Getting It All With C++: Abstraction, Reusability, Performance, And Future-Safety

Ulrich Drepper

redhat.

Goal of Programming

Write with as little effort programs
that run as fast as possible
(and/or use as few resources as possible)

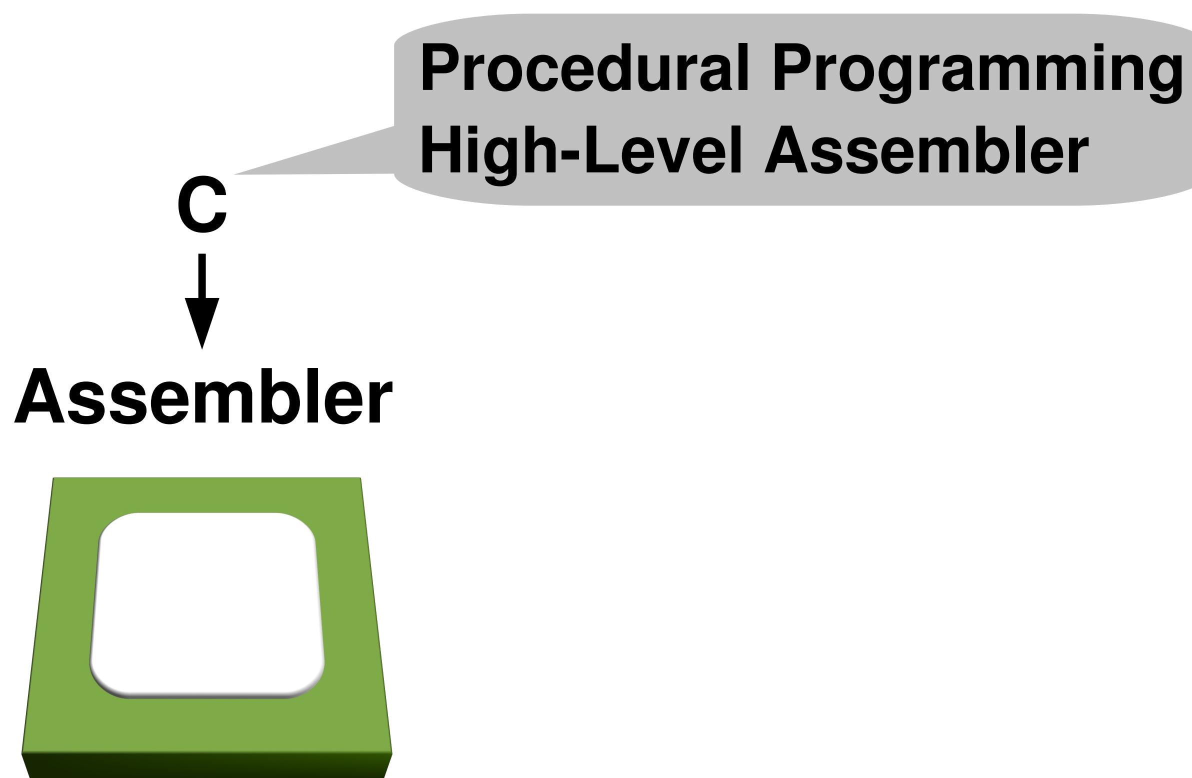
redhat

Ways of Programming

Assembler

redhat

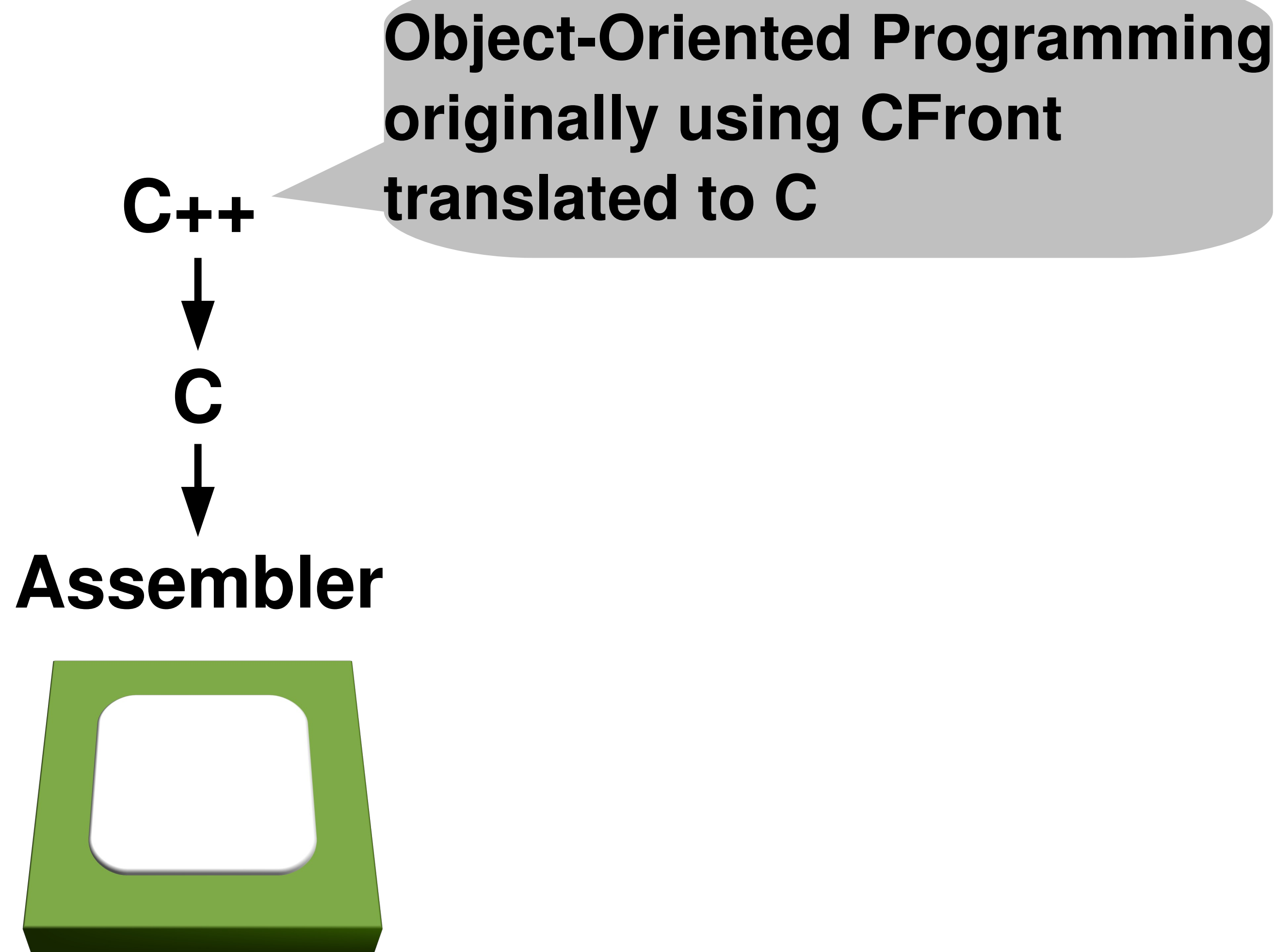
Ways of Programming



redhat

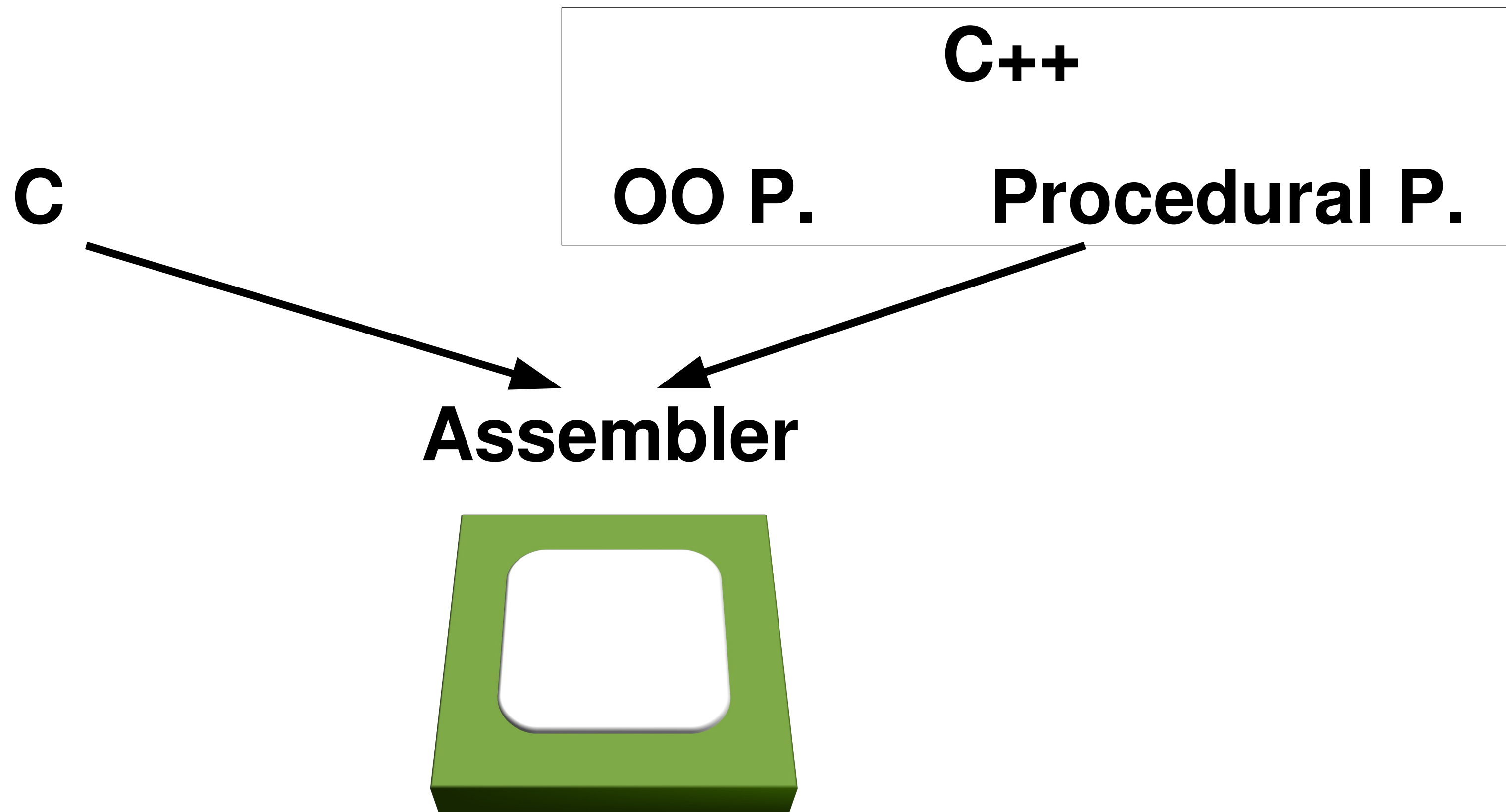
Ways of Programming

- More efficient C:



Ways of Programming

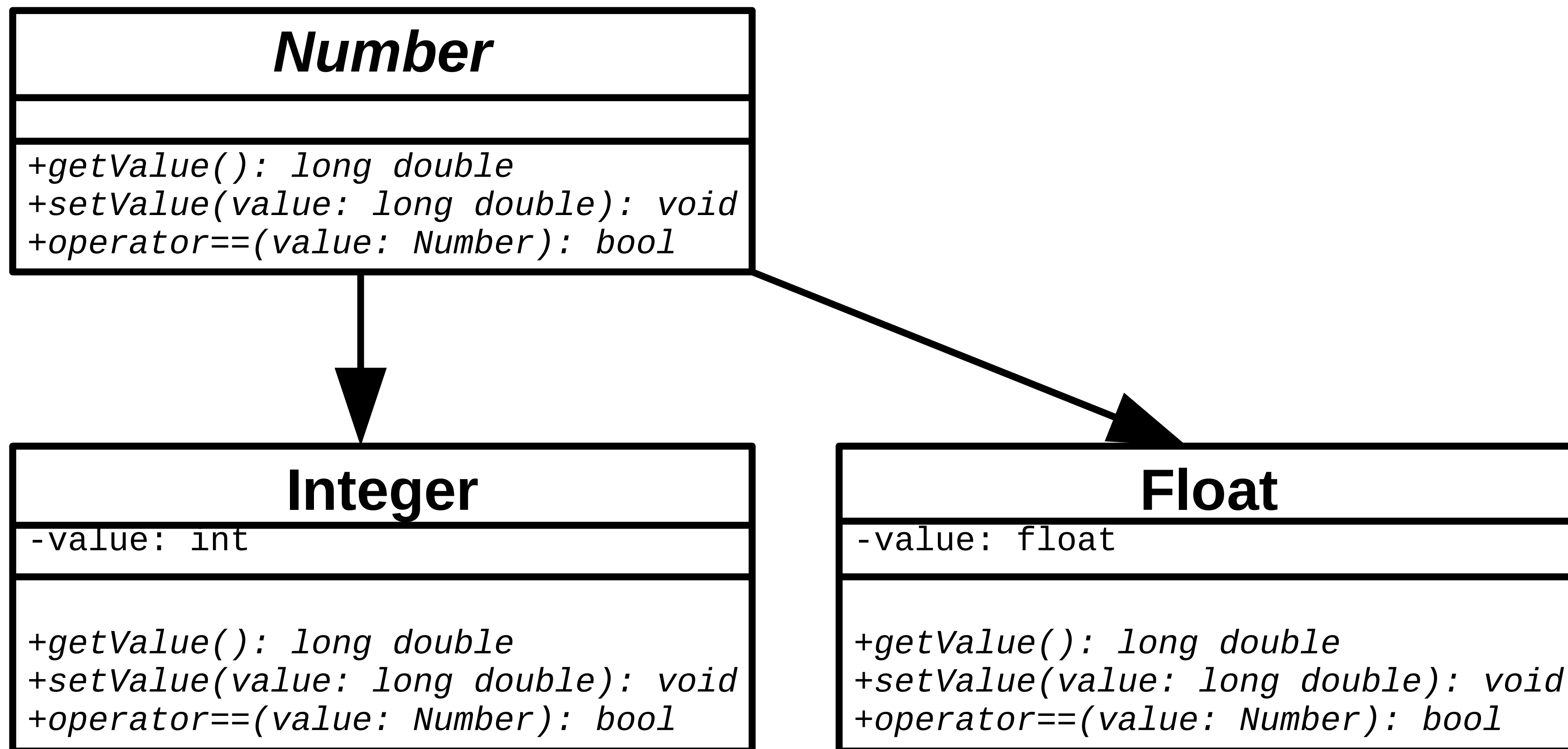
- C++ a language on its own



redhat

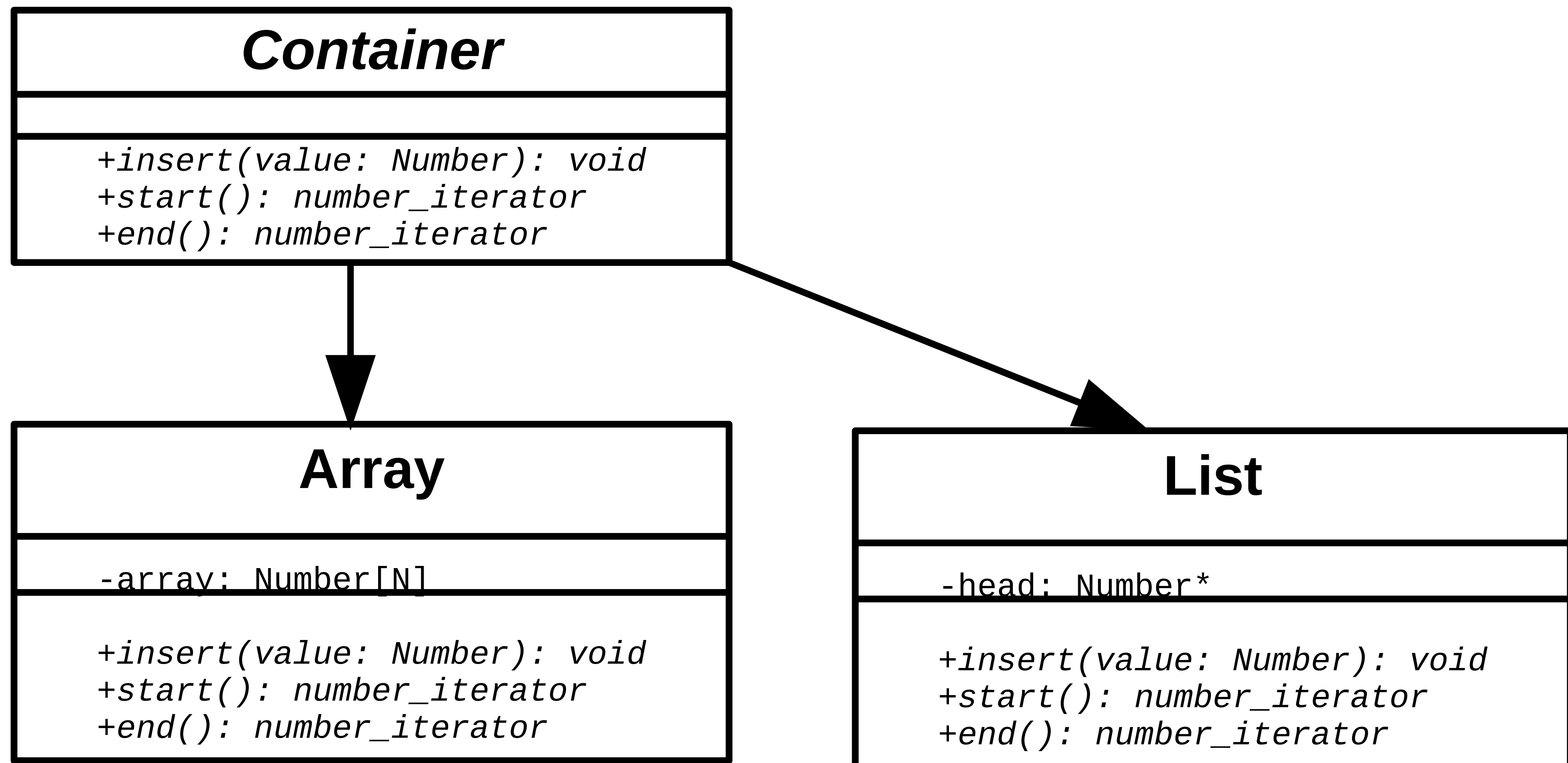
OOP Principles

- Derive and specialize



OOP Principles

- Use base classes in other contexts



OOP Principles

- Advantages:
  - Flexibility
    - Maximum code sharing
    - Minimal ABI
    - Stable ABI for library/DSO interface
  - “Natural” abstraction
- Disadvantages:
  - Inefficient
  - Often requires “superset type” in interfaces

Use Example Classes

- Search implementation

```
bool find(Container &c, Number &n) {
  for (number_iterator i = c.start(); i != c.end(); ++i)
    if (*i == n)
      return true;
  return false;
}
```

- Cost:
  - 2 virtual function calls in prologue
  - 3 virtual function calls pro loop iteration

Virtual Functions

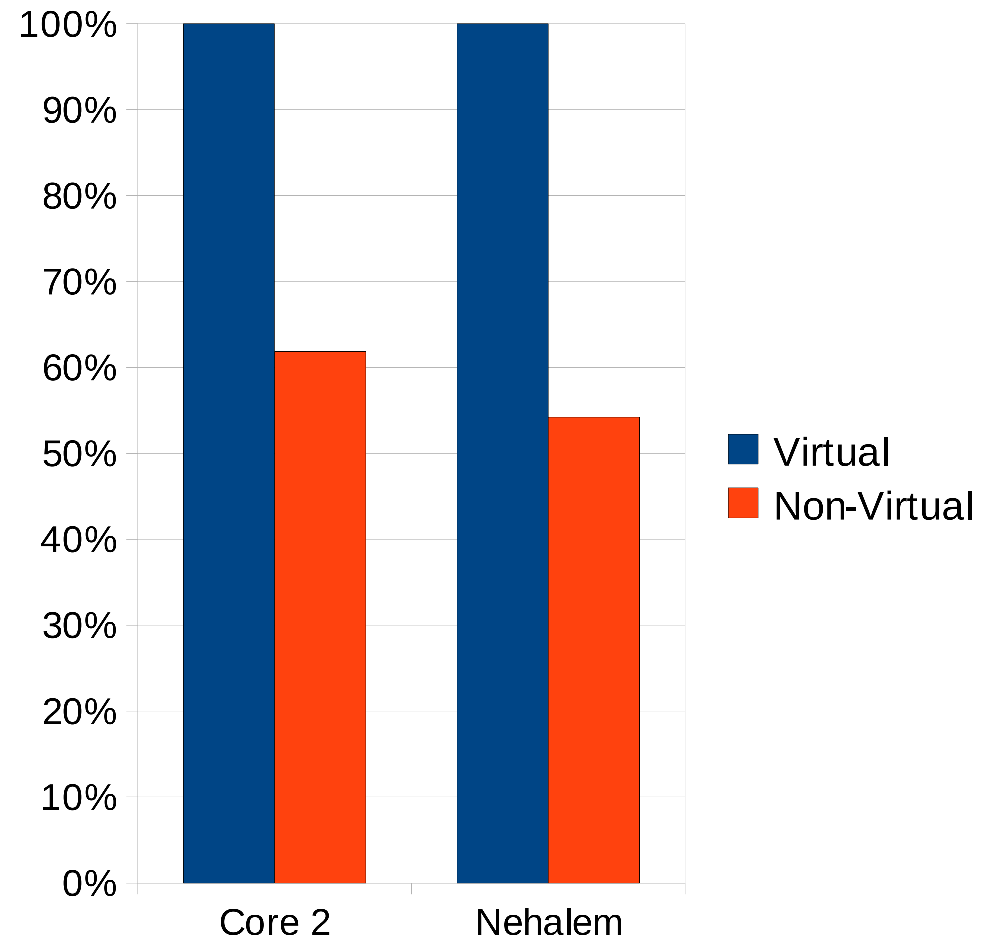
- Virtual function call
  - Cannot be inlined (in general)
  - Indirect function call → prevents prefetching of CPU
  - Often trivial operations dwarfed by cost of call
- Increased size of objects (virtual function table)

redhat

Virtual Functions

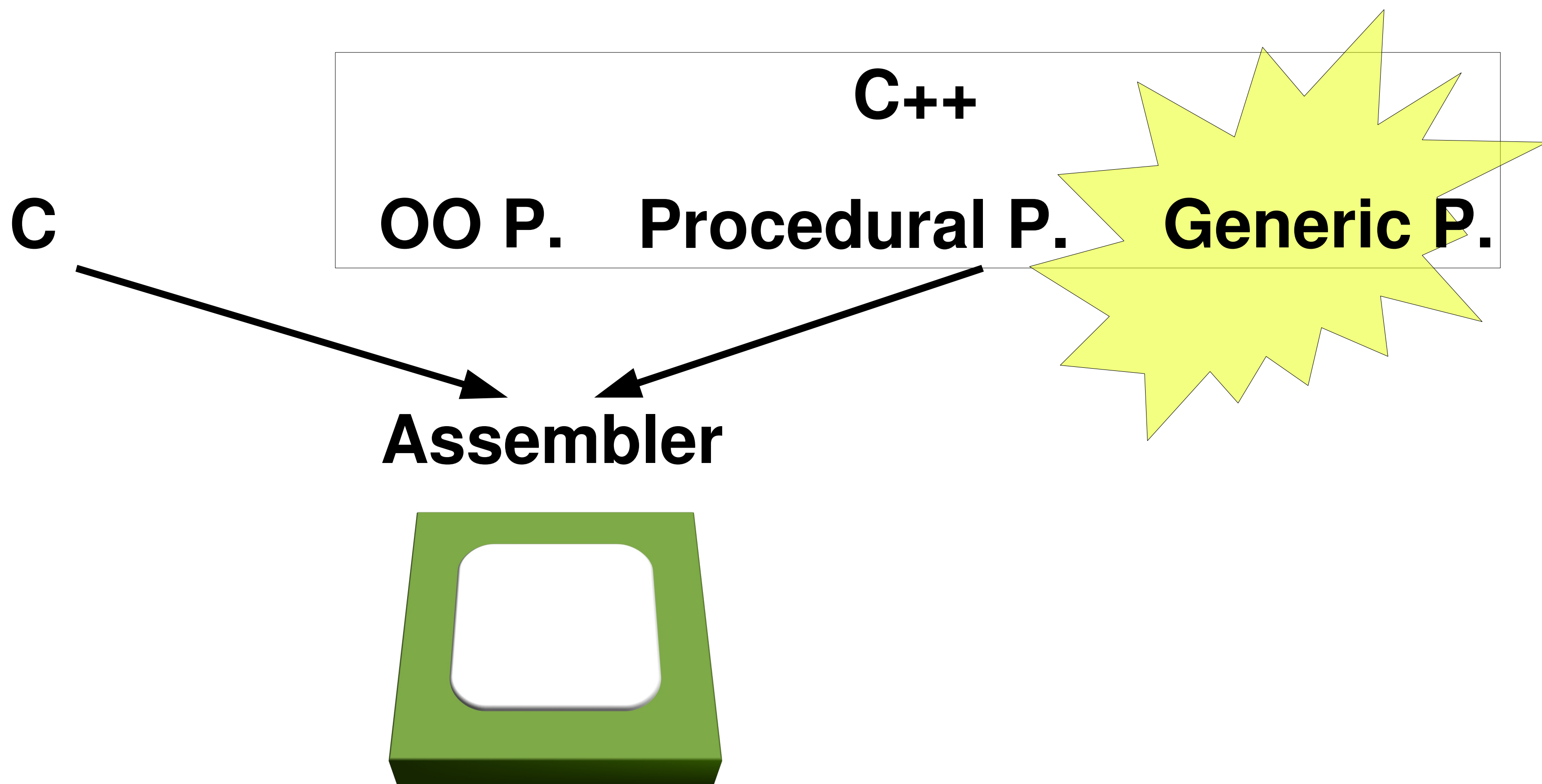
- Example:

```
struct vc {
  int a;
  vc(int b) : a(b) {}
  virtual bool t(int) const;
  virtual void u(int);
};
bool vc::t(int b) const {
  return a < b;
}
void vc::u(int b) {
  a += b;
}
```



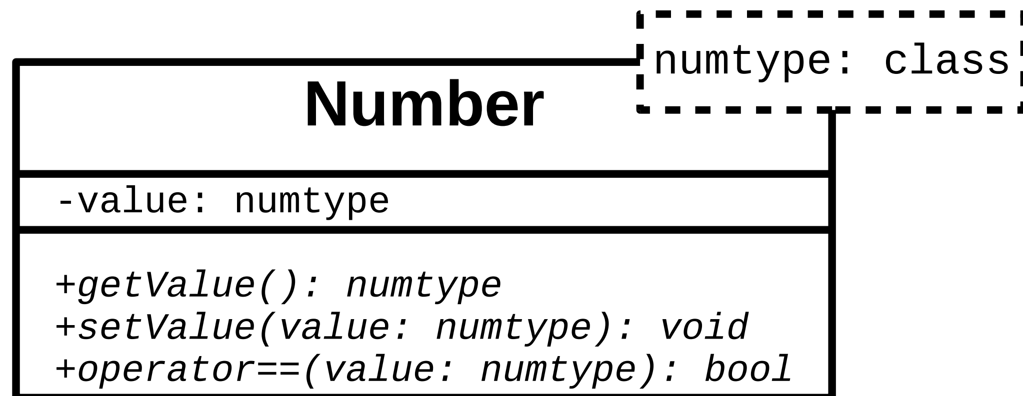
Extend Possibilities

- Templates and the STL



Template Examples

- Continue Example with templates:



- Avoid virtual calls for use of number objects
- Smaller objects

redhat

Results of Templates

- Advantages:
  - Code reuse easy
  - Efficient code through inlining
  - Optimization through specialization
    - Iteratively
- Disadvantages:
  - Code duplication per instantiation
  - Tricky to achieve type safety of template parameters

Reference: *What is Generic Programming?* Gabriel Dos Reis and Jaakko Järvi, 2005

redhat

ISO C++ 1998 Problems

- Too often temporaries
- Type safety of template parameters
  - Implemented using traits
  - Impossible to decode error messages

$$s = \sum_i (\vec{a} \times f + \vec{b} - \vec{c})_i - \sum_i (\vec{d} \times g + \vec{e} \times h)_i$$

redhat

ISO C++ 1998 Problems

```
template<typename F, int N>
struct vec {
   F e[N];
   F &operator[] (size_t idx) { return e[idx]; }
   F operator[] (size_t idx) const { return e[idx]; }
};
template<typename F, int N>
vec<F,N>
operator+(const vec<F,N> &src1, const vec<F,N> &src2) {
  vec<F,N> res;
  for (int i=0;i<N;++i) res[i]=src1[i]+src2[i];
  return res;
}
```

ISO C++ 1998 Problems

```
template<typename F, int N>
vec<F,N>
operator-(const vec<F,N> &src1, const vec<F,N> &src2) {
   vec<F,N> res;
   for (int i=0; i<N; ++i) res[i]=src1[i]-src2[i];
   return res;
}
template<typename F, int N>
vec<F,N> operator*(const vec<F,N> &src, F f) {
  vec<F,N> res;
  for (int i=0; i<N; ++i) res[i]=src[i]*f;
  return res;
}
```

redhat

ISO C++ 1998 Problems

```
template<typename F, int N>
F sumvec(const vec<T,N> &src) {
   F res = 0.0;
   for (int i=0; i<N; ++i) res += src[i];
   return res;
}


{ ...
   s = sumvec(a * f + b - c) – sumvec(d * g – e * h);
   ...
}
```

redhat

C-Style Memory Management

```
template<typename F, int N>
struct vec {
   F e[N];
   F &operator[] (size_t idx) { return e[idx]; }
   F operator[] (size_t idx) const { return e[idx]; }
};
template<typename F, int N>
void addvec(vec<F,N> &dst, const vec<F,N> &src1,
            const vec<F,N> &src2) {
  for (int i=0;i<N;++i) dst[i]=src1[i]+src2[i];
}
```

C-style Memory Management

```
template<typename F, int N>
void subvec(vec<F,N> &dst, const vec<F,N> &src1,
            const vec<F,N> &src2) {
   for (int i=0; i<N; ++i) dst[i]=src1[i]-src2[i];
}
template<typename F, int N>
void scalevec(vec<F,N> &dst, const vec<F,N> &src, F f) {
  for (int i=0; i<N; ++i) dst[i]=src[i]*f;
}
```

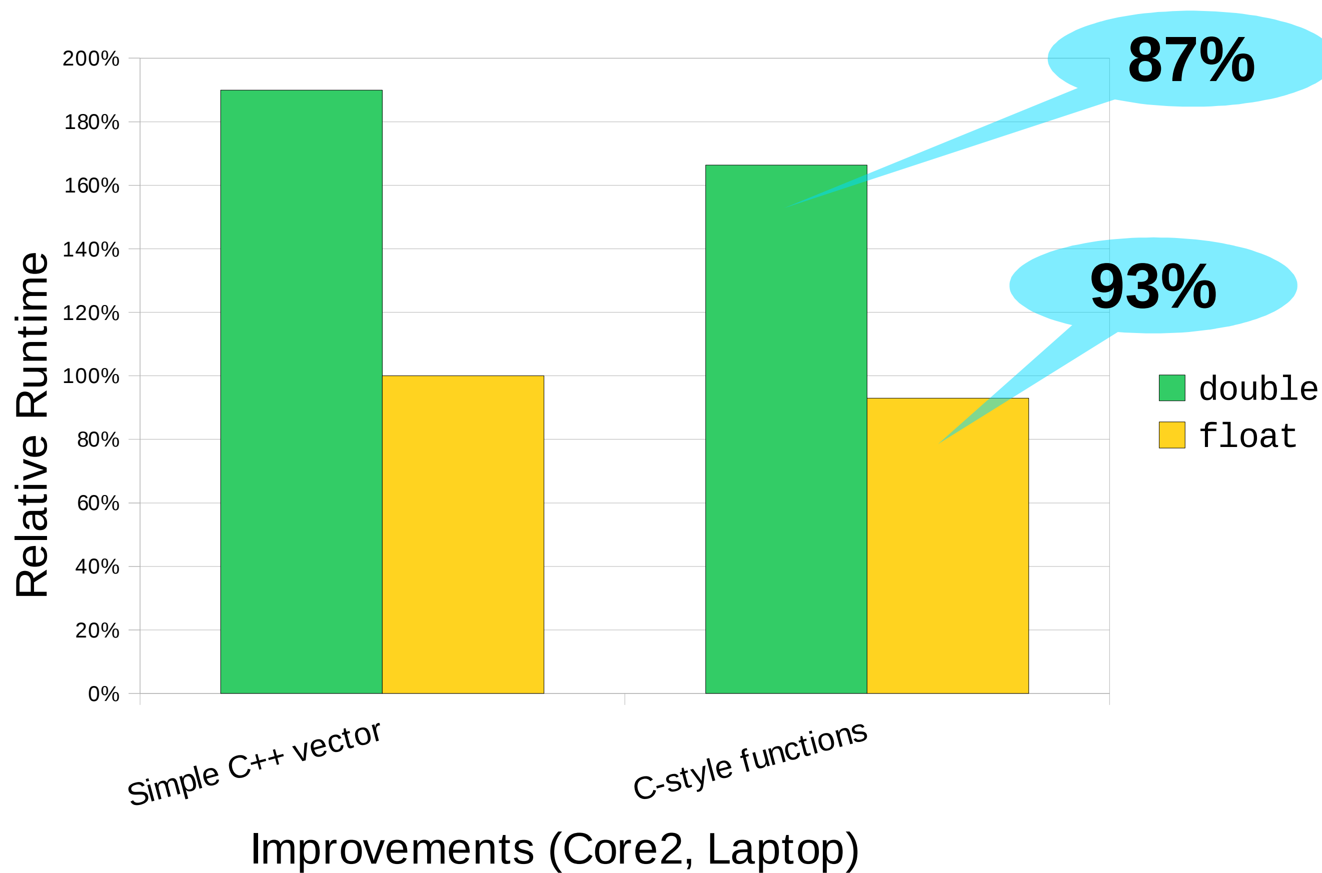
C-style Memory Management

```
template<typename F, int N>
F sumvec(const vec<T,N> &src) {
   F res = 0.0;
   for (int i=0; i<N; ++i) res += src[i];
   return res;
}
{  vec<float,100000> t;
   scalevec(t,a,f); addvec(t,t,b); subvec(t,t,c);
   s = sumvec(t);
   scalevec(t,d,g); addvec(t,t,e);
   s -= sumvec(t);
   ...
}
```

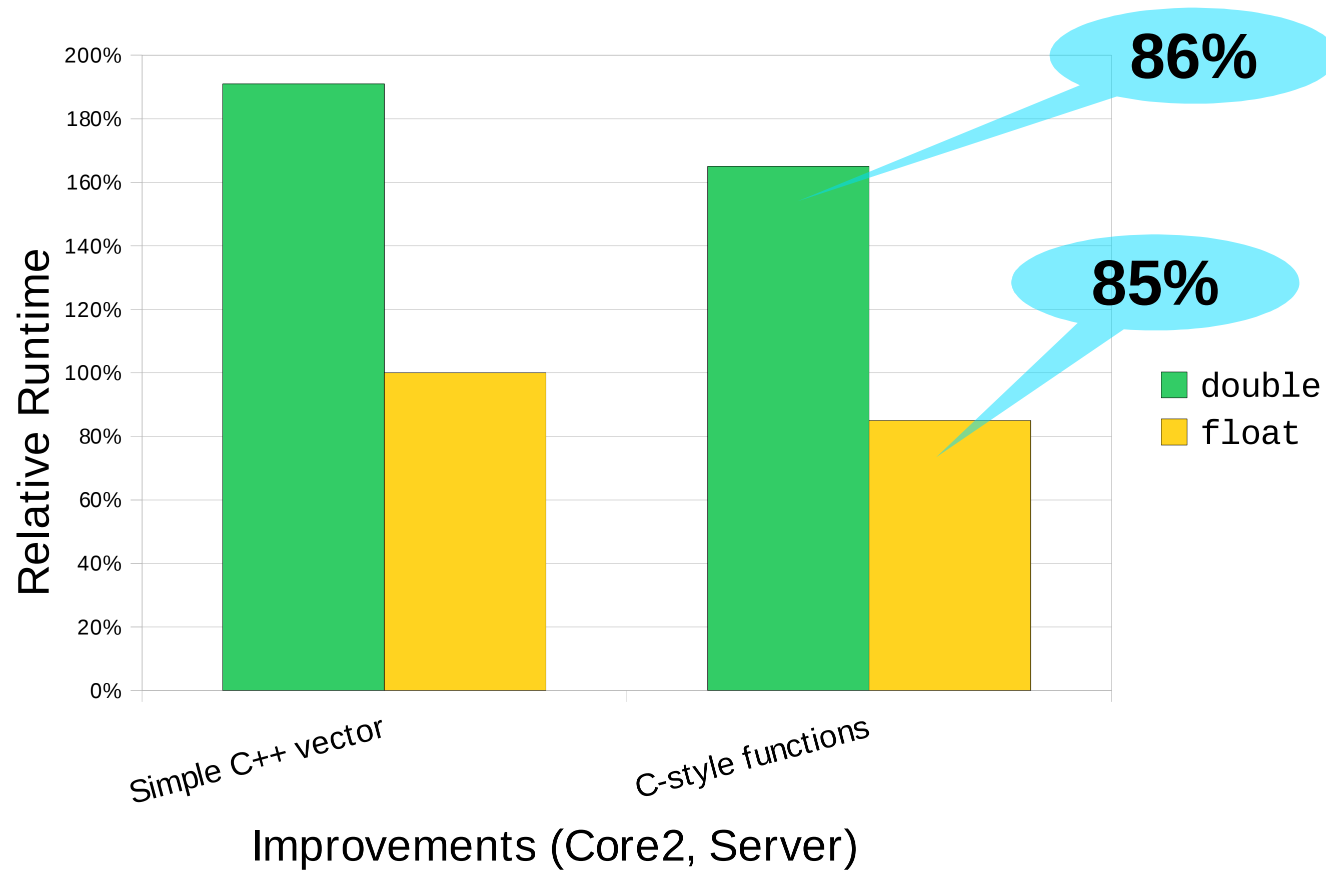
UGLY!

redhat

Ugly&Unmaintainable, But Faster

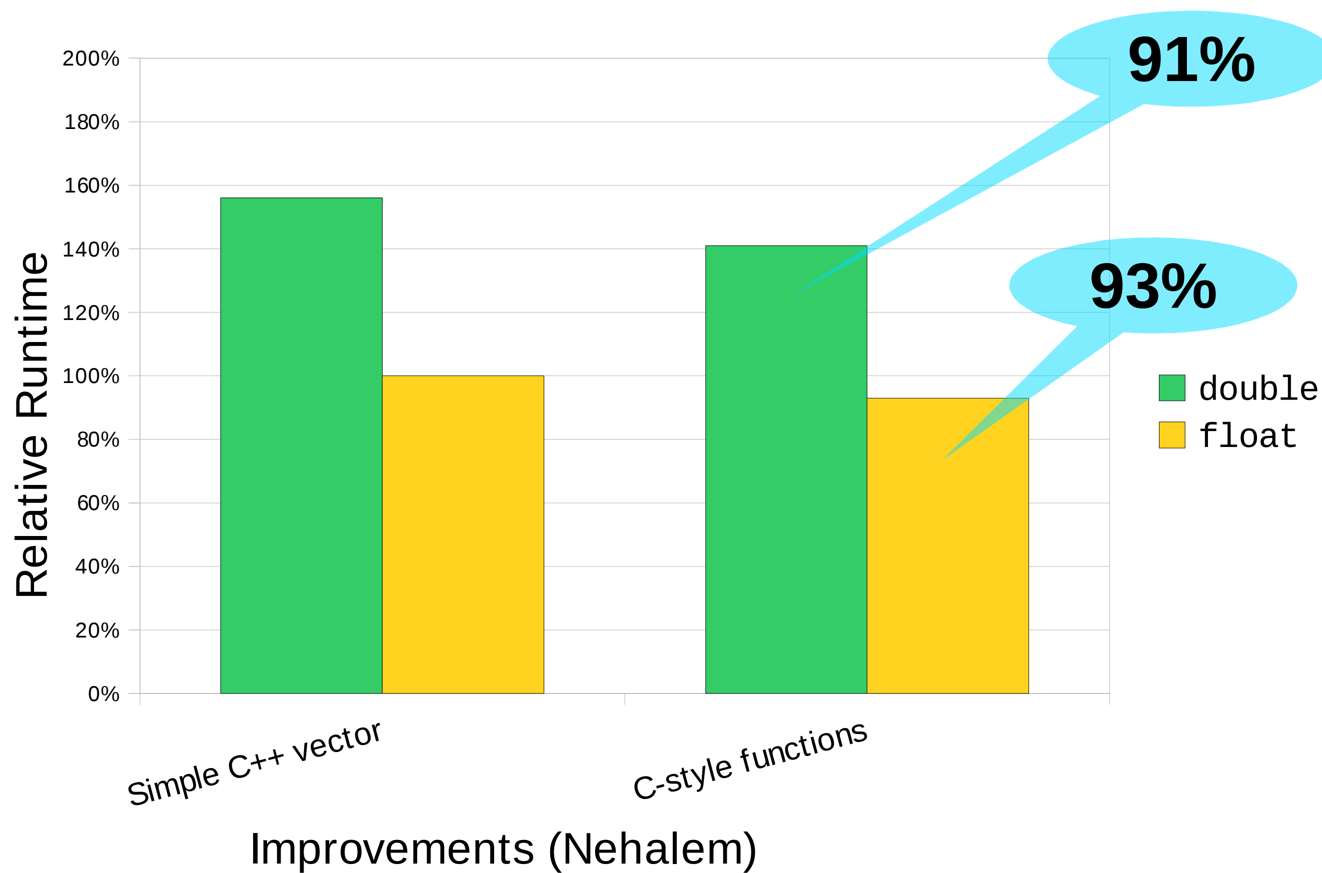


Ugly&Unmaintainable, But Faster



redhat

Ugly&Unmaintainable, But Faster



Move Semantic/rvalue References

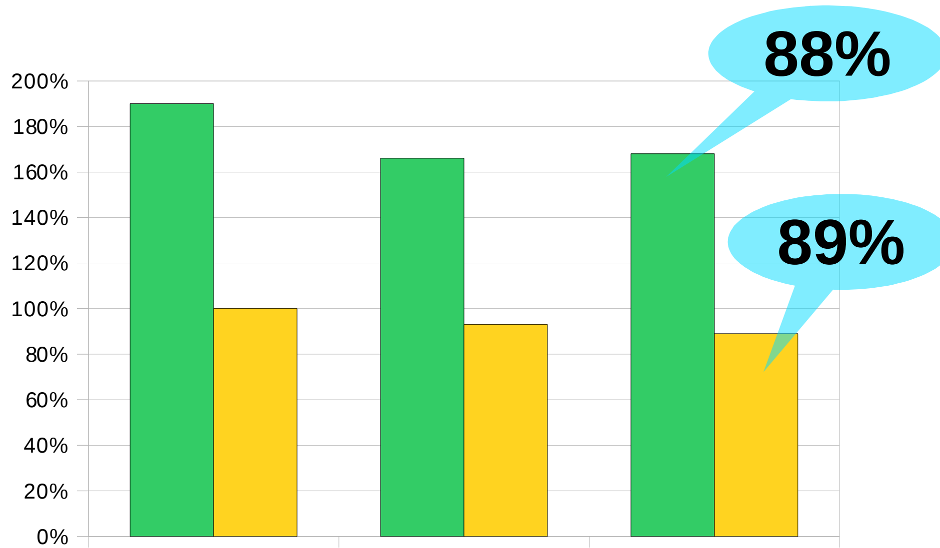
```
template<typname F, int N>
vec<F,N> &&operator+(vec<F,N> &&src1, const vec<F,N> &src2) {
   for (int i=0; i<N; ++i) src1[i] += src2[i];
   return src1;
}

template<typename F, int N>
vec<F,N> &&operator-(vec<F,N> &&src1, const vec<F,N> &src2) {
   for (int i=0; i<N; ++i) src1[i] -= src2[i];
   return src1;
}
```

Move Semantics/rvalue References

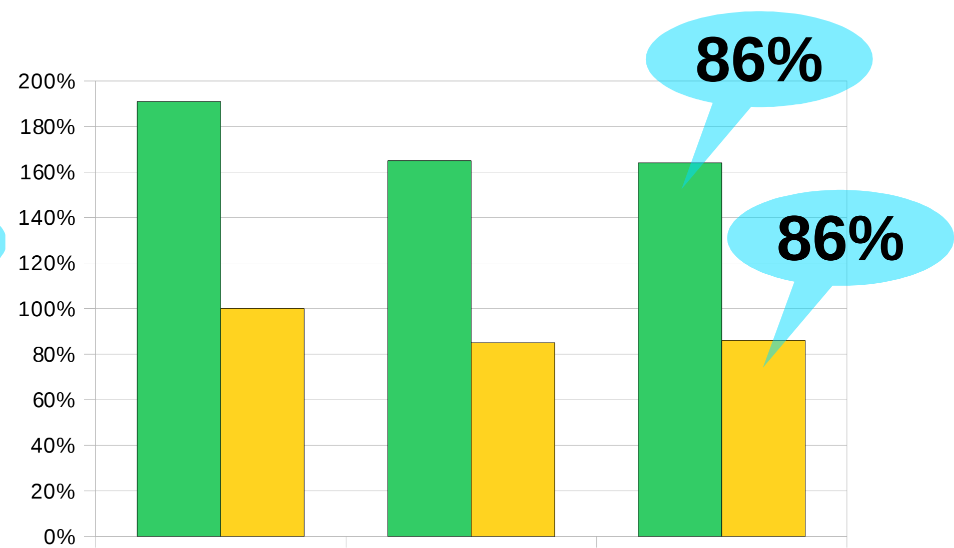
```
{ ...
   s = sumvec(a * f + b - c) – sumvec(d * g – e * h);
   ...
}
```

Identical to simple C++ vector type

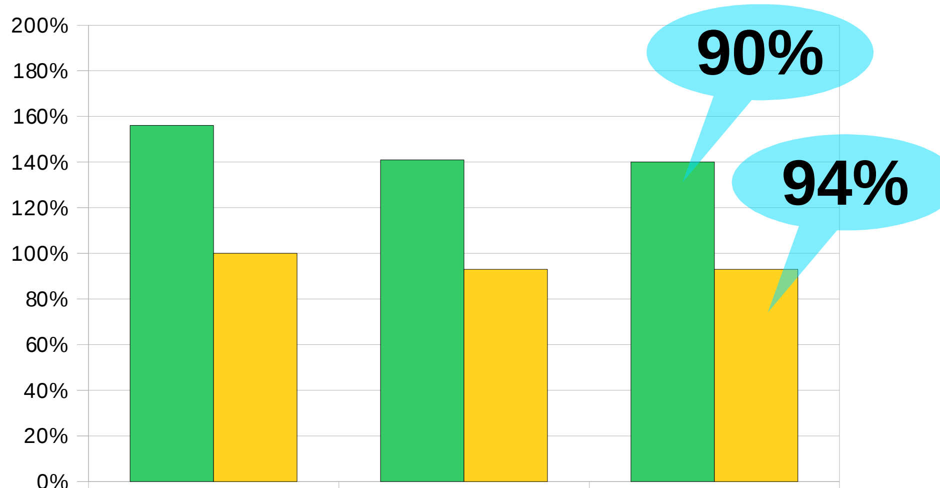
Usability & Performance



Improvements (Core2, Laptop)



Improvements (Core2, Server)



Improvements (Nehalem)

Using R-Value References

redhat

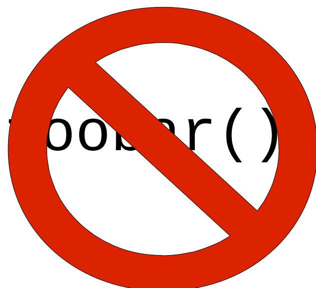
Optimization Phase

- All code concentrated in headers and library functions
- Code for library users trivial
- Full control over memory handling
- Apply additional optimizations. For example:
  - Combine operation
  - Vectorization
  - Parallelization (not shown)

C++ Template Problem

- Class lookups cut short with templates:

```
template <class f> struct foo {
   ...
};
template <class f> struct foobar {
   ...
   operator foo<f>() { ... }
};
void fct(foo &e);
                                ...fct(foo(foobar()))...
   ...  fct(foobar())  ...
```



redhat

Combination

Additional class:

```
#define F float
#define N 100000
struct vecscale {
   const vec &v;
   F f;
   vecscale(const vec &va, F fa) : v(va), f(fa) {}
};
```

redhat

Combination

Additional operators:

```
vecscale operator*(const vec &src, F f) {
   return vecscale(src, f);
}

vec operator+(const vecscale &src1, const vec &src2) {
   vec dst;
   for (int i=0; i<N; ++i)
      dst[i] = src1.v[i] * src1.f + src2[i];
   return dst;
}
```

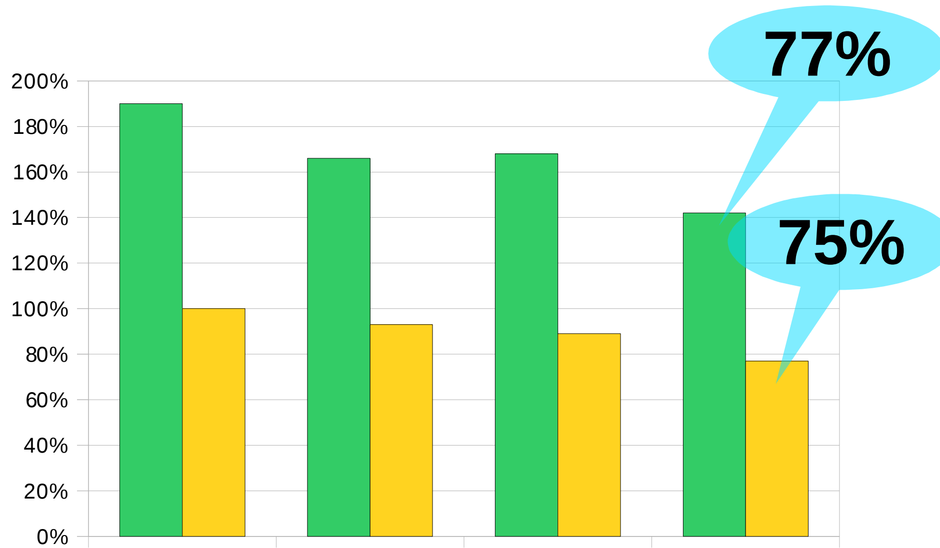
Combination

```
{ ...
   s = sumvec(a * f + b - c) – sumvec(d * g – e * h);
   ...
}
```

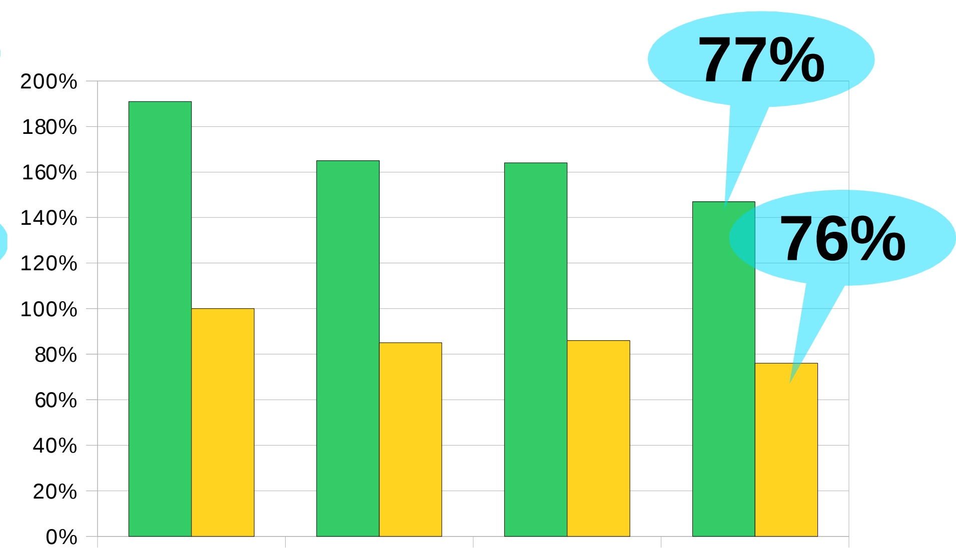
Still not changed!

redhat

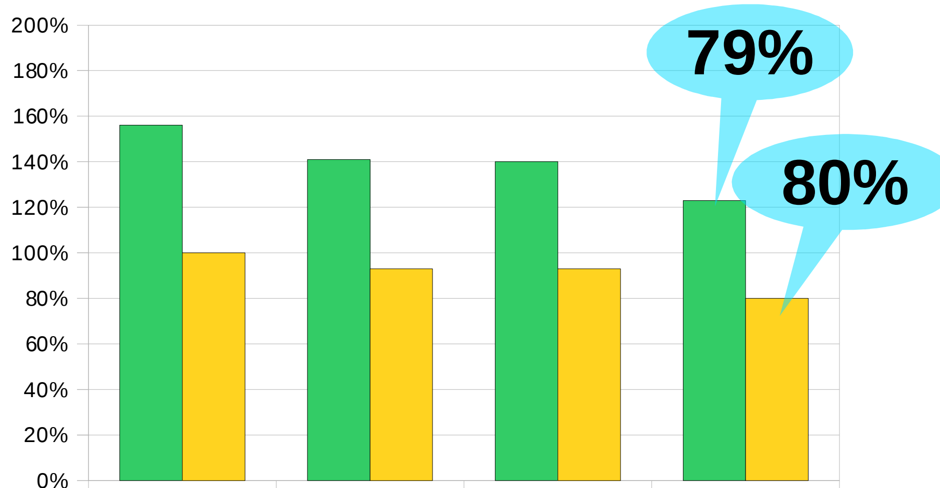
And Better...



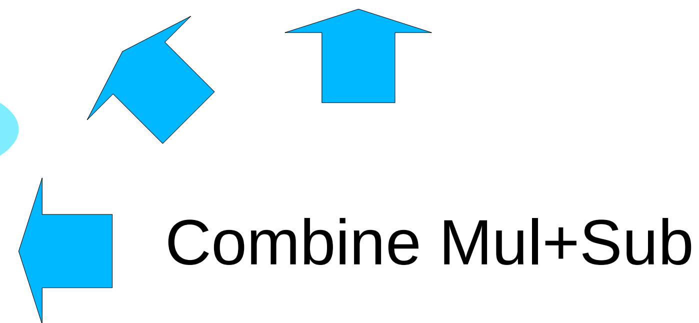
Improvements (Core2, Laptop)



Improvements (Core2, Server)



Improvements (Nehalem)



Combine Mul+Sub

Why Stop There?

Yet another class:

```
struct vecscalesub {
   const vec &v1;
   F f;
   const vec &v2;
   vecscalesub(const vec &va1, F fa, const vec &va2)
   : v1(va1), f(fa), v2(va2) {}
   operator vec() const;
};
```

redhat

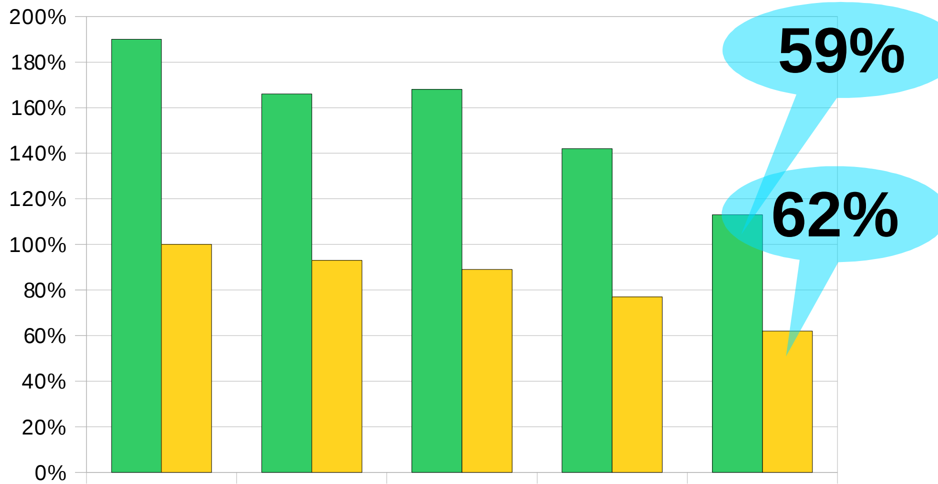
Why Stop There?

```
vecscalesub operator-(const vecscale &src1,
                      const vec &src2) {
   return vecscalesub(src1.v, src1.f, src2);
}
vecscalesub operator-(const vecscale &src1,
                      const vecscale &src2) {
   return vecscalesub(src1.v, src1.f, vec(src2));
}
vecscalesub::operator vec() const {
   vec dst;
   for (int i=0; i<N; ++i)
      dst[i] = v1[i] * f - v2[i];
   return dst;
}
```

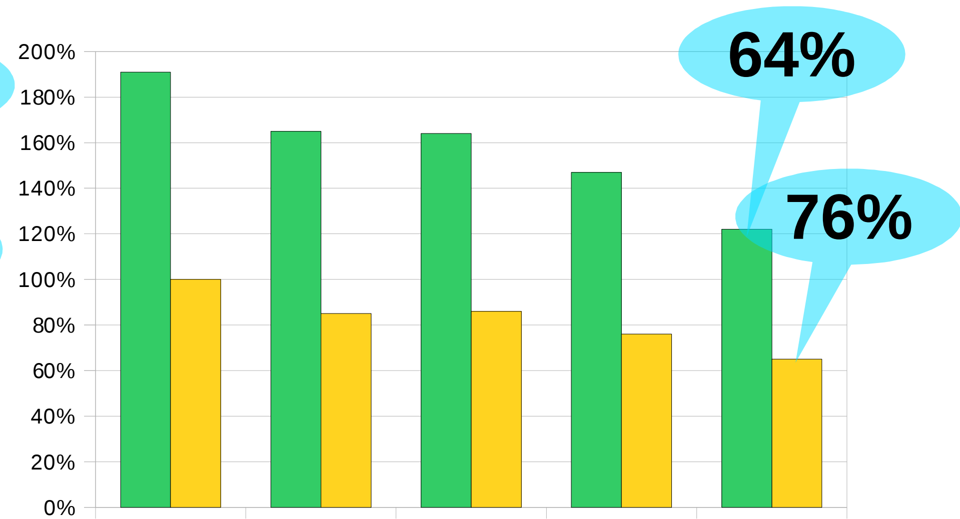
Why Stop There?

```
{ ...
   s = sumvec(a * f + b - c) – sumvec(d * g – e * h);
   ...
}
```

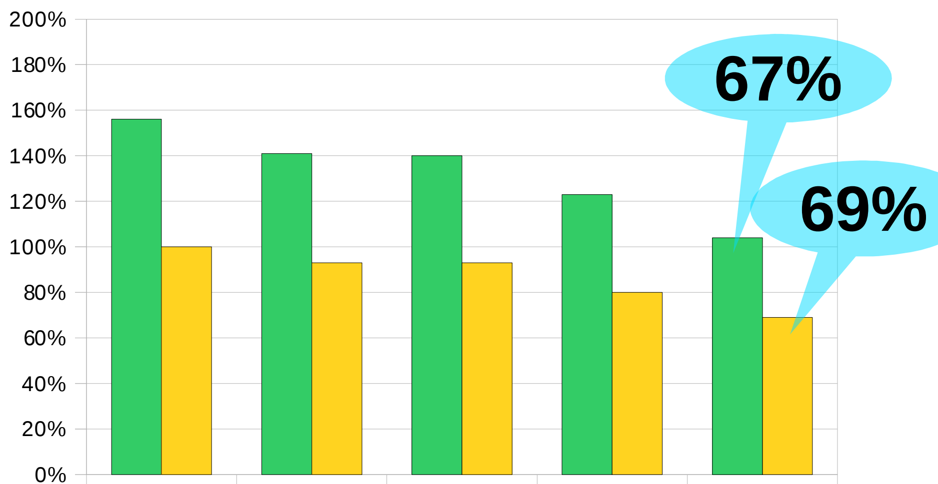
...and Better...



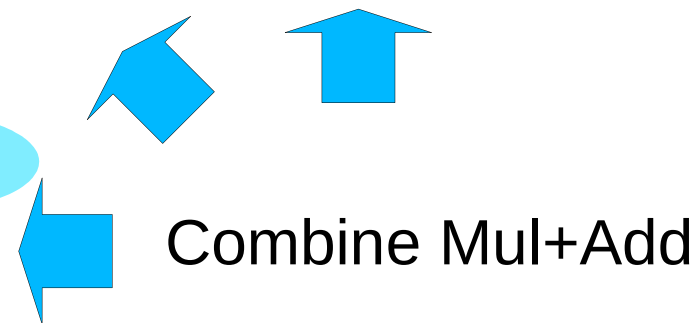
Improvements (Core2, Laptop)



Improvements (Core2, Server)



Improvements (Nehalem)



Combine Mul+Add

redhat

More Complex Operation

- Combine even more:

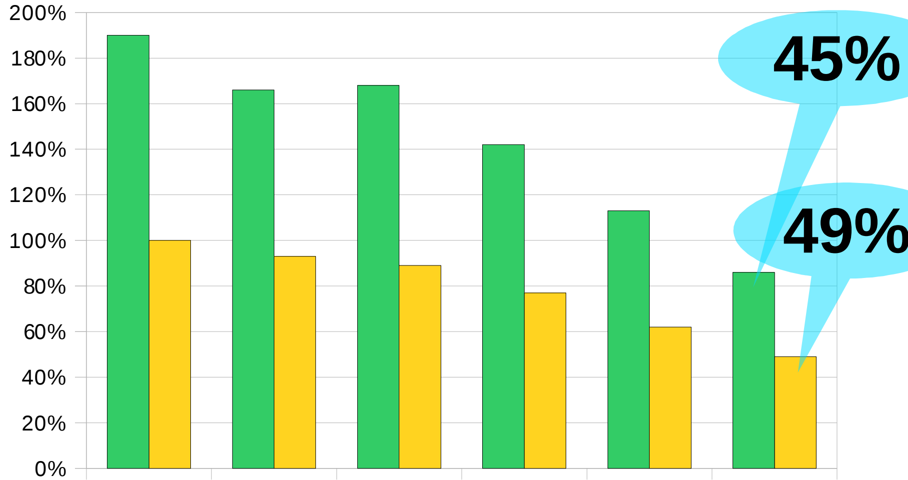
```
vec operator-(const vecscale &vs1,
              const vecscale &vs2) {
   vec res;
   for (int i=0; i<N; ++i)
      res[i] = vs1.v[i]*vs1.f-vs2.v[i]*vs2.f;
   return res;
}
```

Why Stop There?

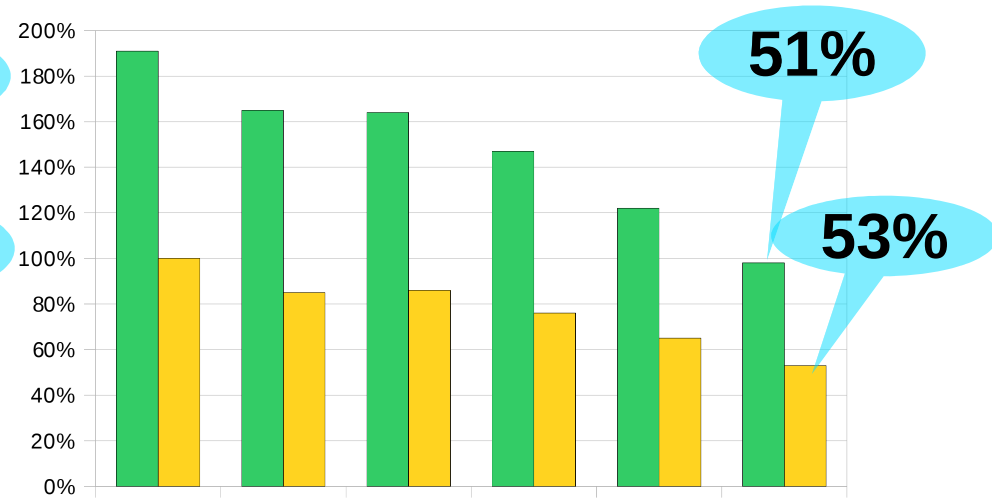
```
{ ...
   s = sumvec(a * f + b - c) – sumvec(d * g – e * h);
   ...
}
```

redhat

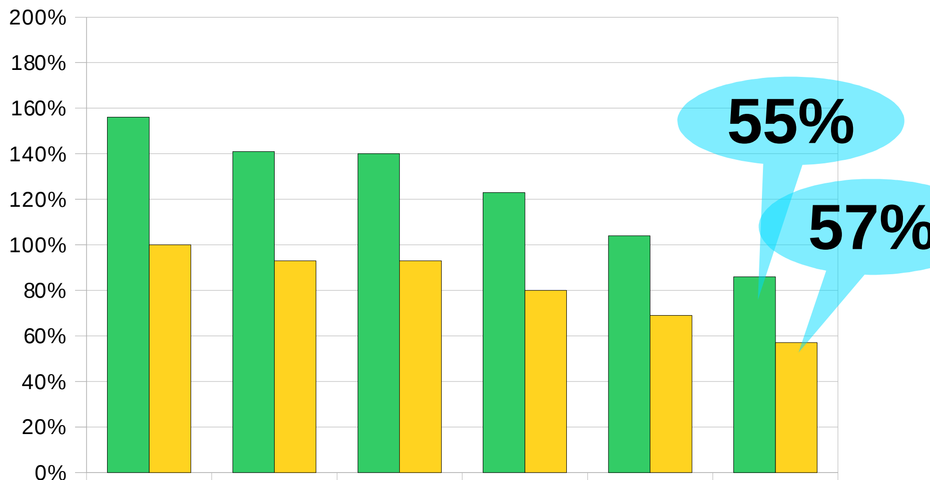
...and Better...



Improvements (Core2, Laptop)



Improvements (Core2, Server)



Improvements (Nehalem)

Combine Mul+Mul+Sub

Vectorization

```
struct vec {
   union {
      F e[N];
      __m128 m[N / 4];
      __m128d d[N / 2];
   };
   F &operator[](size_t n) { return e[n]; }
   F operator[](size_t n) const {return e[n]; }
};
```

Vectorization

```
vec operator-(const vecscale &src1,
              const vecscale &src2) {
   __m128 fv1 = _mm_set1_ps(src1.f);
   __m128 fv2 = _mm_set1_ps(src2.f);
   vec res;
   for (int i=0; i<N/4; ++i)
      res.m[i] = _mm_sub_ps(_mm_mul_ps(src1.v.m[i],
                                       fv1),
                            _mm_mul_ps(src2.v.m[i],
                                       fv2));
   return res;
}
```

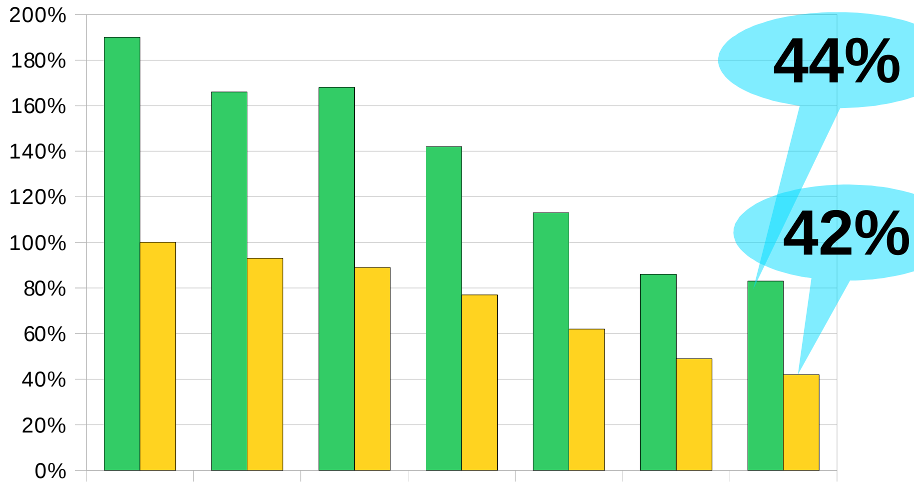
redhat

Vectorization

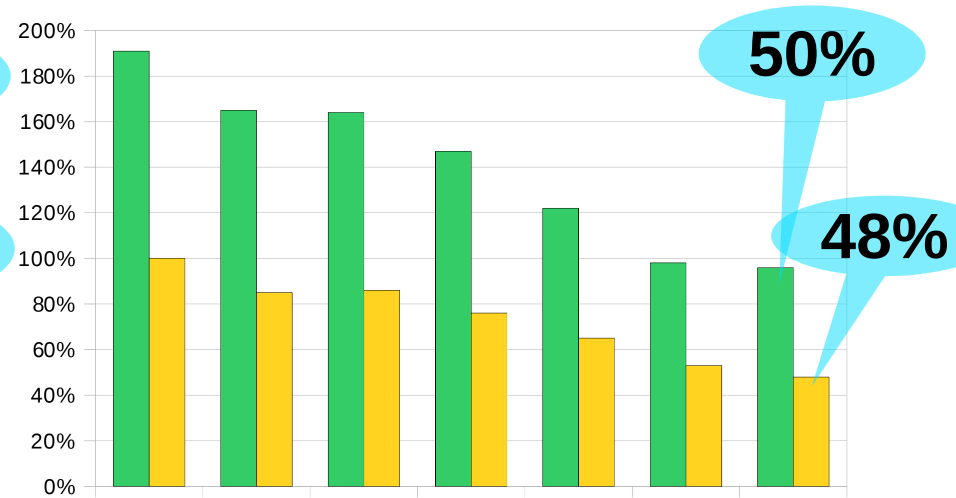
```
{ ...
   s = sumvec(a * f + b - c) – sumvec(d * g – e);
   ...
}
```

You guessed it: no change

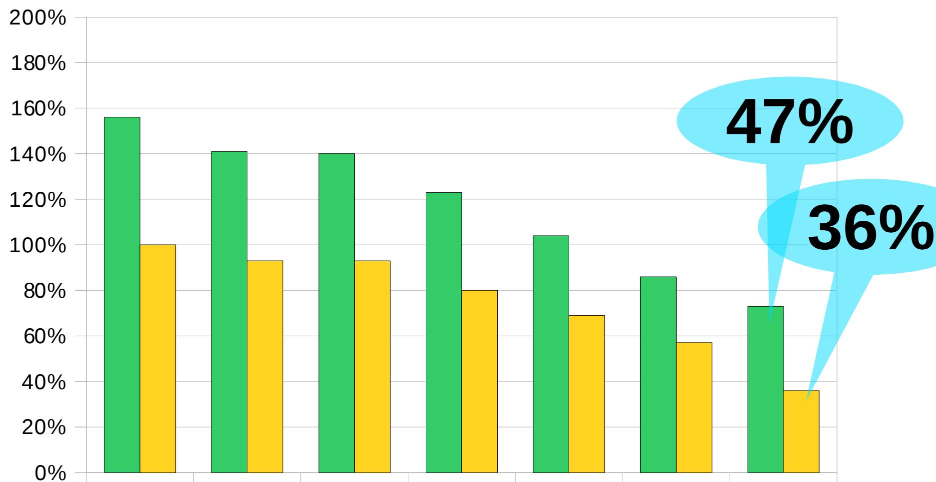
...and better



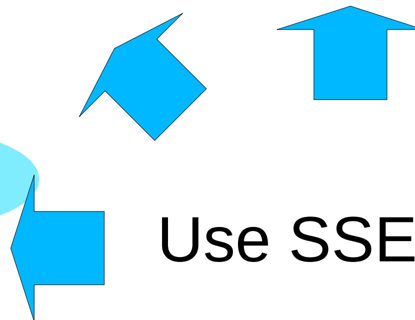
Improvements (Core2, Laptop)



Improvements (Core2, Server)



Improvements (Nehalem)



Use SSE

redhat.

More than ever

- Structuring program important
  - Recognize building blocks
  - Implement in library functions
  - Optimize, if necessary, by experts

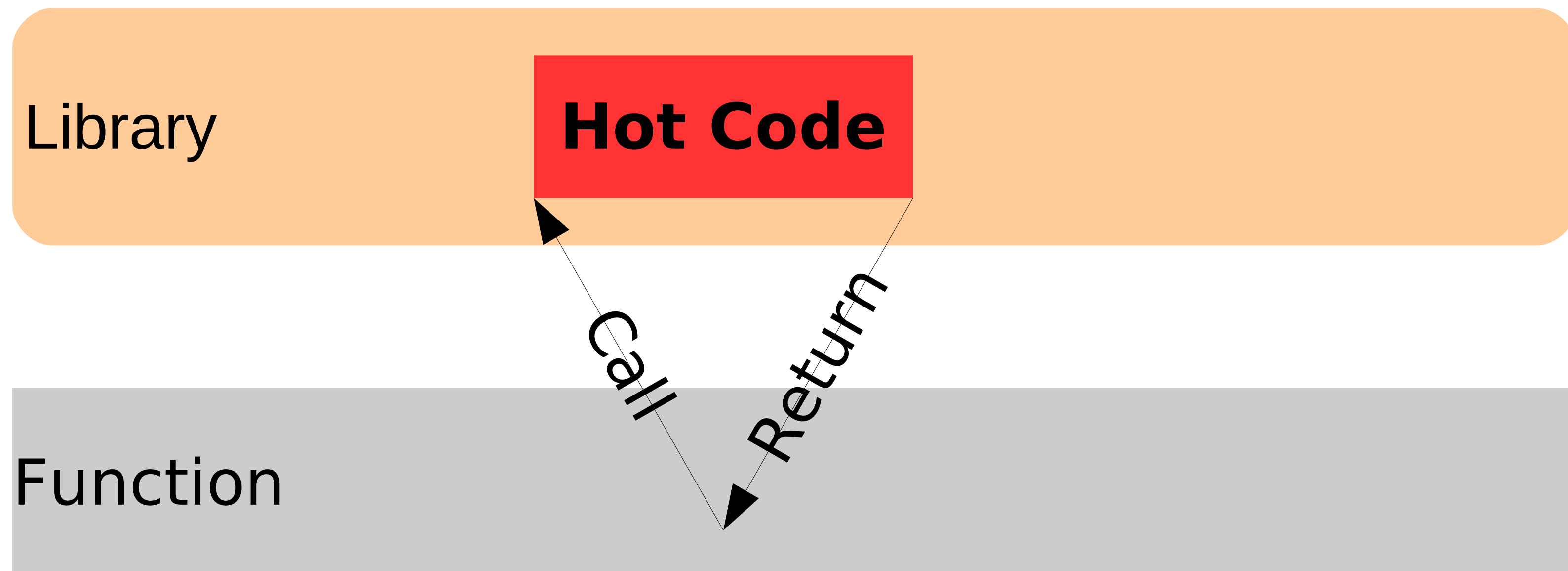
Function

Hot Code

redhat

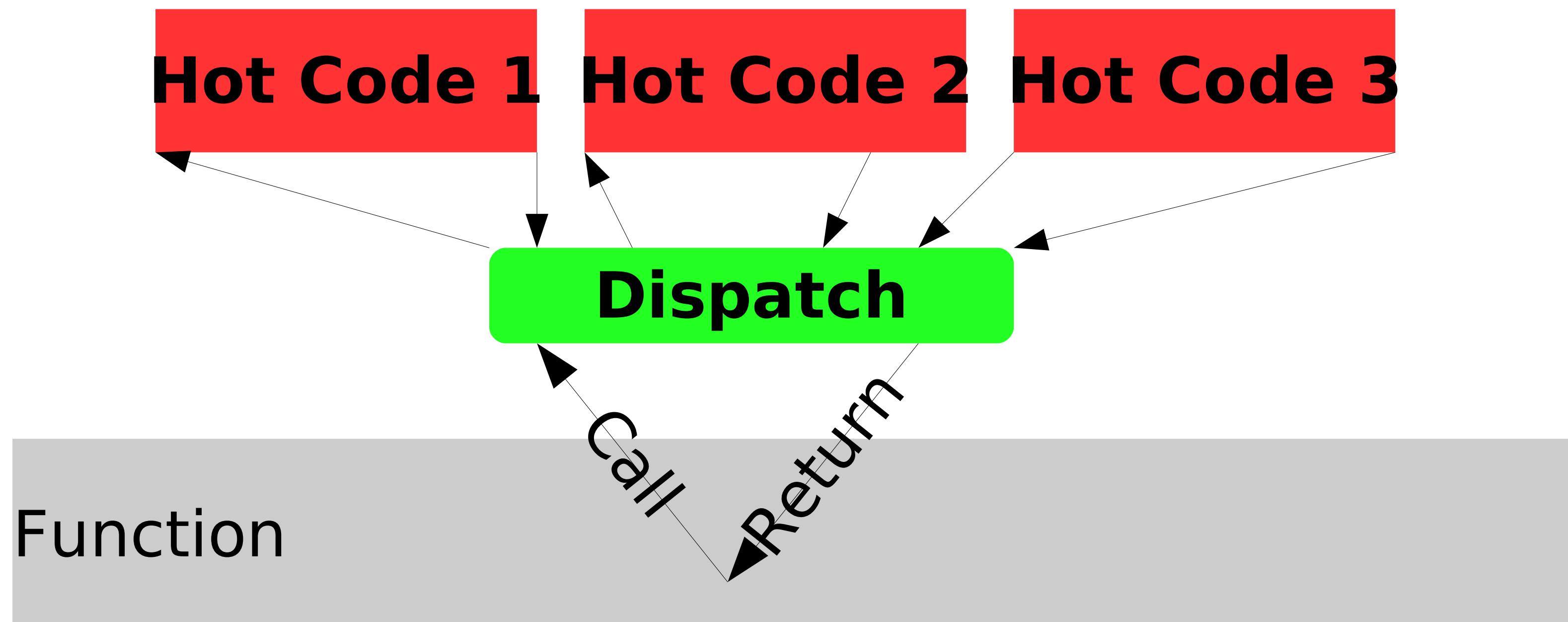
More than ever

- Structuring program important
  - Recognize building blocks
  - Implement in library functions
  - Optimize, if necessary, by experts



More than ever

- Structuring program important
  - Recognize building blocks
  - Implement in library functions
  - Optimize, if necessary, by experts



redhat

Functional Programming

- C++ closes in
  - Efficient way to describe functions
  - Stable interface
  - Even lambdas with closures available

```
void print_minmax(container<int> &a) {
  int max=0, min=0;
  a.iter([&max,&min](int v)
         {if (v<min) min=v; if (v>max) max= v; });
  cout<<”min=”<<min<<” max=”<<max<<endl;
}
```

redhat.

Benefits of this Approach

- Functional programming allows easy optimization
  - Combine functions
- Combines functions run longer
  - Less startup cost

➔ Ideal starting point for accelerators

- GPGPUs, special processors, FPGAs
  - Code needs to be loaded and managed
  - Data must be transfered back and forth

Conclusion

- Functional programming better suited for iterative optimization
- C++ first-grade language for function programming
- Additional benefit: close to hardware
  - Can express lowlevel details
  - As close to direct management as possible
    - Memory is the main performance bottleneck
  - Easy to use all processor instructions
  - Easy to interface with other system components without wrappers

Questions?

drepper@redhat.com | people.redhat.com/drepper