HTT* or *HT*) and is now widely available in desktops/laptop PC and servers using Intel architectures (most of them). As a result, this is the first architecture allowing task parallelism our study will consider.

2.3.2 Multicore

During decades, the miniaturization of transistors has enabled their increasing number per chip quite easily, allowing the design of complex architectures implied by instruction level parallelism. Transistor miniaturization and their density on the same chip are both blocked by physic limitations (*power gate*, heat dissipation problems...). As a consequence, the clock rate of processors is difficult to improve. A more powerful serial approach being not possible yet, developing hardware parallelism seems to be the only way to go on improving performances. Multiprocessors architectures, beeing used for long in the high performance computing domain are not suitable for everyday uses, on desktop and servers, as the number of processors that can cohabit on the same board is limited by space and the cost than can greatly raise considering the price of a processor and the fact that such architectures use several ones by definition. An affordable, scalable true parallel architecture had to

appear to let common devices to enter the parallel era. That's what *multi-core* microprocessors, or *chip multiprocessors* (*CMPs*) have brought. The multiplication of independent execution units, i.e. *cores*, inside the same processor packaging (*chip*) allows parallelism with a single processor. Unlike multithreading architectures, cores does not share any execution resources (registers, arithmetic and logic units...), and contrary to multiprocessing cores can share resources more easily than separated processors thanks to the proximity. For instance, cores inside the same chip can share memory caches. This particular point will be developed in next section. Multi-core architectures, called simply *multicore* in the rest of the report, have bridged the gap between the two pre existing kinds of hardware parallelism.

Multicore adds the central notion of core to the existing and commonly used terms of processor, *Central Processing Unit* (aka *CPU*), processing unit, execution resources. Strictly speaking, multicore have removed the notion of CPU. There are several processing units, i.e.*cores*, which can enables simultaneous usage of their *execution resources* in the case of SMT. A *processor* refers to the whole architecture integrated on a *processor chip* (the integrated circuit on which the cores and every processor component are fixed).

2.4 Memory architecture

2.4.1 Hierarchical memory and caches

Memory is where data and instructions of running programs resides. Memory technologies speed has improved slower than processors. As a consequence, the communication with the main memory potentially limits the speed of computation. This is called the *memory wall*. In order to lower its impact, a different memory banks are integrated inside computer architectures, at different distance from processors. The closest are the *registers*. They are inherent to processor architectures and are used to store the data involved in current computations. Then comes several levels of memory caches, usually 3. The first one, called *cache L1*, is inside the core. It can be divided in two parts, one for data the other for instruction. The cache L2 can be shared by several cores or can resides inside each core. A possible cache L3 is shared by all the cores within a same processor. The last cache is often refers as *LLC* (last level cache). The next level is main memory. The farer from core a memory bank is, the slower and the bigger it is. This is called a *memory hierarchy*.

2.4.2 Uniform and non-uniform memory access

In multiprocessing architectures, processors can access the main memory following two patterns :

- ◇ *Uniform Memory Access* (*UMA*) or *Shared Memory Processing*. Memory can be accessed in the same way by every processor. This kind of architecture is called *symmetric multiprocessor* (*SMP*) or *centralized shared-memory multiprocessor*. Interconnection resources that are involved to access main memory (bus, crossbar switch) are shared among all processors, limiting this architecture to a restricted number of processors (eight is a frequently given scale).

- ◇ *Distributed Shared Memory* (*DSM*) or *Non-Uniform Memory Access* (*NUMA*). In order to lower contention on a single bus shared by every processors, memory is partitioned and each partition is dedicated to a subset of processors. Processors sharing a memory bank are said belonging to a same *NUMA domain*. The term *Asymmetric Multiprocessing* (*AMP*) is also used. The time to access memory bank of a domain from another domain depends of the number of intermediary cores between them (number of *hop*).

2.5 Asymmetric architectures

Most designs targeting desktops, laptops and servers explore parallelism through collaboration of identical cores. These are *homogeneous* architectures. However, *heterogeneous* (also said *asymmetric*) architectures may also be considered. Two levels of heterogeneity are possible, depending on whether the cores implements the same instruction sets architecture (*ISA*).

- ◇ *Same ISA*. This level of heterogeneity is transparent to users. However, cores may have different performances or hardware features. The same hardware complexity is not necessary for the optimal execution of every type of program. Thus, systems which are intended to run programs with complementary needs of hardware complexity could fit such architecture.

- ◇ *Different ISA*. Different ISA means deeper changes : programs are usually compiled to a single instruction set and thus can not be run on every cores. The IBM Cell processor [10] embedded in the PlayStation 3 of Sony is an example of this type of architecture.

Heterogeneous architectures could be popularized in the next few years, to reduce power consumption of processors while improving performances [11]. They imply

task scheduler to take into account the characteristics of cores to assign each task on the more suitable one to answer.

3 Operating systems tasks scheduling for parallel architectures

This section aims to categorize and synthesise scheduling works, in the context of general purpose operating systems.

3.1 Jargon

- ◇ In operating systems, a running program, existing through the notion of *process*, can be potentially executed by many concurrent cores. A *thread* is the basic unit of core utilization [12, 6]. At a given time, any thread is executed by a single core. A *task* is the basic scheduling entity. Depending on the scheduling model, a task can be equivalent either to process or thread. Scheduling models considering multiprocessor tasks for instance, consider tasks that can be run simultaneously by many processors [13]. In this case the considered tasks correspond to processes.

- ◇ A *scheduling policy* is a scheduling algorithm, in other terms a strategy to share the use of one or several resources between tasks ;

- ◇ A *non-preemptive* scheduler waits for the running task to finish its work before selecting an other. Conversely, a *preemptive* scheduler can choose to interrupt a running tasks at any time in favour on a more priority one ;

- ◇ In a preemptive operating system, the *timeslice* or *quantum* is the the period of time a task is allowed to run without being preempted ;

- ◇ Real-time systems have to process each treatment with strict timing constraints under penalty of system crash. Soft real-time systems can tolerate a degree of lateness. A real-time task typically has a worst case execution time *WCET* indicating the time given to the system to process it ;

- ◇ A *resource contention* is a conflict over access to a shared resource.

3.2 Classical approaches

Scheduling for monocore systems consists in sharing the time between the different tasks of the system. Multi-core architectures grants it with a new responsibility : deciding on which processing units each task runs, in other term *space sharing*. Space sharing strategies had early been practically tackled considering that the global load has to be equally balanced between available processors (*load balancing*) [14, 15] and that the profitability of hardware caches can be exploited by scheduling enforcement of a task on a processor on which it has already run short before (*cache affinity*) [16, 17]. These are the two space-sharing policies which are implemented in Linux. Their implementation does not imply complex structures. The scheduler needs to know the burden of each processing units to proceed to balancing, and decides the average time period during which a task can be considered to have affinity with a processing unit. These models simplify the vision of multi-core architecture and only take into consideration independent processing units that does not interfere in any way one another.

3.3 Contention-aware scheduling

Scheduling a thread over impacts resources implicitly used by it. These resources can be shared with other tasks in the case of multiprocessor, multicore and simultaneous multithreading architectures. This sharing influences the number of instruction per cycle (IPC) a thread can achieve in its timeslice. For instance, a last level cache that has been polluted by a co-runner enforces a costly communication with the main memory. Thus, fairness can not be ensured with the only control of tasks timeslices. The rate of progress a task achieves during a timeslice is not directly correlated with time [18]. Given a set of tasks and a set of processors, a mapping that generate less contention than the other exists. The problem of finding the optimal task to processor mapping can be reduced to the bag packing problem. It is a NP-complete problem [19, 20] : no known algorithm can find a solution in polynomial time on actual machines. Heuristics have been proposed in order to fight contention at different architectural level. Two approaches are globally used, combined or not : *reactive* scheduling, where contention is monitored thanks to performance monitoring units (PMU), and *proactive* scheduling based on performance prediction [21, 22]. As it has been done few years before for multiprocessing architecture [23], the presented articles below study possible scheduling for simultaneous and multicore.

3.3.1 Simultaneous multithreading

Simultaneous multithreading gives the opportunity to multiple threads to run together in order to maximize the chances to use the full potential of a core at each cycle, and, as a result, to speedup their overall execution time compared to their sequential execution. This speedup is only possible if co-scheduled threads have a complementary use of the execution resources they share.

- ◇ Snavely and Tullsen approach [24] is based on a sample phase in order to determine the instruction-per-cycle (IPC) of each task when running alone on a simultaneous multithreading core. The co-scheduling of tasks on the same SMT core can then be measured against the sampled results, and the benefit of cooperation be evaluated. The duration of the co-scheduling phase before launching a new sampling phase depends on how often the set of tasks change. Mutual dependency of co-runners is metaphorically defined as *symbiosis*]. Depending of the throughput impact of the cooperation, the symbiosis is said negative or positive.

- ◇ In 2002, Snavely and Tullsen extended their work on symbiotic scheduling the study of how two tasks of opposed priority co-run, unlike the traditional priority operating systems approach resulting in scheduling of tasks of same priority, which is demonstrated to be less effective.

Each parallelism level imply contention on specific resources. Simultaneous multithreading imply contention on executions units shared by tasks that run on the same core. Multi-core architectures raise contention problems at other levels.

3.3.2 Multicore

Unlike simultaneous multithreading, multicore provides a total thread parallelism : thread that run on two different cores dos not share any execution resources. However, they may share their last level cache (L2 or L3), the memory controllers in charge of dispatching the requests to the main memory, and pre-fetchers [25]. These hardware managed resource are thread-unaware, and thus does not schedule requests in a fair way. As a result, performances may vary as certain thread-to-core mapping will result in less contention than others. A deep transformation into processor architectures to better suit parallelism would ultimately be the solution to fight contention. It is however a long-term task that behoves hardware designers. Thus, hardware contention has been tackled with transformation at the software. Most of the works in this domain has been surveyed last year in [25].

Research works that focus on scheduling mainly tackle cache contention. Threads competition upon caches avoids their optimal use and leads to unpredictable performances [26, 27, 28, 29]. Approaches similar to the works of Snavelly and Tullsen on SMT have been proposed by [30, 31, 32]. Based on the fact that contention can be lowered by combining theads with complementary resource usage, these studies applies reactive scheduling to enforce such combination.

Zhuravlev et al. [31] show that memory bus and memory controllers can also have equivalent impacts on performance. The articles claims that scheduler should consider the three of them to fight contention efficiently.

3.4 Real-time scheduling

Real-time scheduling adds deadline constraint to the classical task scheduling problem. Of course, general purpose operating systems are not designed to answer to hard real-time needs, but the ubiquity of multimedia contents makes soft-real time scheduling approach to multimedia applications. Three models are enlightened on the criteria of the tasks arrival pattern : periodic (regular occurrence with a known period), aperiodic (period changes randomly) and sporadic. The resource usage of a non-periodic tasks is unpredictable and as such cannot be scheduled with any deadline guarentee in a real sytem. The sporadic model answers the problem by adding a *minimum interarrival time* (*MIT*) property to aperiodic tasks. In this way, the system can assume a minimum resource usage.

Anderson and Calandrino (2006) [33] early studied cache-aware real-time scheduling on multicore architectures. In a related paper of the same year, Calandrino et al. [34] explored the two categories of space sharing approaches in the context of real-time : partitioned scheduling, where task are statically assigned to core once and does not migrate. *dynamic* partitioning where tasks assignment to core can change at runtime. Three static partitioning policies implementing variants of *Earliest Deadline First* (*EDF*) are compared to two pfair scheduling policies. Results demonstrates that dynamic partitioning can be a viable alternative to static partitioning.

3.5 Asymmetric architectures

Asymmetric multicores architectures could invades desktop and servers in a near future. They change task scheduling models as resources characteristics are heterogenous : the type of workload must be taken into account as a type of core can be more

appropriate than an other one. Taking its roots in HPC research, a vast literature is available. Ghiasi et al. [35] is an example of one of the early works (around 2005) examining scheduling heterogeneous multi-cores architectures. The article considers the implications of heterogeneous cores, with differing voltages and frequencies on task scheduling. An online application profiling is done in order to anticipate their needs and assign them to the core that fits the best, e.g. a memory-bound application does not need a high speed core. significant power savings. The article exposes significant power saving compared to traditional homogenous approaches.

3.6 Hierarchical scheduler

Monolithic operating systems incorporates a one-size-fits all tasks scheduler which gives as only manipulable task property a priority index. This approach does not allow different kind of workload to be served in adequation with their needs, victim of this genericity. Scheduling algorithm is achieved with a optimality goal in mind given a set of resources and a type of task, upon which a resulting schedule can be evaluated. Theory calls it *criteria*. Each kind of workload has a different usage of the available processing units, and different needs that the meeting represent their success criteria. It is application specific. Given these observations, Goyal et al. (1996) [36] proposed a hierarchical partitioning of the processor bandwidth (i.e. the time) between different scheduling strategies (*scheduling classes*) : *hierarchical scheduling*. Each thread must belong to exactly one class of scheduling. The logic of bandwidth partition can bee seen as a tree where each child-node defines what percentage of its father-node bandwidth it takes. Each leaf is a scheduling strategy, and each no-leaf node is a class aggregation than divides bandwith between its child according to the proportional amount they defined. John Regehr thesis works (1998-2001) addressed the development of scheduling hierarchy on mono-core and multi-core architectures [37, 38].

3.7 Bossa

Actual operating systems implements task scheduler as a part of the entire system, making it difficult for users to change its behaviour. Separation of the concrete algorithms that get work done (the *mechanism*) and the decision that trigger the work (the *policy*) is a well-known pattern. Bossa (2002) [39, 40] applies it to the development of task scheduler. Bossa allows the scheduling policy to be written with a language specifically designed to express time sharing on a monoprocessor system, a Domain Specific Language (*DSL*). A compilation time (DSL to C) validate of its

correctness : . The mechanism, integrated in the kernel, enable the composability of policies and the bandwidth partition of the processor among them, through hierarchical scheduling. *Non-leaf* scheduler, as they were defined in [36] are here called *Virtual Scheduler* : scheduler of schedulers. As such, they are definable with the Bossa langage, allowing the definition of alternative partitioning policies to the proportional one defined by Goyal et al. When a relevant event occurs, the mechanism triggers the corresponding policies handlers, and if needed, delegates the scheduling decisions to the policy selected by the hierarchy walk through, according to virtual schedulers policies. An extension of Bossa, *Bossa Nova* [41], introduced modularity to address the need to factorize similarities of policies belonging to a same family. Few years after Bossa, Roper et al. suggested the use of a DSL to write application-specific schedulers in embedded systems : CATAPULT [42]. Their aim was to separate the scheduling logic in a replaceable component, to allow better static analysis and multiple threading libraries to be targeted in a research of performance gain. The DSL, not being designed for safety, is less restrictive than Bossa.

3.8 Scheduler design

4 Conclusion

Whatever architecture is chosen, parallelism in a shared address space involves resource sharing : independent processing units have to share communication roads to main memory, and have to compete for each level of the memory hierarchy. Two corunners of a simultaneous multithreading processor compete for executing resources, cores in the same chip and concurrent processors share the communication mechanisms to access memory. Whatever the architecture is, the understanding of how resources are shared at each parallelism level and in what conditions parallelism is optimal is crucial to design a scheduler that leads to good performances.

Two policies cohabit in task schedulers : time sharing and space sharing. The expression of time-sharing has been explored in Bossa. How to express space-sharing is an ongoing work.

References

[1] Infra-jvm project official webpage. www.agence-nationale-recherche.fr/programmes-de-recherche/[...], 2009.

[2] LaBRI. Progress research team webpage. www.labri.fr/index.php?n=LSR.LSR, 2013.

[3] Dean Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *In 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[4] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ASPLOS VII, page 2–11. ACM, 1996.

[5] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, page 36–43. ACM, 2000.

[6] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 3 edition, December 2007.

[7] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901 – 1909, December 1966.

[8] John Von Neumann. *First Draft of a Report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania, 1945.

[9] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. *SIGARCH Comput. Archit. News*, 20(2):136–145, April 1992.

[10] Ibm cell project. researcher.watson.ibm.com/... [Online; last access 7-april-2013].

[11] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32 – 38, November 2005.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 edition, September 2006.

[13] Maciej Drozdowski. Scheduling multiprocessor tasks—an overview. *European Journal of Operational Research*, 94(2):215–230, 1996.

[14] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 23(5):159–166, November 1989.

[15] Steve J. Chapin. Distributed and multiprocessor scheduling. *ACM Computing Surveys*, 28, 1996.

[16] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, page 26–40, New York, NY, USA, 1991. ACM.

[17] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors: a summary. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '93, page 272–274, New York, NY, USA, 1993. ACM.

[18] Ravi Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, page 257–266, New York, NY, USA, 2004. ACM.

[19] J. D. Ullman. NP-complete scheduling problems. *J. Comput. Syst. Sci.*, 10(3):384–393, June 1975.

[20] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, page 220–229, New York, NY, USA, 2008. ACM.

[21] Yunlian Jiang and Xipeng Shen. Exploration of the influence of program inputs on CMP co-scheduling. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, page 263–273, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proceedings of the 15th international conference on Parallel*

architectures and compilation techniques, PACT '06, page 13–22, New York, NY, USA, 2006. ACM.

[23] Frank Bellosa and Martin Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):113 – 121, 1996.

[24] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGARCH Comput. Archit. News*, 28(5):234–244, November 2000.

[25] Sergey Zhuravlev, Juan Carlos Saez, Serguey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1), November 2012.

[26] G.E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Eighth International Symposium on High-Performance Computer Architecture, 2002. Proceedings*, pages 117 – 128, February 2002.

[27] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, page 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

[28] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, page 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Alexandra Fedorova, Margo Seltzer, Christoper Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, page 26, Berkeley, CA, USA, 2005. USENIX Association.

[30] R. Knauerhase, P. Brett, B. Hohlt, Tong Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54 –66, June 2008.

[31] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, page 129–142, New York, NY, USA, 2010. ACM.

[32] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, page 153–166. ACM, 2010.

[33] James H. Anderson, John M. Cal, and Umamaheswari C. Devi. Real-time scheduling on multicore platforms. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 179–190. Chapman Hall/CRC, Boca, 2006.

[34] LITMUS^RT : A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, pages 111–126, Washington, DC, USA, 2006. IEEE Computer Society.

[35] Soraya Ghiasi, Tom Keller, and Freeman Rawson. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd conference on Computing frontiers*, CF '05, pages 199–210. ACM, 2005.

[36] Pawan Goyal, Xingang Guo, and Harrick M. Vin. Readings in multimedia computing and networking. pages 491–505. Morgan Kaufmann Publishers Inc., 2001.

[37] J. Regehr. *Hierarchical Loadable Schedulers*. PhD thesis, Apr, 1999.

[38] John Regehr. Hls: A framework for composing soft real-time schedulers. In *In Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001*, page 3–14. IEEE, 2001.

[39] Luciano Porto Barreto and Gilles Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, page 19–31, Paris, France, March 2002.

[40] Gilles Muller, Julia L. Lawall, and Herve Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In

Proceedings of the Ninth IEEE International Symposium on High-Assurance Systems Engineering, HASE '05, page 56–65, Washington, DC, USA, 2005. IEEE Computer Society.

[41] Julia L. Lawall, Hervé Duchesne, Gilles Muller, and Anne-Francoise Le Meur. Bossa nova : Introducing modularity into the bossa domain-specific language. In *Lecture notes in computer science*, pages 78–93. Springer, 2005.

[42] Matthew D. Roper and Ronald A. Olsson. Developing embedded multithreaded applications with CATAPULTS, a domain-specific language for generating thread schedulers. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '05, page 295–303, New York, NY, USA, 2005. ACM.