# Elastic Manycores[⋆]

## How to bring the OS back into the scheduling game?

Marcus Völp and Michael Roitzsch

Technische Universität Dresden, Germany
{voelp,mroi}@os.inf.tu-dresden.de

**Abstract.** By introducing asynchronous lambdas, many programming languages have leaped ahead in the race for programmable manycore systems, leaving the operating system and its scheduler behind. Instead of hiding application-inherent parallelism behind pools of threads with opaque behavior, asynchronous lambdas allow programmers to explicitly state which parts of a program can be executed in parallel and when this form of parallelism is available. Introducing stretch as a universal performance metric and externalizing part of the lambda-provided knowledge not only to the runtime but also to the operating system scheduler, this paper tries to lay the foundation for OS scheduling to catch up on the road towards heterogeneous *elastic manycore systems*.

## 1 Introduction

With the gear shift of Moore's Law from frequency scaling to expanding concurrency, we have seen programmability and coordination to become increasingly challenging: To draw optimal performance and energy efficiency out of the given transistor budget, we can no longer rely on homogeneous manycore systems, where all cores are clones of a single kind. Instead, we find our common application mixes much better supported by increasingly heterogeneous systems [17,18]. Starting with instruction extensions, which are no longer available on all cores, we find systems today, where the resources for these extensions are shared, where multiple types of cores co-exist simultaneously and where special purpose and general purpose accelerators (such as H.264 [19] and AES accelerators or GPGPUs) reside next to large and small general purpose cores.

Applications that want to exploit these systems must provide binary code for the best suited cores or accelerators and be able to extract the parallelism that is available in their algorithms. At the same time, the code must have alternative implementations ready to adjust to inferior cores and to run at reduced parallelism if the premium resources are currently required for more urgent needs.

Languages and runtimes such as X10 [7], OpenCL [14], Grand Central Dispatch (GCD) [1], C++ [12], Lithe [20] and the Parallel Pattern Library [13] have

[⋆] This work is in part funded by the DFG through the cluster of excellence "Center for Advancing Electronics Dresden" and by the EU and the state of Saxony through the ESF young researcher group IMData.

already made important steps in this direction: To extract the available parallelism, they encourage a programming concept we call *asynchronous lambdas*. Programmers encapsulate independent work into lambda functions and submit them to queues for asynchronous execution. OpenCL already provides limited support for generating alternative binaries for different instruction sets from the same source code. Finally, static and dynamic analysis of the source code allows us to associate lambdas with meta data such as a lambda dependency graph, heuristics on the use of FPU or accelerator instructions, or information on how lambdas access the data they process. However, to be useful for short term scheduling decisions we need to find a compact representation for all the above information.

This paper investigates elasticity and stretch as guiding metrics, which we believe can serve as such a compact representation. Rather than describing each lambda individually, we summarize alternatives and homogeneously parallel lambdas to a single elastic lambda and stretch this lambda both in time and in the number of cores when we schedule it in our heterogeneous system.

After taking a closer look at asynchronous lambdas in Section 2, we shall see in Section 3 how stretch characterizes the above choices. As guiding examples, we shall use a JPEG blur filter and a database join operator, which we introduce in the following section.

## 2 Asynchronous Lambdas

The programming paradigm of asynchronous lambdas relieves the developer from explicitly managing parallelism with threads. Instead, the code expresses latent parallelism: The developer states explicitly which code pieces can potentially run in parallel. The runtime layer can decide dynamically, which pieces do run concurrently. It thus translates the latent parallelism to actual parallelism.

Asynchronous lambdas combine a programming pattern based on asynchronous invocation and a lambda programming language feature to deliver unique properties.

**Asynchronous invocation** originates from event-based programming [16]. High-throughput network servers experience scalability bottlenecks when assigning each incoming request to a dedicated thread. Performance improves when organizing the server such that queues collect pieces of work and execute them asynchronously [22]. Queuing work instead of explicitly spawning a thread allows the runtime layer to control the number of threads [6], employing strategies like thread pooling.

**Lambdas** automatically transfer state by capturing variables from the enclosing lexical scope. This relieves the programmer from manually collecting contextual data. Lambdas also preserve the logical locality of the code, because they appear inline and not as separate functions. The code looks like a serial flow of instructions and is easier to read than code written for threaded or callback parallelism.

We call the combination of both concepts asynchronous lambdas, but different terms have been coined within different language communities. Table 1 presents a selection of popular implementations. Our ideas are most closely related to GCD.

**Table 1.** Complete or Partial Implementations of Asynchronous Lambdas

| Implementation | Term |
|---|---|
| Apple Grand Central Dispatch (GCD) | Block |
| Microsoft Parallel Patterns Library | Lambda |
| Intel Threading Building Blocks | Task |
| C++11 | Lambda |
| Google Go | Goroutine |
| X10 | Activity |
| Lithe | Block |

### 2.1 Dispatch Queues

Dispatch queues allow the programmer to express parallelism by submitting lambdas for asynchronous execution. Such queues come in two flavors: serial and parallel queues. Lambdas submitted to a serial queue execute one after the other, never running two of them at the same time. This guarantee not only implies protection for critical sections [3], it also allows developers to reason about the order of lambda execution.

Lambdas on a parallel queue execute concurrently and may complete in any order. Parallel queues also support barrier lambdas, which represent serial islands within otherwise parallel work. Barrier lambdas always execute exclusively on their queue and do not start until all previously submitted lambdas have completed.

Because execution on queues occurs asynchronously, lambdas are submitted to the queues by the program before they are executed. Dispatch queues therefore *expose dynamic program structure* to the runtime library but not yet to the OS scheduler. Instead, implementations like GCD inspect the queues and decide how many threads to employ. In fact, GCD on OS X actually interacts with the kernel in a scheduler activation [2] style.

But we think there is a lot more information to harvest. The queues essentially provide a limited look into the future execution of the application. Serial queues and barriers express dependencies between the lambdas, which allows to derive a job graph for the scheduler. Although no asynchronous lambda runtime implements this today, we even imagine alternative binary versions of the same source-level lambda to exploit specialized units of heterogeneous manycores. An example will illustrate:

### 2.2 Example 1: Blurring a JPEG Image

Imagine a little app that applies a blur filter to a compressed JPEG image. The user selects the file to manipulate and the program filters it in the background to keep the app responsive. The code looks like this (in GCD's block syntax):

```
dispatch_barrier_async(queue, ^{
      jpeg_decode(file, image);
});
size_t pixels = image->w * image->h;
dispatch_apply(pixels, queue, ^(size_t i){
      blur_kernel(image, i);
});
dispatch_barrier_async(queue, ^{
      jpeg_encode(image, file);
});
```

**Fig. 1.** Dispatch Code of Blur Filter

First, the JPEG file is decoded asynchronously. The barrier prevents the subsequent filtering to start before decoding is complete. Then, the blur compute kernel is dispatched once for every pixel of the image. A final barrier lambda encodes the in-memory image to a JPEG representation again. This code runs without performing any work synchronously. All lambdas execute in the background. Therefore, the dispatch queue exposes a complete picture of the pending work to the runtime, including the dependency fan-out from the decode step and the dependency fan-in on the encode step. Fig. 2 shows these dependencies. Lambdas are denoted as blocks, barriers as horizontal bars. A number of execution options arise:

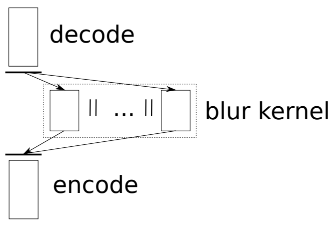

**Fig. 2.** Dispatched Lambdas and the Resulting Dependencies

– The blur kernel can execute in parallel, because it must run for every pixel. The runtime may decide to exploit all, some, or none of this parallelism, taking the cost of data transfer between cores into account.
– Even if available cores share the same ISA, there may be variations between them. Imagine a mixture between complex out-of-order cores and smaller in-order cores. JPEG decoding and encoding may run faster on an out-of-order core, because the single-threaded JPEG code benefits from hidden memory latencies. But those complex cores may be scarce and could be occupied by other applications.
– Within a heterogeneous manycore, some of the cores may support special instructions like a fused multiply-add, which the blur kernel can exploit. This requires alternative binaries for the blur kernel so that the runtime can

pick the one that best matches the available instructions. Such alternative binaries may even be generated at runtime [9].

– Offloading massively parallel work to a GPU can also be beneficial [21]. But in addition to a different ISA and programming environment, we also have to orchestrate explicit data transfer to and from such accelerators and account for its costs.

### 2.3 Example 2: Database Join Operation

Similar decisions come up when executing a large-scale database join, an operation that processes two data sets and outputs all pairs that match a given predicate. The placement of execution relative to data influences processing times considerably. Additionally, data transfer characteristics may change depending on the form of the data. For sorted data or when sorting is cheap, a parallel merge join algorithm [5] can be employed. Otherwise, a partitioned hash join [15] is called for, which pre-selects data into cache-friendly partitions to accelerate the core's evaluation of the join predicate on all possible pairs. This inter-core data transfer adds to the execution cost, so the physical core layout should match the transfer pathways to reduce traffic.

We believe decisions as outlined in the two examples must be made by an operating system scheduler. Only the scheduler can aggregate knowledge from multiple applications and mediate between their competing needs. But currently, much of the knowledge needed to make those decisions is buried within applications. We want to analyze the program structure available in the dispatch queues and export a suitable compact representation to the scheduling layer. The next section describes this representation.

## 3 Elasticity and Stretch

Clearly, exporting all dispatched lambdas together with their dependencies and metadata gives the operating system an accurate and detailed view on application needs. However, at the same time, pushing all this information down to the operating-system scheduler would increase scheduling complexity beyond acceptable bounds. Our approach to counteract this complexity is to aggregate groups of similar lambdas into a single elastic job and to stretch this job both vertically along the timeline and horizontally in terms of number and types of used cores.

In addition to the scheduling options, execution constraints are passed to the scheduler as deadlines, so it can separate desirable from unwanted schedules. Timing constraints, e.g., in the form of application-provided end-to-end deadlines, also allow the scheduler to explore non-work-conserving scheduling opportunities. The real-time community is also beginning to explore runtimes for task models with intra-task parallelism [10].

In the following, we detail the aggregation of lambdas that encapsulate alternative implementations of the same functionality. We also discuss homogeneously parallel lambdas and the factors that influence their stretch.

## 3.1 Alternatives

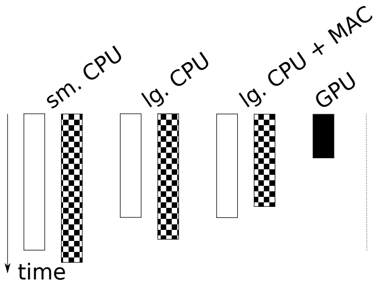

**Fig. 3.** Aggregation of Lambdas, which Encapsulate Alternative Implementations

Fig. 3 shows the time required to execute three alternative implementations of the blur filter from Example 1 on different CPUs. We assume that the small in-order core and the two types of large out-of-order cores share the same ISA. Executing the general purpose CPU alternative (white) on any of the out-of-order cores may improve performance due to the large cores' better latency hiding abilities. However, due to their size and power hunger, we have to expect only few large cores in future manycore systems. By emulating the fused multiply-add instruction, we may also execute the second alternative (checkered) on all cores. However, its performance improvement is likely lost in the emulation overheads. As usual for this type of code, we expect the highest performance improvements from running the GPU kernel of the blur filter (black). However, keep in mind that we need to transfer the data to and from this accelerator. We shall return to this point in Section 3.3.

How long does it take to blur a single pixel in the worst or in the best case, or with a given probability $p$? Clearly, giving answers to such questions requires knowledge about the minimum, maximum or $p$-percentile of the execution time distribution. However, if we assume to know these values for all alternatives and types of CPUs, we can aggregate this information in a type-agnostic way by taking the smallest of these values as the base (worst/best/$p$-percentile) execution time $c$ of the lambda and by taking the distance to the largest of these values as a metric for how far this elastic lambda can be stretched. More precisely, we consider both constant influences $b$ (such as just-in-time compilation of the code for a respective target) and influences that grow linearly with the base execution time. That is, the stretch $s$ of a lambda is given by:

$$s = ac + b \tag{1}$$

where $a$ is a linear factor. The execution time of the lambda may therefore vary between $c$ and $c + s = (a + 1)c + b$. It is easy to see that minimizing this lambda's stretch minimizes its execution time. However, it remains to be seen whether we can extend this desired property also to the entire application, to parts of the system or even to the elastic manycore as a whole. What we do get however, is a guideline on how much effort we have to spend on finding the optimal placement for this application. As long as the application meets its deadlines while all of its lambdas are maximally stretched, we are free to allocate the lambdas to arbitrary types of cores. Vice versa, losing the freedom to stretch a lambda beyond a certain point means we can only run the lambda on cores where the corresponding alternative is able to complete in time.

Our second example not only entails alternative implementations of the same algorithm but also implementations of alternative algorithms providing the same functionality. For example, Graefe et al. [11] make the point that database management systems (DBMS) should support both hash- and sort-based join operators. In our setting, both variants appear as alternatives. More precisely we see the DBMS guided dispatch of a number of lambdas to sort the data in parallel followed by a similar block of parallel activity for merging the results according to the join criterion. For the alternative, similar blocks of independent lambdas are dispatched for the data partitioning and pre-selection step and for the actual join within these partitions.

### 3.2 Homogeneous Parallelism

The second, horizontal dimension along which we stretch elastic jobs spans spatially across the cores of the system. This time however, all aggregated lambdas must be executed to complete the encapsulated functionality. To simplify the discussion, we focus on homogeneous parallelism. That is, we are going to aggregate lambdas with negligible variations in their individual execution times and with approximately equal data transfer times.

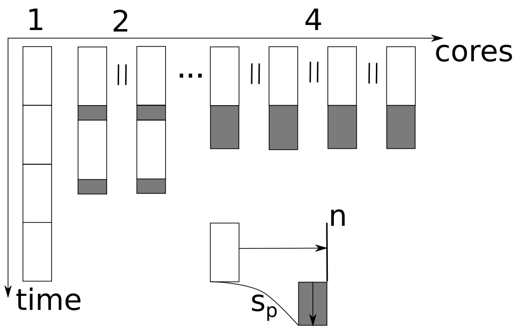

**Fig. 4.** Aggregation of Homogeneous Parallelism

Fig. 4 illustrates our approach. Parallel loop constructs such as the `dispatch_apply` function in Fig. 1 Line 5, dispatch a number of independent lambdas with the options to execute them sequentially, in parallel, or as parallel groups of sequentially executed lambdas. Synchronization,[1] cache traffic when accessing shared objects and the overhead of the parallelization constructs may prolong the individual lambdas although the loop in its entirety is likely to complete faster if we spread it over additional cores. We record this vertical stretch of the lambda execution times in

[1] Although locks may still be used inside lambdas, programming concepts such as GCD discourage their use. Instead, critical sections should be dispatched as separate lambdas. We will look at synchronization overhead to accommodate for these legacy uses and to extend our approach to more advanced synchronization patterns such as transactional memory.

the function $s_p$, which maps the assigned cores to the parallelization overhead. For the horizontal stretch it is more convenient to regard sequential execution as maximally stretched and the complete parallel case as the base. This way we preserve the property that losing the possibility to stretch lambdas beyond a certain point limits our freedom to allocate lambdas flexibly on fewer cores.

Unlike Collette et al. [8], we do not extend our elastic jobs to also cover the vertical stretch caused by executing lambdas sequentially. If we need the execution time for the construct for $k$ cores, we can simply obtain it as

$$\left\lceil \frac{n}{k} \right\rceil (c + s_p(k)) \tag{2}$$

where $c$ is the base execution time of the elastic job. We believe we can build on the additional flexibility of our model by allowing the number of allocated cores to change over time. Assume the lambda is executed on $k_0$ cores for $c + s_p(k_0)$ and on $k_1$ cores for $c + s_p(k_1)$. Then $n' = n - k_0 - k_1$ lambdas remain and Equation 2 gives us the required execution time if we replace $n$ by $n'$.

## 3.3 Data Transfers

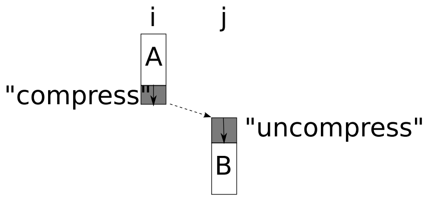

**Fig. 5.** Stretch due to Data Transfers

Data transfers and alternatives are similar in the way they stretch elastic many-cores: the execution time of a lambda may be prolonged depending on how the data is transferred to its core. Overheads may arise from implicit transfers, for example in the form of cache access latencies [4] that cannot be hidden behind the execution of the lambda. However, transfers may also involve explicit data preparation steps. The fundamental differences are that at least two lambdas are involved in determining the stretch and both are affected by its consequences. Fig. 5 illustrates this point. Lambda $A$ executing on core $i$ produces some data that is later on required by lambda $B$ on core $j$. Data may be transferred in uncompressed form. However, one option to speed up this transfer is by compressing the data on core $i$ thereby stretching $A$'s execution time and decompressing it after the transfer on core $j$ with a similar influence on $B$. In a sense, the JPEG en-/decode step of Example 1 can be seen as such a compression/decompression step. In our second example, data partitioning and pre-selection are essential steps for the performance of the partitioned hash join operator.

In Section 3.1, we have distinguished a base execution time and a stretch of this base execution time. This is not possible for data transfers because more than one lambda is involved in the transfer. A minimal stretch of $A$ (e.g., when snooping the data out of core $i$'s private caches) may well impose a significant overhead on $B$ whereas other transfer methods may result in a reduced effort for $B$ at the cost of prolonging $A$'s execution time. For this reason, we attribute data transfers completely as stretch to the lambdas which access this data during their computation.

Given the placement of the data sources, we determine the source and destination stretch by taking into account the minimum and maximum execution times and influences of the available transfer methods and by aggregating these results for all possible destination cores. Again, the more stretch we are able to tolerate on the source cores, the more destination cores and transfer methods remain at the destination side. At the same time, these remaining transfer methods determine the minimum stretch that we have to tolerate at the destination side. The more of this stretch the destination side can accept, the more freedom we have in choosing the destination core and transfer method.

## 4 Conclusions

Catching up with recent improvements in programming languages, this paper argues for elasticity and stretch as guiding principles for scheduling in heterogeneous manycore systems. Although stretch-based scheduling is still in its infancy, we have seen that elastic asynchronous lambdas, which we spread horizontally over available cores and vertically on the timeline, are a suitable representation of application-internal knowledge. Latent parallelism and implementations for alternative instruction sets can be communicated to the scheduler. Together with deadlines as execution constraints, this knowledge strengthens the position of the operating system and allows it to schedule work on the complex hardware of the future.

## References

1. Introducing blocks and grand central dispatch. In: Mac Developer Library. Apple Inc. (August 2010), `https://developer.apple.com/library/mac/#featuredarticles/BlocksGCD/`
2. Anderson, T.E., Bershad, B.N., Lazowska, E.D., Levy, H.M.: Scheduler activations: Effective kernel support for the user-level management of parallelism. ACM Transactions on Computer Systems 10(1), 53–79 (February 1992)
3. Best, M., Mottishaw, S., Mustard, C., Roth, M., Fedorova, A., Brownsword, A.: Synchronization via scheduling — techniques for efficiently managing shared state. In: Programming Language Design and Implementation (2011)
4. Blagodurov, S., Zhuravlev, S., Dashti, M., Fedorova, A.: A case for numa-aware contention management on multicore systems. In: Proceedings of the 2011 USENIX Annual Technical Conference. pp. 1–16. USENIX ATC, USENIX (June 2011)
5. Blasgen, M.W., Eswaran, K.P.: Storage and access in relational data bases. IBM Systems Journal 16(4), 362–377 (1977)
6. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 207–216. PPOPP, ACM, New York, NY, USA (July 1995)

7. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 519–538. OOPSLA, ACM, New York, NY, USA (October 2005)
8. Collette, S., Cucu, L., Goossens, J.: Integrating job parallelism in real-time scheduling theory. Information Processing Letters 106(5), 180–187 (May 2008)
9. DeVuyst, M., Venkat, A., Tullsen, D.M.: Execution migration in a heterogeneous-isa chip multiprocessor. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 261–272. ASPLOS, ACM, New York, NY, USA (March 2012)
10. Ferry, D., Li, J., Mahadevan, M., Agrawal, K., Gill, C., Lu, C.: A real-time scheduling service for parallel tasks. In: Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium. RTAS, IEEE, Los Alamitos, CA, USA (April 2013)
11. Graefe, G., Linville, A., Shapiro, L.D.: Sort versus hash revisited. IEEE Trans. Knowl. Data Eng. 6(6), 934–944 (1994)
12. ISO: ISO/IEC 14882:2011: Programming Languages – C++, 3 edn. (September 2011)
13. Kerr, K.: Visual C++ 2010 and the parallel patterns library. MSDN Magazine (February 2009)
14. Khronos OpenCL Working Group: The OpenCL Specification, 1.2 edn. (November 2012)
15. Kim, C., Sedlar, E., Chhugani, J., Kaldewey, T., Nguyen, A.D., Blas, A.D., Lee, V.W., Satish, N., Dubey, P.: Sort vs. hash revisited: Fast join implementation on modern multicore cpus. In: Int. Conference on Very Large Data Bases (VLDB). Lyon, France (August 2009)
16. Krohn, M., Kohler, E., Kaashoek, M.F.: Events can make sense. In: Proceedings of the 2007 USENIX Annual Technical Conference. USENIX ATC, USENIX, Berkeley, CA, USA (June 2007)
17. Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P.r., Tullsen, D.M.: Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. pp. 81–. MICRO 36, IEEE Computer Society, Washington, DC, USA (2003)
18. Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P., Farkas, K.I.: Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: Proceedings of the 31st annual international symposium on Computer archit ecture. pp. 64–. ISCA '04, IEEE Computer Society, Washington, DC, USA (2004)
19. Kun, Y., Chun, Z., Guoze, D., Jiangxiang, X., Zhihua, W.: A hardware-software co-design for h.264/avg decoder. In: Solid-State Circuits Conference, 2006. ASSCC 2006. IEEE Asian. pp. 119 –122 (nov 2006)
20. Pan, H., Hindman, B., Asanovic, K.: Lithe: Enabling efficient composition of parallel libraries. In: HotPar (August 2009)
21. Rossbach, C.J., Currey, J., Silberstein, M., Ray, B., Witchel, E.: Ptask: Operating system abstractions to manage gpus as compute devices. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles. SOSP, ACM, New York, NY, USA (October 2011)
22. Welsh, M., Culler, D., Brewer, E.: Seda: An architecture for well-conditioned, scalable internet services. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles. pp. 230–243. SOSP, ACM, New York, NY, USA (October 2001)