Software Carry-Save: A case study for instruction-level parallelism

David Defour, Florent de Dinechin

ENS Lyon, 46 allée d'Italie, 69364 Lyon, France
David.Defour@ens-lyon.fr, Florent.de.Dinechin@ens-lyon.fr

Abstract. This paper is a practical study of the performance impact of avoiding data-dependencies at the algorithm level, when targeting recent deeply pipelined, superscalar processors. We are interested in multiple-precision libraries offering the equivalent of quad-double precision. We show that a combination of today's processors, today's compilers, and algorithms written in C using a data representation which exposes parallelism, is able to outperform the reference GMP library which is partially written in assembler. We observe that the gain is related to a better use of the processor's instruction parallelism.

1 Introduction: Modern Superscalar Processors

The increase of performance of recent microprocessors is largely due to the ever-increasing internal parallelism they offer [8]:

- All the workstation processors sold in 2003 possess several functional units which can execute instructions in parallel: between 2 and 4 memory units, usually 2 double-precision floating-point (FP) units, and between 2 and 6 integer units. The capabilities of these units vary widely.
- All these processors are also pipelined, currently with 8 to 20 pipeline stages. More specifically, we focus in the following on the pipeline of integer processing units, characterized by its *latency* and *throughput* as given in Table 1. Pipelines also means parallelism: The table shows for instance that 4 integer multiplications may be running in parallel at a given time in the Pentium-III multiplier.

Integer addition is an ubiquitous operations in typical code, and one-cycle adder units are cheap, so all processors offer several of them. Most processors (Alpha, Pentium III, Athlon, PowerPC) also possess one integer multiplier. However, a recent trend (Pentium IV, UltraSPARC, Itanium) is to make without this integer multiplier, and to delegate the (relatively rare) integer multiplications to an FP multiplier, at the expense of a higher latency due to additional translation costs. As the Itaniums have two identical FP units each capable of multiplication, they are the only architectures in this table on which more than one multiplication can be launched each cycle.

Architecture	concurrent simple integer (Latency/Throughput)	concurrent multiplications (Latency/Throughput)
Pentium III	2 (1/1)	1 (4/1)
UltraSPARC II	2 (1/1)	1 (5-35/5-35)
Alpha EV6/EV7	4 (1/1)	1 (7/1)
AMD Athlon XP	3 (1/1)	1 (4-6/3)
Pentium IV	3 (0.5-1/0.5-1)	1 (15-18/5)
PowerPC G4	3 (1/1)	1 (4/2)
Itanium	4 (1/1)	2 (18/1)
Itanium 2	6 (1/1)	2 (16?/1)

Table 1. Integer unit characteristics. Simple integer means add/subtract, boolean operations, and masks. A latency of l means that the result is available l cycles after the operation has begun. A throughput of n means that a new instruction may be launched every n cycles. This data is extracted from vendor documentation and other vendor-authored papers, and should be taken with caution as many specific architectural restrictions apply. The reader interested in these questions is invited to browse the `mpn` directory of the GMP source code [1], probably the most extensive and up-to-date single source of information on the integer capabilities of processors.

As processors offer ever more parallelism, it becomes increasingly difficult to exploit it. Instruction parallelism is limited by *data-dependencies* of several kinds, and by *structural hazards* [8]. Compilers and/or hardware try to allocate resources and schedule instructions so as to avoid them.

In this paper, we consider several algorithms for multiple-precision, and we show experimentally that on the latest generations of processors, the best algorithm is not the one which executes less operations, but the one which exposes more parallelism.

2 Multiple-Precision as an Algorithmic Benchmark

Most modern computers obey the IEEE-754 standard for floating-point arithmetic, which defines the well-known *single* and *double precision* FP formats. For applications requiring more precision (numerical analysis, cryptography or computational geometry), many general-purpose multiple-precision (MP) libraries have been developed [4–6, 9, 1]. Some offer arbitrary precision with static or dynamic precision control, other simply offer a fixed precision which is higher than IEEE-754 double precision. Here we focus on libraries able to offer *quad-double* precision, *i.e.* 200-210 bits of precision. This is the precision required for computing elementary functions correctly rounded up to the last bit, which is the subject of our main current research.

All libraries code MP numbers as arrays of *machine numbers*, *i.e.* numbers in a native format on which the microprocessor can directly compute: Integer, or IEEE-754 FP numbers. They all also use variations of the same basic multiple-precision algorithms for addition and multiplication, similar to those learnt in

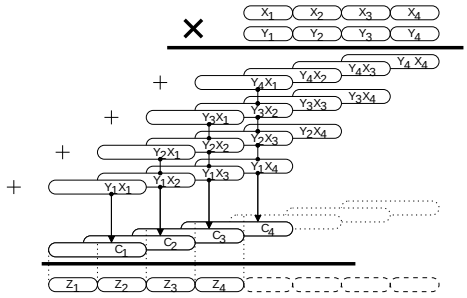

Fig. 1. Multiple-Precision multiplication

elementary school for radix-10 numbers.[1] Figure 1 depicts the algorithm for the multiplication. This figure represents the two input numbers X and Y, decomposed into their n digits x_i and y_i (with $n = 4$ on the figure). Each digit is itself coded in m bits of precision. An array of partial products $x_i y_j$ (each a $2m$-bit number) is computed, then summed to get the final result.

There is a lot of intrinsic parallelism in this algorithm: The partial products can all be computed in parallel, as can the column sums. However the intermediate sums may require up to than $2m + \log_2 n$ bits, while digits of the result are expected to be m-bit numbers like the inputs. Some conversions of large numbers to smaller ones must therefore take place. For example, in the classical pencil-and-paper algorithm in base 10, this conversion takes the form of a *carry propagation*, with right-to-left data-dependencies that do not appear on Fig. 1. These dependencies are a consequence of the representation of the intermediate results, constrained here to be single digits.

There are many other ways to implement Fig. 1, depending on the data representation of the digits, which entail in turn specific data-dependencies. This explains the variety of MP algorithms.

Dense high-radix representation The GNU Multiple-Precision (GMP) package uses a direct transposition of the pencil-and-paper sequential algorithm. The difference is that the digits are machine integers (of 32 or 64 bits on current processors). In other words the radix of the representation is 2^{32} or 2^{64} instead of 10. Carry propagation uses processor-specific *add-with-carry* instructions, which are present in all processors but inaccessible from high-level language. This is one reason for which GMP uses assembly code for its inner loops. The other reason is, of course, performance.

However, on pipelined processors, these carry-propagation dependencies entail pipeline stalls, which GMP programmers try to avoid by filling the pipeline bubbles with useful operations like loop handling and memory accesses (see the

[1] Other algorithms exist with a better asymptotic complexity, for example Karatsuba's algorithm [10]. They are relevant for precision much larger than quad-double.

well-commented source [1]). For recent processors this is not enough, and the latest versions of GMP try to compute two lines of Fig. 1 in parallel. All this needs a deep insight in the execution behaviour of increasingly complex processors.

Bailey's MPFUN [3] is a dense high-radix MP package where the digits are FP numbers instead of integers. In this case, there is no carry, but one has to recover and propagate FP rounding errors, using fairly different algorithms. Due to lack of space we do not describe them here.

Software Carry-Save Another option is to avoid the previous right-to-left carry propagation altogether, by ensuring that all the intermediate results of Fig. 1 (including intermediate sums, not shown) fit on a machine number. To achieve this, the digits of the inputs and output don't use all the precision available in the machine format: Some of the bits are reserved (set to zero), to be used by the MP algorithms to store intermediate carries. The *carry-save* denomination is borrowed from a similar idea widely used in hardware [11, 12].

This idea is first found in Brent's MP library [4] with integer digits. His motivation seems to have been *portability*: Where GMP uses assembler to access the *add-with-carry* instructions, in carry-save MP all the operations are known in advance to be exact, without overflow nor rounding. Therefore algorithms only use basic, and thus portable, arithmetic. The idea has been resurfacing recently: It seems to be used by Ziv [2] with FP digits. Independently, the authors developed the *Software Carry-Save* (SCS) library [7]. Initially we experimented with FP and integer digits, and found that integer was more efficient.

Our motivations for using carry-save MP were again portability (we use the C language), but also *efficiency*: Carry-save MP allows carry-free algorithms which, in addition of being simpler, exposes more intrinsic instruction-level parallelism. Note that there is a tradeoff there: More SCS digits are needed to reach a given precision than in the dense high-radix case, due to the reserved bits. Therefore more elementary operations will be needed.

The actual implementation of SCS uses a mixture of 32-bit and 64-bit arithmetic (well-supported by all processors/compilers and easy to express in the C language in a *de-facto* standard way). For quad-double precision, we use $n = 8$ digits, each digit using $m = 30$ bits of a 32-bit machine word. MP addition uses only 32-bit arithmetic. MP multiplication uses 64-bit arithmetic. As the partial products use 60 bits out of 64, a whole column sum can be computed without overflow. There is only one final carry-propagation in the MP multiplication, although with 36-bit carries. It is written in C using AND masks and shifts.

To sum it up, the SCS representation exposes the whole of the parallelism inherent to the MP multiplication algorithm. The following of the paper shows that the compiler can be trusted to detect and exploit this parallelism.

The library `scslib` is available under the GNU LGPL from
`www.ens-lyon.fr/LIP/Arenaire/`

3 Experiments and Timings

This section gives experimental measures of the performance of four available MP librairies ensuring about 210 bits of precision, on four recent microprocessors. The libraries are our SCS library, GMP [1] (more precisely it floating representation MPF), and two FP-based libraries, Bailey's quad-double library [9], and Ziv's library [2]. The systems considered are the following:

- *Pentium III* with Debian GNU/Linux, gcc-2.95, gcc-3.0, gcc-3.2
- *Pentium IV* with Debian GNU/Linux, gcc-2.95, gcc-3.0, gcc-3.2
- *PowerPC G4* with MacOS 10.2 and gcc-2.95
- *Itanium* with Debian GNU/Linux, gcc-2.95, gcc-3.0, gcc-3.2

The results are relatively independent on the compiler (we also tested other compilers by Sun and Intel). Each result is obtained by measuring the execution times on 10^3 random values (the same values are used for all the libraries). To leverage the effect of operating system interruptions, the tests are run several times and the minimum timing is reported. Care has also been taken to prefill the instruction caches with the library code before timing (by executing a few untimed operations), to chose a number of random values that fits in all the data-caches, and in general to avoid cache-related irrelevant effects.

We have timed multiplication, addition, and conversions to and from MP format for each library. We have also implemented a test on a "lifelike" application: The evaluation of a correctly rounded double-precision logarithm function. This application converts from double to MP, evaluates a polynomial of degree 20 which makes heavy use of multiplication and addition, then converts back to double. Results are summarized in Fig. 2.

A first glance at these graphs, given in the order of introduction of the respective processors, shows that the performance advantage of SCS over the other libraries seems to increase with each generation of processor. We relate this to the increase of internal parallelism, which favors the more parallel SCS approach. FP-based libraries suffer more, because FP addition is a multicycle, pipelined operation of increasing depth, whereas integer addition remains a one-cycle operation. This is the main reason why we chose integer arithmetic in SCS.

Concerning the timings of the conversions to and from FP, the two integer-based libraries have comparable performance, while the FP-based library have the potential of much simpler conversions. The differences observed reflect the facilities offered by the processors to convert machine integers to/from machine doubles. We didn't investigate the bad result of the FP-based Ziv library.

Concerning the arithmetic operations, GMP and SCS have a clear lead over the FP-based libraries. In the following, we therefore concentrate on these two libraries. Let us review the effects which may contribute to a performance difference between SCS and GMP:

1. The SCS library (like IBM's and Bailey's) provides fixed accuracy selected at compile time, whereas GMP is an arbitrary-precision library. This means that the former use almost only fixed loop (which can be unrolled), whereas the latter must handle arbitrary-length loops.

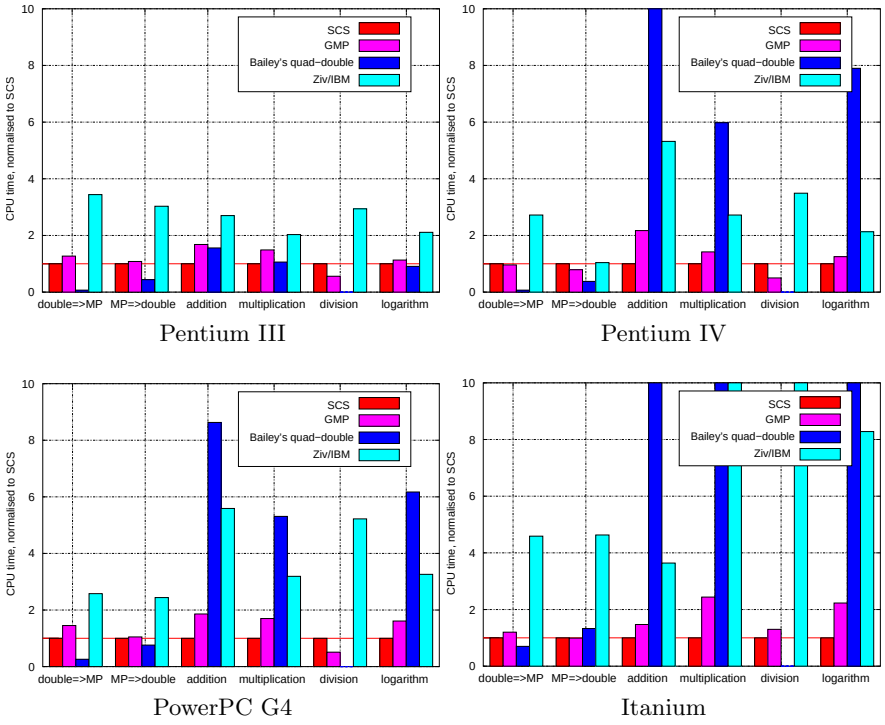

Fig. 2. Compared MP timings on several processors. For the sake of clarity we have normalised results to the SCS timing for each function on each tested architecture: The bars do not represent absolute time. An absent bar means that the corresponding operation showed compilation or runtime errors on this architecture.

2. SCS performs less carry propagations, and therefore less work per digit.
3. GMP uses assembly code, and uses processor-specific machine instructions (the so-called "multimedia extensions") when they help, for example on the Pentium IV architecture.
4. GMP needs less digits for a given precision.
5. SCS exposes parallelism.

Addition benefits from simplicity The first effect accounts for the performance difference in the addition. The algorithms for SCS and GMP addition present similar complexity and data-dependencies, and should exhibit similar performance. However, the cost of loop handling (decrement the loop index, compare it to zero, branch, with a possible pipeline hazard) far exceeds the cost of the actual computation (one *add-with-carry*). The only reason why SCS is faster than GMP here is therefore that its loops are static and may be unrolled.

Multiplication benefits from parallelism On those architectures which can only launch one multiplication each cycle (all but Itanium), the performance ad-

vantage for the multiplication is similar to that of the addition, and for the same reasons. However, on the Itanium architecture, which can launch two pipelined multiplications each cycle, the performance advantage of SCS multiplication over GMP is much higher than that of the addition. This tends to show that GMP fails to exploit this parallelism. To verify that SCS does exploit it, we had a look at the SCS machine code generated by the compiler. The Itanium machine language is interesting in that it explicitely expresses instruction-level parallelism. We could observe that among the 40 fused multiply-and-add involved in the computation of one SCS multiplication, there were 9 places where two multiplications were lauched in parallel. An example of this code is given below.

```
(...)
;;
getf.sig r18 = f6
xma.l f7 = f33, f11, f0
xma.l f6 = f37, f15, f0
;;
add r14 = r18, r14
xma.l f11 = f13, f11, f9
xma.l f8 = f14, f12, f0
;;
(...)
```

The `;;` delimitate *bundles* of independant expressions that can be launched in parallel.

`xma` is the integer multiply-and-add instruction.

Only 9 out of 40 is a relatively disappointing result. Should we blame the compiler ? Remember that each multiply-and-add instruction needs to be surrounded with two long-latency instructions which transfer the data from the integer datapath to the FP datapath and back (the `getf` instruction above). Initially loading the input digits from memory is also a long-latency operation. These structural hazards probably prevent exploiting the full parallelism of Fig. 1.

Applications: Division and logarithm Concerning division, the algorithms used by SCS and GMP are completely different: SCS division is based on a Newton-Raphson iteration, while GMP uses a digit-recurrence algorithm [11, 12]. These results suggest an obvious improvement to the SCS library.

Finally, the logarithm performance is very close to the multiplication performance: The bulk of the computation time is spent in performing multiplications. We believe that this is a typical application. It clearly justifies the importance of exploiting parallelism in the MP multiplication.

4 Conclusion and Future Work

We have presented and compared measures of performance of several multiple-precision libraries. Our main result is that a MP representation which wastes space and requires more instructions, but exposes parallelism, is a sensible choice on today's deeply pipelined, superscalar processors. Although written in a high-level language in a portable way, our SCS library is able to outperform GMP, a library partially written in handcrafted assembly code, on a range of processors.

It may be safely expected that future processors will offer even more parallelism. This may take the form of deeper pipeline, although the practical limit is not far from beeing reached [13]. We also expect that future processors will be able to lauch more multiplications each cycle, either in the Itanium fashion (several fully symmetric FP units each capable of multiplication and addition), or through ever more powerful multimedia instructions. The current trend towards hardware multithreading also justifies increasing the number of processing units.

In this case, the SCS approach will prove increasingly relevant, and multiple-precision computing may become another field where assembly programming is no longer needed. Using Brent's variant [4], where carry-save bits impose a carry-propagation every 2^{M-m} bits, these ideas may even find their way into the core of GMP. The pertinence of this approach and the tradeoffs involved remain to be studied.

References

1. GMP, the GNU multi-precision library. `http://swox.com/gmp/`.
2. IBM accurate portable math. library. `http://oss.software.ibm.com/mathlib/`.
3. David H. Bailey. A Fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4):379–387, 1995.
4. Richard P. Brent. A Fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, 4(1):57–70, 1978.
5. K. Briggs. Doubledouble floating point arithmetic. `http://members.lycos.co.uk/keithmbriggs/doubledouble.html`.
6. Marc Daumas. Expansions: lightweight multiple precison arithmetic. In *Architecture and Arithmetic Support for Multimedia*, Dagstuhl, Germany, 1998.
7. D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, Beijing, China, 2002. Updated version of LIP research report 2002-08.
8. John L. Hennessy and David A. Patterson. *Computer architecture: A quantitative approach (third edition)*. Morgan Kaufmann, 2003.
9. Yozo Hida, Xiaoye S. Li, and David H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In Neil Burgess and Luigi Ciminiera, editors, *15th IEEE Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, June 2001.
10. Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962.
11. I. Koren. *Computer arithmetic algorithms*. Prentice-Hall, 1993.
12. B. Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. Oxford University Press, 2000.
13. Y. Patt, D. Grunwald, and K. Skadron, editors. *Proceedings of the 29th annual international symposium on Computer architecture*. IEEE Computer Society, 2002.

Acknowledgements

The support of Intel and HP through the donation of an Itanium based machine is gratefully acknowledged. Some experiments were also performed thanks to the HP TestDrive program.