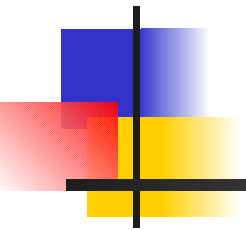
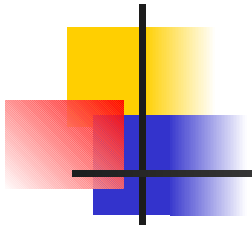


Blocking Techniques with Fast Expression Templates

Jochen Härdtlein (*haerdtlein@cs.fau.de*)

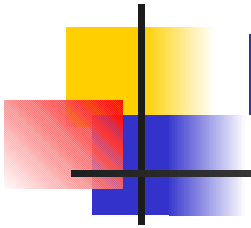
University of Erlangen-Nuremberg, Germany

Department of Computer Science 10



Starting Point

- Library: e.g., for solving PDEs
  - High-performance
  - Flexible in use
  - User-friendly interface
- C++
  - Provides flexibility, user-friendliness
  - Suffers from performance lacks



Implementation Policies

- Traditional operator overloading
  - Temporary objects
  - Memory-intensive data-transfer
- Expression Templates (ET)
  - Tricky to implement and unterstand
  - Overhead mainly for small vectors
  - Lacks in aliasing of pointers

Aliasing Problems with ET

- E.g., `c = a + b * a;`

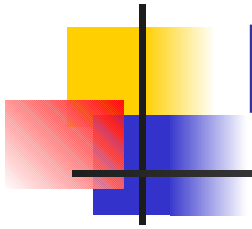
```
Expr<Add<Vector,
        Expr<Mult<Vector,Vector> > > >
```

- Should be inlined to

```
c.data[i] = a.data[i] + b.data[i] * a.data[i];
```

- Could just be converted to something like

```
c.data[i] = x.data[i] + y.data[i] * z.data[i];
```



Fast Expression Templates

- Enhance Expression Templates
  - Enumeration of vectors by template integer
  - Declare all data and functions as static
  - Call evaluation on expression types
  - Derive vector and operating classes from wrapper class
- Changes for users
  - Template enumeration of all variables

The Interface

- Wrapper class
  - Parent to basic class
  - Parent to operating classes

```
template <class A>
struct Expr { };
  const A& a_;
 public:
  Expr(const A& a) : a_(a) {}
  double at(int i) {return a_.at(i);}
};
```

The Basic Class

```
template <int num>
class Vector  : public Expr <Vector<pos> > {
  static double * data_; int N_;
 public: // ... Constructors, destructor
  static double at(int i) {return data_[i];}
  template <class A>
  void operator= (const Expr<A>&  v){
    for(int i=0; i<N_; ++i)
      data_[i] = A::at(i);
  }
};
 template <int num> double* Vector<num>::data_;
```

The Operating Class for “+”

```
template <class A, class B>
struct Add : public Expr<Add<A,B> > {
  static double at(int i) {return A::at(i) + B::at(i);}
};
```

```
// Only one operator +
template <class A, class B>
 Add<A,B>
 operator + (const Expr<A>& a, const Expr<B>& b){
    return Add<A,B>();
 }
```

Application of FET

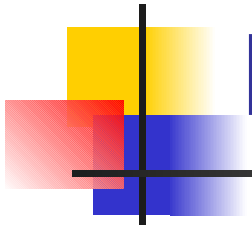
- Easy math-like usage

```
Vector<1> a; Vector<2> b; Vector<3> c;
c = a + b * a;
```

```
Add<Vector<1>, Mult<Vector<2>,Vector<1> > >
```

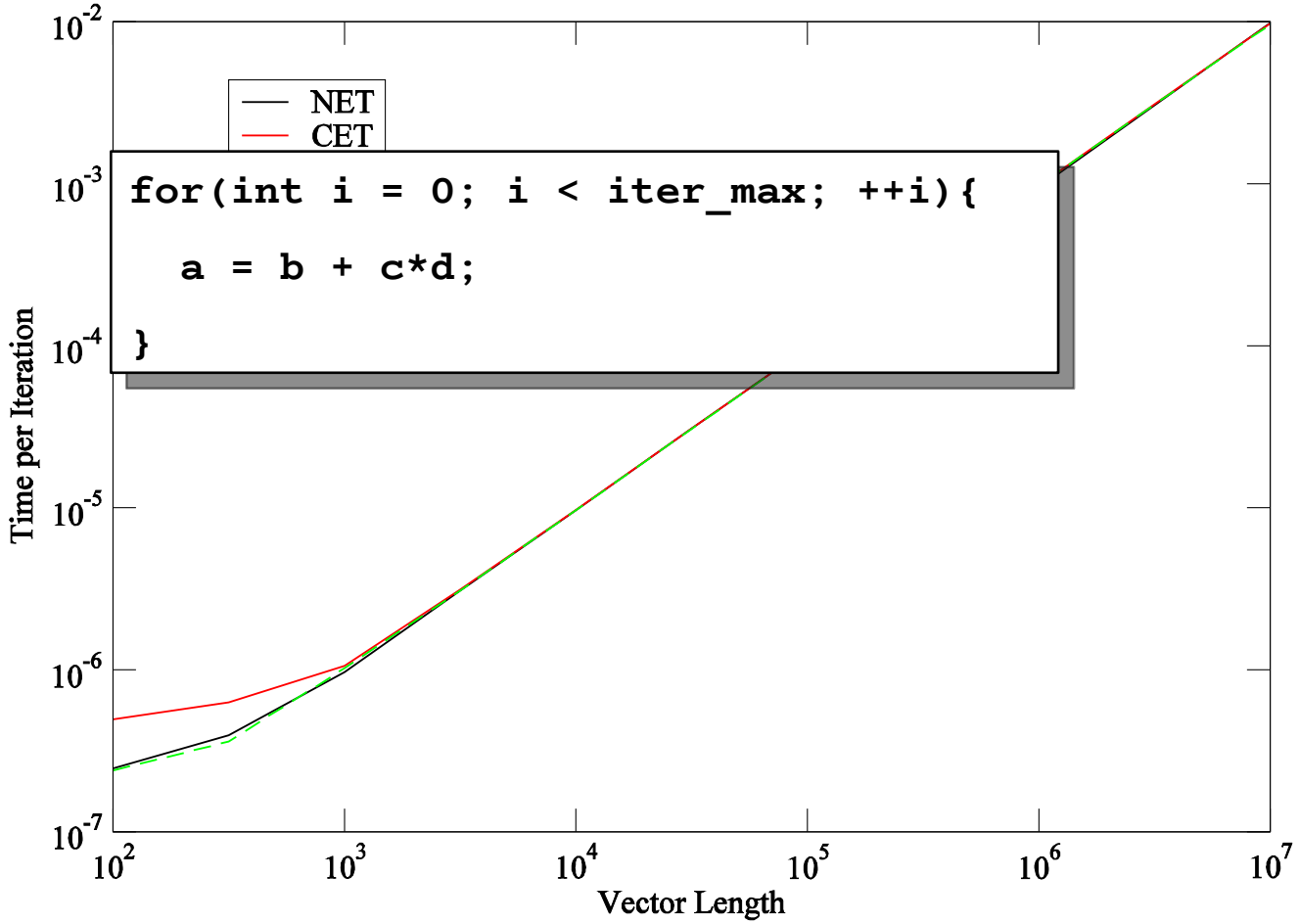
Is really inlined to

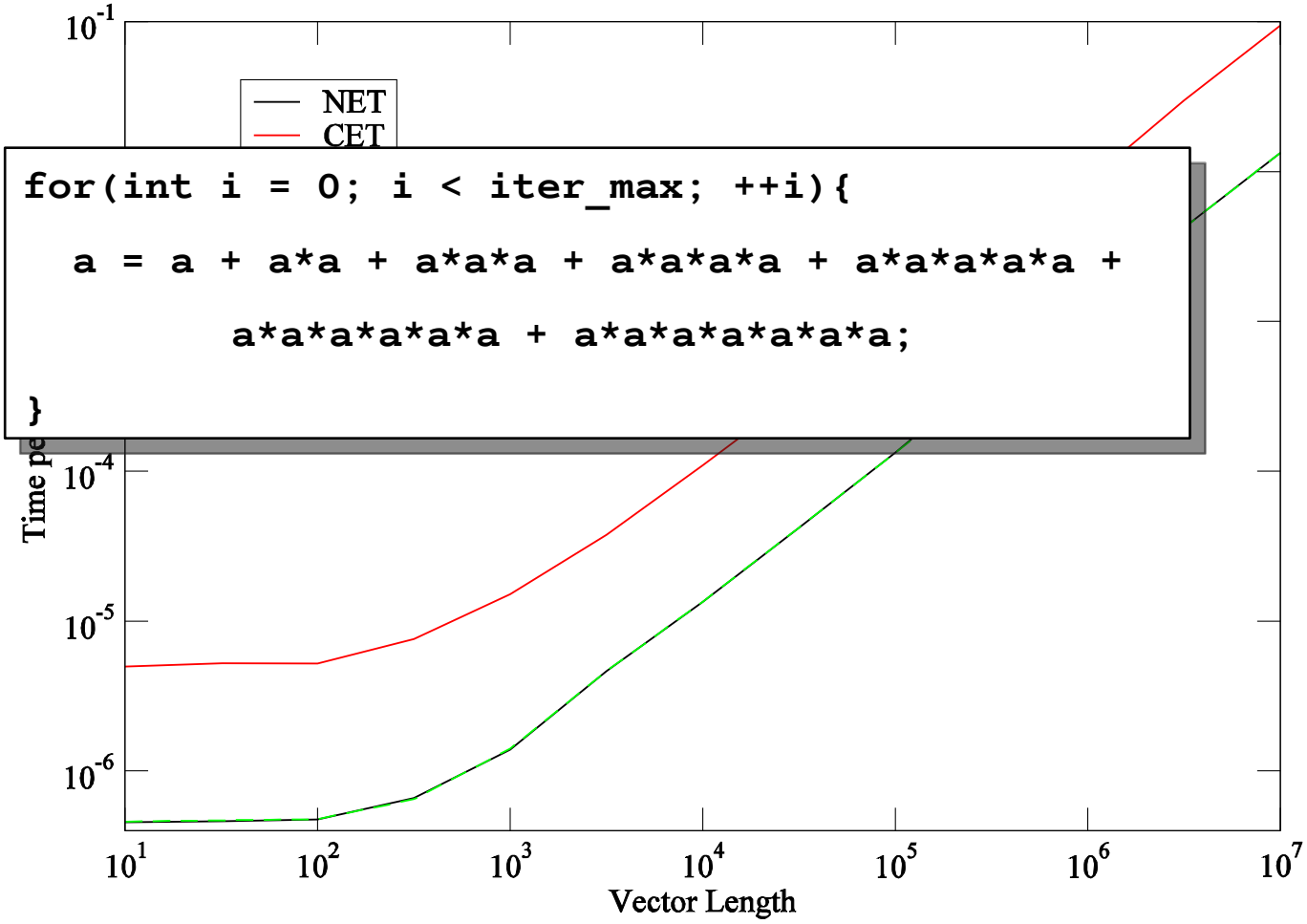
```
c.data_[i] =  a.data_[i] +
              b.data_[i] * a.data_[i];
```



Performance Results

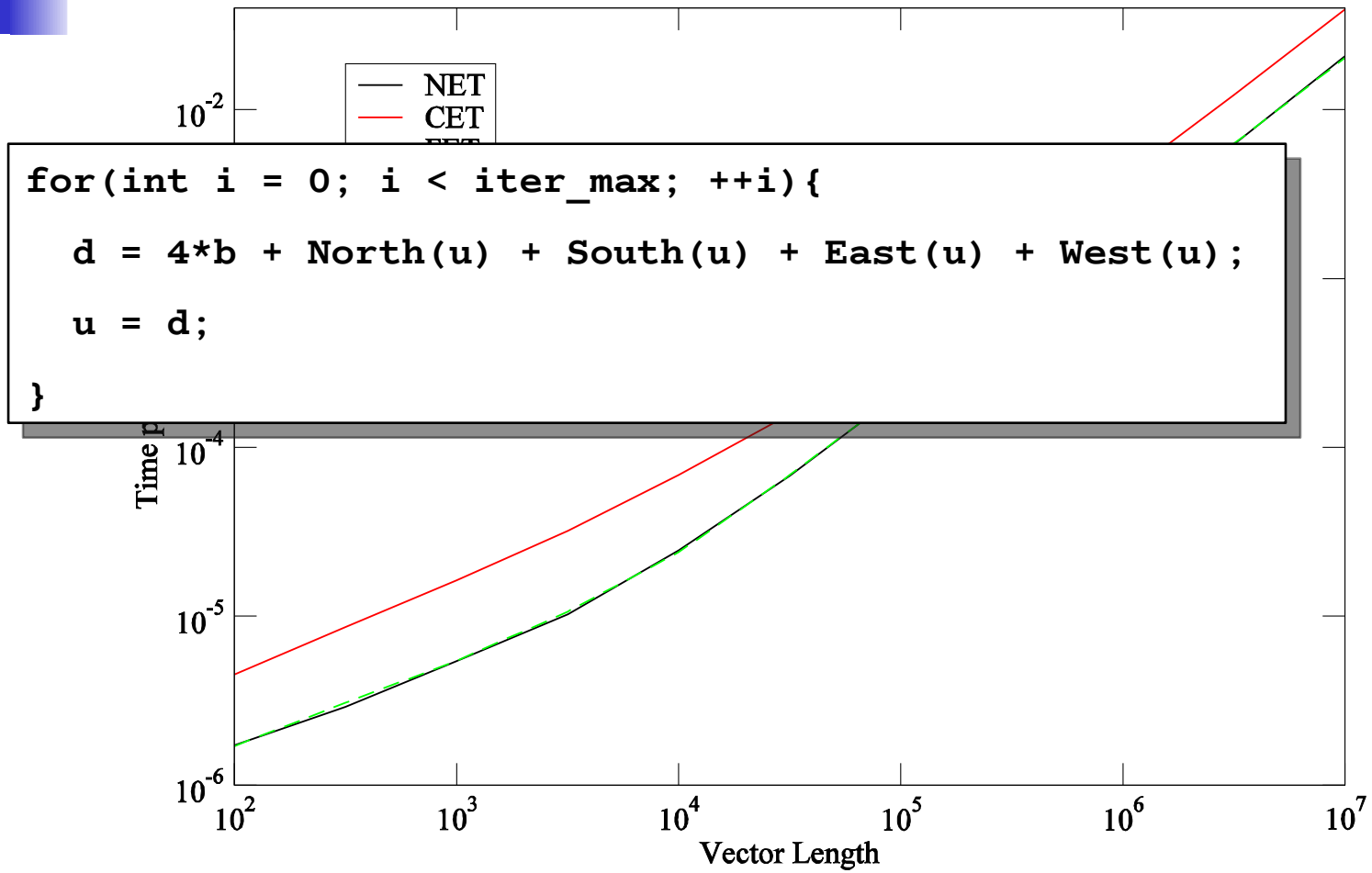
- Platforms
  - Intel Pentium 4, using Gnu compiler
  - Hitachi SR8000 pseudo vector machine
  - NEC SX 6 vector machine
  - AMD Opteron Cluster, Intel Compiler 8.1, using OpenMP


NET
CET
for(int i = 0; i < iter_max; ++i){
a = b + c*d;
}
Time per Iteration
Vector Length
10^-2
10^-3
10^-4
10^-5
10^-6
10^-7
10^2
10^3
10^4
10^5
10^6
10^7




```
for(int i = 0; i < iter_max; ++i){
  a = a + a*a + a*a*a + a*a*a*a + a*a*a*a*a +
      a*a*a*a*a*a + a*a*a*a*a*a*a;
}
```

Solving a 2D PDE: Poisson's Problem with Finite



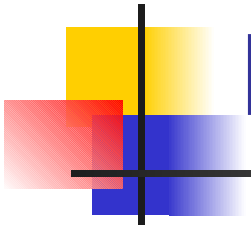

```
for(int i = 0; i < iter_max; ++i){
  d = 4*b + North(u) + South(u) + East(u) + West(u);
  u = d;
}
```

Blocking with FET

- Separator between loops caused by nested template constructs

```
for(int i = 0; i < iter_max; ++i)
  u = c*f + North(u) + South(u) + East(u) + West(u);
```

```
for(int i = 0; i < iter_max; ++i)
  u.operator=(
   Add<CMult<Vector<2> >,
        Add<North<Vector<1> >, Add<South<Vector<1> >,
        Add<East<Vector<1> >, West<Vector<1> > > > > x)
```



Delay of Evaluation

Store the types for evaluation

```
template <class A, class B>
  struct Evaluate { };
```

Create evaluation object at assignment

```
template <class A>
 inline Evaluate<Vector<num>, A>
  operator== (const Expr<A>& a){
     return Evaluate<Vector<num>,A>();
  }
```

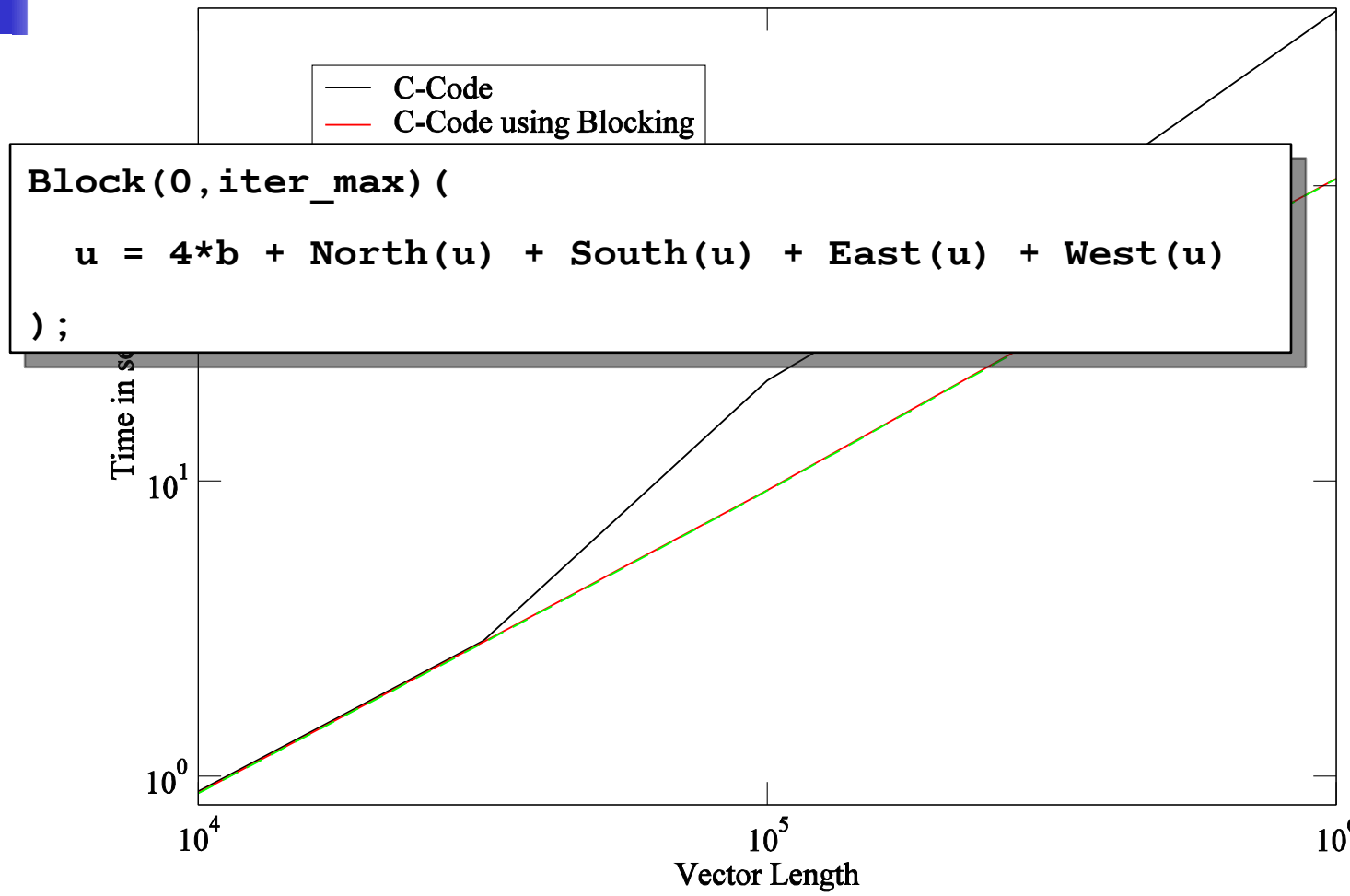
Fusion of Loops

```
template <class A>
class Loops{
 protected:
  static int o_beg, o_end, i_end;
 public:
  Loops(int b, int e);
  template <class B, class C>
  inline void operator() (const Evaluate<B,C>& ev){
    i_end = B::size();
    A::template iteration<B,C>();
  }
};
```

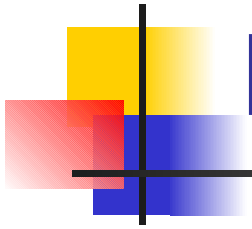
Defining Loops

```
struct For : public Loops<For>{
  For(int b, int e) ;

  template <class A, class B>
  static inline void iteration(){
    for(int k = o_beg; k < o_end; ++k)
      for(int i = 0; i < i_end; ++i)
        A::Set(i) = B::Give(i);
  }
};
```

Poisson's Problem with Finite Elements (Blocking)

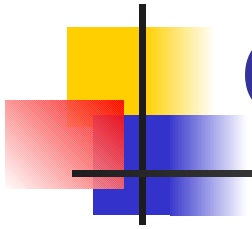



```
Block(0,iter_max)(

  u = 4*b + North(u) + South(u) + East(u) + West(u)

);
```



Further Advantages

- defining several loops
- compile-time counting of variables using typelists
- expression-dependent computing of blocksizes



Conclusions and Further Work

- FET perform better than classical ET
- Easier to implement and understand
- Enable loop modification techniques

- Self-acting enumeration
- COLSAMM