# Efficient implementation of polynomial arithmetic in a multiple-level programming environment

**Xin Li & Marc Moreno Maza**

ORCCA

**September 2, 2006**

**ICMS'2006, Castro Urdiales, SPAIN.**

# Motivations (I)  Generic Code

**Need for generic code of polynomial arithmetic.**

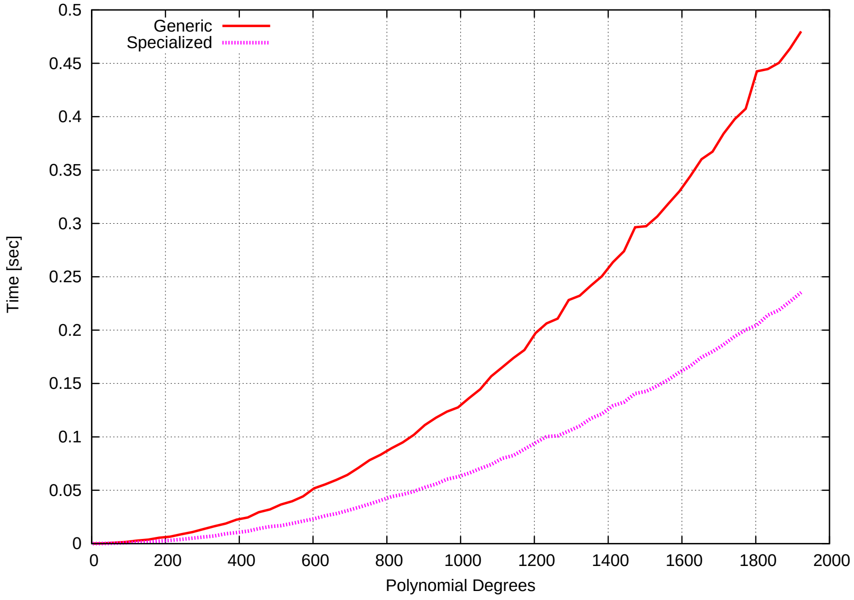
- **Suited to generic programming.**
- **Code reuse, maintainability.**
- **Reasonable speed after compiler optimization.**

**We use *AXIOM* and** ALDOR**.**

- **Parametric polymorphism.**
- **Dependent type.**

# Motivations (II)    Specialized Code

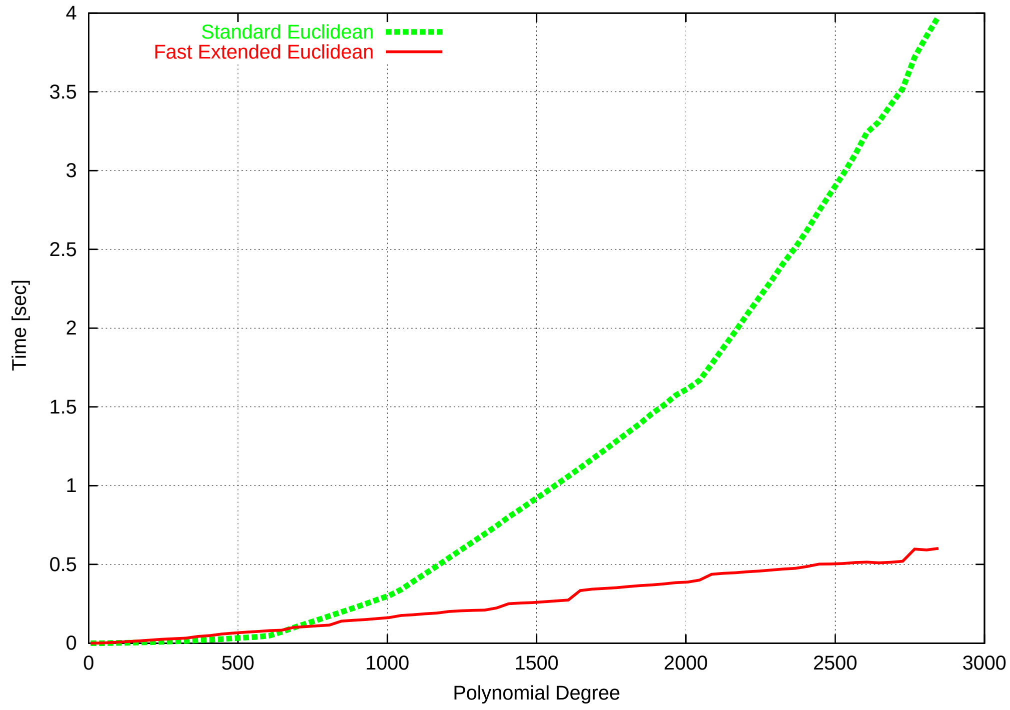
- **“Particular” domains deserve their own specialized implementations.**
- **More efficient.**




**Univariate Generic F.E.E.A *vs.* Specialized F.E.E.A Modulo a 27-bit Prime.**

# Motivations (III) Fast Algorithms

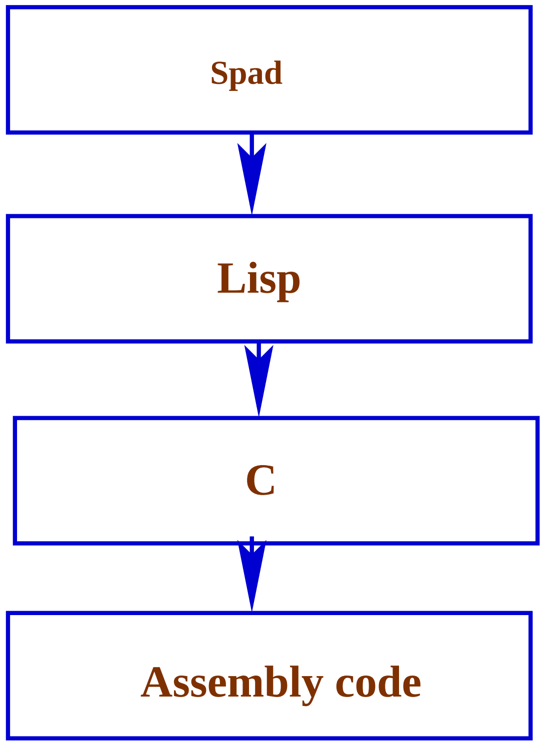
- **We are interested in fast polynomial arithmetic.**




**Univariate Fast E.E.A. vs. Quadratic E.E.A. (both generic code.) Modulo a 27-bit Prime.**

## Motivations (IV) Implementation Issue

- **Focus on implementation issues in *AXIOM*.**
- **Open *AXIOM* has a multiple-language level construction.**




- **Mix code at each level for high performance.**

# The SPAD level

**In *AXIOM* at SPAD level:**

- **A two-level type system – *categories* and *domains*.**

**E.g.** `Ring` **is the** AXIOM **category.**

- **The domain** `SUP(R)`**, where** `R` **has** `Ring` **type, implements** `UnivariatePolynomialCategory(R)` **with sparse data representation.**

- **The domain** `SMP(R,V)`**, where** `R` **has** `Ring` **type,** `V` **has** `VariableSet` **type, implements** `RecursivePolynomialCategory(R, V)` **with recursive sparse data representation.**

**Our implementation for better support dense algorithms:**

- **The domain** `DUP(R)` **implements the same category as** `SUP(R)` **does. The data representation is dense.**

- **The domain** `DRMP(R,V)` **implements the same category as** `SMP(R,V)` **does. The data representation is recursive dense.**

- **The benefit of dense domains for dense algorithms in later slides.**

## The GNU Common Lisp (*GCL*) level

- ***Lisp*** **is a symbolic-expression-oriented language.**
- **To speed up the performance of *GCL* code.**
  - **Using statically typed feature.**
  - **Suppressing run-time type checking.**
  - **Choosing adapted data structures.**

## Example-1: Array access in *GCL*

**SPAD code:** **array1.i := array2.i**

**Compiled Lisp code:** **(SETELT |array1| |i| (ELT |array2| |i|))**

**Hand-written Lisp code:** **(setf ( aref array1 i ) ( aref array2 i ))**

```
case t_vector:
if (index >= seq->v.v_fillp) goto E;
return(aref(seq, index));
```

**Without array bound checking, "aref" is slightly faster.**

# Example-2: Vector-based recursive dense polynomials

- **Implement $Z/pZ[x_1, \ldots, x_n]$ domain using the vector-based representation proposed by R. Fateman.**

- **Each polynomial encoded by a vector, each slot is pointer to another polynomial or a number.**

- **At *SPAD* level using Union type for this disjunction.**

- **At *Lisp* level using the property of dynamic typing.**

**Spad Union type is a “cons”.**

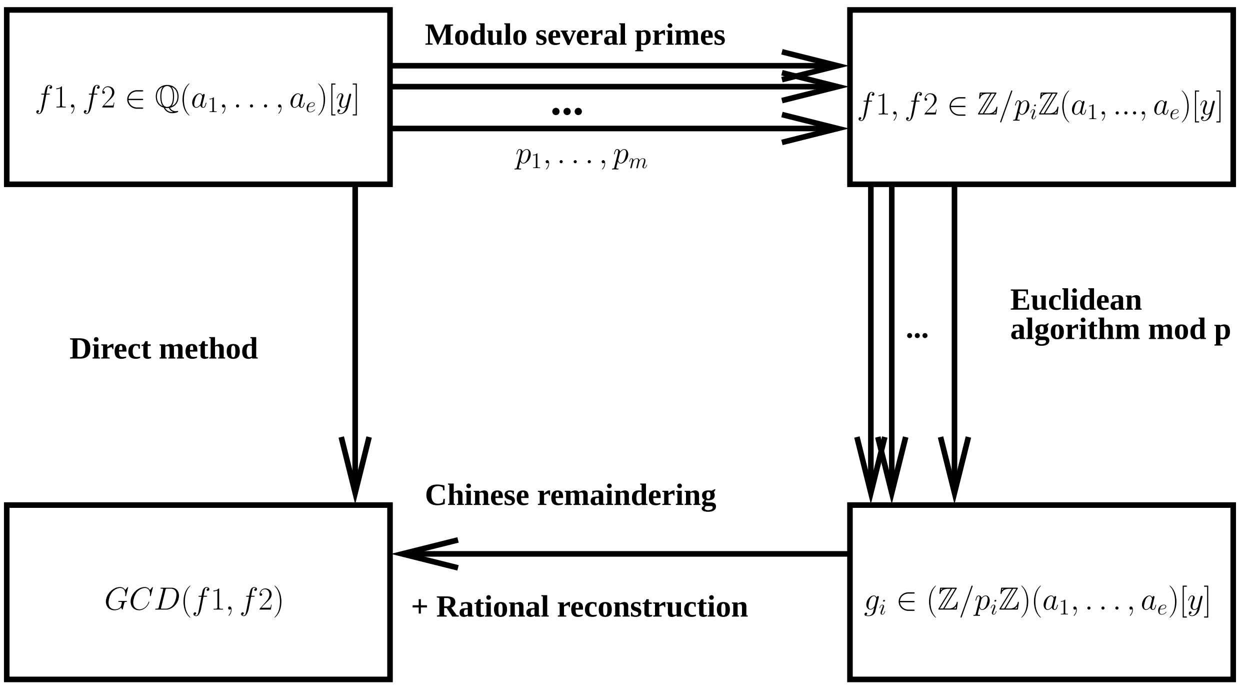
**“cdr” keeps the type info. “car” keeps the value.**

```
struct cons {
FIRSTWORD;
object c_cdr;
object c_car; };
```

# Benchmark (I)

- MMA – **Multivariate Modular Arithmetic domain.**
- MMA(p,V) **implements the same operations as** DRMP(PF(p),V).
- **Benchmark: van Hoeij and Monagan Modular GCD algorithm [1].**

**Input:** $f1, f2 \in \mathbb{Q}(a_1, \ldots, a_e)[y]$
**Output:** $GCD(f1, f2)$





**[1] A Modular GCD algorithm over Number Fields presented with Multiple Extensions.van Hoeij, Michael Monagan. ISSAC'02 proceedings, 109-116, (2002).**

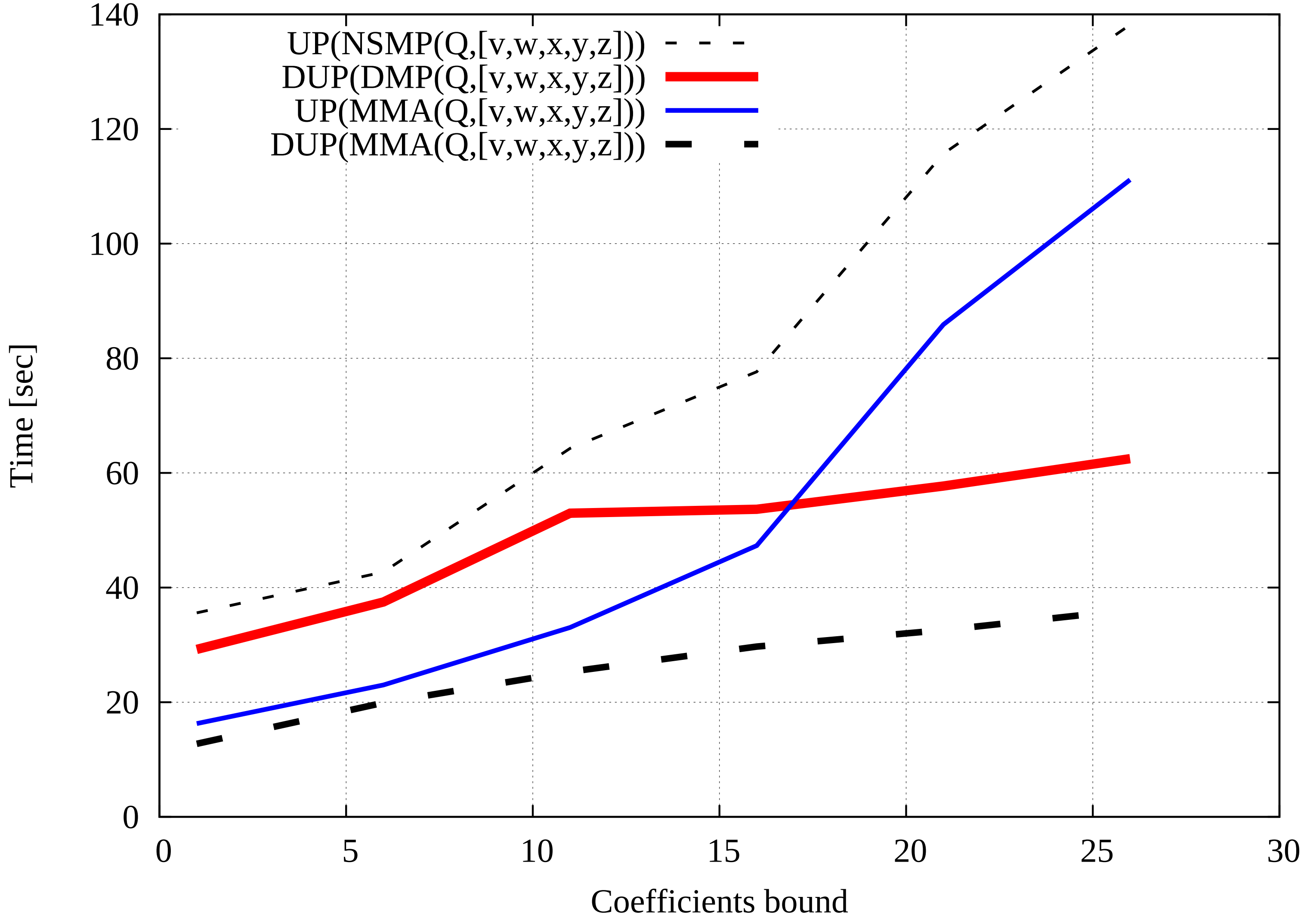
# Benchmarks (II)

**In *AXIOM* we implemented/used:**

| $\mathbb{Q}(a_1, a_2, \ldots, a_e)$ | $\mathbb{K}[y]$ |
|---|---|
| NSMP **in** SPAD | SUP **in** SPAD |
| DMPR **in** SPAD | DUP **in** SPAD |
| MMA **in** LISP | SUP **in** SPAD |
| MMA **in** LISP | DUP **in** SPAD |

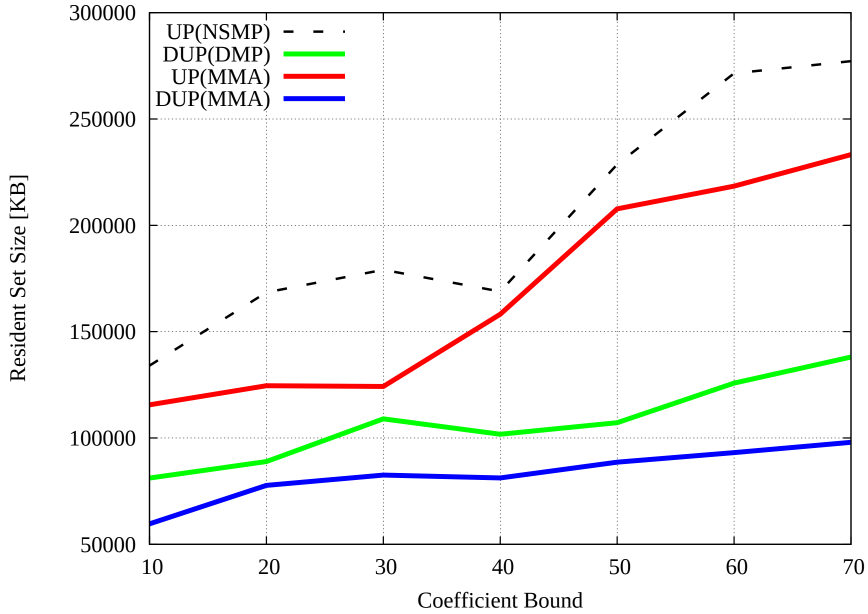
# Benchmarks (III)

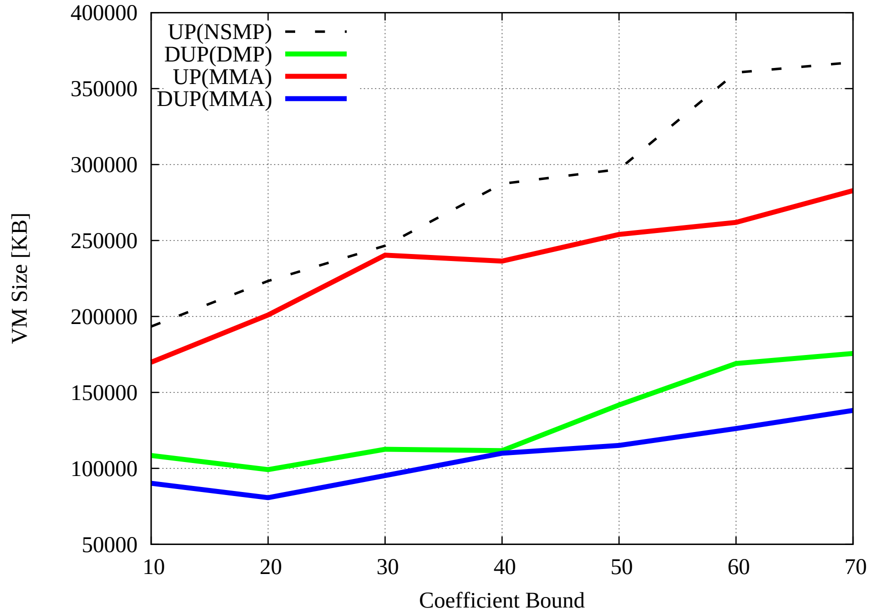
## Timing:



# Benchmarks (IV)

## Memory Consumption:



## The *C* level

**We go to *C* level:**

**• Implementing efficiency-critical operations requires direct access to machine arithmetic and no much symbolic expression manipulation.**

**• *GCL* is written in *C*, we can extend the *GCL* kernel at *C* level.**

**• Direct use of *C**C*++ or *Assembly* based libraries: `NTL`, `GMP`, . . . .**

## Example-1: Modular integer reduction on special Fourier primes.

- $p = c \cdot 2^k + 1,\ c,\ k \in \mathbb{Z}^+,\ (c,\ 2) = 1$**,** $p$ **is a prime;**
- **Input:** $Z$**,** $p \in \mathbb{Z}$**, where** $Z \leq (p-1)^2$**.**

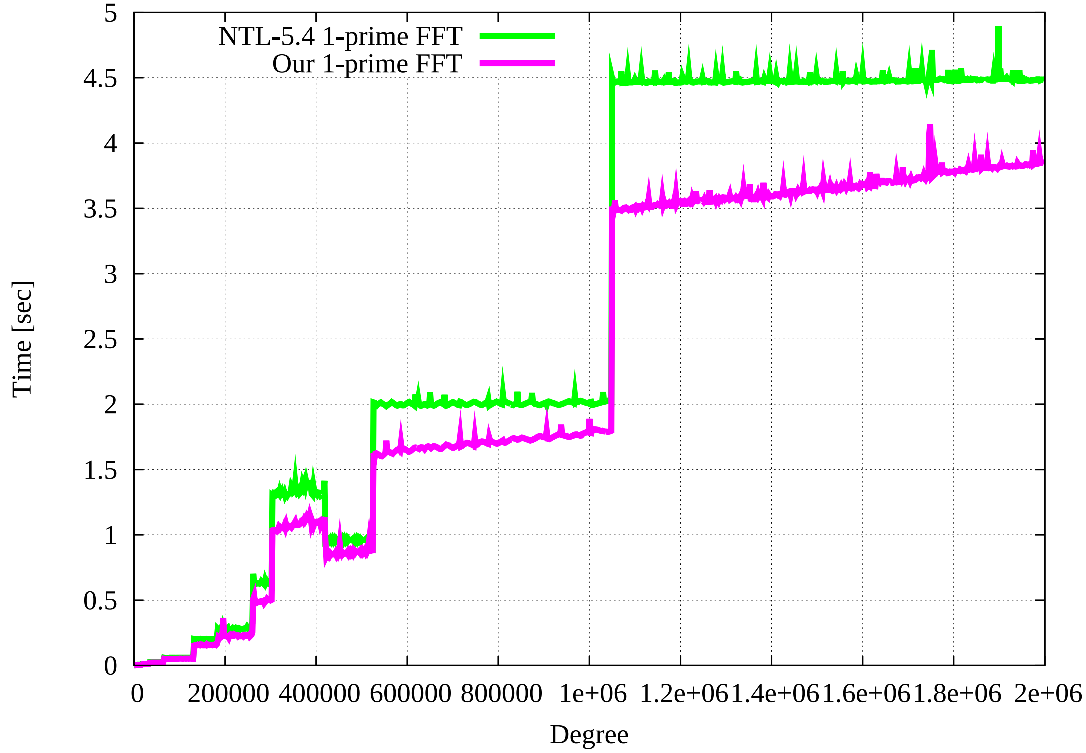
  **Output:** $r' := (c{\cdot}Z\ rem\ p) - \delta$**, where** $\delta < (c+1){\cdot}p$**.**
  - **Consider** $c{\cdot}Z = q{\cdot}p + r$**.**
  - **Define** $q' := \lfloor \frac{Z}{2^k} \rfloor$**.**
  - **Define** $r' := c{\cdot}(Z - q'{\cdot}2^k) - q'$**.**
  - **We have** $r' = r - \delta$**.**
- **This algorithm only needs "shifts" and "adds/subs".**

```
output = c * ((long)Z)&MASK_k - (long)(Z >> k);
```

## Example-2: *FFT* on special Fourier primes.

- **Special Fourier Prime modular reduction used in *FFT*.**
  - **Output:** $r' = (c \cdot Z \; rem \; p) - \delta$.
  - **Cancel** $c$ **by its inverse.**
  - **Control** $\delta$**s in middle stages.**
  - **Remove** $\delta$**s at the end.**
- **It's more efficient than Montgomery trick and floating point trick.**
- **Joint work with prof. Éric schost.**

# Benchmark of FFT




*FFT*-based uni-poly-mul over $\mathbb{Z}/p\mathbb{Z}$, $p$ is a 28-bit prime.

• In above figure, both our and NTL's FFT are 1-prime FFT. (Using `zz_p`::FFTInit($i$) in NTL.)

• Our 3-prime FFT vs. NTL's 3-prime FFT has the same speed-up-ratio as in the Figure. (Using `zz_p`::init($p$) in NTL.)

# Benchmark of FFT

| Event Type | Incl. |
|---|---|
| Instruction Fetch | 166 165 831 |
| Data Read Access | 82 331 696 |
| Data Write Access | 28 625 645 |
| L1 Instr. Fetch Miss | 138 |
| L1 Data Read Miss | 1 104 799 |
| L1 Data Write Miss | 530 870 |
| L2 Instr. Fetch Miss | 77 |
| L2 Data Read Miss | 64 220 |
| L2 Data Write Miss | 62 029 |
| L1 Miss Sum | 1 635 807 |
| L2 Miss Sum | 126 326 |
| Cycle Estimation | 195 156 501 |

NTL-FFT

| Event Type | Incl. |
|---|---|
| Instruction Fetch | 110 363 043 |
| Data Read Access | 33 976 545 |
| Data Write Access | 14 838 926 |
| L1 Instr. Fetch Miss | 130 |
| L1 Data Read Miss | 696 219 |
| L1 Data Write Miss | 236 576 |
| L2 Instr. Fetch Miss | 71 |
| L2 Data Read Miss | 154 837 |
| L2 Data Write Miss | 47 911 |
| L1 Miss Sum | 932 925 |
| L2 Miss Sum | 202 819 |
| Cycle Estimation | 139 974 193 |

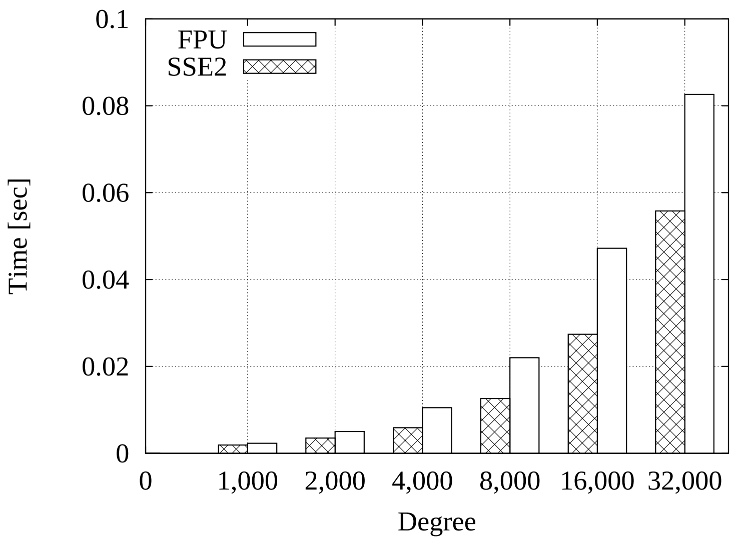
OUR-FFT

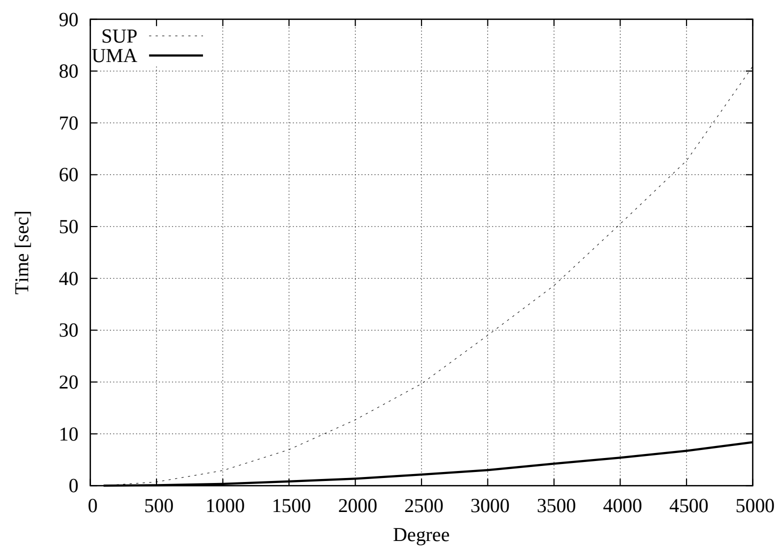
Cache setting:

- desc: I1 cache: 16384 B, 32 B, 8-way associative.
- desc: D1 cache: 65536 B, 64 B, 2-way associative.
- desc: L2 cache: 1048576 B, 64 B, 8-way associative.

# The ASSEMBLY level

- For using specific hardware features or existing code.



## Conclusion and future-work.

- By a few examples, we show that properly mixing code at each *AXIOM* language level may achieve better performance.
- Familiar with strength/weakness of language tools and their optimizer techniques are essential for high performance.
- We hope to construct automatic performance-tuning packages for fast polynomial arithmetic in near future.

* All code can be downloaded from http://www.csd.uwo.ca/∼xli96