C++ PERFORMANCE OPTIMIZATION

Alexander Maslennikov

About the speaker

- Software Development Engineer at Intel
- Responsibilities: transforming a desktop performance profiler, named Intel® VTune™ Profiler, into a cloud service

Do you really need to tune your performance?

You don't need this if:

- You have no demand for higher performance
- You can afford to run your service on better HW

Optimization criteria and stopping condition

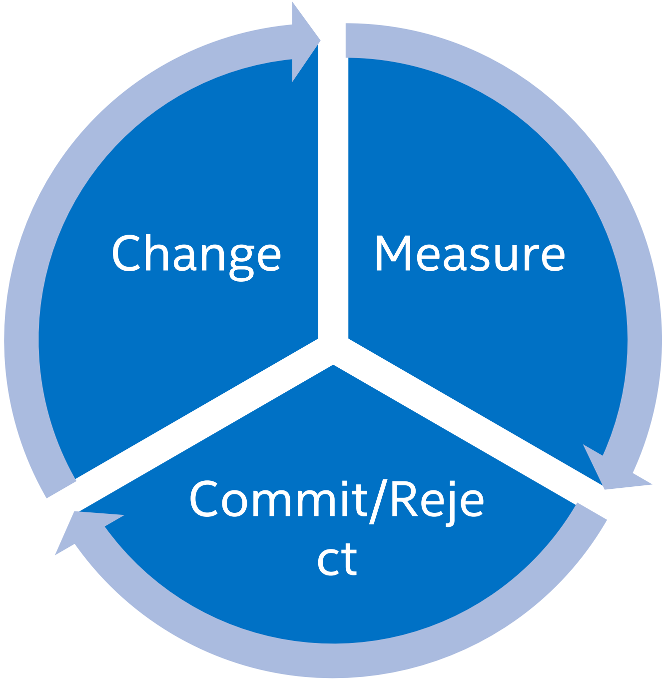
Optimization criterion is a metric by which you evaluate the performance of your workload

- Execution time
- Frame rate
- Throughput
- Latency

Stopping condition defines the optimization criteria value good enough to stop the process

- < 10s
- > 60 FPS
- etc.

Optimization Process



Start with baseline

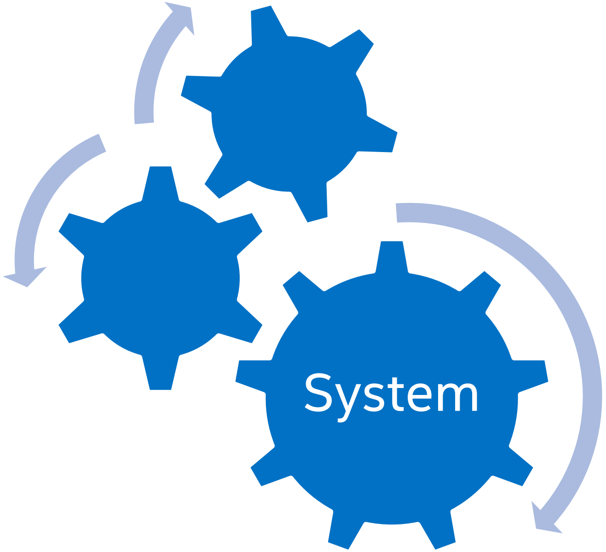
Baseline measurement reflects the absolute best performance the solution is capable of exhibiting in its current state

Break when stopping condition is met

Build a solid baseline

- Use right compiler flags
  - Compile in release mode
  - Compile with optimization flags (O2 or O3 for gcc, icl, and msvc)
- Compile for the target CPU
  - Your target microarchitecture might have support of a more advanced instruction set
  - Benefit from Intel® Advanced Vector Extensions (Intel® AVX, AVX2, AVX-512)
  - -march, -mtune, -mcpu
- Use optimized libraries

What makes a good workload



Workload

- Measurable
- Reproducible
- Representative

Ways to optimize performance and their impact

Performance increase is unknown and might be huge

- Algorithmic optimization
- Design optimization

Limited performance increase

- Vectorization
- Parallelization
- Other microarchitecture optimizations

Let’s get down to business

Sample application

- A small model of in-memory database
- Keeps data about employees:
    - Name
    - Position
    - Age
    - Salary
- Allows applying filters and getting results
- Holds 50 million records

Sample application data flow

1. Client makes a request and provides filters
2. Registry, that holds all records, performs filtration
3. Filtered data is returned in JSON response

Sample workload

- **Workload:** filtering operations
- **Performance metric:** Queries per Second (QPS) (higher is better)
- **Stopping condition:** QPS > 30
- Instrumentation and Tracing technology (ITT) API is used for code instrumentation and extracting the workload from a system

Baseline and setup

- Intel® Xeon® Gold 6152 Processor
  - 22 cores, 2 sockets
  - Supports AVX-512
- 128GB RAM
- Intel® Compiler 19.0
- Baseline performance: **3.9 QPS**

Algorithmic optimization

- Choose algorithms of the best complexity and the most suitable
- Use libraries that contain algorithms optimized for target HW

Design optimization

Revise code architecture and design to avoid performance bottlenecks

- Eliminate unnecessary copies
- Use async operations instead of active waits
- Use caching
- etc.

Hotspots

- Don’t guess - use a profiler
- Focus on unexpected hotspots
- Focus on large hotspots – doesn’t make sense to optimize insignificant hotspots
- Potential gain might be the most significant if algorithms aren’t optimal

Top hotspots

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
std::vector<filtering::Employee, std::allocator<filtering::Employee> >::_Emplace_reallocate	baseline.exe	5.122s
filtering::EmployeeRegistry::filter	baseline.exe	3.363s
func@0x1401be877	ntoskrnl.exe	1.909s
std::getline	baseline.exe	1.574s
filtering::EqualsFilter<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >::match	baseline.exe	1.373s
[Others]		15.301s

Unexpectedly long copy operation – who calls?

Callers
std::vector<filtering::Employee, std::allocator<filtering::Employee> >::_Emplace_reallocate
filtering::EmployeeRegistry::filter
main

Design optimization

```
std::vector<Employee> EmployeeRegistry::filter(
    IFilter<std::string>::Ptr nameFilter,
    IFilter<std::string>::Ptr positionFilter,
    IFilter<int>::Ptr ageFilter,
    IFilter<float>::Ptr salaryFilter) const
{
    std::vector<Employee> result;

    std::copy_if(
        m_employees.begin(),
        m_employees.end(),
        std::back_inserter(result),
        [&](const Employee& employee)
    {
        bool match = true;

        // Filtering stuff

        return match;
    });

    return result;
}
```

- Return a vector of Employee objects after filtration
- Copy filtered employees into result vector
- No space reservation for result vector

Remove unnecessary copying

```
std::vector<size_t> EmployeeRegistry::filter(
    IFilter<std::string>::Ptr nameFilter,
    IFilter<std::string>::Ptr positionFilter,
    IFilter<int>::Ptr ageFilter,
    IFilter<float>::Ptr salaryFilter) const
{
    std::vector<size_t> result;
    result.reserve(m_employees.size());

    for (size_t i = 0; i < m_employees.size(); i++)
    {
        bool match = true;

        // Filtering stuff

        if (match) result.push_back(i);
    }

    return result;
}
```

- Return a vector of indices of filtered records in the main container
- Return a vector of all records in a separate method
- Performance after the change: **9.3 QPS**
- $\frac{9.3\ QPS}{3.9\ QPS} \approx \mathbf{2.38x}$ performance gain against baseline

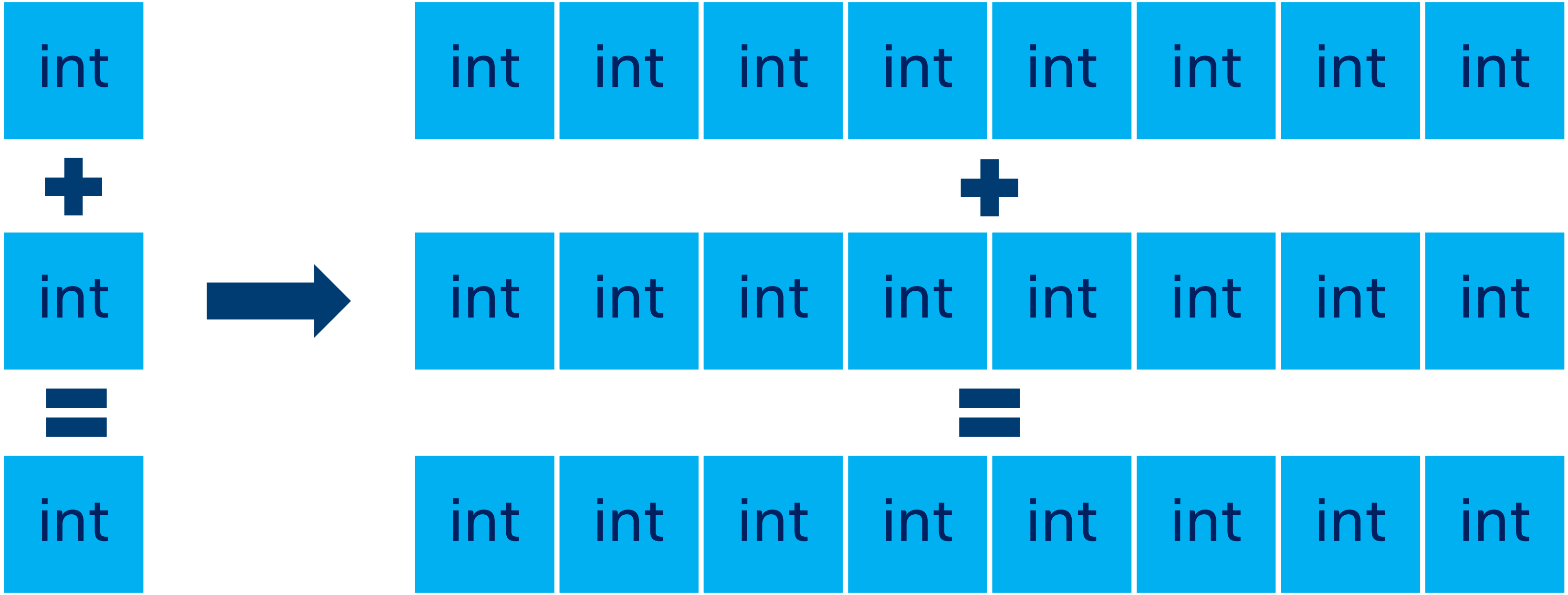
Top hotspots after removing unnecessary copies

Expensive copy operation is not a top hotspot anymore

filtering::EmployeeRegistry::filter	1.643s
filtering::EqualsFilter<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >::match	0.665s
filtering::NotEqualsFilter<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >::match	0.554s
intel_fast_memcmp	0.542s
filtering::GreaterFilter<int>::match	0.283s
filtering::LessFilter<float>::match	0.198s
filtering::AnyFilter<float>::match	0.196s
filtering::GreaterFilter<float>::match	0.193s
filtering::LessFilter<int>::match	0.161s

Now top hotspots are simple filtering functions and it doesn’t make sense to continue with algorithmic optimization

Vectorization



Vectorization boundaries

AVX-512 zmm register size

int	int	int	int	int	int	int	int	int	int	int	int	int	int	int	int	16x 32-bit int

double	double	double	double	double	double	double	double	8x 64-bit double

Vectorization report

Function Call Sites and Loops	Type	Why No Vectorization?
filtering::NotEqualsFilter<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >::match	Function	
filtering::LessFilter<int>::match	Function	
filtering::LessFilter<float>::match	Function	
filtering::IEmployeeRegistry::load	Function	
filtering::GreaterFilter<int>::match	Function	
filtering::GreaterFilter<float>::match	Function	
filtering::EqualsFilter<float>::match	Function	
filtering::EqualsFilter<class std::basic_string<char,struct std::char_traits<char>,class std::allocator<char> > >::match	Function	
filtering::EmployeeRegistry::filter	Function	
filtering::EmployeeRegistry::EmployeeRegistry	Function	
filtering::Employee::Employee	Function	
filtering::AnyFilter<float>::match	Function	

No loops were detected in filtering functions

Before

```
std::vector<size_t> EmployeeRegistry::filter(
    IFilter<std::string>::Ptr nameFilter,
    IFilter<std::string>::Ptr positionFilter,
    IFilter<int>::Ptr ageFilter,
    IFilter<float>::Ptr salaryFilter) const
{
    std::vector<size_t> result;
    result.reserve(m_employees.size());

    for (size_t i = 0; i < m_employees.size(); i++)
    {
        bool match = true;

        match &= nameFilter->match(m_employees[i].name);
        match &= positionFilter->match(m_employees[i].position);
        match &= ageFilter->match(m_employees[i].age);
        match &= salaryFilter->match(m_employees[i].salary);

        if (match) result.push_back(i);
    }

    return result;
}
```

```
template<class T>
class IFilter
{
    ...
    virtual bool match(const T& value) const = 0;
};

template<class T>
class EqualsFilter final : public ISingleValueFilter<T>
{
    ...
    bool match(const T& value) const override
    {
        return value == m_value;
    }
};
```

After

```
std::vector<size_t> EmployeeRegistry::filter(
    IFilter<std::array<char, 32>>::Ptr nameFilter,
    IFilter<std::array<char, 32>>::Ptr positionFilter,
    IFilter<int>::Ptr ageFilter,
    IFilter<float>::Ptr salaryFilter) const
{
    std::vector<char> matched(m_size, 1);
    nameFilter->match(m_names, matched);
    positionFilter->match(m_positions, matched);
    ageFilter->match(m_ages, matched);
    salaryFilter->match(m_salaries, matched);

    std::vector<size_t> result;
    result.reserve(matched.size());

    for (size_t i = 0; i < m_names.size(); i++)
    {
        if (matched[i]) result.push_back(i);
    }

    return result;
}
```

```
template<class T, class Filter>
void match(const std::vector<T>& values, std::vector<char>& result, Filter filter)
{
    const int valuesSize = values.size();
#pragma ivdep
#pragma vector always
    for (int i = 0; i < valuesSize; i++)
    {
        result[i] = result[i] & filter(values[i]);
    }
}

template<class Filter>
void match<std::array<char, 32>, Filter>(const std::vector<std::array<char, 32>>& values, std::vector<char>
& result, Filter filter)
{
    const char* pValues = reinterpret_cast<const char*>(values.data());
    int counter = 0;
    const int valuesSize = values.size();
    // Inner loop is already vectorized, no need for #pragma vector here
    for (int i = 0; i < valuesSize * 32; i += 32, counter++)
    {
        result[counter] = result[counter] & filter(pValues + i);
    }
}

template<>
class EqualsFilter<std::array<char, 32>> final : public ISingleValueFilter<std::array<char, 32>>
{
    ...
    void match(const std::vector<std::array<char, 32>>& values, std::vector<char>& result) const override
    {
        filtering::match(values, result, [this](auto value)
        {
            bool result = true;
#pragma ivdep
#pragma vector always
            for (int i = 0; i < 32; i++) if (value[i] != m_value[i]) result = false;

            return result;
        });
    }
};
```

Vectorization report

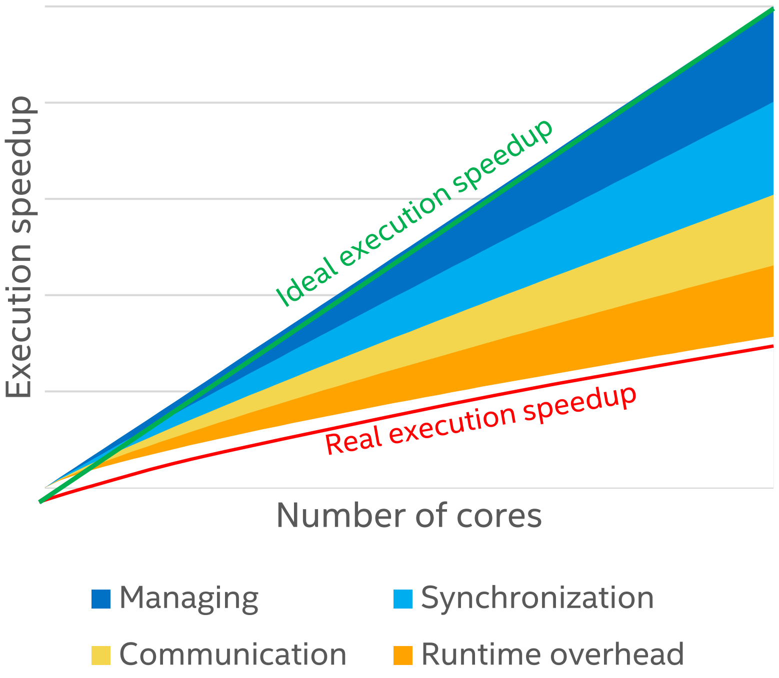
Function Call Sites and Loops	Type	Why No Vectorization?
FilterLogger::~FilterLogger	Function	
[Import thunk std::basic_streambuf<char,struct std::char_traits<char> >::_Lock]	Function	
[Import thunk std::basic_streambuf<char,struct std::char_traits<char> >::_Unlock]	Function	
[loop in filtering::EmployeeRegistry::EmployeeRegistry at xiosbase:33]	Scalar	exception handling for a call prevents vectorization
[loop in filtering::EmployeeRegistry::EmployeeRegistry at xlocale:33]	Scalar	
[loop in filtering::EmployeeRegistry::filter at employee_registry.cpp:74]	Scalar	exception handling for a call prevents vectorization
[loop in filtering::EmployeeRegistry::filter at employee_registry.cpp:74]	Scalar	
[loop in filtering::EmployeeRegistry::filter at vector:74]	Scalar	
[loop in filtering::EmployeeRegistry::filter at xmemory0:74]	Scalar	
[loop in filtering::EmployeeRegistry::filter at xmemory0:74]	Scalar	
[loop in filtering::EmployeeRegistry::filter at xmemory0:74]	Scalar	
[loop in filtering::EqualsFilter<class std::array<char,32> >::match at filter.hpp:102]	Vectorized (Body) Completely Unrolled	
[loop in filtering::EqualsFilter<class std::array<char,32> >::match at filter.hpp:94]	Scalar	inner loop was already vectorized
[loop in filtering::GreaterFilter<float>::match at filter.hpp:164]	Vectorized (Body)	
[loop in filtering::GreaterFilter<int>::match at filter.hpp:615]	Vectorized (Body)	
[loop in filtering::GreaterOrEqualsFilter<float>::match at filter.hpp:180]	Vectorized (Body)	
[loop in filtering::LessFilter<float>::match at filter.hpp:196]	Vectorized (Body)	
[loop in filtering::LessFilter<int>::match at filter.hpp:196]	Vectorized (Body)	
[loop in filtering::LessOrEqualsFilter<float>::match at filter.hpp:212]	Vectorized (Body)	
[loop in filtering::NotEqualsFilter<class std::array<char,32> >::match at filter.hpp:137]	Scalar	inner loop was already vectorized
[loop in filtering::NotEqualsFilter<class std::array<char,32> >::match at filter.hpp:145]	Vectorized (Body) Completely Unrolled	

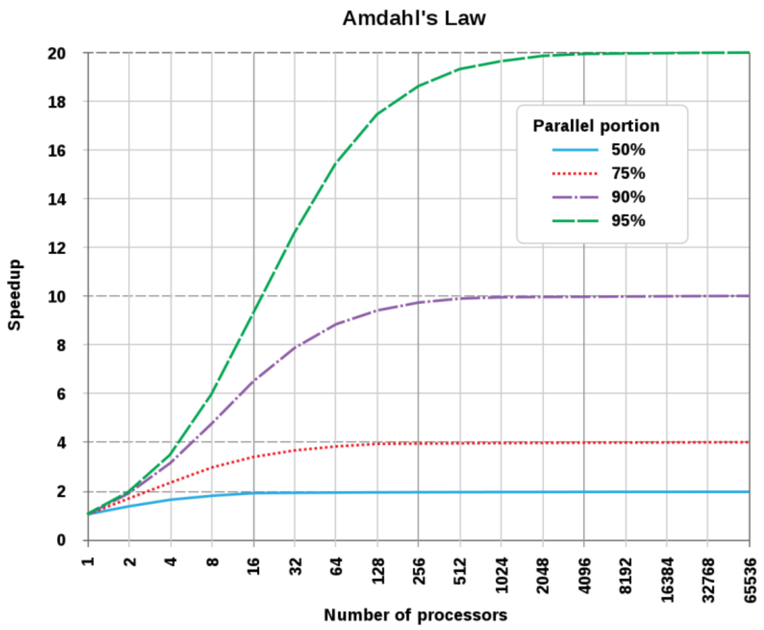
All filter loops were vectorized and some were completely unrolled

Performance gain

- Performance after vectorization: **24.0 QPS**
- **6.2x** faster than the baseline
- ~2.6x faster than the previous step
- Still not good enough

Parallel execution speedup

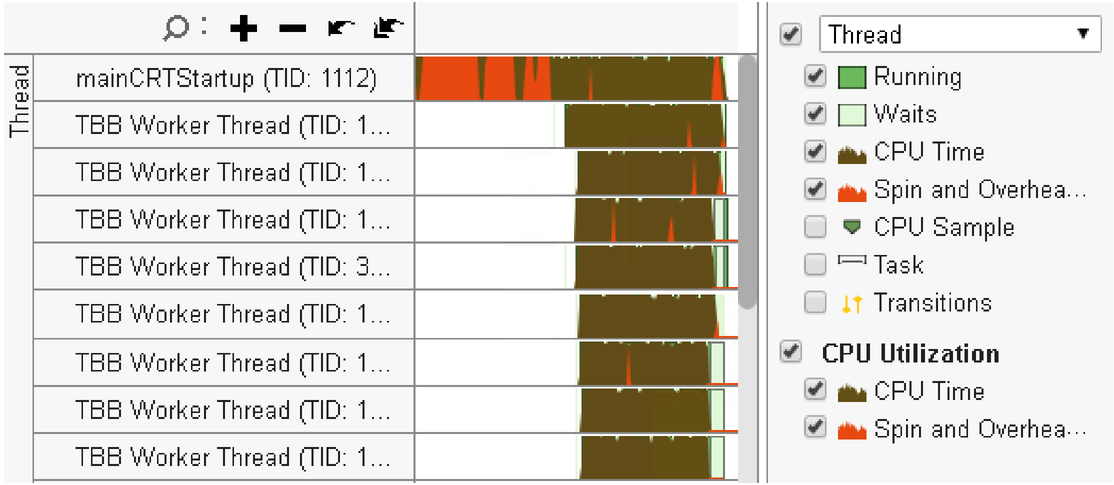




Threading Building Blocks

```
#include <tbb/task_scheduler_init.h>
#include <tbb/task_group.h>

...
tbb::task_scheduler_init();

tbb::task_group group;

...

for (auto filter : filters)
{
    group.run([filter, &registry]
    {
        ... // Initializing filters

        auto result = registry->filter(
            std::move(nameFilter),
            std::move(positionFilter),
            std::move(ageFilter),
            std::move(salaryFilter));
    });
}

group.wait();
```

- Performance: **130.5 QPS**
- **33x** faster than the baseline
- Stopping condition is met

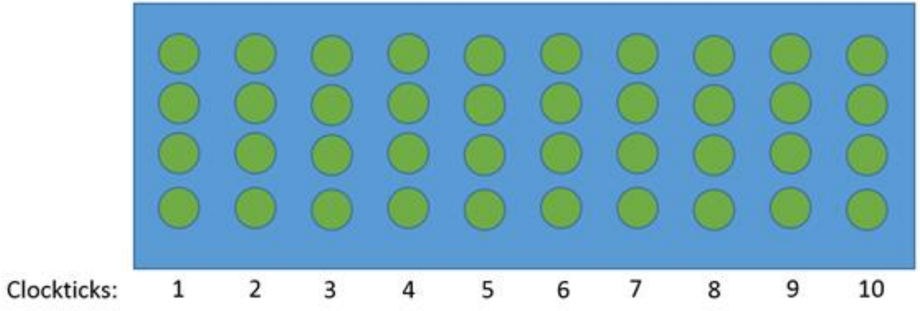


Bold attempt to parallelize baseline

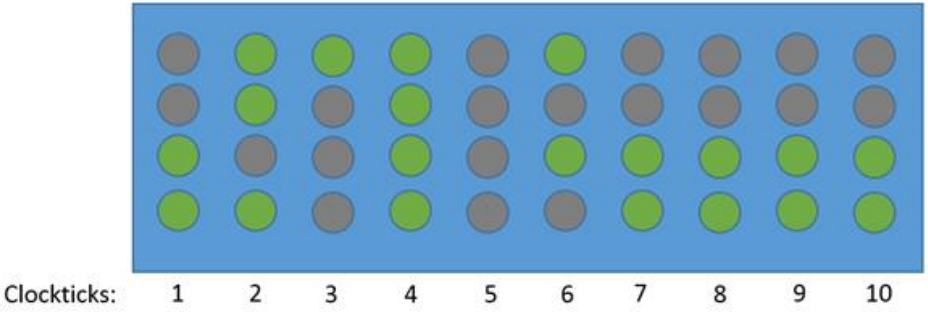
- Performance: **5.2 QPS**
- 25x slower than optimized code
- < 2x faster than the baseline
- Optimization of the single thread performance is multiplied after parallelization

uArch optimizations

- µArch performance gain is limited by ideal CPI (Clocks Per Instruction) of 0.25
- CPI does not depend on CPU frequency which may change

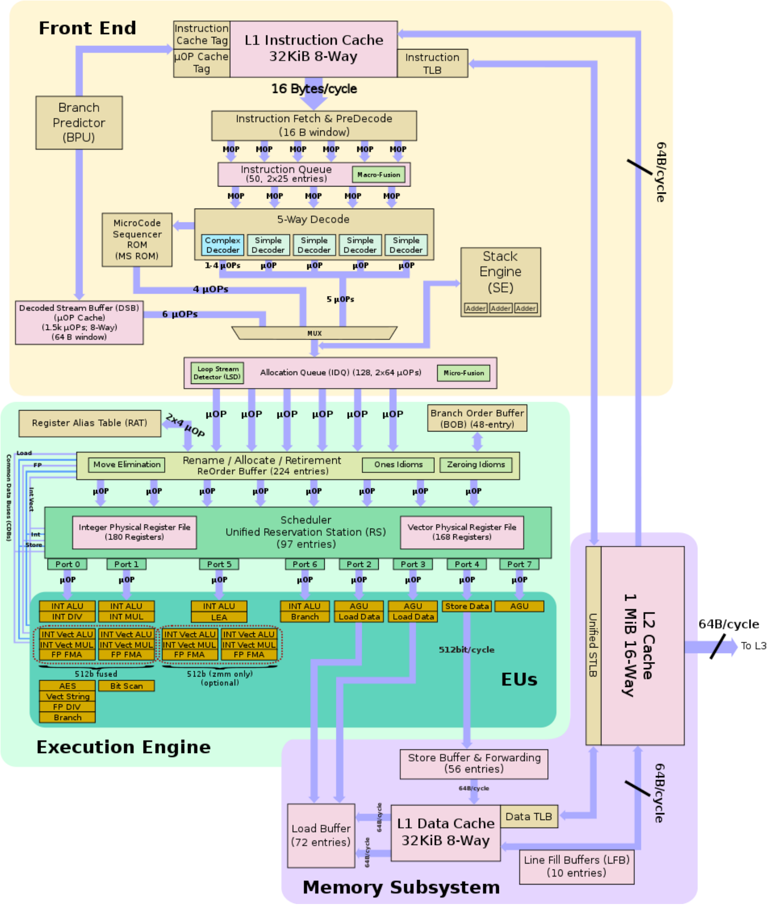


Ideal CPU pipeline



Real CPU pipeline

Modern CPU core architecture

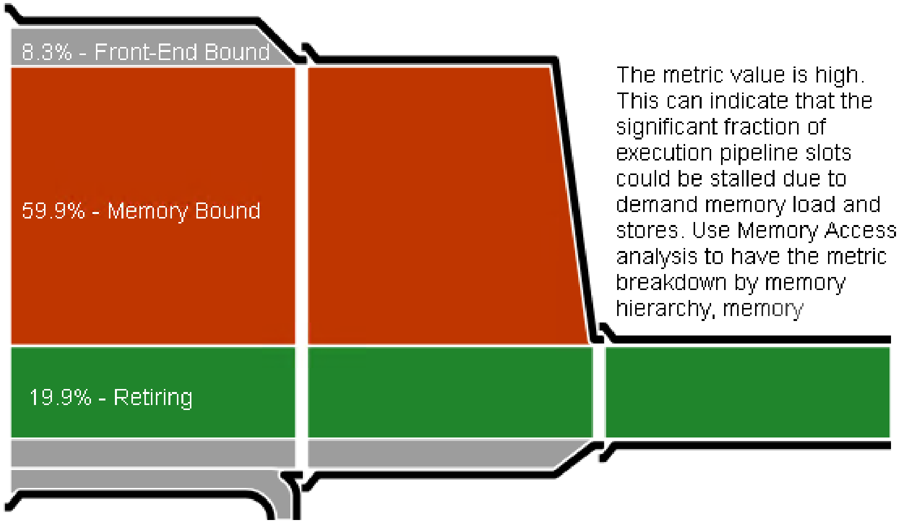


Why instructions might not retire:

- Front-end bound
- Back-end bound
- Bad speculation

Microarchitecture efficiency

- CPI rate: 1.627
- > 50% of instruction are stalled due to demand of memory

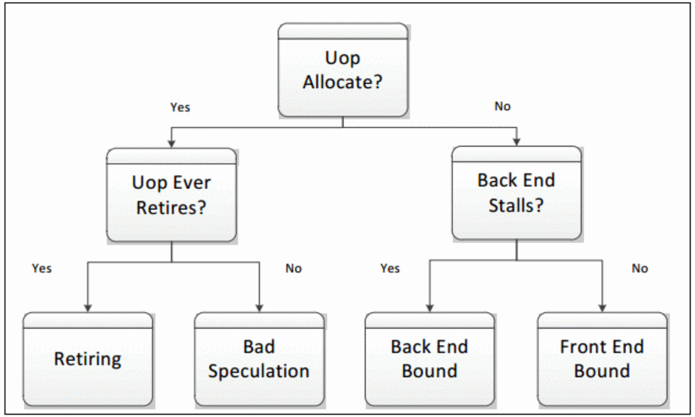


µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Dig deeper?

- Rewrite everything in assembly?
- Whatever you do – measure the CPI (Clocks Per Instruction) (use Top-down approach and focus on execution problems)
- CPI in general is not actionable – look for CPI of specific operations
- Only if you're desperate
- You're left behind when a new HW is released



Top-Down characterization

Summary

1. Use optimized libraries
2. Get a solid baseline
3. Define optimization criterion for your workload
4. Profile your code
5. Fix your code design and algorithms first
6. Vectorize and parallelize
7. Help the compiler rather than go wild with assembly code

Useful links

- Sample project: https://github.com/alexandermaslennikov/cpp-performance-optimization
- Intel® VTune™ Profiler: https://software.intel.com/vtune
- Intel® Advisor: https://software.intel.com/advisor
- Intel® Threading Building Blocks: https://software.intel.com/tbb
- Top-down Microarchitecture Analysis Method: https://software.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html