Efficient Implementation of Polynomial Arithmetic in a Multiple-Level Programming Environment

Xin Li and Marc Moreno Maza

University of Western Ontario, London N6A 5B7, Canada

Abstract. The purpose of this study is to investigate implementation techniques for polynomial arithmetic in a multiple-level programming environment. Indeed, certain polynomial data types and algorithms can further take advantage of the features of lower level languages, such as their specialized data structures or direct access to machine arithmetic. Whereas, other polynomial operations, like Gröbner basis over an arbitrary field, are suitable for generic programming in a high-level language.

We are interested in the integration of polynomial data type implementations realized at different language levels, such as LISP, C and ASSEMBLY. In particular, we consider situations for which code from different levels can be combined together within the same application in order to achieve high-performance.

We have developed implementation techniques in the multiple-level programming environment provided by the computer algebra system AXIOM. For a given algorithm realizing a polynomial operation, available at the user level, we combine the strengths of each language level and the features of a specific machine architecture. Our experimentations show that this allows us to improve performances of this operation in a significant manner.

1 Introduction

In a general purpose computer algebra system, generic code implementing univariate and multivariate polynomial over an arbitrary ring or field is a central feature. This is the case, for instance, in the computer algebra systems AXIOM [13], ALDOR [2], MAGMA [3] and SINGULAR [11]. On the other hand, the quest for high-performance in critical areas, such as modular algorithms for polynomial GCDs, leads naturally to develop low-level specialized code for univariate and multivariate polynomials over finite fields. Such code is available in the software systems mentioned above and others.

The works of [14,7,8] present efficient implementations of asymptotically fast algorithms for polynomial arithmetic in a high-level programming environment. In the reported experiments, for FFT-based univariate polynomial multiplication and the Half-GCD algorithm, the speed-up ratio between the pure low-level specialized code and the pure high-level generic code is in the range $2 \cdots 4$. (We refer to [10] and [16] respectively for these algorithms.) Therefore, high-performance

A. Iglesias and N. Takayama (Eds.): ICMS 2006, LNCS 4151, pp. 12–23, 2006.
© Springer-Verlag Berlin Heidelberg 2006

in polynomial arithmetic can be achieved in a high-level programming environment, such as AXIOM and ALDOR.

However, linkage to specialized code is a substantial bonus when low-level implementation can take advantage of special software or hardware features. The purpose of this study is to investigate implementation techniques for polynomial arithmetic in a multiple-level programming environment. We are interested in the integration of polynomial data type implementations realized at the different code levels. In particular, we consider situations for which code from different levels can be combined together within the same application in order to achieve high-performance.

As a driving example, we use the modular algorithm of van Hoeij and Monagan [12]. We recall its specifications. Let $\mathbb{K} = \mathbb{Q}(a_1, a_2, \ldots, a_e)$ be an algebraic number field over the field $\mathbb{Q}$ of the rational numbers. Let $f_1, f_2 \in \mathbb{K}[y]$ be univariate polynomials over $\mathbb{K}$. The algorithm of van Hoeij and Monagan computes $\gcd(f_1, f_2)$. To do so, for several prime numbers p, a tower of simple algebraic extensions $\mathbb{K}_p$ of the prime field $\mathbb{Z}/p\mathbb{Z}$ is used. Arithmetic operations in $\mathbb{K}_p$ are performed by means of operations on multivariate polynomials over $\mathbb{Z}/p\mathbb{Z}$, whereas the operations on the images of f_1, f_2 modulo p are performed in the univariate polynomial ring $\mathbb{K}_p[y]$. Therefore, several types of polynomials are used simultaneously in this algorithm. This is why it is a good candidate for our study.

We chose AXIOM as our implementation environment based on the following observations. AXIOM has a high-level programming language, called SPAD, which possesses all the essential features of object-oriented languages. Libraries written in SPAD implement a hierarchy of algebraic structures (groups, rings, fields, ...) and a hierarchy of algebraic domains ($\mathbb{Q}$, $\mathbb{A}[x]$ for a given ring $\mathbb{A}$, ...).

The SPAD compiler translates SPAD code into COMMON LISP, then invokes the underlying LISP compiler to generate machine code. Today, GCL [17] (GNU COMMON LISP) is the underlying LISP of AXIOM [1]. The design of GCL makes use of the native C compiler for compiling to native machine code. In addition, GCL employs the GNU Multi-Precision library (GMP) [9] for its arbitrary precision number arithmetic. Therefore, AXIOM is an efficient multiple language level system. Moreover, the complete AXIOM system is open-source. Hence, we can implement our packages at any language level and even modify the AXIOM kernel. This allows us to take advantage of each language level's strength and access machine arithmetic directly when necessary. Therefore, we believe that AXIOM, with its different implementation levels, all in open source, provides an exceptional development environment among all computer algebra systems, for the purpose of our study.

In Sections 2, 3 and 4 and 5 we discuss the strength (in view of our objectives) of the SPAD, LISP, C and ASSEMBLY level, respectively, together with our implementation techniques. In Section 6, we report on our experimentation.

Our results suggest that choosing adequate and optimized data structures is essential for polynomial arithmetic with high-performance. At the same time, implementing them at a suitable language level or levels may impact the running

time of an algorithm implementation in AXIOM by a factor in the range $2 \cdots 4$ for large input data.

2 The SPAD Level

The AXIOM computer algebra system possesses an interactive mode for user interactions and the SPAD language for building library modules. The SPAD language has a two-level object model of *categories* and *domains* that is similar to *interfaces* and *classes* in Java. For instance, `Ring` is the AXIOM category for all rings (in the algebraic sense, see for instance [5]) with units and `Integer` is the AXIOM domain for the ring of integer numbers (see [13] for more details about the AXIOM hierarchies of categories and domains).

The user can define a new category and domain, which, then, can be added to the library modules. To do so, this new SPAD code must implement an AXIOM type constructor (or a package), to be compiled with the SPAD compiler. An AXIOM *type constructor* is simply a function which returns an AXIOM type, that is a category or a domain. For instance, `SparseUnivariatePolynomial`, abbreviated to `SUP`, is a type constructor, which takes an argument `R` of type `Ring` and returns an implementation of the ring of univariate polynomials over `R`, with a sparse representation (see below). Actually, `SUP(R)` implements `Ring` and other operations specific to polynomials (evaluation, differentiation, ...). The whole interface of `SUP(R)` is `UnivariatePolynomialCategory(R)` where `UnivariatePolynomialCategory` is a category constructor.

The SPAD language supports *conditional exports*. This permits to implement the following statement: if `R` has type `Field` then `SUP(R)` implements `EuclideanDomain`. SPAD supports also *conditional implementation*. For instance, if `R` has type `FiniteFieldCategory`, one can use formulas such *Little Fermat Theorem* to speed up some operations of `SUP(R)`, such as exponentiation. These features of the SPAD language are important for combining different data types and achieving high-performance.

Implementing a new domain constructor requires the programmer choosing a data structure for representing the objects defined by this domain. After a newly defined domain or category is compiled, it becomes an AXIOM data type which can be used just like any system provided data type.

In the light of these properties of the SPAD language, we describe briefly the polynomial type constructors that we use in this study. Please, see [13] and [14] for more details. Let `R` be an AXIOM `Ring` and `V` be an AXIOM `OrderedSet`.

`SUP or UP`. As mentioned above, the domain `SUP(R)` implements the ring of univariate polynomials with coefficients in `R`. More precisely, it satisfies the AXIOM category `UnivariatePolynomialCategory(R)`. The representation of these polynomials is *sparse*, that is, only non-zero terms are encoded.

`DUP`. The domain `DUP(R)` implements `UnivariatePolynomialCategory(R)` as well. The representation is dense: all terms, null or not, are encoded.

`NSMP`. The domain `NSMP(R,V)` implements the ring of multivariate polynomials with coefficients in `R` and variables in `V`. (To be precise, it implements the

AXIOM category `RecursivePolynomialCategory(R, V)`.) A non-constant polynomial f of `NSMP(R)`, with greatest variable v, is regarded as a univariate polynomial in v implemented as an element of `SUP(NSMP(R))`. Therefore, the representation is recursive and sparse.

DRMP. The domain `DRMP(R,V)` implements the same category as `NSMP(R,V)`. The representation is also recursive. However, it is based on `DUP` rather than `SUP`.

The constructors `SUP` and `NSMP` are provided by the AXIOM standard distribution, whereas `DUP` and `DRMP` were implemented by us at SPAD level. Our motivation is the implementation of modular methods based on the Euclidean Algorithm (or its variants) which tend to "densify" the computations, even if the input polynomials are sparse. This is the case, for instance with the algorithm of van Hoeij and Monagan, that we have implemented for this study.

3 The Lisp Level

The domain constructors `SUP`, `DUP`, `NSMP` and `DRMP` allow the user to construct polynomials over any AXIOM `Ring`. So we say that their code is generic. Observe that `R` has no influences on the representation scheme of the objects of `SUP(R)` or `DUP(R)`.

Ideally, one would like to use also *conditional data representations*. For instance, one could think of a domain `U(R)` implementing univariate polynomials over `R`, an AXIOM `Ring`, such that sparse polynomials (polynomials with frequent null terms) have a sparse representation and dense polynomials (polynomials with few null terms) have a dense representation. In addition, if `R` implements a prime field $\mathbb{Z}/p\mathbb{Z}$ for a machine word size prime p, one could require encode each dense polynomial of `U(R)` by an array of machine words (such that the slot of index i contains the coefficient in the term of degree i). But the code of this ideal type constructor `U` would be quite complicated and harder to optimize from the compilation point of view. Indeed, many tests would be needed for selecting the appropriate representation. Instead, we believe that specialized domain constructors (say, dense univariate polynomials over a prime field) to be called by a package (implementing, for instance, the algorithm of van Hoeij and Monagan) are a better choice. Moreover, polynomials over prime fields are such an importance case, for modular methods, that they deserve an independent treatment.

For these reasons, we have defined at the SPAD level a polynomial type constructor `MultivariateModularArithmetic`, abbreviated to `MMA`, taking a prime integer `p` and `V`, an `OrderedSet`, as arguments, such that `MMA(p,V)` implements the same operations as `DRMP(PF(p),V)`. (The domain `PF(p)` implements the prime field of characteristic `p`.) In fact, `MMA(p,V)` is just a wrapper for an implementation in Lisp. At this inner level of AXIOM, we have realized two implementations of `MMA(p,V)`: one for the case where `p` fits in a machine word and one for the case where it does not.

In these implementations, we used the *vector-based recursive dense* representation proposed by Richard J. Fateman [6]: a multivariate polynomial f is encoded

by a number (to be precise, an integer modulo p) if f is constant and, otherwise, by a LISP vector storing the coefficients of f w.r.t its leading variable. At the SPAD level, such disjunction would be implemented by a union type bringing an extra indirection. This can be avoided at the LISP level, thanks to the so-called predicate functions which can judge the type of an object.

In addition, we can tell the LISP compiler to use machine integer arithmetic, if p fits in a machine word and, otherwise, to use the functions for big integer arithmetic from the GMP library [9].

Similarly for univariate polynomials, we have defined at the SPAD level a univariate type constructor `UnivariateModularArithmetic`, abbreviated to `UMA`, taking a prime integer p as argument and implementing the same operations as `DUP(PF(p))`. It is also a wrapper for two LISP implementations: one for Small p's and for one big p's. Each of these univariate polynomials is implemented using `fixnum-array` (a C-like array) in GCL. It is possible direct using C arrays to encode univariate polynomials over $\mathbb{Z}/p\mathbb{Z}$, but we prefer the LISP level garbage collection system which is more efficient and convenient.

All these specialized implementations at the LISP level yield significant speed up, as reported in Section 6.

4 The C Level

GCL is implemented in C language and uses the native optimizing C compiler to generate native machine code. This allows us to extend the functionalities of the LISP level of AXIOM with a given C function by either integrating this function into the GCL kernel, or by integrating it into a GCL library.

This interoperability between LISP and C has at least two benefits for achieving high-performance in the AXIOM environment. First, ASSEMBLY code (written for some efficiency critical operation, see Section 5) can be into the LISP level via C. Second, we can use existing C libraries providing efficient implementations of polynomial and integer arithmetic, such as GMP library [9] or NTL [15]. We illustrate these two benefits by an important example: the implementation of dense univariate polynomials over the prime field $\mathbb{Z}/p\mathbb{Z}$.

Recall that we have two implementations for these polynomials at the LISP level: one for small primes p (that fit in a machine word) and one for big primes p. They are both available at the SPAD level via the wrapper domain constructor `UMA`. For both the small and big prime case, we have integrated in the GCL kernel:

- classical multiplication, addition and Chinese remaindering algorithm written in ASSEMBLY,
- FFT-based multiplication written in C with ASSEMBLY sub-routines.

Moreover, in the big prime case, we distinguish between the double precision case and the multiple precision case. In the former one, we have developed our ASSEMBLY routines which improve the performance of the GMP multiple precision functions. (See [14] for details.) In the latter one, we rely on the ASSEMBLY

routines of GMP. This distinction is motivated by the importance of the double precision case. For instance, most prime numbers used in the modular method of [4] are of that size.

5 The Assembly Code Level

As mentioned in Section 4, we make use of the Assembly functions of the GMP library in order to accomplish low-level operations with univariate polynomial over $\mathbb{Z}/p\mathbb{Z}[x]$. There are two other reasons for taking advantage of Assembly code in our AXIOM environment. We discussed them below.

5.1 Controlling Register Allocation

In a modern computer architecture, CPU registers seat at the top level of the memory hierarchy. Although optimizing compilers devote special efforts to make good use of the target machine's register set, this effort can be constrained by numerous factors, such as:

- difficulty to estimate the execution frequencies of each part of the program,
- difficulty to allocate or evict ambiguous values,
- difficulty to take advantage of some new hardware features on specific platforms.

Therefore, a high-performance oriented application requires programmers to better exploit the power of registers on a target machine. In fact, we have spent a great effort in this direction in our implementation.

First, we directly program the efficiency-critical parts in Assembly language in order to explicitly manipulate data in registers. For example, for dense univariate polynomials over $\mathbb{Z}/p\mathbb{Z}$, we write the classical multiplication algorithm in both C and Assembly language. The Assembly version is faster than the C version since we always try to keep frequently used variables in registers instead of a memory location. Although in C we can declare a variable to be of "register" type, this does not guarantee that the register is reserved for this variable. According to our benchmark results, our explicit register allocation method is always faster than the C compiler's optimization.

Beside the general purpose registers, we also can use MMX, XMM registers when they are available. Keeping the working set in registers will yield significant performance improvement comparing to keeping them in main memory.

5.2 Using Architecture Specific Features

Polynomial arithmetic in $\mathbb{Z}/p\mathbb{Z}[x]$ makes an intensive use of integer division. This integer operation has a dominant cost in crucial polynomial operations like the FFT-based multiplication in $\mathbb{Z}/p\mathbb{Z}[x]$. Therefore, improving the performance of integer division is one of the key issues in our implementation.

In the NTL library [15], the single precision modular reduction is implemented by means of floating point arithmetic, based on the following formula

$$a \equiv a \lfloor a\ 1/p \rfloor\ p$$

This formula can be implemented directly in C in two or three lines of code. However, one can further improve the performance by writing assembly code.

We have also implemented this trick in ASSEMBLY language for the Pentium IA-32 with SSE2 support. The SSE/SSE2 instruction sets use XMM registers. Each of these registers is of 128-bit and can be used to pack 2 double precision floating point numbers. In fact, SSE/SSE2 instructions can compute on multiple data packed in one register, in parallel.

Our implementation of the FFT-based polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$ uses this technique. It is faster than using FPU unit, as reported in Section 6.

6 Experimentation

6.1 Benchmarks for the LISP Level Implementation

The goal of these benchmarks is to measure the performance improvements obtained by our specialized multivariate polynomial domain constructor `MMA` implemented at the LISP level and described in Section 3. We are also curious about measuring the practical benefit of dense recursive polynomial domains in a situation (polynomial GCD computations over algebraic number fields) where AXIOM libraries traditionally use sparse recursive polynomials.

As announced in the introduction, our test algorithm is that of van Hoeij and Monagan [12]. Recall that, given an algebraic number field $\mathbb{K} = \mathbb{Q}(a_1, a_2, \ldots, a_e)$, this algorithm computes GCDs in $\mathbb{K}[y]$ by means of a small prime modular algorithm, leading to computations over a tower of simple algebraic extensions $\mathbb{K}_p$ of $\mathbb{Z}/p\mathbb{Z}$. Recall also that the algorithm involves two polynomial data types:

- a multivariate one for the elements of $\mathbb{K}$ and $\mathbb{K}_p$,
- a univariate one for the polynomials in $\mathbb{K}[y]$ and $\mathbb{K}_p[y]$.

Figure 1 shows the different combinations that we have 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

Note that:

- the first two combinations, that is `NSMP + SUP` (sparse polynomial domains) and `DMPR + DUP` (dense polynomial domains), involve only SPAD code,

- the other two combinations use MMA - our dense multivariate polynomials implemented at the LISP level and SUP/DUP - univariate polynomials written at the SPAD level.

We would like to stress the following facts:

- the algorithms for addition, multiplication, division of `DRMP` and `MMA` are identical,
- none of the above polynomial types uses fast arithmetic, such as FFT-based or Karatsuba multiplication.

Remember also that:

- the SPAD constructors `NSMP`, `DMPR`, `UP`, and `DUP` are generic constructors, i.e. they work over any AXIOM ring,
- however, our dense multivariate polynomials implemented at the LISP level (provided by the `MMA` constructor) only work over a prime field.

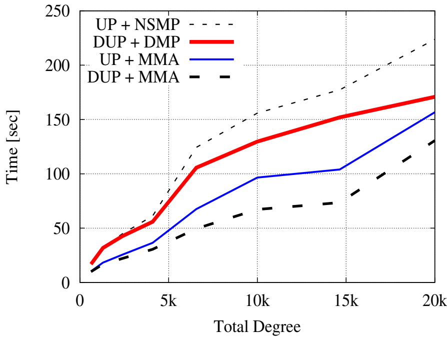
Therefore, we are comparing here is the performances of

- specialized code at the LISP level versus generic code at the SPAD level,
- sparse representation versus dense representation.

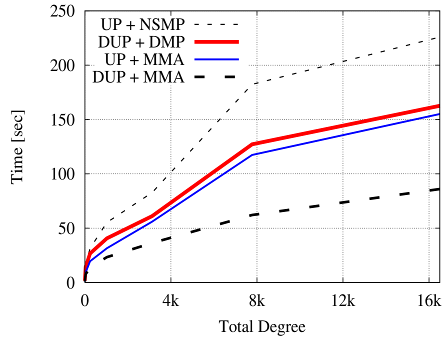
We have run the algorithm of van Hoeij and Monagan for different degrees of the extension $\mathbb{Q} \rightarrow \mathbb{K}$, different degrees of the input polynomials and different sizes for their coefficients. Figure 1 p. 20 shows our benchmark results. First, we fix the coefficient size bound to 5 and increase the total degree (degree of the extension plus maximum degree of an input polynomial). The charts (a), (b) and (c) correspond to towers of 3, 4 and 5 simple extensions respectively. Second, we fix the total degree to 2000 and increase the coefficient bound. The charts (d), (e) and (f) correspond to towers of 3, 4 and 5 simple extensions respectively. We observe the following facts.

Charts (a), (b), (c). For univariate polynomial data types, `DUP` outperforms `SUP` and, for the multivariate polynomial data types, `MMA` outperforms `DRMP`, which outperforms `NSMP`. For the largest degrees, the timing ratio between the best combination, `DUP + MMA`, and the worst one, `SUP + NSMP` is in the range $2 \cdots 3$.

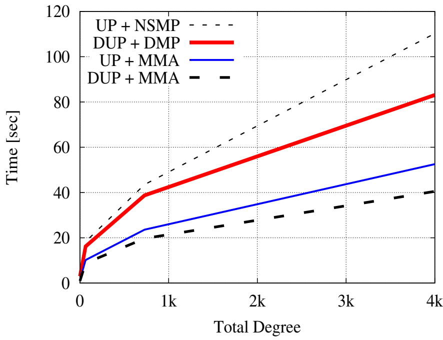
Charts (d), (e), (f). The best and the worst combinations are the same as above, however the timing ratio is in the range $3 \cdots 4$. Interestingly, the second best combination is `SUP + MMA` for small coefficients and `DUP + DRMP` for larger ones. This fact has probably the following double explanation. First, the `SUP` constructor relies on some fast routines which allows it to compete with the `DUP` constructor for small input data. Second, garbage collection of polynomials built with `DUP + DRMP` appears to be more efficient than for `SUP + MMA` polynomials, for large data.



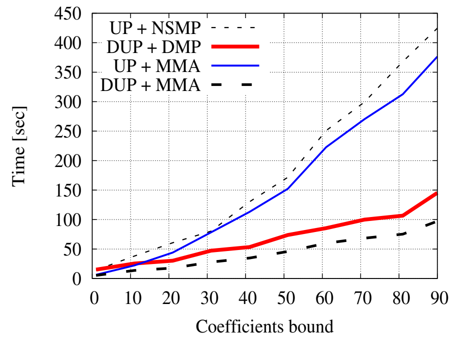
(a)



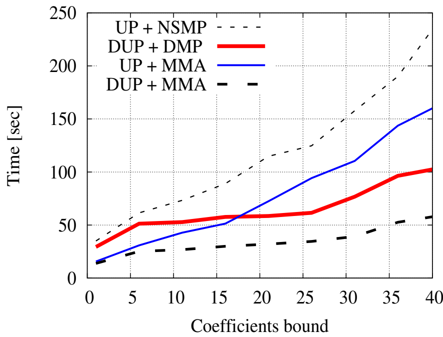
(b)



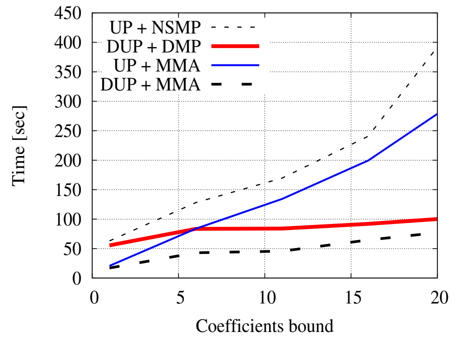
(c)



(d)



(e)



(f)

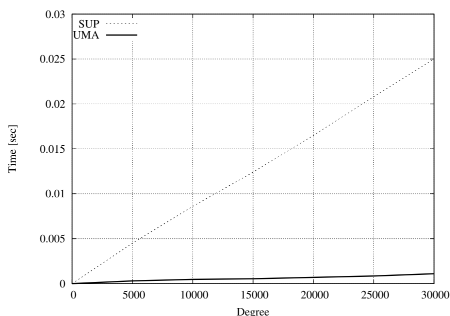
Fig. 1. In (a), (b) and (c) fixing the coefficient size bound, and increase the total degree of input polynomials. Conversely In (d), (d), and (f) fixing the total degree and increase the coefficient size bound.

6.2 Benchmark for the C and Assembly Level Implementation

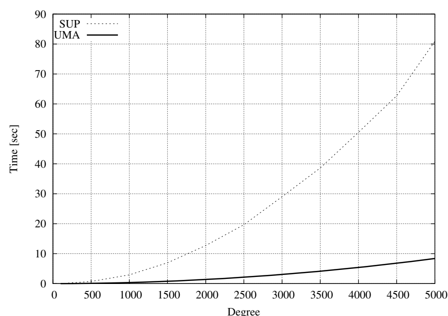
The goal of these benchmarks is to measure the benefit provided by the C and Assembly levels to the SPAD level. Figure 2 shows benchmarks for addition and classical multiplication in $\mathbb{Z}/p\mathbb{Z}[x]$ for a 64-bit prime p, between

- the code of the `SUP` constructor (from the SPAD level), and
- the `UMA` constructor (written in Lisp with C and Assembly sub-routines).

This clearly illustrates the benefits of our implementation at Assembly level comparing to its SPAD level counterpart.



Assembly level polynomial addition.



Assembly level polynomial multiplication.

Fig. 2.

Figure 3 illustrates the performance difference between

- the generic Assembly code using integer arithmetic and,
- the SSE2 Assembly code using floating point arithmetic.

The benchmark data of Figure 3 are obtained with our implementation of FFT-based univariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}[x]$ for a 27-bit Fourier prime p. This benchmark shows that our SSE2-based implementation is significantly faster than our generic Assembly version.

6.3 Benchmark of the Multi-level Implementation

The goal of this benchmark is to compare an AXIOM function, involving code at all levels, SPAD, Lisp, C and Assembly, versus its counterpart in a similar computer algebra system, namely MAGMA. We choose multivariate polynomial multiplication based on Kronecker substitution (which for us is implemented at SPAD and Lisp levels, since it is independent from machine arithmetic) and FFT-based univariate multiplication (which for us is implemented at C and Assembly level, as shown in previous benchmark). Our implementation outperforms MAGMA's counterpart as shown in Figures 4.

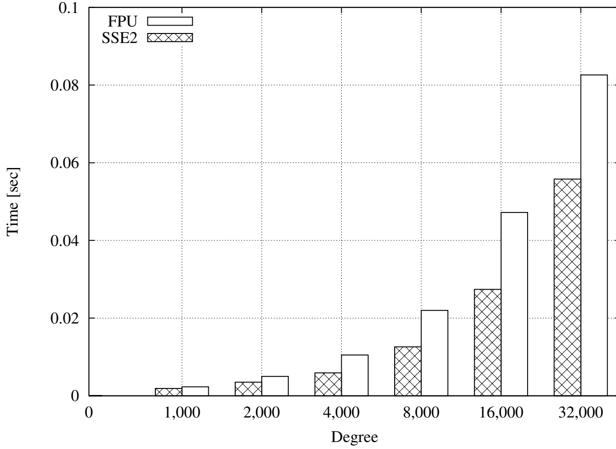


Fig. 3. Generic assembly vs. SSE2 version assembly of FFT-based univariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$

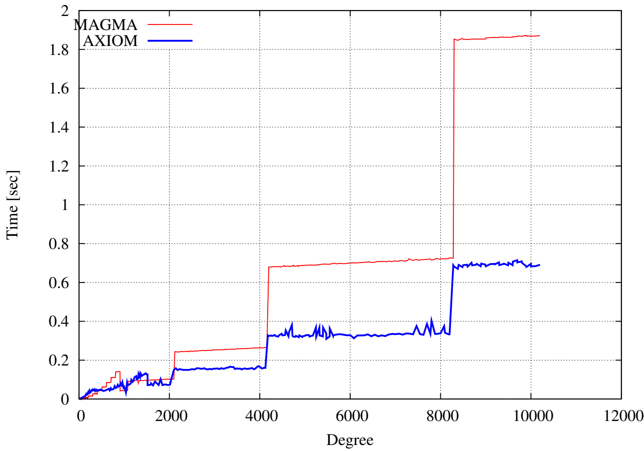


Fig. 4. Bivariate multiplication modulo a 64-bit prime

7 Conclusion

We have investigated implementation techniques for polynomial arithmetic in the multiple-level programming environment of the AXIOM computer algebra system. Our benchmark results show that careful integration of data structures and code from different levels can improve the performances in a significant manner (a ratio of 2...4 speed up reported in 6). The integration process requires deep understanding of polynomial arithmetic, machine arithmetic and compiler optimization techniques. However, we believe that it should be implemented in a transparent way for the end-user.

References

1. *The* AXIOM *developers web site.* http://page.axiom-developer.org.
2. aldor.org. *Aldor 1.0.2.* University of Western Ontario, Canada, 2004. http://www.aldor.org.
3. The Computational Algebra Group in the School of Mathematics and Statistics at the University of Sydney. *The MAGMA Computational Algebra System for Algebra, Number Theory and Geometry.* http://magma.maths.usyd.edu.au/magma/.
4. X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *ISSAC'05*, pages 108–115. ACM Press, 2005.
5. David S. Dummit and Richard M. Foote. *Abstract algebra.* Simon and Schuster, 1993.
6. R. J. Fateman. Vector-based polynomial recursive representation arithmetic. http://www.norvig.com/ltd/test/poly.dylan, 1999.
7. A. Filatei. Implementation of fast polynomial arithmetic in Aldor, 2006. University of Western Ontario.
8. A. Filatei, X. Li, M. Moreno Maza, and É Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Proc. ISSAC'06*. ACM Press, 2006.
9. Free Software Foundation. *GNU Multiple Precision Arithmetic Library.* http://swox.com/gmp/.
10. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.
11. G.-M. Greuel, G. Pfister, and H. Schönemann. SINGULAR 3.0. A Computer Algebra System for Polynomial Computations, Centre for Computer Algebra, University of Kaiserslautern, 2005. http://www.singular.uni-kl.de.
12. M. van Hoeij and M. Monagan. A modular gcd algorithm over number fields presented with multiple extensions. In Teo Mora, editor, *Proc. ISSAC 2002*, pages 109–116. ACM Press, July 2002.
13. R. D. Jenks and R. S. Sutor. *AXIOM, The Scientific Computation System.* Springer-Verlag, 1992. AXIOM is a trade mark of NAG Ltd, Oxford UK.
14. X. Li. Efficient management of symbolic computations with polynomials, 2005. University of Western Ontario.
15. V. Shoup. *The Number Theory Library.* http://www.shoup.net/ntl.
16. C.K. Yap. *Fundamental Problems in Algorithmic Algebra.* Princeton University Press, 1993.
17. Taiichi Yuasa, Masami Hagiya, and William F. Schelter. *GNU Common Lisp.* http://www.gnu.org/software/gcl.