Formal proof for delayed finite field arithmetic using floating point operators

Sylvie Boldo, Marc Daumas, Pascal Giorgi

Formal proof for delayed finite field arithmetic using floating point operators*

Sylvie Boldo
INRIA (Saclay - le-de-France)
sylvie.boldo@inria.fr

Marc Daumas
ELIAUS (EA 3679 UPVD)
marc.daumas@ens-lyon.org

Pascal Giorgi
LIRMM (UMR 5506 CNRS–UM2)
pascal.giorgi@lirmm.fr

Abstract

Formal proof checkers such as Coq are capable of validating proofs of correction of algorithms for finite field arithmetics but they require extensive training from potential users. The delayed solution of a triangular system over a finite field mixes operations on integers and operations on floating point numbers. We focus in this report on verifying proof obligations that state that no round off error occurred on any of the floating point operations. We use a tool named Gappa that can be learned in a matter of minutes to generate proofs related to floating point arithmetic and hide technicalities of formal proof checkers. We found that three facilities are missing from existing tools. The first one is the ability to use in Gappa new lemmas that cannot be easily expressed as rewriting rules. We coined the second one "variable interchange" as it would be required to validate loop interchanges. The third facility handles massive loop unrolling and argument instantiation by generating traces of execution for a large number of cases. We hope that these facilities may sometime in the future be integrated into mainstream code validation.

1 Introduction

Introducing a new algorithm is a difficult task. Authors have to persuade readers that their algorithm is correct and efficient. Such goals are usually attained by providing pen-and-paper proofs of correction more or less interlaced with the description of the algorithm. Authors may also provide results of tests to guarantee correction and efficiency on random cases and on known or new hard cases. Alas, this process is known to fail on mundane as well as notorious occurrences [16].

Developing a proof of correction in a formal proof checker using higher order logic such as Coq [1] would be a nice alternative but such a task usually represents a large amount of work outside the fields of expertise of most authors.

*This work has been partially founded by PICS 2533 of the CNRS, project EVA-Flo of the ANR, project CerPAN of the ANR and PPF Surna.

The delayed solver studied here works on a $N \times N$ unitary triangular matrix on $\mathbb{Z}/p\mathbb{Z}$ finite field. The key improvement of this algorithm compared to state of the art lies in the fact that delayed algorithms use floating point units to perform operations with no rounding error and delay computations of remainders as much as possible. Operations on floating point numbers are limited to three functions. The other functions use combinatorial logic.

The first function (`DGEMM_NEG`) performs a naive matrix multiplication and Gappa may soon be able to handle the proof obligation generated by a tool such as the Why platform [9] and the corresponding floating-point annotations [2]. The second function (`DTRSM`) is invoked only under the `delay` predicate. This is enforced by the condition on the induction of the invoking function (`LZ_TRSM`) for N between 2 and 54 with p prime varying between 2 and $94,906,266$. Variable interchange in the `delay` predicate allows to limit the proof to the 53 different values of N where a naive user would consider the $94,906,265$ different values of p.

Proof obligations are usually derived from a static analysis of the source code considered. Our work showed that generating proof obligations from traces of execution after most parameters have been instantiated may also be useful. We have set up a C++ class to provide such proof obligations but we hope that such capability will be provided by Why and similar tools in the future.

We present some background information in the remaining of the introduction. We continue with a new lemma that might be used by Gappa for inductive linear bounds in Section 2 and with our prototyping variable interchange developments in Sections 3. We conclude this work in Section 4.

1.1 Finite field arithmetic and application to linear algebra

Finite field arithmetic plays a crucial role in nowadays applications. One of the most extensively studied application of finite fields is cryptography. Another key application of finite field arithmetic arises with exact linear algebra computation where modular techniques (e.g. CRT or P-adic lifting) allow some control on expression swell with high performances (see [8] and references herein). While cryptographic applications need finite fields of large cardinality for security purpose, most exact linear algebra restrains to machine word size prime field (e.g. 32 or 64 bits) in order to benefit from machine arithmetic units.

A classical way to perform one arithmetic operation in a prime field, here we refer to integers modulo a prime number, is to first perform the operation on integers and second reduce the result to the destination field. Let $x, y \in \mathbb{Z}/p\mathbb{Z}$ and $* \in \{+, \times\}$. One may compute $z = x * y \in \mathbb{Z}/p\mathbb{Z}$ by computing $t = x * y \in \mathbb{Z}$ and a modular reduction $z = t \mod p$.

When one deals with fixed precision prime field arithmetic, two majors issues arise: performances and cardinality limitation. The latter issue can have a non-negligible impact on the former one. As was just said, the classical way to perform arithmetic operations over a prime field is to perform operations on integers and reduce intermediate results. Therefore, all integers between 0 and $(p-1)^2$ must be representable to correctly perform multiplications over $\mathbb{Z}/p\mathbb{Z}$. This limitation slightly increase to perform an `AXPY` operation (a multiplication followed by an addition) with only one reduction step. This implies that all integers between 0 and $p \times (p-1)$ must be representable.

Using word-size machine integers and classic arithmetic we obtain the following cardinality limitation: $p < 2^{16}$ on 32 bit architectures and $p < 2^{32}$ on 64 bit architectures with `unsigned` types. An alternative to increase cardinality of word-size prime fields is to use floating point numbers. According to the IEEE 754 standard [10], mantissas of double precision floating

point numbers can store 53 bit integers (including the implicit bit). Therefore, we can perform prime field arithmetic with cardinality up to 2^{26} using `double`. Note that, the reduction is easily obtained by the `fmod` function available in standard libraries. This approach is quite interesting in practice since floating point multiplications and divisions may be faster than their integer counterparts.

On selected classes of algorithms, delayed prime field arithmetic sustains better performances. The idea is to perform several integer operations before reduction into the field. It has been very fruitful for exact linear algebra [7]. Delayed exact linear algebra computations also benefit from optimized numerical BLAS (e.g. ATLAS [18], GOTO [11]) libraries for exact computations and they often reach peak FPU throughput for operations over a finite field.

Beside basics linear algebra operations such as matrix-vector products and matrix multiplications, delayed arithmetic over a prime field is valuable when expressions swell largely such as solving systems of linear equations. This approach works perfectly for unitary triangular system (only ones along the diagonal) despite the exponential growth of the intermediate variables.

1.2 Formal proof checking and Gappa

Gappa [3] has been created to generate formal certificates of correction for programs that use floating point arithmetic [6, 15, 14] and is related to other developments [12, 5]. It is available from

`http://lipforge.ens-lyon.fr/www/gappa/`.

It will in the future be able to interact seamlessly with Why [2], a tool to certify programs written in a generic language. C and Java can be converted to this language.

Gappa manipulates arithmetic expressions on real and rational numbers and their evaluations on computers. Exact and rounded expressions are bounded using interval arithmetic [13], forward error analysis and properties of dyadic fractions. To the authors' best knowledge, Gappa is the first tool that can convert some of the simple tasks performed here into formal proofs validated by an automatic proof checker. Such goal has previously been quoted as *invisible formal methods* [17] in the sense that Gappa delivers formal certificates to users that are not expected to write any piece of proof in any formal proof system.

Gappa produces a Coq file for a given input script. Users do not need to be able to write the Coq file but they can check the work of Gappa by reading it or parsing it automatically. It contains `Variable`s, `Definition`s, `Notation`s, `Lemma`s and comments are between `(*` and `*)` signs. Although enclosure is the only predicate available to users, Gappa internally relies on more predicates to describe properties on expressions. All the properties of the input script are defined in the Coq file. Validity of proofs can automatically be checked by Coq. More insights to Gappa are presented in [4].

2 A new lemma that might be used by Gappa for inductive linear bounds

A key application in exact linear algebra is the resolution of triangular systems over finite fields presented in Figure 1. A delayed prime field arithmetic version of this algorithm can be constructed by simply doing a delayed matrix multiplication on the operation $B_1 := B_1 - A_2X_2$. Listing 1 performs such matrix multiplication `DGEMM_NEG` with no reduction. We used naming conventions of BLAS and LAPack for the function and the parameter names. For the

Input: $A \in \mathbb{Z}/p\mathbb{Z}^{N\times N}$, $B \in \mathbb{Z}/p\mathbb{Z}^{N\times K}$.
Output: $X \in \mathbb{Z}/p\mathbb{Z}^{N\times K}$ such that $AX = B$.

if N=1 **then**
$X := A_{1,1}^{-1} \times B$.
else *(splitting matrices into* $\lfloor \frac{N}{2} \rfloor$ *and* $\lceil \frac{N}{2} \rceil$ *blocks)*

$$\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}}^{B}$$

X_2 :=`LZ_TRSM(`A_3, B_2`)`.
$B_1 := B_1 - A_2X_2$.
X_1 :=`LZ_TRSM(`A_1, B_1`)`.
return X.

Figure 1. First algorithm for `LZ_TRSM(`A, B`)`

sake of simplicity some parameters have be omitted and some function names were slightly modified.

The `DREMM` function computes the remainder modulo p of all the components of a matrix. We are overspecifying it for the purpose of this presentation as our proof never use the exact value of these remainders but the sole property that they are between 0 and $p-1$. In the definition of `is_exact_int_mat`, we use the predicate `exists`. The property described is that $X[i \times LDX + j]$ is not any float, but it is an integer. To describe this, we require (or prove) that there exists an integer equal to the floating-point value of $X[i \times LDX + j]$. We also use `\old` in the annotations: the value `\old(X)` represents the value of the variable `X` before the function execution. It allows us to specify the outputs depending on the inputs, even if the pointed values are modified. Some ghost variables are introduced in our example to ease the proofs. A mechanism not presented here prevents such ghost variables to remain in the code once compiled.

Let M be the width of matrix A_2 or the height of vector X_2 and `FP_EPSILON` be the machine ϵ for `fp` in `float`, `double` or `long double` and `FP` in `FLT`, `DBL` or `LDBL` respectively. The `DREMM` reduction can be delayed until the end of `DGEMM_NEG` provided

$$(p-1)^2 \times M \leq 2/\texttt{FP\_EPSILON},$$

as all the number between 0 and $2/$`FP_EPSILON` can be represented exactly with type `fp`.

Assertions on the `DGEMM_NEG` function generate proof obligation

$$-1 \leq \frac{Y[i \times LDY + k]}{(p-1)^2 \times (j+1)} = \frac{oYik - A[i \times LDA + j] \times X[j \times LDX + k]}{(p-1)^2 \times (j+1)}$$

for all iterations defined by i, j and k. It can be proved by induction on j with the following lemma that we proved in Coq and similar ones for the other relations.

$$\forall\ a, b, c, d, e \in \mathbb{R} \quad ; \quad e \leq \frac{a}{b} \ \wedge\ e \leq \frac{c}{d} \ \wedge\ 0 < b \times d \quad \Longrightarrow \quad e \leq \frac{a+c}{b+d}$$

Listing 1. Matrix-matrix multiplication Y <- Y - AX and component-wise remainder

```
#include <math.h>
typedef double fp;
#define FP_ESPILON DBL_EPSILON

/*@ logic real epsilon() { 2^^(-53) } @*/
/*@ logic real max_int() { 2 / epsilon() } @*/

/*@ predicate is_exact_int_mat (fp *X, int LDX, int N, int M) {
  @   \valid_range(X,0,LDX*N) && M <= LDX &&
  @   \forall int i; \forall int j; 0 <= i < N && 0 <= j < M =>
  @      \round_error(X[i*LDX+j])==0 && \exists int v;  X[i*LDX+j]==v
  @ } @*/

/*@ predicate is_exact_int_mat_bounded_by
  @ (fp *X, int LDX, int N, int M, int min, int max) {
  @   is_exact_int_mat(X,LDX,N,M) && \forall int i; \forall int j;
  @     0 <= i < N && 0 <= j < M => min <= X[i*LDX+j] <= max
  @ } @*/

/*@ requires (p-1)*(p-1)*M <= max_int() &&
  @   is_exact_int_mat_bounded_by(Y,LDY,N,K,0,p-1) &&
  @   is_exact_int_mat_bounded_by(A,LDA,N,M,0,p-1) &&
  @   is_exact_int_mat_bounded_by(X,LDX,M,K,0,p-1)
  @ assigns Y[..]
  @ ensures
  @   is_exact_int_mat_bounded_by(Y,LDY,N,K,(1-p)*(p-1)*M,p-1) @*/
void DGEMM_NEG (int N, int M, int K, int p,
                fp *A, int LDA, fp *X, int LDX, fp *Y, int LDY) {
  int i, j, k; fp oYiK;
  for (i = 0; i < N; i++)
    for (j = 0 ; j < M; j++)
      for (k = 0; k < K; k++) {
        oYik       = Y[i*LDY+k];
        Y[i*LDY+k] = Y[i*LDY+k] - A[i*LDA+j] * X[j*LDX+k];
        /*@ assert -1 <= Y[i * LDY + k] / ((p-1)*(p-1)*(j+1)) &&
                         Y[i * LDY + k] <= p-1                @*/
      }
}

/*@ requires is_exact_int_mat(X,LDX,N,K)
  @ assigns X[..]
  @ ensures is_exact_int_mat_bounded_by(X,LDX,N,K,0,p-1) &&
  @   \forall int i; \forall int j; 0 <= i < N && 0 <= j < K =>
  @     \exists int d; X[i*LDX+j] == \old(X[i*LDX+j]) + d*p @*/
void DREMM (int N, int K, int p, fp *X, int LDX) {
  int i, k;
  for (i = 0; i < N; i++) for (k = 0; k < K; k++) {
    X[i*LDX+k] = fmod (X[i*LDX+k], p);
    if (X[i*LDX+k] < 0) X[i*LDX+k] += p;
  }
}
```

Gappa does not handle arrays so we have to rename $A[i \times LDA + j]$ to `Aik`, $X[j \times LDX + k]$ to `Xjk` and $Y[i \times LDY + k]$ to `Yik`. The following Gappa text should be sufficient to prove the generic case of the induction if our lemma is added to Gappa.

```
{
  p in [2,1b53]                                    ->
  j in [1,1b53]                                    ->
  oYik in [-1b53,1b53]                             ->
  Aij/(p-1) in [0,1]                               ->
  Xjk/(p-1) in [0,1]                               ->
  oYik / ((p-1)*(p-1)*j) >= -1                     ->
  ((p-1)*(p-1)*j) * ((p-1)*(p-1)) >= 0             /\
  -Aij*Xjk / ((p-1)*(p-1)) >= -1                   /\
  (oYik - Aij*Xjk) / ((p-1)*(p-1)*(j+1)) >= -1
}
-Aij*Xjk / ((p-1)*(p-1)) ->
  - (Aij/(p-1)) * (Xjk/(p-1)) { (p-1) <> 0 };
(oYik - Aij*Xjk) / ((p-1)*(p-1)*(j+1)) ->
  (oYik + (- Aij*Xjk)) / (((p-1)*(p-1)*j) + ((p-1)*(p-1)))
  { ((p-1)*(p-1)*(j+1)) <> 0 , ((p-1)*(p-1)*j) * ((p-1)*(p-1)) <> 0 };
```

A Gappa file usually starts with aliases. Gappa uses them for its outputs and in the formal proof instead of machine generated names. We do not need any alias here. Identifiers are assumed to be universally quantified over the set of real numbers the first time Gappa encounters them.

The main assertion is written between brackets ({ }). The hypotheses end with the last line finished by an implication sign (`->`). Each states that a variable or an expression is within an interval or bounded. Note that $p1 \rightarrow p2 \rightarrow p3$ is logically equivalent to $p1 \wedge p2 \rightarrow p3$. The goal is next. It is a conjunction (`/\`). Statements about intermediate variables are given as goals to force Gappa to establish them first.

The statement after the brackets are hints that propose replacements. Gappa replaces the left side of the `->` sign by the right side as soon as it encounters the former. It also tries to prove that the replacements are valid. They help Gappa identify the proper theorems syntactically. Conditions on the validity of the hints are expressed between brackets.

3 Variable interchange in a predicate

For the sake of completeness we recall an optimal bound on integer coefficients growth during backward substitution.

Corollary. *[7, corollary 3.3] Let $A \in \mathbb{Z}^{N \times N}$ be a unit diagonal upper triangular matrix, and $b \in \mathbb{Z}^N$, with $|A|, |B| \leq p-1$. Then $x \in \mathbb{Z}^N$ the solution of the system $Ax = B$ is such that*

$$|x| \leq \frac{p-1}{2} \left[p^{N-1} + (p-2)^{N-1} \right],$$

and this bound is optimal.

As this formula also bounds all the intermediate values, enough bits are available to combine a few recursion steps of `LZ_TRSM` without reduction and guarantee that all numerical results are exact. We present in Listing 2 an improved version for the delayed prime field arithmetic by replacing the last levels of the recursion by calls to `DTRSM` numerical solver according to

Listing 2. Delayed solution of a unitary triangular system over a finite field

```
/*@ predicate delay(int N, int p) @*/
/*@ logic int l_pmax(int n) @*/

// Floating point exact solution to a small unitary triangular system
/*@ requires N <= 54 &&
  @   is_exact_int_mat_bounded_by(X,LDX,N,K,0,l_pmax(N)-1) &&
  @   is_exact_int_mat_bounded_by(A,LDA,N,N,0,l_pmax(N)-1)
  @ assigns X[..]
  @ ensures
  @   is_exact_int_mat_bounded_by(X,LDX,N,K,-max_int(),max_int()) @*/
void DTRSM (int N, int K, fp *A, int LDA, fp *X, int LDX) {
  int i, j, k;
  for (i = N-2; i >= 0; i--)
    for (j = i+1; j < N; j++)
      for (k = 0 ; k < K; k++)
        X[i*LDX+k] = X[i*LDX+k] - A[i*LDA+j] * X[j*LDX+k];
}

/*@ requires (p-1)*(p-1)*N <= max_int() &&
  @   is_exact_int_mat(A,LDA,N,N) && is_exact_int_mat(B,LDB,N,K)
  @ assigns B[..]
  @ ensures is_exact_int_mat(B,LDB,N,K) @*/
void LZ_TRSM (int N, int K, int Nmax, int p,
              fp *A, int LDA, fp *B, int LDB) {
  if (N <= Nmax) {
    /*@ assert N <= 54 && delay(N,p) @*/
    DTRSM (N, K, A, LDA, B, LDB); DREMM (N-1, K, p, B, LDB);
  } else {
    int P = N/2, G = N - P;
    LZ_TRSM (G, K, Nmax, p, A+P*(LDA+1), LDA, B+P*LDB, LDB);
    DGEMM_NEG (P, G, K, p, A+P, LDA, B+P*LDB, LDB, B, LDB);
    DREMM (P, K, p, B, LDB);
    LZ_TRSM (P, K, Nmax, p, A, LDA, B, LDB);
  }
}


/*@ ensures \forall int N; N <= \result => delay (N, p) @*/
int Nmax (int p) {
  fp pp = 1, p2 = 1; int N;
  for (N = 0; ((p-1)*(pp+p2))/2 < 2 / 2^53; N++)
    {pp *= p; p2 *= p-2;};
  return N;
}

/*@ ensures \result==l_pmax(N) &&
  @   \forall int p; p <= \result => delay (N, p) @*/
int pmax (int N) {
  int p; for (p = 1; N <= Nmax(p); p++);
  return p-1;
}
```

Table 1. Maximum value `pmax` of parameter `p` for each value of parameter `N` allowed

N	2	3	4	5	6	7	8	9	10	11	12
$p_{\max}$	94906266	208064	9739	1553	457	191	97	59	39	29	19

N	13	14	15	$16 \cdots 19$	$20 \cdots 23$	$24 \cdots 34$	$35 \cdots 54$
$p_{\max}$	17	13	11	7	5	3	2

Table 2. Time to establish that no round-off occurred and generate a Coq proof script

Script	N	2	3	4	5	6	7	8	9	10	11
no	time(s)	00.02	00.06	00.10	00.19	00.29	00.43	00.57	00.74	00.93	01.15
Coq		00.02	00.06	00.13	00.22	00.35	00.50	00.87	01.35	01.84	02.43

Script	N	12	13	14	15	17	19	23	34	54
no	time(s)	01.38	01.63	01.91	02.90	02.90	03.73	05.93	13.71	38.32
Coq		03.22	03.77	05.26	07.13	12.22	19.50	38.13	421.4	5767

the above corollary. Recursion is stopped for the maximal integer $N_{\max}$ such that

$$\frac{p-1}{2}\left[p^{N_{max}-1}+(p-2)^{N_{max}-1}\right] \leq 2/\texttt{FP\_EPSILON}. \tag{1}$$

Function `Nmax` of Listing 2 uses a strict inequality in equation (1) to control the loop because it computes the bound with the same floating point format than the one used by `LZ_TRSM`. We focus now on proving that the `DTRSM` function invoked by `LZ_TRSM` never produce any round-off error.

One could port the proof of Corollary 3.3 [7] to Coq to finish the proof of correction. We decided to use Gappa and simple techniques that could be made automatic. Syntactically, the `DTRSM` function is invoked by `LZ_TRSM` only if the condition $delay(N,p)$ is fulfilled. We may define it by:

$$delay(N,p) = N \leq N_{\max}(p).$$

Gappa does not handle loops and branches. We perform a case analysis but p may vary from 2 to $94,906,266$. On the other hand, `N` varies only between 2 and 54. Table 1 presents the value computed by the `pmax` function. Variable interchange should allow to prove that `DTRSM` is invoked only on the condition

$$delay(N,p) \Longleftrightarrow p \leq p_{\max}(N).$$

A C++ class produces a trace for `N` between 2 and 54 where all branches and control statements have been removed. Each trace contains only floating point operations. Table 2 gives the time needed by Gappa on a 1.86 GHz Quad-Core Intel Xeon Processor with 2x4MB L2 cache and 2x1GB of memory, to produce each of the proofs that no round-off error occurred for all the values of N between 2 and 54. That ends the proof of correction of the algorithm.

4 Perspectives and concluding remarks

This report presents a new use of Gappa based on proof obligations generated from a trace of execution [14]. It would enable us to prove in Coq in the future that expression swell within the studied algorithm of delayed finite field arithmetic does not introduce round off errors [7]. This full certification will not be obtained by porting the initial proof to Coq but by a case analysis on the 53 possible values of one argument N. Pieces of the formal proof have been generated by Gappa for each individual value of N.

Our approach can be easily reproduced to other exact linear applications over finite fields. More precisely, the FFLAS-FFPACK project has been successful on using delayed prime field arithmetic for linear algebra applications.

`http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/FFLAS/`

References

[1] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development.* Springer-Verlag, 2004.

[2] S. Boldo and J.-C. Fillitre. Formal verification of floating point programs. In *Proceedings of the 18th Symposium on Computer Arithmetic*, Montpellier, France, 2007.

[3] M. Daumas and G. Melquiond. Generating formally certified bounds on values and round-off errors. In *Real Numbers and Computers*, pages 55–70, Dagstuhl, Germany, 2004.

[4] M. Daumas and G. Melquiond. Generating certified properties for numerical expressions and their evaluations. Technical Report hal-00127769, Centre pour la Communication Scientifique Directe, Villeurbanne, France, 2007.

[5] M. Daumas, G. Melquiond, and C. Muñoz. Guaranteed proofs using interval arithmetic. In P. Montuschi and E. Schwarz, editors, *Proceedings of the 17th Symposium on Computer Arithmetic*, pages 188–195, Cape Cod, Massachusetts, 2005.

[6] F. de Dinechin, C. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1318–1322, Dijon, France, 2006.

[7] J.-G. Dumas, P. Giorgi, and C. Pernet. FFPACK: Finite field linear algebra package. In *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 63–74, New York, NY, USA, 2004. ACM Press.

[8] W. Eberly, M. Giesbrecht, P. Giorgi, A. Storjohann, and G. Villard. Solving sparse rational linear systems. In *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 63–70, New York, NY, USA, 2006. ACM Press.

[9] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, pages 173–177, Berlin, Germany, 2007.

[10] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–47, 1991.

[11] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Technical report, University of Texas, 2002. FLAME working note #9.

[12] J. Harrison. Formal verification of floating point trigonometric functions. In W. A. Hunt and S. D. Johnson, editors, *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 217–233, Austin, Texas, 2000.

[13] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied interval analysis*. Springer, 2001.

[14] G. Melquiond and S. Pion. Formally certified floating-point filters for homogeneous geometric predicates. *Theoretical Informatics and Applications*, 2007. To appear.

[15] R. Michard, A. Tisserand, and N. Veyrat-Charvillon. Optimisation d'oprateurs arithmtiques matriels base d'approximations polynomiales. In *Symposium en Architecture de Machines*, pages 1318–1322, Perpignan, France, 2006.

[16] J. Rushby and F. von Henke. Formal verification of algorithms for critical systems. In *Proceedings of the Conference on Software for Critical Systems*, pages 1–15, New Orleans, Louisiana, 1991.

[17] A. Tiwari, N. Shankar, and J. Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, 2003.

[18] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, Jan. 2001. `www.elsevier.nl/gej-ng/10/35/21/47/25/23/article.pdf`.