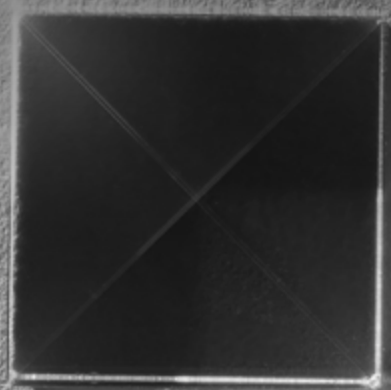
THE CAPCO INSTITUTE
JOURNAL
OF FINANCIAL TRANSFORMATION

ALTERNATIVE RISKS

An implementation framework to guide system design in response to FRTB requirements

OLIVIER COLLARD | CHARLY BECHARA
GILBERT SWINKELS



ALTERNATIVE CAPITAL MARKETS

#49 APRIL 2019

THE CAPCO INSTITUTE

JOURNAL OF FINANCIAL TRANSFORMATION

RECIPIENT OF THE APEX AWARD FOR PUBLICATION EXCELLENCE

CONTENTS

ALTERNATIVE MODELS

ALTERNATIVE RISKS

ALTERNATIVE MARKETS

DEAR READER,

Welcome to edition 49 of the Capco Institute Journal of Financial Transformation.

Disruptive business models are re-writing the rules of our industry, placing continuous pressure on financial institutions to innovate. Fresh thinking is needed to break away from business as usual, to embrace the more rewarding, although more complex alternatives.

This edition of the Journal looks at new digital models across our industry. Industry leaders are reaching beyond digital enablement to focus on new emerging technologies to better serve their clients. Capital markets, for example, are witnessing the introduction of alternative reference rates and sources of funding for companies, including digital exchanges that deal with crypto-assets.

This edition also examines how these alternatives are creating new risks for firms, investors, and regulators, who are looking to improve investor protection, without changing functioning market structures.

I am confident that you will find the latest edition of the Capco Journal to be stimulating and an invaluable source of information and strategic insight. Our contributors are distinguished, world-class thinkers. Every Journal article has been prepared by acknowledged experts in their fields, and focuses on the practical application of these new models in the financial services industry.

As ever, we hope you enjoy the quality of the expertise and opinion on offer, and that it will help you leverage your innovation agenda to differentiate and accelerate growth.

Lance Levy, **Capco CEO**

AN IMPLEMENTATION FRAMEWORK TO GUIDE SYSTEM DESIGN IN RESPONSE TO FRTB REQUIREMENTS

OLIVIER COLLARD | Principal Consultant, Capco
CHARLY BECHARA | Director of Research & Innovation, Tredzone
GILBERT SWINKELS | Partner, Capco

ABSTRACT

The changes that must be made to a bank's infrastructure to implement the "fundamental review of the trading book" (FRTB) standards are transformational. The data and process requirements are such that pricing platforms need a complete overhaul to meet performance and latency goals. This article will present a viable design process, and the supporting framework, to fully leverage today's multi-core environments, be it cloud or otherwise.

1. INTRODUCTION

In order to implement the "fundamental review of the trading book" (FRTB), banks have to deal with requirements imposed by the Internal Model Approach (IMA) and Standard Approach (SA). A majority of banks will opt for an IMA approach, at least for a large part of their trading activities, in order to optimize their capital requirements. But even if banks go for IMA, they will have to compute SA in parallel to compare both.

Complying with FRTB requirements generally requires a significant rework of the front-to-back trading infrastructure to cope with "orders of magnitude" increases in the number of computations, an equally massive increase in volumes of data consumed and produced, and a need to harmonize the use of pricing and risk data and models across a complex process chain.

This article explains how the Reactive software design approach and the supporting Simplx open source framework from Tredzone™ can help address some of these challenges.

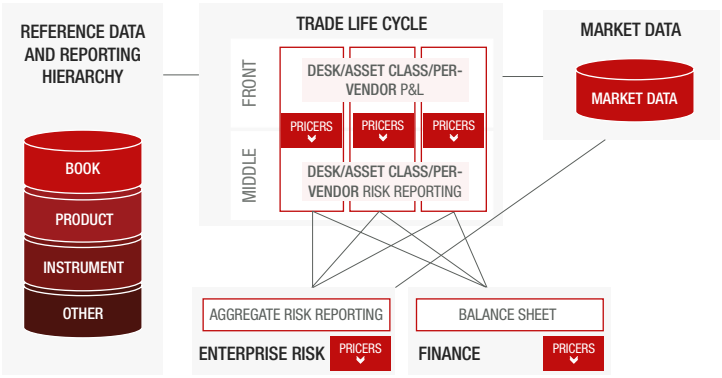
2. FRTB TECHNOLOGY IMPACTS

FRTB will have a number of technological implications for banks, including:

Massive increase in the number of computations
FRTB changes the approach to model risk for banks, based on notions like Expected Shortfall (ES), "Standardized Approach," and the concept of Non-Modellable Risk Factors for capital requirement computations. As a consequence current sensitivities-based optimizations will need to be reconsidered, using full revaluation methods instead. This requirement is expected to result in a tenfold increase in the number of P&L calculations required, further magnified by the ever-increasing need to move to real-time stress testing to provide transparency into capital consumption.

Harmonized processes and forms of governance
FRTB favors a realignment of governance and approaches between the front office and the risk department, leading to a consolidation of front-office risk engines. This trend challenges the existing reliance on trading and

Figure 1: Typical front-to-back trading architecture



risk packages from different vendors, often deployed based on the preferences of individual desks depending on the assets they trade. The P&L and risk attribution will be checked at the trading desk level. Consistency requirements mean that pricing data and libraries need to be streamlined across desks and across aggregated risk reporting stacks.

Data quality requirements and volumes increase consistency in pricing and risk calculations across the firm can only be achieved by ensuring that there is a single source of trade data, that market data, and everything that is calculated from it (like volatility surfaces and curves), have a single and common source, and that reference data used to enrich the trades (e.g., product taxonomies) are unique across the firm.

3. CURRENT ARCHITECTURE AND LIMITATIONS

Organizations often rely on a number of front-office systems, sourced from independent software vendors, with each responsible for a subset of a bank's assets or trading desks. This results in a "siloed" infrastructure, where separate risk reports are built for each platform.

Banks would address this lack of systems interoperability by deploying an additional, enterprise-wide risk layer. While successful at building a cross-asset view of the risk, this approach still relies on each underlying system's specifics about pricing algorithms, shock scenarios, risk factors, and reference data definitions (e.g., using different yield curves).

This often results in hard-to-explain valuation discrepancies among the business, risk, and finance views, with additional costs in computing hardware provisionning.

4. SYSTEM INFRASTRUCTURE REQUIREMENTS

The increase in calculations required by FRTB will likely push many conventional grids over their limits, thereby increasing the need to review systems in a fundamental way.

The requirement for computing power and the shift to a unified pricing framework motivates adoption of technologies and architecture models that (i) leverage highly parallel and resilient hardware grids for horizontal and vertical scale out, possibly spread across private and public clouds; (ii) deploy computing application frameworks that favor data and processing colocation (shared-memory, ideally in-process) for efficient processing of large datasets; (iii) are "implementation-technology aware," efficiently scheduling computations including pricing modules implemented as native C++ code or concurrently accessing GPGPUs; (iv) have low and deterministic scheduling overhead to match the front-office near-real-time requirements for pre-trade analysis; and (v) provide full control of systems with a rich and holistic development environment, supportive of agile approaches.

The introduction of new architecture components is an opportunity to consider technologies backed by open-source projects, reducing the silo (duplication, model coherence) effect that would result from mixing off-the-shelf solutions from different vendors.

5. SOLUTION DESIGN

5.1 Target architecture

Key to the IMA approach is the requirement to ensure the desk-by-desk P&L attribution. This requires the deployment of a unique cross-asset pricing framework, servicing the front office, risk, and finance functions alike with one "golden source" for trade, market, and reference data. Using a shared pricing framework also allows for efficient allocation of computing resources, with expected savings on infrastructure, better accountability and traceability, and back-testing.

Figure 2: Target front-to-back architecture

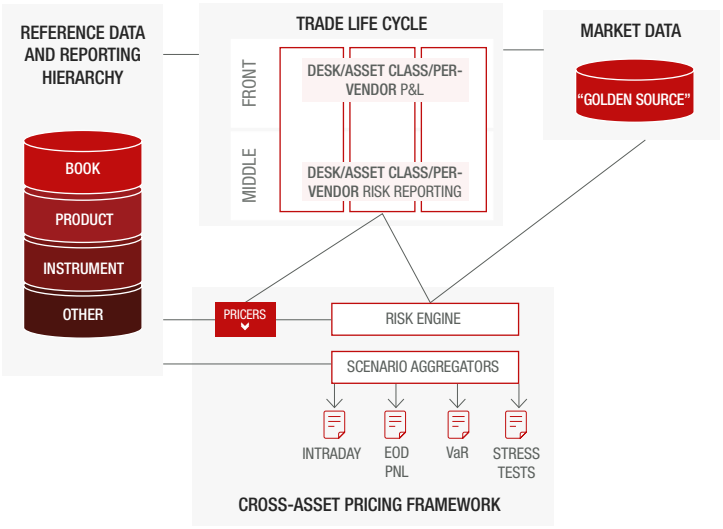
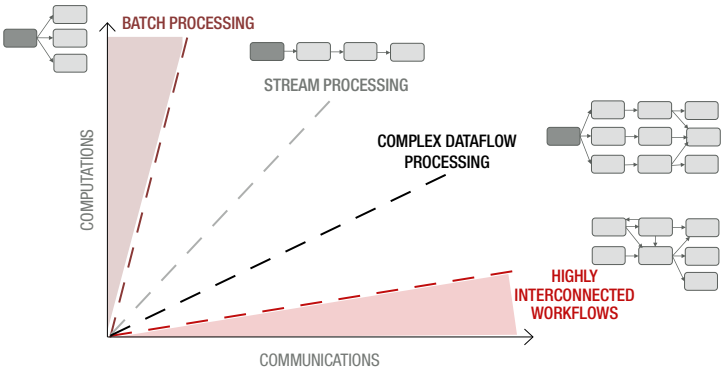


Figure 3: Computational application profiles



This "cross-asset" pricing framework becomes the single source of truth for all valuations, conforming to the intraday front office P&L computations for IMA and SA risk reporting.

The remainder of the article describes an approach to implementing such a pricing framework.

5.2 FRTB computational profile

Designing performance-driven applications requires careful characterization of the computational profile and a good understanding of the often-overlooked low-level execution environment. This is even more true when the solution design involves highly-parallel architectures.

Each application has a computational profile that results from a balance between the amount of CPU computations and the amount of communication between software modules. This spectrum is described as follows:

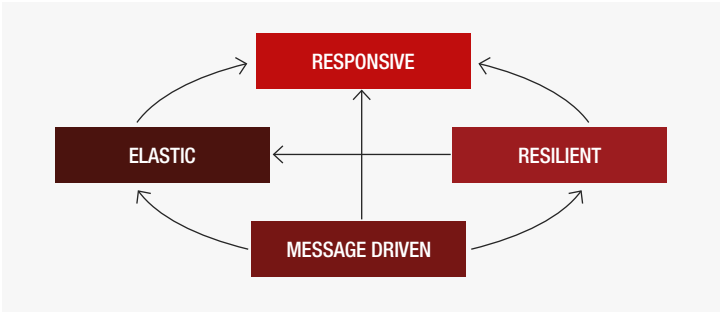
- **Batch processing** (a.k.a. "embarrassingly parallel processing"): consists of modules that require little to no synchronization (i.e., there is no or little need for tasks to communicate results between them).
- **Dataflow processing** (a.k.a. "stream processing"): software modules have only static dependencies upon each other, allowing the application to easily exploit a limited form of parallel processing. In this type of application one can emphasize the movement of data and model the application as a series of connections. Explicitly defined inputs and outputs connect the modules, which function like black boxes. A module runs as soon as all its inputs become valid, which makes the overall application inherently parallel.
- **Complex dataflow processing:** a more complex form of dataflow processing with multi-stage task dependencies. Efficient task scheduling can still be done statically.
- **Highly-interconnected workflows** (a.k.a. "complex event processing"): highly-interconnected workflows, combining data from multiple sources, where the frequency of communication between modules and their inter-dependencies are high and dynamic. Task scheduling needs to be done dynamically.

Each category has a corresponding set of proven solution patterns, programming models, frameworks, libraries, compiler features, or even dedicated hardware (e.g., GPGPU or general-purpose computing on graphics processing units).

FRTB system infrastructure requirements singularly mix all these computation models:

- The Expected Shortfall requires the computation of a high number of pricings, based on historical data or Monte Carlo scenarios. This matches both the "complex dataflow" and "batch processing" categories, typically addressed using a mix of multicore CPUs and/or GPGPUs.
- Sensitivities calculations have a "streaming" profile, where smart memory reuse between iterations is key to CPU performance optimization.

Figure 4: Actor Model



- Efficiently scheduling and refreshing computations based on trade, market, and reference data updates, qualifies for the highly interconnected workflow computation profile, with a strong requirement on multi-core scheduling in the context of large data transfers.

Implementing different computation models requires careful application design, with significant consequences on code complexity and maintainability. This complexity can be visualized using the Roofline performance model,[1] an intuitive approach to a platform performance expectations analysis in the context of a specific hardware.

The model identifies five performance ceilings that constrain runtime performance: processor peak performance (floating point operations per second or FLOPS), memory bandwidth, inter-process communication (instruction pre-fetching, non-uniform memory access), computation (instruction-level and task-level parallelism), and data locality (cache misses qualified as compulsory, capacity, or conflict misses).

A valid solution design under FRTB requirements must balance its module parallelization so that CPUs can be kept busy, avoiding waiting for data from remote or local storage, memory, cache on the CPU, or from OS thread synchronization, etc. The design must, therefore, carefully leverage the hardware (storage, cache, and memory) and properly schedule both IOs and computation threads on multicore platforms.

The FRTB requirements (more computations on growing data volumes combined with a need for real-time processing) is generally considered a strong case for asynchronous, distributed microservice-based architectures.

[1] https://bit.ly/2NB0ZFd
[2] https://bit.ly/2I2HXud

6. SOLUTION DESIGN METHODOLOGY

Reactive software is a design philosophy and a paradigm shift that combines building both large-scale reactive microservices and fine-grain reactive applications (one process). Based on asynchronous message-passing design, there exist a plethora of concurrent programming models that allow for building a reactive software from the ground up. The actor model is one such battle-proven programming model and is the design model supported by the C++ framework we will discuss below.

Reactive systems are software systems that satisfy the four properties depicted in the Reactive manifesto:

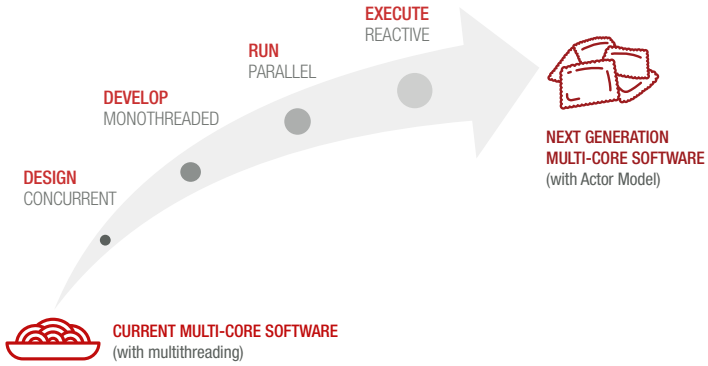
- **Responsive:** to the real-time user demands, as well as internal system components demands. Ensure service continuity.
- **Resilient:** system stays responsive in the face of failure. Resilience is achieved by replication, containment, isolation, and delegation.
- **Elastic:** system stays responsive under varying workload, achieving elasticity in a cost-effective way on commodity hardware and software platforms.
- **Message-driven:** systems rely on asynchronous message-passing to establish a boundary between components that ensure loose coupling, isolation, and location transparency.

The **Actor Model** is a concurrent computation model that uses "actors" as the universal primitive of concurrent computation. It was invented by Carl Hewitt in 1973. Back then, the CPU computing power (single core, 1 MHz, ~5000 transistors, slow I/O, expensive memory) and the application requirements (few concurrent users, small datasets, latency in seconds) did not justify or allow for such complex distributed and parallel systems.

Since then, the theory and practice supporting the Actor Model have matured and proved their worth to software developers and architects for concurrent applications development: actors ended up at the core of the highly respected Erlang programming language, and actors constitute the perfect ground for building Reactive Software, as depicted in the Reactive Manifesto.[2]

Actors form the base constituents of an application logic. Actors communicate with asynchronous events, relying on an execution infrastructure for routing and load-balancing messages transparently. The infrastructure is responsible for optimal actor distribution and monitoring, effectively

Figure 5: Structuring multi-core development



decoupling functional logic from the application's technical execution or deployment. The runtime transparently handles interactions between actors, using in-process communications when possible, and falling back to the network otherwise.

Actors are a very efficient concept, supporting the whole development to production lifecycle. By being directly mapped to functional concepts, actors shorten the distance between business and functional architectures; they encapsulate the logic at a level granular enough for splitting work between developers; they are directly usable concepts for testing; and they allow administrators to decide the topology dynamically, based on available hardware and application load.

7. SOLUTION FRAMEWORK

Implementing a reactive system requires an IT stack with full control over execution, as well as the development flexibility to express the computational problems we described.

A plethora of concurrent programming models and frameworks exist, defining abstractions intended to simplify the developer's job of mapping functional logic to computational resources. Some of these frameworks are successful at hiding the complexity of resource sharing and contention, especially in data-driven cluster deployments, but often fail at efficiently scheduling computations, trading off hardware resources for ease of implementation.

CPUs implement highly sophisticated architectures with multiple levels of parallelization:

- **Instruction-level parallelism (ILP):** several execution units per core and multiple instructions per cycle.
- **Data-level parallelism (DLP):** single instruction, multiple data (SIMD) instructions for vector-computing – multiple processing elements that perform the same operation on multiple data points simultaneously.
- **Thread-level parallelism (TLP):** several processing cores (a.k.a. "multicore") per CPU chip.

Data and instruction-level parallelisms are technical, low-level optimizations, available to native code compilers or virtual machines only. Thread-level parallelism, however, is where application developers and application frameworks come into the picture. This type of design is considered among the most challenging for developers to get correct: inter-thread synchronization and performance considerations (like thread scheduling, thread contention, and coherence) abound.

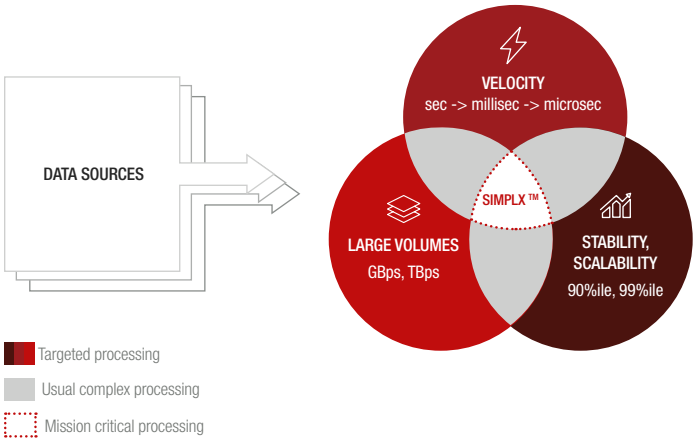
The downside of typical microservice architectures, and the frameworks they build upon, is a focus on the development lifecycle and functional decoupling of deployed artefacts, often trading resources overhead for operational efficiency. While the network resources latency overhead is generally well controlled, the subtler effects of the execution model and the performance bottlenecks raised by multicore execution and memory data transfers are less understood, especially by developers used to JVM development. Yet, mission-critical applications must optimize for multi-core systems and properly schedule communications between functional modules, avoiding message bus intermediaries (Appendix 1).

The ideal software stack allows for building a holistic system with the right balance between high volume processing, fast velocity, infinite scalability, and extreme stability. This calls for a framework fostering in-memory and in-cache computing, core-aware communications, asynchronous inter-thread, and process communication.

Functional drivers for the framework:

- Have **highly-available platform** with native resilience and recovery features.
- Have support for **service-oriented** development patterns.
- Have **loosely coupled** business and technical layers, supporting component composability and reusability, as well as code maintenance.

Figure 6: Ensuring a stable system under heavy computational loads handling large data volumes



- Enable **platform reactivity and responsiveness** with respect to more demanding user activities as well as burst activities.
- Increase **throughput scalability** with number of deployed hardware cores, and with **deterministic response time** (stability > 90 percentile).
- On demand, **compute the right functionalities** to enable the real-time processing based on the infrastructure resources.
- Target-deployment agnostic, addressing grids and clouds (whether hybrid or native) alike, through reconfiguration only.
- Have **real-time monitoring** of software transactions and in-process activities without impacting the performance.

8. SIMPLX™: OPEN SOURCE ACTOR MODEL FRAMEWORK AS ENABLER

Most of the frameworks implementing the Actor Model target JVM environments (e.g., Akka, Scala). Tredzone Reactive Toolset (a.k.a. Simplx™) is the only solution available for mission-critical and latency-sensitive applications implemented in C++ (Appendix 2).

The core technology is a multicore-optimized Actor Model runtime that is integrated in an application as a simple C++ API, and which is responsible for (i) managing the thread's lifecycle and multithreading low-level synchronizations, (ii) managing the cache memory and memory recycling, (iii) managing actor concurrency and scheduling on all cores, (iv) managing communications between actors and CPU cores, all in-memory, (v) adaptive communication performance: embedded throttling in API, (vi) low-level real-time performance monitoring, (vii) on-the-fly multicore deployment, (iix) multicore hardware optimizations and abstractions, and (ix) error handling and management.

Figure 7: Simplx™ ecosystem

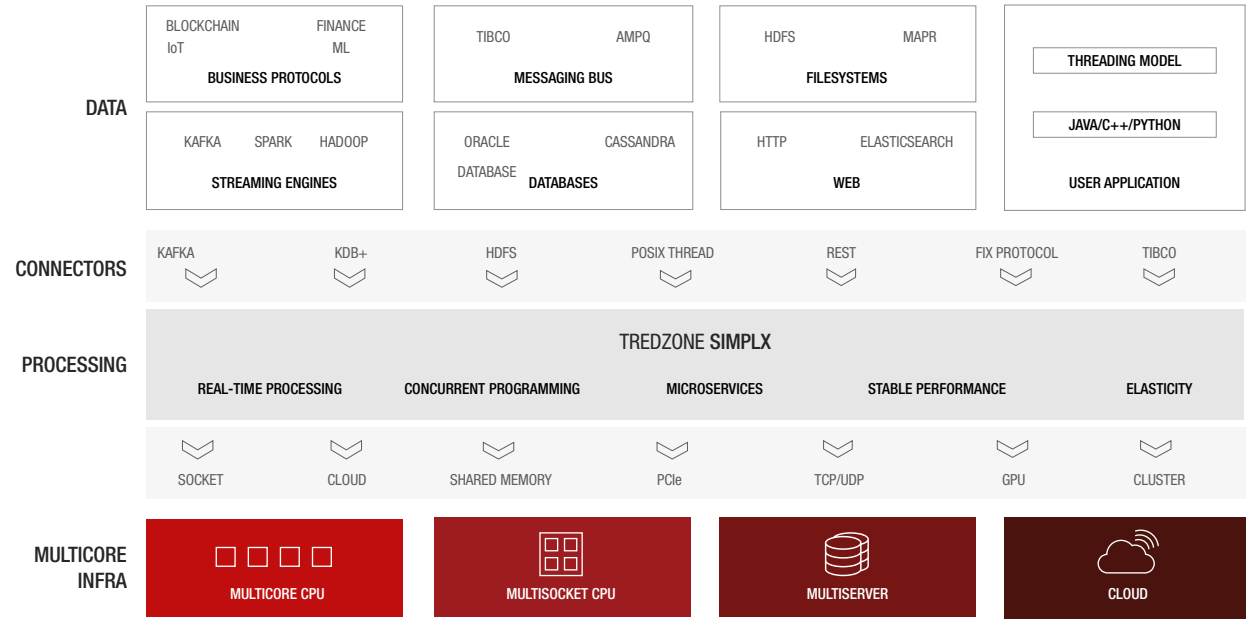
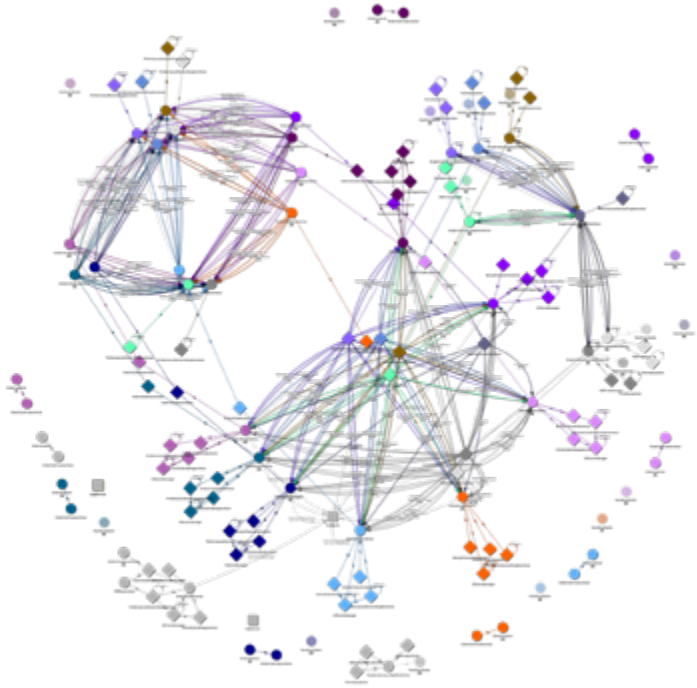


Figure 8: Simplx™ live application monitoring example



The core runtime technology has the following characteristics:

- **No vendor lock-in**, as open sourced under the Apache 2 license[3].
- **Low memory footprint, no "third-party" dependencies, in-memory and in-cache computing.**
- **Single threaded per core**, 100% distributed runtime architecture, enabling vertical scalability by adding more cores with no centralization bottleneck.
- **Clustered runtime architecture** allows for the composition of multiple runtimes to form a communicating cluster (for microservices), hence scaling **horizontally** when adding machines.
- **The core runtime is built in C++ 11.**
- **With multicore hardware portability** the runtime is portable to any multicore hardware architecture (x86, SPARC, ARM, etc.), and even exotic ones (Xeon PHI, Cavium, Kalray, etc.).
- **Operating system portability** works on Linux, Windows, and Mac.
- **Language agnostic** allows for the integration of the native C++ API with a Java or C# API, allowing the mix of actors implemented using a variety of programming languages in the same system. This may prove useful for building a complete FRTB platform, integrating modules from quants, market data providers, risk systems, etc.

In addition to the open-source runtime, Tredzone provide a rich set of DevOps tools to help with debugging, live monitoring (Figure 8), profiling and testing, as well as an ecosystem of convenience libraries and connectors to interface with databases, message buses, or GPGPU hardware.

9. CONCLUSION

FRTB requirements are pushing financial firms to consider alternatives to their existing risk platforms, including resorting to custom implementations. As banks outline a vision of their future infrastructure, their design should reflect the objectives outlined above: reactive and scalable, consistent through golden sources, as well as efficient through the use of standardized design patterns supported by proven frameworks.

This paper explained how a Reactive software design approach,[4] as implemented in Tredzone's Reactive Toolset™, can help address these challenges.

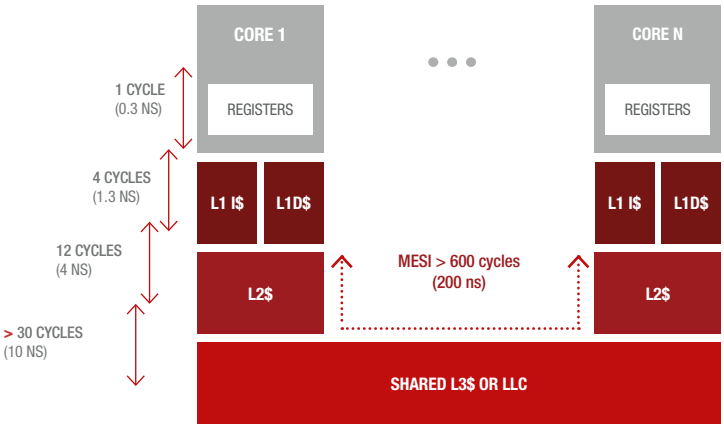
APPENDIX 1: MULTICORE CPU HARDWARE

The mid-2000s marked the end of the Moore's law era, when upgrading to higher-frequency hardware brought automatic performance enhancements. Instead, the industry turned to a model where performance gains come from adding more execution units (cores). But leveraging multicore architectures requires extra development and testing efforts, and a naïve approach of adding more threads often falls short of scalability expectations.

An application relying excessively on threads and/or synchronizations between threads will defeat operating system and hardware schedulers as application threads will inefficiently compete for data access. As documented in Figure A1, the main parameters that impact scalability are:

[3] https://bit.ly/2H3SRMp
[4] www.reactivemanifesto.org

Figure A1: CPU cache access latency



Assuming Freq = 3 GHz

- **Core resources contention** is due to hyperthreading for example. Several software threads want to access simultaneously to the same execution units, and thus processing is serialized and prioritized.
- **Cache memory contention** occurs when there is contention from the same core on its private cache memory (L1/L2) in case of operating system context switch, or when the problem size is too big to be stored in the private cache memory. This phenomenon is called cache thrashing. In multicore, cache thrashing occurs on the shared L3 cache (or LLC last level cache) between the cores. Hence, the impact is several hundreds of nanoseconds lost in latency. The solution is an intelligent software that improves the data locality in-cache processing in order to also improve the memory bandwidth utilization while keeping the CPU core busy doing local computation; this is a very difficult problem, and there is no easy solution today. This is a typical problem in HPC stencil computation.
- **Cache coherency** is a hardware mechanism that allows for core to core seamless communication whenever there is a software synchronization (mutex, barrier, etc.) or access to the same memory area by both cores. A simple cache coherency (one cache line) costs at least 600 cycles.

APPENDIX 2: THE EURONEXT USE CASE

In spring 2014, Euronext started a new phase of their history as an independent listed firm, span-off from ICE.

They immediately identified a major strategic priority: upgrade their technical infrastructure in order to make it easier to follow the fast-changing business requirements. This resulted in contradictory constraints: make performance fully predictable and reduce latency, cope with increasing and in practice unpredictable volumes, and cope with new functionalities, hence increasing complexity. These contradictory requirements sounded like an impossible mission. As is often the case, when faced with engineering challenges, the first reaction was to seek hardware capability improvements (newer multicore-based machines). However, close analysis concluded that relying on hardware upgrades alone would not significantly improve performance, while adding to the complexity of managing a large infrastructure.

The project itself was raising contradictory interests between stakeholders, making communication increasingly difficult and creating an increasingly tense dialog between implementation teams.

Tredzone demonstrated the value of its Simplx™ reactive toolset to Euronext. The inner features of its reactive-design approach, its optimal handling of cluster resources scheduling, and its extensive set of productivity tools proved essential to the performance, stability, and monitoring of Euronext's new platform. Tredzone's technology became the foundational backbone of Euronext's Optiq® trading platform. A distributed team iteratively released the new Optiq platform components over less than three years.

Optiq® achieved dramatic performance improvements, running 10 times faster (tens of microseconds) and at a high level of stability (99th percentile), while dividing the hardware footprint by four. This resulted in significant savings and a positive return on investment.

ABOUT CAPCO

Capco is a global technology and management consultancy dedicated to the financial services industry. Our professionals combine innovative thinking with unrivalled industry knowledge to offer our clients consulting expertise, complex technology and package integration, transformation delivery, and managed services, to move their organizations forward.

Through our collaborative and efficient approach, we help our clients successfully innovate, increase revenue, manage risk and regulatory change, reduce costs, and enhance controls. We specialize primarily in banking, capital markets, wealth and asset management and insurance. We also have an energy consulting practice in the US. We serve our clients from offices in leading financial centers across the Americas, Europe, and Asia Pacific.

WORLDWIDE OFFICES

APAC
Bangalore
Bangkok
Hong Kong
Kuala Lumpur
Pune
Singapore

EUROPE
Bratislava
Brussels
Dusseldorf
Edinburgh
Frankfurt
Geneva
London
Paris
Vienna
Warsaw
Zurich

NORTH AMERICA
Charlotte
Chicago
Dallas
Houston
New York
Orlando
Toronto
Tysons Corner
Washington, DC

SOUTH AMERICA
São Paulo

WWW.CAPCO.COM

CAPCO