```
        ; Before applying VFCInstrument

        %30 = fsub float %28, %29
        %31 = load float, float* %7, align 4
        %32 = fsub float %30, %31
        store float %32, float* %6, align 4
        %33 = load float, float* %8, align 4
        store float %33, float* %5, align 4
        br label %34

        ; After applying VFCInstrument

        %30 = call float @_floatsub(float %28, float %29)
        %31 = load float, float* %7, align 4
        %32 = call float @_floatsub(float %30, float %31)
        store float %32, float* %6, align 4
        %33 = load float, float* %8, align 4
        store float %33, float* %5, align 4
        br label %34
```

Figure 6.2: A short IR excerpt from a Kahan summation algorithm before and after applying *VFCInstrument* pass.

VFCFuncInstrument instruments library and user defined functions. This pass provides multiple features.

First, the pass maintains a lightweight stack-trace of the called functions; allowing us to retrieve the calling context. The stack-trace is updated at each function prologue and epilogue. Stack unwinding libraries such as *libunwind* or the `backtrace()` call also provide the stack-trace. But these functions need to unwind the frame-pointer chain for each call. Verificarlo has a trace mode where the calling context is recorded for each FP call; in this case unwinding the stack is much too costly compared to updating the stack-trace twice per function.

Second, the pass captures and instruments the scalar arguments and return values of functions. This is useful to build a profile of the range and precision used for each argument and return value. Optionally, *VFCFuncInstrument* can apply the vprec backend on each argument or return value to reduce its precision. Using this feature, one can measured the effect of reducing the precision during function calls; such as calls to mathematical library functions.

6.2 Monte Carlo arithmetic backend

As previously discussed in chapter 5, Monte Carlo Arithmetic is a powerful framework to understand the numerical stability of a function or program.

We recall that the result in Monte Carlo arithmetic of $x \circ y$ with $\circ \in \{+, -, *, /\}$ and $x, y \in \mathbf{F}$ is computed as:

- $\text{mca}(x \circ y) = \text{round}(\text{inexact}(\text{inexact}(x) \circ \text{inexact}(y)))$ in full MCA mode,
- $\text{mca}(x \circ y) = \text{round}(\text{inexact}(x \circ y))$ in RR mode,

The inexact function models FP errors with ξ an uniformly distributed random variable in the range $\left(-\frac{1}{2}, \frac{1}{2}\right)$

- When z is representable, $\text{inexact}(z) = z$
- When z is inexact, $\text{inexact}(z) = z + 2^{e_z - t}\xi$

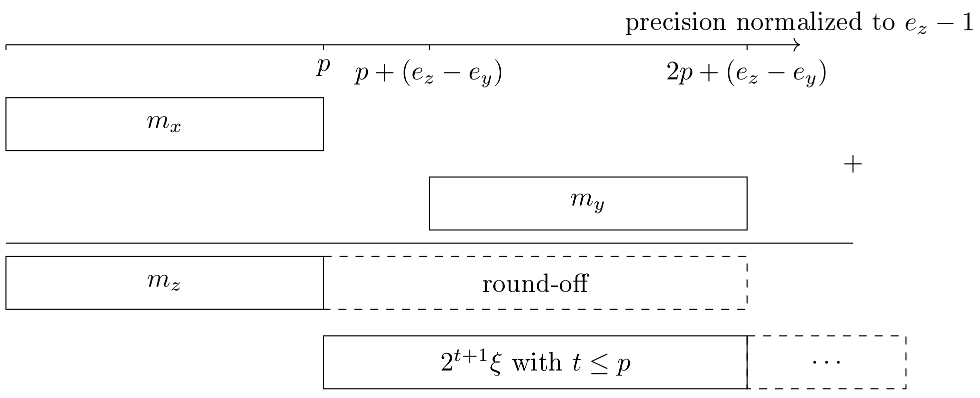

Figure 6.3: MCA computation $inexact(z) = inexact(x + y)$ with $e_z = e_x$.

where t is the virtual precision and $e_z = \lfloor\lfloor \log_2(z) \rfloor\rfloor + 1$ is the order of magnitude of z.

In Stott Parker model's, all the operations performed in the right-hand side before the final rounding must be done in infinite precision. The final round operator, round the result using round-to-nearest to the machine precision p, ensuring that the MCA computation fits into the memory allocated for the original IEEE 754 result. Therefore, when instrumenting a program with Monte Carlo backend Verificarlo does not need to change the memory layout.

6.2.1 Implementing MCA with limited precision

When implementing MCA in software we cannot use infinite precision to compute the inexact terms. Let us consider the addition between two positive representables $z = x + y$ with $e_x \geq e_y$ and $e_x = e_z$. When summing, both values are normalized to the largest exponent e_z; so the mantissa of y is shifted to the right by $(e_z - e_y)$ bits as shown in figure 6.3.

$$inexact(z) = x + y + 2^{e_z - t}\xi = 2^{e_z - 1}(m_x + 2^{e_y - e_z}m_y + 2^{t+1}\xi)$$

Because $t \in [0, p]$, the ξ mantissa left position is between 1 and $p + 1$. The mantissa of ξ has an infinite number of random bits, but it is sufficient to sum together the bits of ξ that coincide with round-off bits. Since there is an infinite number of ξ bits to the right of $2p + (e_z - e_y)$, they can simply be taken into account in the final rounding by setting the sticky bit to one.

Nevertheless, this leaves us with a computation that must be performed with $2p + (e_z - e_y)$ bits. With binary64 numbers, $p = 53$ and normal exponents are in $[-1022, 1023]$, therefore we have a required precision of 2151 bits. Such a computation is much too costly.

To be efficient, our implementation will limit both the precision in which inexact terms are computed and the number of random bits in ξ. In verificarlo we choose to use p random bits in m_ξ and an extended precision of $2p$. This follows Stott Parker recommendations [136, p. 38].

This choice is coherent, indeed for the maximum virtual precision $t = p$, the stochastic term is shifted just after the mantissa,

$$inexact(z) = z + 2^{e_z - p}\xi = (-1)^{s_z} 2^{e_z - 1}(m_z + 2^{p+1}\xi)$$

If we represent the stochastic error with p random bits, then we need $2p$ bits to store the resulting mantissa $m_z + 2^{p+1}\xi$.

Similarly, when working with the full MCA mode, the inexact function is applied on the operands and the computation is performed with $2p$ bits of precision.

6.2.2 Quad and MPFR backends

The original MCA backend of Verificarlo relied on mcalib [49]. Mcalib uses the GNU MPFR multi-precision floating-point library to perform the operations in extended precision. This simplifies the implementation since the extended precision operations are handled by a general library, but is not particularly efficient.

To optimize the original backend, we decided to use more efficient extended precision computations. When the original operands' type is binary32, we can use the standard binary64 IEEE-754 computations to compute the MCA result since it offers more than twice the precision of binary32. This is particularly efficient because binary64 computations are done in hardware.

When the original operands's type is binary64, we use GCC libquadmath. This library implements a quadruple precision format with a mantissa of 112 bits; on current architectures the quadruple operations are emulated in software. This faster MCA backend is called the *quad* backend.

Using binary64 and quadruple types for performing the computation handles nicely the case of denormalized values. Should x or y be denormals, once converted to the extended precision they become normal.

In both MPFR and Quad backends, ξ is represented as a FP value and the inexact(z) function is implemented with FP addition. This raises an interesting issue when z is close to a power of two.

Rounding issues Stott Parker defines nearest rounding as [136, p.13], $\text{round}(x) = 2^{e-p}\llbracket 2^{p-e}x + \frac{1}{2}\rrbracket$, noting that the definition is invalid when a carry-out happens during rounding.

First, let us assume that no carry-out happens and that this definition is valid. We can show that MCA RR mode at $p = t$ always rounds to one of the two closest representables of the result.

For z representable, the result follows from the definition: $\text{inexact}(z) = z$.

Let us consider the case where z is inexact with $\lfloor z \rfloor < z < \lceil z \rceil$ and $e = e_z$.

$$\text{inexact}(z) = \text{round}(z + 2^{e-p}\xi)$$

Because $-\frac{1}{2} < \xi < \frac{1}{2}$ and round is monotonous we have,

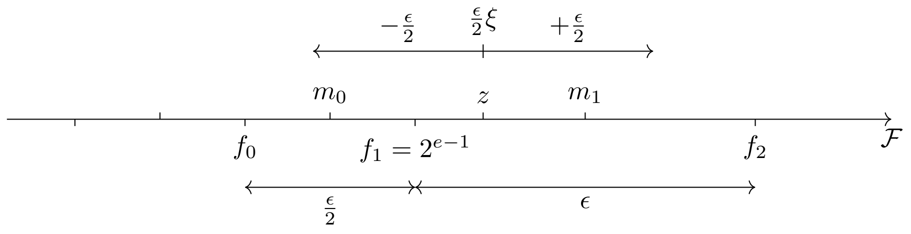

Figure 6.4: Rounding carry-out when z is close to a power of two.

$$\begin{aligned}
\mathrm{round}(z - 2^{-e-p-1}) &\leq \mathrm{round}(\mathrm{inexact}(z)) \leq \mathrm{round}(z + 2^{e-p-1}) \\
2^{e-p}\llbracket 2^{p-e}(z - 2^{e-p-1}) + \frac{1}{2}\rrbracket &\leq \mathrm{round}(\mathrm{inexact}(z)) \leq 2^{e-p}\llbracket 2^{p-e}(z + 2^{e-p-1}) + \frac{1}{2}\rrbracket \\
2^{e-p}\llbracket 2^{p-e}z\rrbracket &\leq \mathrm{round}(\mathrm{inexact}(z)) \leq 2^{e-p}\llbracket 2^{p-e}z + 1\rrbracket \\
\lfloor z \rfloor &\leq \mathrm{round}(\mathrm{inexact}(z)) \leq \lceil z \rceil
\end{aligned}$$

Therefore, *when no carry-out happens* during rounding, MCA RR at $t = p$ correctly rounds towards one of the two closests representables.

Let us now study the case when the mantissa underflows at rounding. This can happen when the rounded value is close to 2^{e-1}, and the exponent changes after rounding. To simplify the analysis we will consider $z > 0$, but the same reasoning applies for $z < 0$.

Let us consider the representable $f_1 = 2^{e-1}$. The next representable is $f_2 = f_1 + \epsilon$ with $\epsilon = 2^{e-p}$. The previous representable is $f_0 = f_1 - \frac{\epsilon}{2}$ because the distance between two representables is halved when crossing a power of two. The midpoints between f_0 and f_1, and between f_1 and f_2, are noted m_0 and m_1 respectively. Figure 6.4 places these three representables and their midpoints on the $\mathcal{F}$ axis.

We recall that

$$\begin{aligned}
\mathrm{round}(z - 2^{-e-p-1}) \leq &\mathrm{round}(\mathrm{inexact}(z)) \leq \mathrm{round}(z + 2^{e-p-1})) \\
\mathrm{round}(z - \frac{\epsilon}{2}) \leq &\mathrm{round}(\mathrm{inexact}(z)) \leq \mathrm{round}(z + \frac{\epsilon}{2})
\end{aligned}$$

This interval is represented as a double arrow in the top of figure 6.4. An incorrect rounding can only happen if the inexact value is lower than the midpoint m_0

$$\begin{aligned}
z - \frac{\epsilon}{2} <& m_0 \\
z <& f_1 - \frac{\epsilon}{4} + \frac{\epsilon}{2} \\
z <& 2^{e-1}(1 + 2^{-p-1})
\end{aligned}$$

Therefore, when $2^{e-1} \leq z < 2^{e-1}(1+2^{-p-1})$ the rounding can be incorrect; in that case the result will be f_0 which is neither $\lfloor z \rfloor = f_1$ nor $\lceil z \rceil = f_2$.

We can illustrate the issue with a numerical example taking $p = 3$, let's choose $z = 2^0(1 + 2^{-5}) = 1.00001$, so $e = 1$, $f_1 = 1.00$, and $f_0 = 0.111$, all floating point numbers noted in binary.

We know that $2^{e-p}\xi \in (-2^{e-p-1}, 2^{e-p-1})$, so -0.000111 is an admissible value for which $\text{round}(\text{inexact}(z)) = \text{round}(1.00001-0.000111) = \text{round}(0.111011) = 0.111 = f_0$. In this example, because we chose z close to a power of two, $\text{round}(\text{inexact}(z))$ has three possible realizations 0.111, 1.00 and 1.01. This does not match our expectations for a well behaved stochastic rounding operator since z is not always rounded to one of its closest representables. To avoid this issue we consider a third implementation of MCA where the inexact function is slightly altered.

6.2.3 MCA integer backend

The julia library `StochasticRounding.jl` [84] uses integer operations to perform the stochastic rounding. It is possible to extend their approach to compute the MCA inexact function using integers. We recall the original definition $inexact(z) = z + 2^{e_z - t}\xi = (-1)^{s_z}2^{e_z-1}(m_z + 2^{t+1}\xi)$.

First, we interpret z as an integer i. The first α bits of i contain the sign and the exponent, then bits $\alpha + 1$ to $\alpha + p$ contain m_z.

Then, we generate p random bits to form m_ξ the mantissa of ξ. The first bit acts as a sign bit in two-complement. We shift the mantissa to the right by $\alpha + t - 1$ bits, ensuring that the sign is extended during the shift. Next, we add the shifted random mantissa and i together,

$$i + (m_\xi >> (\alpha + t - 1))$$

Finally, we interpret back the result as a FP value and round it.

It is possible that the addition of i with the shifted random bits trigger a carry-out that increases or decreases the final exponent. Fortunately, this case is correctly handled on the integer computation because the IEEE-754 biased exponent representation ensures that an underflow or overflow from the mantissa carries on correctly to the exponent.

This particular case where an underflow changes the final exponent, corresponds exactly to the problematic case where z is close to a power of two. Interestingly, working with integers solves the problem raised in the previous section. Indeed, the stochastic noise in the mantissa will be naturally reinterpreted when the exponent is decreased. So after moving across a power of two boundary, the stochastic noise will be halved.

Taking the previous example in section 6.2.2 with $z = 1.00001 \times 2^0$. We will only consider the pseudo-mantissas and ignore the sign and exponent bits in our notations. The pseudo-mantissa of z is $m_z = 00001$ since the initial 1 bit is implicit.

The minimal ξ mantissa in two-complement is 101, we arithmetically shift it to the right by $t-1 = p-1 = 2$ bits getting 11101. Now we can compute the sum as $00001 + 11101 = 11110$. The mantissa underflows in two-complement so the resulting exponent decreases by one. The final result is $\text{inexact}(z) = 1.1111 \times 2^{-1}$

		verificarlo backends		
	original	IEEE	MCA quad	MCA int
Kahan binary32	1.34s	2.36s (×1.7)	6.28s (×4.7)	7.76s (×5.8)
Kahan binary64	1.34s	2.34s (×1.7)	105s (×78)	64s (×48)
NAS CG A	0.80s	6.41s (×8)	173s (×216)	128s (×160)

Table 6.1: E×ecution time (and slowdown) for a Kahan sum of 100 millions elements and for the NAS CG A using different Verificarlo backends.

which is above the midpoint 0.1111, so after the final rounding the result will be 1.0.

Unlike the quad backend, MCA int backend guarantees that for RR and $t = p$ the only possible results are the closest representables, even in the case of a mantissa underflow during rounding.

The inexact′ function implemented by MCA int can be written for $z \geq 0$ as:

- for $2^{e-1}(1+2^{-p-1}) \leq z < 2^e$, $\text{inexact}'(z) = \text{inexact}(z)$, because no underflow happens at rounding.
- for $2^{e-1} \leq z < 2^{e-1}(1 + 2^{-p-1})$,

$$\text{inexact}'(z) = \begin{cases} \text{inexact}(z) & \text{if inexact}(z) \geq \lfloor 2^{e-1} \rfloor \\ \frac{2^{e-1} - \text{inexact}(z)}{2} & \text{if inexact}(z) < \lfloor 2^{e-1} \rfloor \end{cases}$$

When no underflow happens, $inexact(z)$ and z have the same exponent. In that case, MCA int implements exactly the inexact function as defined by Stott Parker: $inexact(z) = z + 2^{e_z - t}\xi$.

One should note that with the new function, inexact(z) values are no longer uniformly distributed when crossing a power of two boundary. MCA int implements therefore a slightly different MCA arithmetic where the magnitude of the ξ noise is halved when the computed result underflows a power of two boundary. This new definition always rounds to one of the two closest representatives, which is a nice property for a rounding operator.

6.2.4 Performance evaluation of the MCA backends

Simulating MCA on software is costly. Here we evaluate the overhead of two benchmarks: the compensate Kahan summation algorithm and the NAS CG conjugate gradient benchmark.

Experiments were performed on a 6 core Coffee Lake i7-9850H at 2.60GHz with 15Gb of memory. The Kahan summation algorithm was run on an array with 100 million elements. NAS CG benchmark was run on dataset class A. The setup for both benchmarks can be found within the `tests/` directory shipped with verificarlo source-code.

Table 6.1 shows the execution time and overheads for the different backends. The IEEE backend in verificarlo, is a dummy backend where the FP hook mirror the standard IEEE-754 operation. It offers debugging possibilities and also provides a baseline to measure the instrumentation cost. On the Kahan