Performance Analysis of Decimal Floating-Point Libraries and Its Impact on Decimal Hardware and Software Solutions

Michael J. Anderson[1], Charles Tsen[1], Liang-Kai Wang[2], Katherine Compton[1], and Michael J. Schulte[1]
[1]*Department of Electrical and Computer Engineering, University of Wisconsin - Madison*
[2]*Advanced Micro Devices, Inc.*
{*mjanderson1, stsen*}*@wisc.edu, liangkai.wang@gmail.com,* {*kati, schulte*}*@engr.wisc.edu*

***Abstract*— The IEEE Standards Committee recently approved the IEEE 754-2008 Standard for Floating-point Arithmetic, which includes specifications for decimal floating-point (DFP) arithmetic. A growing number of DFP solutions have emerged, and developers now have many DFP design choices including arbitrary or fixed precision, binary or decimal significand encodings, 64-bit or 128-bit DFP operands, and software or hardware implementations.**

There is a need for accurate analysis of these solutions on representative DFP benchmarks. In this paper, we expand previous DFP benchmark and performance analysis research. We employ a DFP benchmark suite that currently supports several DFP solutions and is easily extendable. We also present performance analysis that (1) provides execution profiles for various DFP encodings and types, (2) gives the average number cycles for common DFP operations and the total number of each DFP operation in each benchmark, and (3) highlights the tradeoffs between using 64-bit and 128-bit DFP operands for both binary and decimal significand encodings. This analysis can help guide the design of future DFP hardware and software solutions.

I. Introduction

People predominantly perform math using decimal numbers. However, due to the speed and efficiency advantages of binary hardware, most computer systems support binary arithmetic rather than decimal arithmetic. Converting decimal numbers to binary and performing operations on them using binary hardware can lead to inaccuracies. For example, the binary floating-point (BFP) single-precision number closest to the decimal fraction 0.1 is actually equal to 0.100000001490116119384765625. In certain applications, small errors like this can compound and lead to incorrect and unacceptable results [1]. One study by IBM estimates that binary representation and rounding errors of decimal numbers in a telephone billing application can accumulate to a yearly billing discrepancy of over $5 million dollars [2].

Decimal floating-point (DFP) number systems represent floating-point numbers using a radix (i.e., exponent base) of ten rather than two. DFP therefore exactly represents decimal numbers, provided those numbers fit in the format's precision. Two prevailing DFP number encodings have emerged. One, originally proposed by Intel, encodes the significand of the DFP number as an unsigned binary integer, and is commonly called the binary integer decimal (BID) encoding. The other, originally proposed by IBM, encodes the significand as a compressed binary coded decimal (BCD) integer, and is commonly called densely packed decimal (DPD). The IEEE 754-2008 Standard for Floating-point Arithmetic [3] specifies these DFP encodings and their operations, rounding, and exception handling. The standard allows DFP numbers to be encoded either in BID or DPD, and specifies arithmetic operations on 64-bit and 128-bit DFP numbers, known as *decimal64* and *decimal128*, respectively.

The availability and accessibility of DFP has increased recently, making it likely that DFP will move into more general-purpose applications and computer systems. With the addition of built-in DFP types to versions 4.2 and newer of the GCC compiler, programmers have increased access to decimal types. Furthermore, IBM has added dedicated DFP hardware to the Power6, z9, and z10 servers [4], [5], [6] and has developed significant compiler and software interface support for DFP arithmetic on these systems [7]. We expect dedicated DFP hardware to be provided by more processors in the future.

As DFP becomes more prevalent, there is an increasing need for accurate evaluation and understanding of both hardware and software DFP solutions. A benchmark suite that aids this evaluation would help to eliminate the gap in understanding of the tradeoffs between different DFP solutions and give insight into which operations could benefit most from hardware acceleration. Since hardware implementations are not widely available for all DFP formats, current analysis must use available software libraries. Several questions related to IEEE 754-2008 formats and encodings require further investigation, including:

- Which DFP operations are most frequent and require a significant fraction of the overall execution time in DFP applications?
- What are the potential performance benefits of implementing certain DFP operations in hardware?
- What are the differences in performance between decimal64 and decimal128 for the DPD and BID encodings?
- For specific operations, do DPD or BID software libraries offer better performance? How many cycles does it take to implement specific operations in each of these libraries?
- As new solutions emerge, what level of performance do they provide to DFP applications?

978-1-4244-5028-2/09/$25.00 ©2009 IEEE

This paper describes extensive modifications made to previous DFP benchmarks [8] to help answer these questions and provide more modular support for various emerging DFP solutions. In particular, the benchmarks have been updated to support the IBM decFloats modules [9], the Intel DFP Math Library [10], and built-in GCC DFP types [11]. Although the benchmarks previously supported only the IBM decNumber arbitrary-precision library, they now have the added ability to use fixed-precision 64-bit and 128-bit DFP types with either BID or DPD encodings. The addition of built-in DFP types in GCC is especially important, because it allows the benchmarks to be compiled and run on machines with native DFP instruction set and hardware support, such as those provided in recent IBM Power and System z processors [4], [5], [6].

The performance of DFP operations can vary significantly depending on the input values. Thus, to accurately analyze DFP solutions, it is necessary to do so with representative workloads. The enhanced benchmarks were run with representative input sets for each DFP solution for both 64-bit and 128-bit formats and performance statistics of several metrics were recorded. These statistics include the average number of cycles for each DFP operation, number of each DFP operation in each benchmark, performance differences between decimal64 and decimal128 types, percentage of operations that require rounding, total benchmark execution time, and execution time profile.

Contributions of this paper include: (1) the first benchmark suite to implement a wide range of DFP solutions, (2) an analysis of the tradeoffs between various DFP solutions and formats on representative benchmarks, and (3) an improved understanding of potential DFP operations for hardware acceleration. A particularly important unanswered question is, what are the implications of moving from decimal64 to decimal128? This paper helps to answer this and other questions through quantitative analysis.

In the remainder of the paper, Section II gives an overview of the benchmarks and DFP solutions employed. Section III describes the methodology of our DFP benchmark framework, which allows the benchmarks to be used with multiple DFP solutions. Section IV explains the implementation of frequently executed DFP operations. Section V presents and analyzes DFP performance results of the benchmarks for various DFP solutions. Section VI gives our conclusions.

II. Overview of Benchmarks and DFP Solutions

The expanded DFP benchmark suite now supports the seven DFP types shown in Table I. The IBM decNumber library features both fixed-precision and arbitrary-precision types for the DPD encoding. The Intel DFP Math library features fixed types for the BID encoding. The abbreviations in bold are used throughout the paper in place of full names for the DFP types.

The benchmarks also support built-in GCC types and operations for either DPD or BID encodings, depending on the compiler's configuration. The GCC compiler uses either the IBM decNumber library for DPD encodings or the Intel DFP Math library for BID encodings. Therefore, the built-in DFP types are included in the presented analysis

TABLE I
Supported DFP Solutions

Library	Supported types
IBM decNumber	*decNumber* - Arbitrary-precision DPD (**DN64/DN128**) *decDouble* - Fixed-precision decimal64 DPD (**DPD64**) *decQuad* - Fixed-precision decimal128 DPD (**DPD128**)
Intel DFP Math	Fixed-precision decimal64 BID (**BID64**) Fixed-precision decimal128 BID (**BID128**)
Built-in GCC	Fixed-precision decimal64 DPD Fixed-precision decimal64 BID

Five benchmarks are implemented using the decimal API framework. The five benchmarks are described below, except for the telephone billing application, Telco, which is described elsewhere [2]. More detailed information about the other four benchmarks is also available [8] [12].

A. Banking System Benchmark (Banking)

Banking is an important area that is represented by the benchmarks, and requires DFP computations and decimal rounding. Banking systems deal with a variety of tasks, communicate with database systems, and perform daily maintenance routines to keep account information up to date. To emulate daily server activity in an bank, we enhanced an existing banking system benchmark [8], which includes checking, certificate of deposit, credit card, and mortgage accounts.

B. Euro Conversion Benchmark (Euro)

When the euro was introduced, the European Central Bank (ECB) specified several regulations and monetary policies for euro currency conversion [13]. Currency conversion between the euro and legacy currencies, as well as between two legacy currencies, were specified to require decimal arithmetic [13]. The currency conversion benchmark is based on the requirements from the ECB, which include:

1) having a unilateral conversion rate from the euro to each legacy currency unit,
2) not using an inverse conversion rate,
3) having six significant digits, including trailing zeros, in the conversion rates from the euro to other currencies,
4) not rounding or truncating conversion rates, and
5) having an intermediate result in euros when converting between two legacy currency units, in which the intermediate result may be rounded to no less than three decimal digits.

The Euro benchmark performs a series of conversions between currencies using historical exchange rates. To convert between two non-euro currencies, the value is first converted to euros using DFP multiplication. Then it is converted from euros to the second currency using DFP division.

C. Risk Management Benchmark (Risk)

Risk management is a corporate finance field that seeks to measure and manage risk to earn greater returns over a time period. It usually involves several statistical calculations such

as variance and covariance computations. The risk management benchmark is based on existing spreadsheets [14], but uses DFP libraries and types. It computes risk parameters in the capital asset pricing model (CAPM) and predicts prices for a company against a stock market valuation. It calculates risk measures, variance statistics, and expected returns using historical data to determine risk parameters for publicly traded companies.

D. Tax Preparation Benchmark (Tax)

The tax preparation benchmark calculates federal taxes for an individual for the 2006 tax year. This benchmark was modified from the Open Tax Solver from Sourceforge [15]. The code follows the instructions provided in IRS documents 1040, Schedules A, B, C, and D. The code was modified to incorporate DFP arithmetic. The inputs to the program are given by W2 and 1099 forms handed out by employers, universities, banks, etc. The benchmark inputs are generated using directed pseudo-random numbers with constraints to model a realistic setting, which presents the distribution of adjusted gross income on a sample size of about 132 million tax returns for the 2004 tax year [16]. To support the decimal libraries and types, various changes are made to the code. Appropriate library functions are called to perform DFP arithmetic operations and comparisons. Since the inputs are already in the DFP format, reading and storing the inputs does not result in any loss of precision. Furthermore, all DFP calculations are performed with decimal rounding, making the final results more accurate. The tax preparation benchmark reads users' tax form data from a file. After the calculations are made, results are stored in a different file. Hence, the amount of file I/O is directly proportional to the number of tax returns.

III. Methodology

In this section, we present the methodology and usage model of the decimal API framework. This section describes how the benchmarks are enhanced to support multiple DFP solutions. A DFP solution contains two main components that are abstracted in our benchmark API framework:

- **Decimal Types** - The variables containing decimal data. The format and size of this variable can be different for each implementation.
- **Operators** - Operations such as addition, subtraction, comparison and rounding. For most DFP solutions, these operations are explicitly called as functions with DFP parameters.[1]

A. Decimal Types

Our benchmark framework uses the C programming language, so the #define directive configures these abstractions for each DFP solution. Benchmark applications use a generic type called *dec_t* that is parameterized to use the chosen decimal format. In the example below, directives in a header file choose between the IBM decFloats fixed-precision types and the Intel DFP Math Library fixed-precision types. Either DECNUMBER_FIXED or BID must be defined as a compiler directive for the following code.

```
#if defined(DECNUMBER_FIXED)
 #define dec_t decDouble // IBM decFloats
#elif defined(BID)
 #define dec_t BID_UINT64 //Intel BID
#endif
```

The #define directives are pre-processed by the compiler, so there are no performance artifacts introduced by generalizing the benchmark code in this way.

B. Operators

DFP operations such as *Add*, *Subtract*, and *Compare* are typically performed using explicit function calls. For this reason, generic functions are defined for each operation. These generic functions are configured differently for each DFP solution, as in the code below.

```
#if defined(DECNUMBER_FIXED)
 #define ADD(Z, X, Y, C) decDoubleAdd(Z, X, Y, C)
#elif defined(BID)
 #define ADD(Z, X, Y, C)  bid_add(Z, X, Y, C)
#endif
```

In the above example, ADD(Z, X, Y, C) defines the operation Z = X + Y. Each of these function calls requires a context (C), which stores rounding modes and status flags for each DFP operation. This is contained in a special type, *context_t*, that is also defined for each DFP solution.

IV. DFP Operations

This section discusses some of the common DFP operations, and how they are implemented for the DFP solutions employed in the benchmarks.

A. Multiply

For the general case, performing a DFP multiply involves three steps. First, the significands are multiplied. Next, the exponents are added. Finally, if the result exceeds the available precision of the target type, the value is rounded. For DPD-encoded operands, the major challenge is the significand multiplication, which is performed as a series of multiplications and summations of binary coded decimal (BCD) digits. In contrast, the BID-encoded types take advantage of highly optimized binary multipliers for the significand multiplication. However, BID rounding is much more expensive because it uses a wide multiplication of the intermediate result by an integer power of $\frac{1}{10}$. These constants are pre-computed and stored in a table that is indexed by the number of digits to round off [17]. We expect the average number of cycles required for a BID multiply to be very sensitive to the amount of rounding that is performed in the benchmark.

B. Add/Subtract

In the addition operation, the two input operands must be aligned to have the same exponent by shifting one or both significands. Unlike BFP, DFP types are non-normalized, meaning one or more of the significant digits can be zero, and

[1]The GCC compiler supports operator symbols such as +, -, >, and < for DFP arithmetic using its built-in types.

a single decimal number can have multiple possible DFP representations. In this situation, IEEE 758-2008 specifies which representation operations should return. In DPD, alignment is relatively simple. The shift operation is performed on BCD numbers, for which a 1-digit shift corresponds to a 4-bit shift. In BID, however, a decimal shift requires a multiply by a pre-computed power of 10.

After alignment, the significands are added. If overflow occurs as a result of the addition, the exponent is incremented. The result must then be rounded if it does not fit in the precision of the destination operand. In both BID and DPD, the significand addition is relatively straightforward. In BID, when rounding is needed, it is the most costly part of the operation.

C. Quantize

The quantize operation modifies one DFP operand to have a specific exponent equal to that of the other DFP operand. The process is fundamentally the same as either alignment or rounding, depending on the relative difference of the exponents. However, the required left or right decimal shifts are less costly than rounding because of a narrower operand width. For example, in decimal128 multiplication, the intermediate result to be rounded may be up to 226 bits. A decimal128 quantize only needs to consider a 113-bit significand.

D. Copy/Zero

The copy and zero operations correspond to loading one DFP variable with the value of another, or loading a DFP variable with the DFP representation of zero. Although they occur frequently, these operations are relatively simple to implement in either hardware or software.

V. Performance Analysis

In this section, we analyze the performance of the benchmarks from Section II on representative input sets. The goals of this section are to characterize the performance of various DFP encodings and types on individual operations and overall runtime. This provides insight into where hardware acceleration might be best applied.

Previous work has analyzed the execution profiles of a set of benchmarks for decimal64 types in the arbitrary-precision decNumber library [8]. Since then, software libraries have emerged that support fixed-precision decimal64 and decimal128 types and are easily accessible in new versions of GCC. Developers are moving to fixed-precision types for performance reasons, with large amounts of code being developed for the decimal128 type. This paper presents a broader analysis that includes fixed-precision types and 128-bit sizes. These new profiles provide a more complete representation of which operations should be considered for hardware implementations of DFP.

The benchmarks, described in Section II, are written in C and compiled using GCC version 4.1.2 with the -O3 flag. This version of GCC does not feature built-in DFP datatypes, so the IBM decNumber library version 3.61 and the Intel Decimal Floating-Point Math Library version 1.0 update 1 were used for DPD and BID formats respectively. The benchmarks were executed on an Intel(R) Core(TM)2 Duo E8400 64-bit 3GHz processor running Debian Linux. Runtimes of individual DFP operations were calculated using the x86 Time Stamp Counter register [18]. Profiling was performed using gprof.

TABLE II
Sample Input

Banking	100,000 Accounts, 100,000-150,000 events/day, 30 days
Euro	1 million conversions to/from the euro
Risk Management	Risk factor calculations for 30 companies
Tax Preparation	100,000 2006 Federal tax returns
Telco	Billing for 1 million calls

TABLE III
Number of DFP Operations Per Benchmark (In Thousands)

	Banking	Euro	Risk	Tax	Telco
Add	34,068	1,780	3,673	17.277	20,000
Compare	16,770	1,832		3,420	
Copy	25,051	2,022	5,374	17,716	3,620
Divide	21,213	2,772	64	295	
Max	582				
Min				399	
Minus	100				
Multiply	26,970	2,842	3,668	1,611	12,500
Power	2,954		2		
Quantize	23,694	3,512		171	12,500
Sqrt			1		
Subtract	24,472	1,832	1,951	19,513	
ToIntegral	24,864	3,612			
Zero	12,170	1,000	375	320,763	7,499

In our comparisons, the benchmarks are run on identical input values, though the formats for these input values varied. The input values are based on historical data [8]. A summary of our input data is given in Table II.

First, we discuss some of the benchmark characteristics, and present the number of times each DFP operation is executed in each benchmark. Next we analyze the performance of individual DFP operations for various formats. Then we discuss the impact of the different DFP types on the overall performance of the benchmarks. This includes a comparison of total runtime between DFP types and execution profiles for DPD and BID fixed-precision types. Because rounding constitutes a large percentage of total runtime for DFP applications, we investigate it further.

A. Operation Frequency

Table III displays the number of times (in thousands) each DFP operation is executed for each of the benchmarks in our experiments. The Telco and Risk benchmarks operate mostly on data with the same exponent, which means the alignment portion of the Add operation is not needed. The performance of these benchmarks is significantly affected by how each DFP solution handles the special case of pre-aligned inputs. For example, the decNumber library has a fast path for this situation, so it performs well on these two benchmarks. The Euro benchmark has a large proportion of DFP divides compared to other DFP operations, so the performance of this benchmark is strongly affected by the DFP solution's implementation of division.

TABLE IV
AVERAGE NUMBER OF CYCLES PER OPERATION

	DN64	DN128	DPD64	DPD128	BID64	BID128
Add	157	161	154	233	109	213
Copy	23	30	9	10	9	12
Divide	1,039	1,552	627	940	370	1,420
Max	372	437	136	190	182	547
Min	346	348	222	274	204	368
Minus	129	125	40	49	5	14
Multiply	239	253	296	453	117	544
Quantize	220	220	138	211	147	391
Sqrt	21,331	33,998	25,599	38,559	713	7,194
Subtract	227	237	289	580	126	313
ToIntegral	385	385	178	269	148	519
Zero	6	6	5	6	1	1

B. Cycles Per Operation

Table IV shows the average number of cycles per operation for the various DFP solutions across the five DFP benchmarks. For decimal64, the BID encoding performs best for almost all operations. It can be concluded that, for BID64, the performance gain from the currently available high-speed 64-bit binary hardware outweighs the performance loss from expensive rounding. The exception is *Quantize*, which almost always requires rounding or shifting (unless the operands are already aligned), and thus performs best for fixed-precision DPD types.

The performance slowdown of fixed-precision types when moving from decimal64 to decimal128 is shown for each operation in Figure 1. BID has a larger slowdown than DPD for every operation. The test system did not have a 128-bit integer datapath, so BID128 was not able to take advantage of available binary hardware as much as BID64. For operations such as *Multiply* and *Divide* in BID128, rounding quickly becomes very costly. As a result, many operations in BID64 take fewer cycles than in DPD64, but the same operations in BID128 take more cycles than in DPD128.

For most operations with arbitrary-precision DPD types, moving from decimal64 to decimal128 increases the cycle count by a very small amount. This is because there is significant overhead involved with supporting arbitrary-precision that is present in both precisions. Also, the implementations for both sizes are very similar and extra computation is done only when the precision is used. The arbitrary-precision DPD types performed surprisingly well for *Add* and *Multiply* compared to the equivalent operations in the fixed-precision DPD types. This is due to several fast paths that are built into the decNumber arbitrary-precision functions for common cases.

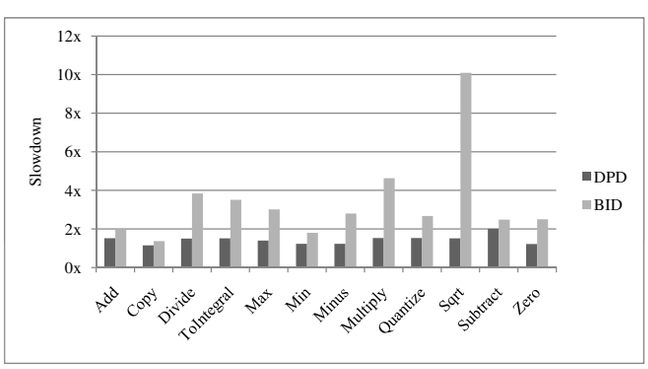

Fig. 1. Slowdown from decimal64 to decimal128 by operation.

Subtract takes significantly more cycles than *Add* for most DFP solutions. Besides requiring an extra negation, we also observed a greater proportion of unaligned operands in subtractions compared to additions. The required alignments contributed to the cycle counts.

Finally, *Zero* is similar in both DPD and BID fixed types. The discrepancy in number of cycles occurs because the BID *Zero* implementation was inline with the code, whereas the DPD operation employed a function call.

C. Benchmark Profiles

Tables V and VI show execution profiles for DPD64 and DPD128; Tables VII and VIII show execution profiles for BID64 and BID128. In the BID profiles, *Power* and *Sqrt* are left out because they are not provided in the library, and were instead approximated using the corresponding BFP functions. Also, the BID *Zero* operation could not be measured using gprof because the operation is inline with the benchmark code and does not require a function call. Its percent of the overall execution time is approximated using the average number of cycles per *Zero* call, the number of times *Zero* is called, and the total number of cycles elapsed in the benchmark.

The profiles suggest which operations would have the most impact on performance if implemented in hardware. Because *Add*, *Subtract*, and *Multiply* are so common, these are good candidates for hardware acceleration. *Quantize* and *RoundToIntegral* could have significant impact on performance if put into hardware, particularly for the BID significand encoding. Though *Divide* is less common, its long execution time contributes significantly to runtime in many applications. *Divide* could therefore benefit from acceleration, especially for the decimal128 format.

Multiply and *Divide* take a smaller portion of total execution time in BID64 than DPD64. This is because BID uses existing high-speed binary integer multipliers to perform these operations. However, in moving to BID128, *Multiply*, *Divide*, and *Quantize* become a significantly greater percentage of total execution time because of the rounding performed in these operations. As a result, *Add* and *Subtract* take a much smaller proportion of total execution time for BID128. In contrast to BID, DPD128 spends more time in *Add* and *Subtract* than DPD64, and about the same percentage of time in *Multiply* and *Divide* as DPD64.

For decimal64 types, almost all of the benchmarks spend more than 70% of their total execution time performing DFP operations. This increases to 80% for decimal128 types, indicating a large potential performance boost due to DFP hardware, especially for decimal128 types.

D. Total Runtime

Table IX shows the total runtime of each benchmark normalized to the runtime of BID64 for each solution. Figure

TABLE V
EXECUTION PROFILE FOR DPD64

	Banking	Euro	Risk	Tax	Telco
Add	14.6%	9.6%	15.5%	24.2%	20.5%
Compare	3.0%	5.2%		3.8%	
Copy	0.4%	0.3%		1.5%	0.8%
Divide	18.2%	31.3%	0.6%	2.0%	
ToIntegral	8.0%	16.1%	0.0%	0.7%	
Max	0.1%		0%		
Min				0.5%	
Minus					
Multiply	16.0%	19.9%	26.8%	6.0%	44.0%
Power*	6.4%		4.2%		
Quantize			15.0%		24.4%
Sqrt			0.3%		
Subtract	11.8%	11.0%	35.2%	3.3%	
To/From Num	0.2%		0.3%	0.0%	
ToString					5.3%
Zero	0.1%	0.0%	0.0%	14.7%	0.3%
Total Decimal	78.8%	93.4%	97.9%	56.7%	95.3%

TABLE VI
EXECUTION PROFILE FOR DPD128

	Banking	Euro	Risk	Tax	Telco
Add	17.2%	12.3%	19.1%	23.4%	21.2%
Compare	3.1%	3.7%		4.6%	
Copy	0.3%	0.3%	0.0%	0.9%	0.6%
Divide	18.1%	30.8%	0.7%	1.5%	
ToIntegral	7.6%	14.4%		0.5%	
Max	0.1%		0.0%		
Min				0.6%	
Minus	0.0%		0.0%		
Multiply	18.1%	19.7%	19.9%	6.5%	43.7%
Power*	7.5%		6.4%		
Quantize			11.4%		24.2%
Sqrt			0.4%		
Subtract	13.4%	13.5%	40.0%	3.3%	
To/From Num	0.2%		0.0%	0.0%	
ToString	0.1%				4.1%
Zero	0.1%	0.0%	0.0%	25.8%	0.0%
Total Decimal	84.4%	94.7%	97.9%	67.1%	95.6%

2 shows the slowdown going from decimal64 to decimal128 for each DFP library. The BID solution used here has no available DFP *Power* operation, so the corresponding BFP operation is used as an approximation. This impacts performance slightly for the Banking and Risk benchmarks, since *Power* is much faster in BFP than in DFP.

In general, the fixed-precision types perform better than arbitrary-precision types. In Risk and Telco, arbitrary-precision types perform very well because those benchmarks contain mostly aligned additions and the decNumber arbitrary-precision library has a fast path for this case. The BID64 library is the fastest for most benchmarks, due to its ability to best take advantage of the existing binary hardware. However, BID pays a large penalty for the increase to decimal128 precision. We attribute this to rounding and shifting that requires large integer multiplies, and the lack of correspondingly wide (greater than 64-bit) binary hardware on the tested processor model.

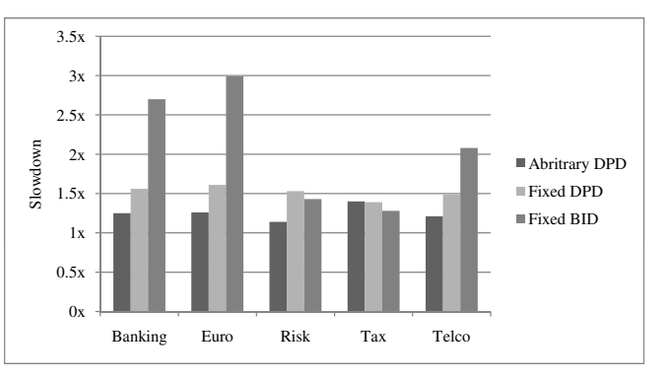

Fig. 2. Runtime Slowdown from decimal64 to decimal128.

TABLE VII
EXECUTION PROFILE FOR BID64

	Banking	Euro	Risk	Tax	Telco
Add	13.3%	8.1%	17.2%	19.8%	34.8%
Compare	3.2%	8.5%		5.4%	0.2%
Copy	0.6%	0.5%	0.0%	1.4%	
Divide	14.2%	26.8%	1.6%	1.7%	
ToIntegral	10.7%	24.1%		1.1%	
Max	0.3%				
Min				1.2%	
Minus	0.0%			0.0%	
Multiply	14.5%	15.1%	12.5%	1.8%	16.7%
Quantize			26.1%		31.2%
Sqrt			0.0%		
Subtract	11.0%	9.4%	38.7%	2.8%	
ToBinary	1.2%		0.0%	0.0%	
ToString					11.3%
Zero	0.1%	0.0%	0.0%	2.6%	0.3%
Total Decimal	69.1%	92.5%	96.6%	37.8%	94.5%

TABLE VIII
EXECUTION PROFILE FOR BID128

	Banking	Euro	Risk	Tax	Telco
Add	11.4%	6.9%	10.6%	18.8%	20.5%
Compare	3.4%	6.1%		7.1%	
Copy	0.5%	0.4%	0.0%	1.9%	
Divide	23.6%	37.3%	1.0%	4.8%	
ToIntegral	12.8%	19.5%		1.2%	
Max	0.3%				
Min				1.3%	
Minus	0.0%			0.0%	
Multiply	24.5%	18.9%	27.5%	7.1%	38.4%
Quantize			30.6%		33.1%
Sqrt			0.2%		
Subtract	10.1%	8.2%	26.4%	3.1%	
ToBinary	1.0%		0.2%		
ToString					3.0%
Zero	0.1%	0.0%	0.0%	2.5%	0.0%
Total Decimal	87.7%	97.3%	96.5%	47.8%	95.6%

E. Rounding

Rounding is an important aspect of DFP calculations. It can be very complex and computationally intensive, especially for operands that use the BID encoding. For example, rounding an intermediate result of a multiplication in BID uses a multiply of width of approximately $2k$ by $2k$, where k is the maximum number of digits in the intermediate result significand [17]. Therefore, the frequency of rounding impacts overall performance. This section examines whether the frequency of rounding changes given the additional precision from decimal64 to decimal128 for our benchmarks.

Table X shows the frequency of rounding for common decimal operations across the benchmarks. It is important to note that, for these operations, rounding only occurs if the infinitely precise result requires more precision than is provided by the result format. The quantize operation is often

TABLE IX
BENCHMARK RUNTIME NORMALIZED TO BID64

	Banking	Euro	Risk	Tax	Telco
DN64	2.12	2.38	1.02	1.01	1.25
DPD64	1.66	1.59	1.12	0.96	1.17
BID64	1	1	1	1	1
DN128	2.65	3	1.16	1.41	1.51
DPD128	2.58	2.56	1.71	1.33	1.74
BID128	2.7	2.99	1.43	1.28	2.08

TABLE X
ROUNDING IN DECIMAL OPERATIONS

	Banking		Euro		Risk	Tax	Telco
	64	128	64	128			
Add/Sub	6.839%	6.837%	0.09%	0.00%	0.06%	0.03%	0%
Divide	6.20%	6.20%	32.8%	32.8%	96.3%	2.84%	0%
Multiply	25.64%	24.6%	11.09%	0.00%	0.19%	0.26%	0%

used after DFP operations to round values to a specified quantum (e.g., the nearest hundredth). For a given operand size, the frequency of rounding is not implementation-dependent, so results are only shown for the decNumber arbitrary-precision library.

Separate columns are shown within Banking and Euro for decimal64 and decimal128. These are the only benchmarks in which rounding frequency differed between the two operand sizes. For these two applications, the frequency of rounding decreases for decimal128 because the increased precision eliminates the need for rounding in some cases. This effect is particularly noticeable in the Euro benchmark in which all addition, subtraction, and multiplication rounding is eliminated by the use of decimal128 precision. This can be explained by the nature of the input data for this benchmark. Euro works on monetary amounts multiplied by six-digit conversion rates. If the original monetary amount exceeds 10 decimal digits (e.g., $100,000,000.00), the intermediate result of the multiply can exceed the 16-digit precision allowed by decimal64 and will require rounding if the decimal64 format is used. However, to exceed the precision of the decimal128 format, the original monetary amount must be at least 28 decimal digits, which is uncommon for this application.

The results in Table X indicate that, for the majority of our benchmarks, the precision provided by decimal64 is sufficient. Furthermore, rounding as part of addition, subtraction, and multiplication is not very frequent in these benchmarks. Hardware DFP implementations may therefore benefit from a variable-latency approach in which the operations can complete early if rounding is not needed.

VI. CONCLUSION

In this paper we analyzed various DFP solutions on representative benchmarks to better understand their performance tradeoffs. We showed that the number of cycles per operation, execution profile, and overall performance can be significantly different depending on the chosen DFP solution. This information is essential for hardware and software designers in choosing how to accelerate DFP.

We also quantified the performance loss that results when moving from decimal64 to decimal128 types. This penalty can be especially high for BID because of the use of expensive rounding methods in decimal128. However, the benchmarks demonstrate that rounding frequency is often similar for both decimal64 and decimal128. This suggests that most pre-rounded results that exceed the precision of decimal64 also exceed the precision of decimal128; therefore, if applications do not need the extra precision of decimal128, they can gain significant performance benefits by using decimal64 instead.

Finally, the profiles indicate that, for most benchmarks, more than 70% of the total execution time is spent in decimal operations when using decimal64 types, and more than 80% when using decimal128 types. For applications that use decimal128 instead of decimal64, there is increased potential for speedup through hardware acceleration.

REFERENCES

[1] M. F. Cowlishaw, "Decimal floating-point: Algorism for computers," in *IEEE Symposium on Computer Arithmetic*, 2003, pp. 104–111.

[2] IBM. (2005, Mar.) The telco benchmark. [Online]. Available: speleotrove .com/decimal/telco.html

[3] *IEEE Standard for Floating-Point Arithmetic*, IEEE Std. 754-2008, 2008.

[4] L. Eisen, J. W. Ward, H.-W. Tast, N. Mäding, J. Leenstra, S. M. Müller, C. J. 0002, J. Preiss, E. M. Schwarz, and S. R. Carlough, "IBM POWER6 accelerators: VMX and DFU," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 663–684, 2007.

[5] A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic, "Decimal floating-point in z9: An implementation and testing perspective," *IBM Journal of Research and Development*, vol. 51, no. 1/2, pp. 217–228, 2007.

[6] E. M. Schwarz, J. S. Kapernick, and M. F. Cowlishaw, "Decimal floating point support on the IBM System z10 processor," *IBM Journal of Research and Development*, vol. 53, no. 1, pp. 4:1–4:10, 2009.

[7] M. Cowlishaw, "Decimal arithmetic bibliography," March 2009. [Online]. Available: speleotrove .com/decimal/decbibalpha.html

[8] L.-K. Wang, C. Tsen, M. J. Schulte, and D. Jhalani, "Benchmarks and performance analysis of decimal floating-point applications," in *International Conference on Computer Design*, 2007, pp. 164–170.

[9] N. Chainani, "Decfloat: The data type of the future," in *www .ibm. com/developerworks/data/library/techarticle/dm-0801chainani*, 2008.

[10] M. Cornea. (2008, Mar.) Intel decimal floating-point math library. [Online]. Available: software. intel .com/en-us/blogs/2008/03/06/intel-decimal-floating-point-math-library/

[11] GCC, the GNU Compiler Collection. [Online]. Available: gcc .gnu. org

[12] L.-K. Wang, "Processor support for decimal floating-point arithmetic," Ph.D. dissertation, Madison, Wis, USA, 2007.

[13] D.-G. for Economic and F. A. C. from the commission to the European council, "Review of the introduction of euro notes and coins," *EURO PAPERS*, Apr. 2002.

[14] A. Damodaran. (2008) Corporate finance. [Online]. Available: pages .stern. nyu .edu/ adamodar/

[15] A. Roberts. (2007, Feb.) Open tax solver. [Online]. Available: opentaxsolver .sourceforge.net/

[16] D. of the Treasury. International Revenue Service. All returns in tax year 2004. [Online]. Available: www .irs. gov/pub/irs-soi/04in01ar.xls

[17] M. Cornea, C. Anderson, J. Harrison, P. T. P. Tang, E. Schneider, and C. Tsen, "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format," in *IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 2007, pp. 29–37.

[18] Intel, "Using the RDTSC Instruction for Performance Monitoring," Intel Corporation, Tech. Rep., 1997.