July 2017

Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error

Stefano Cherubin [a] Giovanni Agosta [a] Imane Lasri [b]
Erven Rohou [b] and Olivier Sentieys [b]
[a] *DEIB - Politecnico di Milano, name.surname@polimi.it*
[b] *INRIA, name.surname@inria.fr*

Abstract. Error-tolerating applications are increasingly common in the emerging field of real-time HPC. Proposals have been made at the hardware level to take advantage of inherent perceptual limitations, redundant data, or reduced precision input [20], as well as to reduce system costs or improve power efficiency [19]. At the same time, works on floating-point to fixed-point conversion tools [9] allow us to trade-off the algorithm exactness for a more efficient implementation. In this work, we aim at leveraging existing, HPC-oriented hardware architectures, while including in the precision tuning an adaptive selection of floating- and fixed-point arithmetic.

Our proposed solution takes advantage of the application domain knowledge of the programmers by involving them in the first step of the interaction chain. We rely on annotations written by the programmer on the input file to know which variables of a computational kernel should be converted to fixed-point. The second stage replaces the floating-point variables in the kernel with fixed-point equivalents. It also adds to the original source code the utility functions to perform data type conversions from floating-point to fixed-point, and vice versa. The output of the second stage is a new version of the kernel source code which exploits fixed-point computation instead of floating-point computation.

As opposed to typical custom-width hardware designs, we only rely on the standard 16-bit, 32-bit and 64-bit types. We also explore the impact of the fixed-point representation on auto-vectorization.

We discuss the effect of our solution in terms of time-to-solutions, error and energy-to-solution.

Keywords. Approximate Computing, Fixed-Point, Compilers

1. Introduction

High Performance Computing (HPC) has been traditionally the domain of grand scientific challenges and industrial sector such as oil & gas or finance, where investments are large enough to support massive computing infrastructures. Nowadays HPC is recognized as a powerful technology to increase the competitiveness of nations and their industrial sectors, including small scale but high-tech businesses – *to compete, you must compute* has become an ubiquitous slogan [2].

The current roadmap for HPC systems aims at reaching the Exascale level (10^{18} FLOPS) within the $2023-24$ timeframe – with a $\times 1000$ improvement over Petascale, reached in 2009, and a $\times 100$ improvement over current systems. Reaching Exascale poses the additional challenge of significantly limiting the energy envelope, while providing massive increases in computational capabilities – the target power envelope for future Exascale system ranges between 20 and 30 MW.

To fulfill the 20 MW target, energy-efficient heterogeneous supercomputers need to be coupled with software stacks able to exploit a range of techniques to trade-off between power, performance, and other metrics of quality to achieve the desired goals without exceeding the power envelope. In the recent years, customized precision has emerged as a promising approach to improve such power/performance trade-offs.

Customized precision originates from the fact that many applications can tolerate some loss of quality during computation, as in the case of media processing (audio, video and image), data mining, machine learning, etc. Error-tolerating applications are increasingly common in the emerging field of real-time HPC. Thus, recent works have investigated this line of research in the HPC domain as a way to provide a breakthrough in power and performance for the Exascale era. For example, several works [21,1] have studied the acceleration of physics simulation via an adaptive reduction of the precision from double- to single-precision floating-point, while in Göddeke et al. [5] the precision of native double-precision solvers is compared with emulated- and mixed-precision solvers of linear systems of equations as they typically arise in finite element discretisations. Other works [8,14] address precision tuning considering only floating-point data types. Finally, Lam and Hollingsworth [7] provide an approach to simulate the effects of precision tuning limited to floating-point data types, to support the programmer in a manual tuning effort.

Proposals have been made at the hardware level to take advantage of inherent perceptual limitations, redundant data, or reduced precision input [20], as well as to reduce system costs or improve power efficiency [19]. Furthermore, [6] is an early application-specific approach dealing with cases where inherent limitations in the data can be exploited.

At the same time, works on floating-point to fixed-point conversion tools [9, 11,3] offer the ability to trade-off the algorithm exactness for a more efficient implementation, providing either analytic or profile-based methods to obtain tight bounds on the numerical precision of fixed-point implementations. Such methods have been traditionally applied to DSP computations, both for fast and power-efficient hardware implementation on FGPAs and for implementation on processors with no floating-point hardware units.

In this work, we aim at leveraging existing, HPC-oriented hardware architectures, while including in the precision tuning an adaptive selection of floating- and fixed-point arithmetic. In particular, we rely on pre-defined C types such as `short`, `int`, and `long`, excluding custom-width types typically used in high-level synthesis. We also study how auto-vectorization is impacted. This is part of a wider effort to provide the programmers with an easy way to manage extra-functional properties of programs, including precision, power, and performance [15,17,16].

Section 2 gives an overview of our solution and its technical implementation. We provide a description of hardware and software setup of the experimental evaluation in Section 3. We conclude and discuss future work in Section 4.

2. Overview of the Proposed Solution

Fixed-point representations are typically used in hardware design, where the width can be arbitrarily chosen for each value, on a per-bit basis. Converting an application from floating-point to fixed-point representations is a sophisticated process [12]. Since the widths of the integer and fractional parts are fixed and pre-computed, they must be carefully chosen to limit the loss of precision. This is accomplished for a given computation by assessing the dynamic range (minimum and maximum) of its input values, and by propagating these ranges through all intermediate values – in a data-flow manner – to the results. Based on all ranges, an appropriate fixed-point representation that minimizes the added noise is selected.

When using general-purpose processors, on the contrary, the actual bit-widths are constrained by the underlying hardware, typically the width of registers. In practice, such containers are 16-bit, 32-bit or 64-bit wide. Still, the cost of floating-point arithmetic, even in optimized hardware implementations, is high enough that it is worth investigating the benefits of fixed-point operation even in the context of high performance computing. In this work, we discuss and compare results for the `x86_64` architecture with Intel, and Intel SSE4.2 and AVX2 vector extensions enabled. We exploit vectorized fixed-point operations to speed up the execution, at the expense of some loss in precision. This loss in precision can actually be reduced if the range of values processed is small enough.

To this end, our proposed solution takes advantage of the programmers' application domain knowledge on the nature of the processed values. In particular, we rely on source code annotations written by the programmer (we consider as input a valid C/C++ source file) to know which variables should be converted to fixed-point. The input annotations for a simple example are shown in Listing 1.

Then, we perform a *value range propagation analysis* to propagate the value range information from annotated variables along data-dependence chains, thus inferring the value range for each variable involved in the computation. The output of this analysis is a fully annotated C/C++ source code having the range of values each variable can assume annotated on its declaration. To perform the value range propagation analysis, we re-purposed the GeCoS[1] framework [4,18,13,3].

From the value ranges, it is then possible to compute the number of bits needed for the representation of the integer part of the fixed-point representation. The width of the fractional part is then obtained as the difference between the architectural constraint on the total bit size and the size of the integer part.

The GeCoS source-to-source compiler [4] takes then care of replacing the annotated floating-point variables with their fixed-point equivalent. It also adds to the original source code the utility functions to perform data type conversions from floating-point to fixed-point and vice versa. The output of this stage is a new version of the kernel source code exploiting fixed-point computation instead of floating-point computation. The fixed-point code can then be compiled as a standard C/C++ source file using GCC or Clang/LLVM. We developed a C++ library that defines a template type `FixedPoint<integer_bits,fractional_bits>` with operators properly defined to make its use convenient.

[1] `http://gecos.gforge.inria.fr`

The output of the source-to-source compilation process for our running example is shown in Listing 3 for the input of Listing 2.

Listing 1 Annotation Example

```
input :

#pragma VARIABLE_TRACKING variable
for (int i=0, i<10, i++) {
  variable=i;
}

output :

variable_min = 0
variable_max = 9
```

Listing 2 Before Source-To-Source

```
#define SIZE1 10
#define SIZE2 10

#pragma VARIABLE_TRACKING m tmp foo
double m[SIZE1][SIZE2];

double tmp;

double foo;

foo = 0;
for (size_t i = 0; i < SIZE1; ++i) {
  for (size_t j = 0; j < SIZE2; ++j) {
    if (m[i][j] > m[j][i]) {
      foo = foo + m[i][j] * m[j][i];
      tmp = m[i][j];
      m[i][j] = m[j][i];
      m[j][i] = tmp;
    }
  }
}
```

Listing 3 After Source-To-Source

```
#define SIZE1 10
#define SIZE2 10

double m[SIZE1][SIZE2];
FixedPoint<3,29> m_fixp[SIZE1][SIZE2];

double tmp;
FixedPoint<3,29> tmp_fixp;

double foo;
FixedPoint<8,24> foo_fixp;

convert2DtoFixP<double,SIZE1,SIZE2>(
                              m,m_fixp);

foo_fixp = 0;
for (size_t i = 0; i < SIZE1; ++i) {
  for (size_t j = 0; j < SIZE2; ++j) {
    if (m_fixp[i][j] > m_fixp[j][i]) {
      FixedPoint<8,24> _s2s_tmp_foo_0;
      _s2s_tmp_foo_0 =
         m_fixp[i][j] * m_fixp[i][j];
      foo_fixp =  foo_fixp +
                  _s2s_tmp_foo_0;
      tmp_fixp = m_fixp[i][j];
      m_fixp[i][j] = m_fixp[j][i];
      m_fixp[j][i] = tmp_fixp;
    }
  }
}

convertScalarToFloat<double>(
                           foo_fixp,foo);
convert2DToFloat<double,SIZE1,SIZE2>(
                              m_fixp,m);
```

Issues with Vectorization The `GCC 5.4.0` and `Clang 4.0.0` compilers are not designed to efficiently vectorize kernels for the `x86_64` architecture when the fixed-point conversion is applied. In particular, sign extension and shift operations that are introduced when performing fixed-point multiplications are not handled automatically by the vectorizers.

Listing 4 shows an example of code from the `saxpy` kernel, compiled with `GCC 5.4.0` to assembly code. It is possible to see that several unpacking instructions are generated to perform the shift operation, which the compiler generates as an independent instruction.

However, it is possible to use the `pmulhw` from the MMX vector extension to replace the 16-bit shift, as shown in Listing 6. A similar solution can be applied in the case of 32-bit operands: it is possible to replace the 32-bit shift by expressing the multiplication as a sequence of `pmuldq` and `pshufd` instructions.

Since the implementation of the vectorizer is beyond the scope of this paper, we did not apply this set of optimizations for the experimental evaluation.

Listing 4 Fixed-point SAXPY kernel, compiled with baseline GCC

```
.L34:
        movdqa    (%rdx), %xmm2
        movdqa    %xmm3, %xmm1
        addq      $16, %rax
        addq      $16, %rdx
        pmullw    %xmm2, %xmm1
        movdqa    %xmm1, %xmm0
        pmulhw    %xmm3, %xmm2
        punpckhwd %xmm2, %xmm1
        punpcklwd %xmm2, %xmm0
        psrad     $16, %xmm1
        psrad     $16, %xmm0
        movdqa    %xmm0, %xmm2
        punpcklwd %xmm1, %xmm0
        punpckhwd %xmm1, %xmm2
        movdqa    %xmm0, %xmm1
        punpcklwd %xmm2, %xmm0
        punpckhwd %xmm2, %xmm1
        punpcklwd %xmm1, %xmm0
        paddw     -16(%rax), %xmm0
        movaps    %xmm0, -16(%rax)
        cmpq      %rax, %rcx
        jne       .L34
```

Listing 5 Fixed-point SAXPY kernel with unsigned multiplication, compiled with baseline GCC

```
.L34:
        movdqa  (%rdx), %xmm0
        addq    $16, %rax
        addq    $16, %rdx
        pmullw  %xmm1, %xmm0
        paddw   -16(%rax), %xmm0
        movaps  %xmm0, -16(%rax)
        cmpq    %rcx, %rax
        jne     .L34
```

Listing 6 Fixed-point SAXPY kernel after post-processing, integer multiplication restored

```
.L34:
        movdqa  (%rdx), %xmm0
        addq    $16, %rax
        addq    $16, %rdx
        pmulhw  %xmm1, %xmm0
        paddw   -16(%rax), %xmm0
        movaps  %xmm0, -16(%rax)
        cmpq    %rcx, %rax
        jne     .L34
```

3. Experimental Evaluation

Hardware Setup The platform used to run the experiments is a NUMA node with two Intel Xeon E5-2630 V3 CPUs (@2.4 – 3.2 GHz Turbo) for a total of 16 cores, with hyper threading enabled and 128 GB of DDR4 memory (@1866 MHz) on a dual channel memory configuration. The selected hardware is therefore representative of modern supercomputer nodes. The operating system is Ubuntu 16.04 with version 4.4.0 of the Linux kernel. We rely on the performance power settings with Turbo Boost activated to effectively drive all of the CPU cores up to 3.2 GHz from the base clock of 2.4 GHz. The compiler in use is GCC 5.4.0.

We collected for each kernel two performance indicators (Time-To-Solution and Energy-To-Solution), as well as the error with respect to the reference version and the instruction mix. Performance measurements are averaged over 100 executions for each same kernel. Time-To-Solution is measured using the `clock()` API from the standard C++ `sys/time.h`. Energy-To-Solution is measured using the Intel RAPL (*Running Average Power Limit*), a set of hardware counters providing energy and power information. These counters are updated automatically by the hardware. Linux provides an interface to read the counter values. Intel defines a hierarchy of power *domains*, where the top-level domain is the *package*. In our experiment we consider Energy-To-Solution the $\sum_i^{all} Energy_{package,i}$. Note that RAPL does not map energy to processes therefore, $Energy_{package,i}$ represents the energy consumed by the package i as a whole. We used a controlled and unloaded machine for our experiments to guarantee that the energy consumption is due to the benchmarks we run.

We measured the error on the output of each kernel with respect to the highest

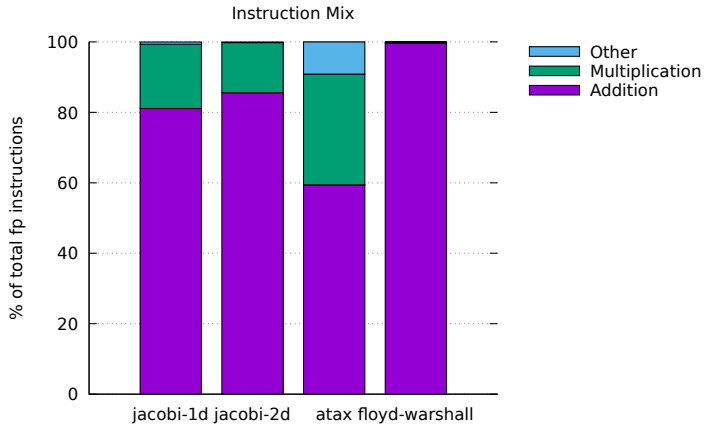

Figure 1. Instruction mix for the selected PolyBench benchmarks.

data precision, which is floating-point quadruple-precision (128 bits). To measure the instruction mix we rely on Intel Software Development Emulator (SDE), a Pin tool [10] that produces instruction traces and classifies them into categories.

Benchmarks PolyBench[2] is a collection of benchmarks consisting of regular kernels written in C language. We evaluated our approach over a subset of the linear algebra family of PolyBench [22]. The subset is chosen based on the ability of the compiler to vectorize the code, which directly impacts the speedups that can be achieved with fixed-point arithmetic. The benchmarks that can be fully vectorized are: `floyd-warshall`, `atax`, `jacobi-1d`, `jacobi-2d`.

`floyd-warshall` Shortest path in a weighted graph
`atax` Matrix Transpose and Vector Multiplication
`jacobi-1d` 1-D Jacobi stencil computation
`jacobi-2d` 2-D Jacobi stencil computation

PolyBench defines five presets of input data sizes for each benchmark: mini, small, medium, large, extralarge. In these experiments, measurements in terms of error, energy, and time are obtained by exploiting the medium data-set size.

The benchmark is characterized by the instruction mix reported in Figure 1. The four benchmarks show a variety of floating-point instruction mixes, ranging from `floyd-warshall`, which has almost only additions, to `atax` which has a more balanced mix of additions and multiplications. The `jacobi-1d` and `jacobi-2d` kernels fall in the middle.

Analysis of Results With the above described setup, we collected for each benchmark measures for the following metrics: time-to-solution, energy-to-solution, and error. We report the time-to-solution and energy-to-solution normalized with respect to the execution of the quadruple-precision floating-point version of each benchmark, which provides the greatest accuracy, but also the slowest time-to-

[2] `https://sourceforge.net/projects/polybench/`

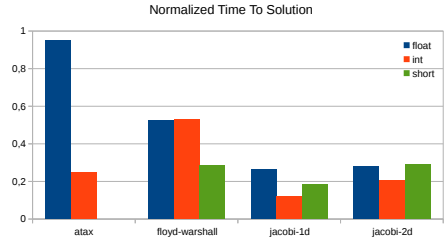

Figure 2. Normalized time-to-solution for each benchmark and data type, normalized to the same benchmark with double precision floating-point arithmetic.

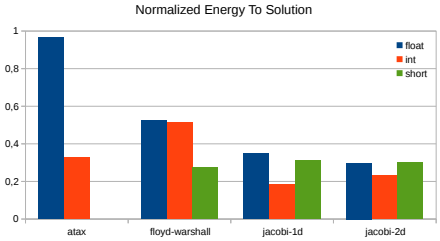

Figure 3. Normalized energy-to-solution for each benchmark and data type, normalized to the same benchmark with double precision floating-point arithmetic.

solution and the largest energy-to-solution. For what concerns the error, we report the *relative solution error* (or relative forward error), defined as follows:

$$\eta = \frac{\|A_{approx} - A\|_F}{\|A\|_F}$$

where $\|A\|_F$ is the Frobenius matrix norm:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{i,j}^2}$$

As it can be seen from Figure 2, different benchmarks achieve maximum performance with different approximation solutions.

In particular, for `atax`, we obtain the best performance/error trade-off using 32-bit fixed-point arithmetic, as with 16-bit fixed-point arithmetic it is impossible to find a good compromise between the need to preserve precision for small numbers and the need to provide a sufficiently large number of integer part bits to avoid overflows. On the other hand, 32-bit fixed-point arithmetic provide a major speed-up at only a limited cost in precision.

For `floyd-warshall`, 16-bit fixed-point arithmetic are sufficient for a reasonably good precision, and provide a good boost in performance. The algorithm does not include multiplications, so impacts on both metrics are more limited than in other cases.

For `jacobi-1d` and `jacobi-2d`, 16-bit fixed-point arithmetic is inefficient due to limited opportunities to vectorize. Indeed, the 16-bit fixed-point becomes slower than 32-bit fixed-point arithmetics, as similar operations are used, but more conversion overhead is incurred. 32-bit fixed-point arithmetic, on the other hand, provide a reasonable boost to performance while incurring in reasonable error.

It is important to note that the approximation also needs to be taken into account, as with fixed-point, and sometimes also with low-precision floating-point, there is typically a price to pay. The graphs in Figure 4 plot the relative solution error against the time-to-solution for each benchmark to highlight Pareto-optimal solutions. The graph shows that, while for `jacobi-1d` and `jacobi-2d` there is a single Pareto-optimal solution (32-bit fixed-point), for `atax` and `floyd-warshall`

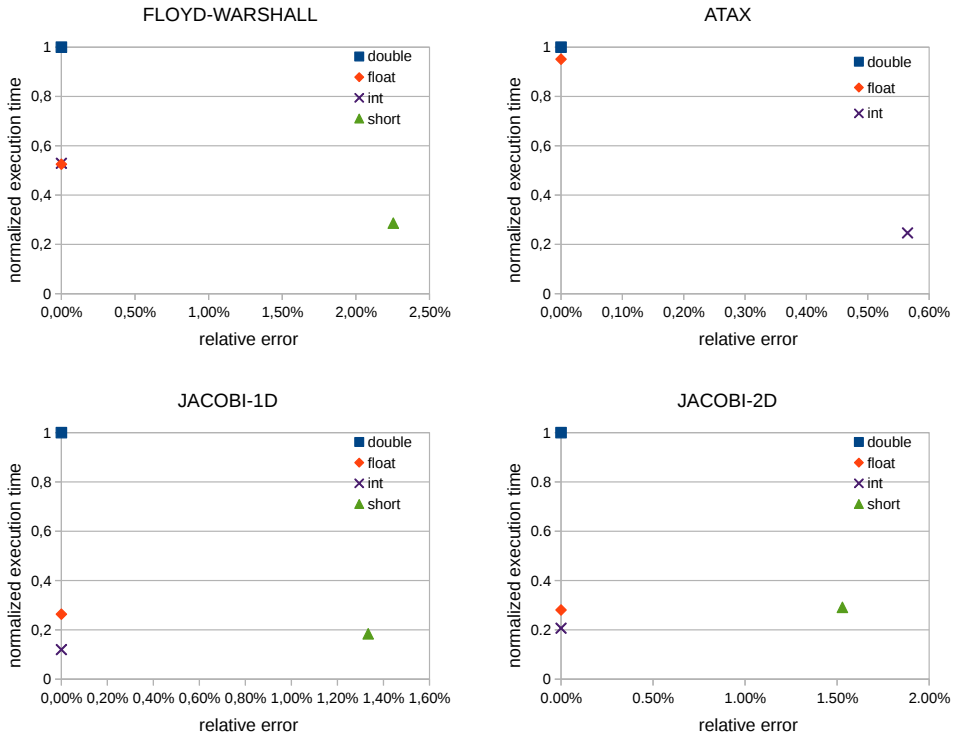

Figure 4. Relative solution error vs time-to-solution

there are two (single-precision floating-point and either 32-bit or 16-bit fixed-point). Therefore, a solution can be selected based on the target relative solution error.

Figure 3 reports the normalized energy-to-solution for each benchmark and data type, normalized to the double precision version. In general, energy to solution is strongly related to time to solution. In most cases, the energy saving is more limited than the performance improvement, though. The only exception is `floyd-warshall`, which differs from the other benchmarks for a distinctly lower use of multiplications. This difference in the instruction mix is reflected in the energy to solution, which is also lower than in other benchmarks when compared to time to solution.

4. Conclusion

In this paper, we assess the interest of fixed-point and reduced-size floating-point arithmetic operations as a tool to provide a trade-off between energy, performance, and precision in High Performance Computing. We employ a subset of the PolyBench benchmark and find out, for each benchmark and data set size, the best trade-off in terms of energy-to-solution and time-to-solution vs. error.

We automate the floating-to-fixed point conversion by retargeting an existing source-to-source compiler, designed for use with hardware implementations, to

produce code suitable for execution in HPC environments. We show how to overcome the current limitations of compilers with respect to vectorization support to improve the time-to-solution and energy-to-solution metrics of smaller size data types, both for fixed- and floating-point data types.

Future works include porting our work to the Intel MIC architecture where the availability of multiplication operations on 32-bit fixed-point data with automated selection of the high bits of the result may further improve the performance of 32-bit fixed-point versions of benchmarks relying heavily on vectorizable multiplications, as well as replacing the current hack with a specialized vectorization pass in the compiler.

Acknowledgments

This work is partially supported by the European Union's Horizon 2020 research and innovation programme, under grant agreement No 671623, FET-HPC ANTAREX. Authors would like to thank Thomas Lefeuvre for his help with the GeCoS framework.

References

[1] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.

[2] Joe Curley. HPC and Big Data. *Innovation*, 12(3), 2014.

[3] Gaël Deest, Tomofumi Yuki, Olivier Sentieys, and Steven Derrien. Toward Scalable Source Level Accuracy Analysis for Floating-point to Fixed-point Conversion. In *International Conference on Computer-Aided Design (ICCAD)*, pages 726–733, 2014.

[4] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys. GeCoS: A framework for prototyping custom hardware design flows. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 100–105, 2013.

[5] Dominik Göddeke, Robert Strzodka, and Stefan Turek. Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):221–256, 2007.

[6] Xuejun Hao and Amitabh Varshney. Variable-precision rendering. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, pages 149–158, New York, NY, USA, 2001. ACM.

[7] Michael O Lam and Jeffrey K Hollingsworth. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications*, pages 1–15, 2016.

[8] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 369–378, New York, NY, USA, 2013. ACM.

[9] Michael D Linderman, Matthew Ho, David L Dill, Teresa H Meng, and Garry P Nolan. Towards program optimization through automated analysis of numerical precision. In *International Symposium on Code Generation and Optimization*. ACM, 2010.

[10] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200. ACM, 2005.

[11] Daniel Menard, Romuald Rocher, and Olivier Sentieys. Analytical fixed-point accuracy evaluation in linear time-invariant systems. *Transactions on Circuits and Systems I: Regular Papers*, 55(10):3197–3208, 2008.

[12] Daniel Menard and Olivier Sentieys. Automatic evaluation of the accuracy of fixed-point algorithms. In *Conference on Design, Automation and Test in Europe*, page 529. IEEE Computer Society, 2002.

[13] Karthick Nagaraj Parashar, Daniel Menard, and Olivier Sentieys. Accelerated Performance Evaluation of Fixed-Point Systems With Un-Smooth Operations. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4):599–612, April 2014.

[14] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1074–1085, New York, NY, USA, 2016. ACM.

[15] C. Silvano, G. Agosta, A. Bartolini, A. Beccari, L. Benini, J. M. P. Cardoso, C. Cavazzoni, R. Cmar, J. Martinovic, G. Palermo, M. Palkovic, E. Rohou, N. Sanna, and K. Slaninova. Antarex – autotuning and adaptivity approach for energy efficient exascale hpc systems. In *2015 IEEE 18th International Conference on Computational Science and Engineering*, pages 343–346, Oct 2015.

[16] Cristina Silvano, Giovanni Agosta, Andrea Bartolini, Andrea R Beccari, Luca Benini, Radim Cmar, Carlo Cavazzoni, Jan Martinovi, Gianluca Palermo, Martin Palkovi, et al. Autotuning and adaptivity approach for energy efficient exascale HPC systems: the antarex approach. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 708–713. IEEE, 2016.

[17] Cristina Silvano, Giovanni Agosta, Stefano Cherubin, Davide Gadioli, Gianluca Palermo, Andrea Bartolini, Luca Benini, Jan Martinovič, Martin Palkovič, Kateřina Slaninová, João Bispo, João M. P. Cardoso, Rui Abreu, Pedro Pinto, Carlo Cavazzoni, Nico Sanna, Andrea R. Beccari, Radim Cmar, and Erven Rohou. The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems. In *International Conference on Computing Frontiers*, CF '16, pages 288–293. ACM, 2016.

[18] N. Simon, D. Menard, and O. Sentieys. ID.Fix-infrastructure for the design of fixed-point systems. In *University Booth of the Conference on Design, Automation and Test in Europe (DATE)*, volume 38, 2011.

[19] Swagath Venkataramani, Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *International Symposium on Microarchitecture*, pages 1–12. ACM, 2013.

[20] Rangharajan Venkatesan, Amit Agarwal, Kaushik Roy, and Anand Raghunathan. Macaco: Modeling and analysis of circuits for approximate computing. In *International Conference on Computer-Aided Design*, pages 667–673. IEEE Press, 2011.

[21] Thomas Yeh, Petros Faloutsos, Milos Ercegovac, Sanjay Patel, and Glenn Reinman. The art of deception: Adaptive precision reduction for area efficient physics acceleration. In *International Symposium on Microarchitecture (MICRO)*, pages 394–406. IEEE, 2007.

[22] Tomofumi Yuki. Understanding PolyBench/C 3.2 kernels. In *International workshop on Polyhedral Compilation Techniques (IMPACT)*, 2014.