Doc No: N3344=12-0034
Date: 2012-01-15
Authors: Stefano Pacifico
Alisdair Meredith
John Lakos

Toward a Standard C++ 'Date' Class

Bloomberg L.P.
Stefano Pacifico (*spacifico1@bloomberg.net*), Alisdair Meredith (*ameredith1@bloomberg.net*), John Lakos (*jlakos@bloomberg.net*)

Abstract

The need to represent date values in software is both important and ubiquitous. Having a Standard Library type to represent date values would help to improve developer productivity and ensure interoperability across disparate components that might use such values in their interfaces. Specifying an interface for such a standard type should be informed by performance implications that interface might impose. In particular, the degree to which explicit precondition checking is part of the contract will significantly affect performance. Moreover, fundamentally different implementation strategies (e.g., field-based, serial-based) are viable, and will result in very different performance trade-offs; hence, a standard interface would ideally admit all such implementations. In this report, we examine the results of applying a suite of eight candidate implementations – each with and without precondition checking enabled – to a carefully chosen set of benchmarks across several common computer platforms. We discuss the relative merits of the various implementation choices, including the overhead of mandatory precondition checking. Based on this analysis, we conclude that such precondition checking would be prohibitively expensive, and we provide the essential interface for a flexible, maximally efficient standard date type admitting all known viable implementations.

Contents

1 Introduction 4

2 Proposed Implementations 6
2.1 Checked vs. Unchecked . . . 7
2.2 Field-Based Implementations . . . 7
2.2.1 `YMD_12` . . . 7
2.2.2 `YMD_4` . . . 8
2.3 Howard Hinnant's Date Types . . . 8
2.3.1 `HH_YMD_4` . . . 9
2.3.2 `HH_YMD_SERIAL_8` . . . 9
2.3.3 `HH_SERIAL_4` . . . 10
2.4 Raw Serial Type . . . 10
2.4.1 `HH_SERIAL_RAW_4` . . . 10
2.5 Bloomberg Date Types . . . 11
2.5.1 Cache Optimization . . . 11
2.5.2 `BDET_CACHED_4` . . . 11
2.5.3 `BDET_UNCACHED_4` . . . 12

3 Benchmarks 13
3.1 Benchmarks Overview . . . 13
3.2 Timing . . . 14
3.3 Benchmark Scenarios . . . 14
3.3.1 Creators . . . 14
3.3.2 Accessors . . . 14
3.3.3 Increase Day . . . 15
3.3.4 Increase Month . . . 15
3.3.5 Array Sorting . . . 15
3.3.6 Modified Following . . . 15
3.3.7 Packed Calendar . . . 17

4 Platforms 18

5 Results 20
5.1 Creators . . . 21
5.2 Accessors . . . 23
5.3 Increase Day . . . 25
5.4 Increase Month . . . 26
5.5 Array Sort . . . 28
5.6 Modified Following . . . 29
5.7 Packed Calendar . . . 30

6 Analysis and Discussion **31**
6.1 Overview 31
6.2 Benchmarks and a Figure of Merit 32
6.3 Serial-Based vs. Field-Based Implementations 33
6.4 The Benefits of Caching for Serial-Based Implementations 33
6.5 The Cost of Checking 34
6.6 An Observation on the Underlying Calendar 34

7 Conclusions **35**

8 Recommendations **37**

Appendix A **i**

Acknowledgments

We would like to deeply thank Tom Marshall, Henry Mike Verschell, Alexei Zakharov for their important technical advise throughout. We would also like to thank Chen He, Rob Shearer, and Jeffrey Schwab for further feedback and proofreading. Thank you!

Source Code Availability

Source code for the `date` implementations described in this report will be made available upon request.

Chapter 1

Introduction

Howard Hinnant presented an early TR2 proposal for date types at the Bloomington meeting, which led to an active discussion online regarding the appropriate interface for date types and, in particular, whether operations should be checked or not, where a checked interface will detect invalid date values, and reject them, e.g., by throwing an exception. While it is clear that checked operations will be more expensive than unchecked, it is unclear just how much more expensive. The initial impetus for this report was to measure the impact of enforced checking, to inform the Library Working Group when designing the interface for a standard date facility. This report also explores several other performance concerns with the design of a standard `date` class, including the relative performance of different date implementations (e.g., serial vs field representations), the choice of calendar (Gregorian vs proleptic Gregorian), and the impact of additional meta-data within the `date` value (such as the day of the week), and a static cache to optimize conversions between serial and field representations.

To properly determine the effect of checking, we measure the performance of both checked and unchecked implementations of common `date` type representations, over a set of common "primitive" operations as well as some representative algorithms. For the purposes of this paper, the `date` class is deliberately primitive, deferring extensive calendar facilities to another component trafficking in such `date` types in its own interface.

There are two primary options for representing a date value, either as separate year, month and day fields (a field representation), or as an offset counting days from a fixed epoch date (a serial representation). We examine several implementations from each category (as well as a hybrid implementation). We also test variations of these basic implementation types that include additional meta-data about the date (such as the day of the week).

Finally, we provide two mostly equivalent implementations, one using a Gregorian calendar, and the other using a proleptic Gregorian calendar, to determine what, if any, impact the choice of calendar has on performance. We believe the performance information presented here will be interesting to any supplier of a standard date facility, regardless of the interface adopted by the standard.

Chapter 2

Proposed Implementations

In this chapter, we describe the various implementations of a `date` class that we will use in our experiments. These implementations are intended to represent a range of possible designs of a `date` class and can be divided into two categories: *field-based* implementations and *serial-based* implementations. Field-based implementations store the information for year, month, and day in three different integral data members (sometimes in a *bitfield*). Serial-based implementations store a date value as the integral number of days from an implementation-defined *epoch* date. We have adopted class names that should suggest the implementation being used. Field-based implementations are indicated by `YMD` in the name, and serial-based implementations are indicated by `SERIAL` in the name. The one hybrid implementation combines both terms in its name, `HH_YMD_SERIAL_8`. In addition, each class name terminates with a numeric suffix that represents the size of the type in bytes. For example, `YMD_12` is a field-based implementation of `date` that holds integral members representing the year, month, and day of a date value, and has a size of 12 bytes.

We expect field-based `date` implementations and serial-based `date` implementations to be better suited to different sets of operations. For example, operations that we might expect to be faster for a serial-based implementation include adding N days to a date, and determining the number of days between two dates. Operations we might expect to be faster for a field-based implementation include constructing a `date` value from individual year, month and day values, and accessing the individual year, month, and day fields of a `date`. Note that we will see that, for serial-based implementations, it is advantageous to use a cache in order to achieve performance comparable to field-base implementations for operations such as setting any of the year, month, and day fields.

2.1 Checked vs. Unchecked

When designing a date type, there are two approaches to dealing with client-supplied values that may be invalid: *wide contracts* that guarantee the detection of such invalid values, and contractually define how such errors will be handled (e.g., by throwing an exception), and *narrow contracts* that simply declare that invalid values result in *undefined behavior*. For this reason, every implementation examined in this report has both a *checked* and an *unchecked* version. The checked implementations discussed in this paper fulfill their wide contract by throwing an exception when an invalid value is detected. The unchecked implementations discussed in this paper do not test for precondition violations (or are built in a mode in which these checks are compiled out), and rely on the client to obey the contract of the class to avoid violating any preconditions. Such narrow contracts allow faster execution for well-formed data. Note that an unchecked implementation does not preclude providing additional methods that perform explicit checks (e.g., `set_yearmonthday_if_valid`).

2.2 Field-Based Implementations

The following two implementations are prototypes we created for the sole purpose of this investigation, are not complete, and implement only those methods that are exercised by our benchmarks. These two implementations are the most straightforward, and serve as a baseline against which the other, more robust implementations are measured. We believe that production-quality implementations of such classes would exhibit comparable performance.

2.2.1 `YMD_12`

The class `YMD_12` represents the most straightforward implementation of a date class, and will be used throughout this paper as the *baseline* implementation. This class holds a date value by storing the year, month, and day values in three separate 32-bit `int` data members, resulting in trivial constructors, accessors, and manipulators. Each object of this class occupies 12 bytes in memory, and can represent dates having years in the range $[INT\_MIN, INT\_MAX]$.

```
class YMD_12 {
//...
    int d_year;
    int d_month;
    int d_day;
//...
};
```

2.2.2 `YMD_4`

The class `YMD_4` is a more space-efficient version of `YMD_12`. Although `YMD_12` is used as the baseline implementation in this investigation, `YMD_4` may be a more realistic example of a simple field-based implementation due to its smaller footprint. Each object of this class occupies 4 bytes in memory, and can represent dates having years in the range $[SHORT\_MIN, SHORT\_MAX]$.

```
class YMD_4 {
//...
    short d_year;
    char  d_month;
    char  d_day;
//...
};
```

2.3 Howard Hinnant's Date Types

The following three `date`-class implementations were provided by Howard Hinnant (`hhinnant@apple.com`) for the Bloomington meeting of the ISO C++ committee. These types are all prefixed with `HH`. All these implementations use the types `chrono::year`, `chrono::month`, and `chrono::day` in their interface. All these implementations also include two additional data members: `n_` and `dow_`. The `n_` and `dow_` data members are set on construction from the corresponding `n_` and `dow_` values embedded in the supplied `chrono::day`. The `n_` and `dow_` in `chrono::day` allow different ways to specify the day value, e.g., the 3rd day of the month, or the 3rd Saturday of the month. In the context of a `date`, the `n_` and `dow_` values also determine how that `date` will respond to date arithmetic (particularly incrementing the month).

2.3.1 HH_YMD_4

The class HH_YMD_4 is a field-based implementation that stores, in addition to the year, month, and day, a field to indicate whether or not the year is a leap year. Two additional fields, n_ and dow_, determine the interpretation of the day value, as discussed in section 2.3. Each object of this class occupies 4 bytes in memory, and can represent dates having years in the range $[-32768, 32767]$.

```
class HH_YMD_4 {
//...
    int16_t y_;           // year
    uint16_t m_    : 4;  // month
    uint16_t d_    : 5;  // day
    uint16_t leap_ : 1;  // leap year flag
    uint16_t n_    : 3;  // date value control field
    uint16_t dow_  : 3;  // day of the week
//...
};
```

2.3.2 HH_YMD_SERIAL_8

The class HH_YMD_SERIAL_8 is an extension of class HH_YMD_4 (see section 2.3.1). This implementation stores an additional 4-byte integer data member holding a serial representation of the date value stored in the other data members. Two additional fields, n_ and dow_, determine the interpretation of the day value, as discussed in section 2.3. Each object of this class occupies 8 bytes in memory, and can represent dates having years in the range $[-32768, 32767]$.

```
class HH_YMD_SERIAL_8 {
//...
    uint32_t x_;          // serial date
    int16_t  y_;          // year
    uint16_t m_    : 4;  // month
    uint16_t d_    : 5;  // day
    uint16_t leap_ : 1;  // leap year flag
    uint16_t n_    : 3;  // date value control field
    uint16_t dow_  : 3;  // day of the week
//...
};
```

2.3.3 HH_SERIAL_4

Unlike HH_YMD_SERIAL_8 and HH_YMD_4, the class HH_SERIAL_4 does not store the fields year, month, day directly, but instead stores only the serial representation of the date in a 4-byte integer data member. Two additional fields, n_ and dow_, determine the interpretation of the day value, as discussed in section 2.3. Each object of this class occupies 4 bytes in memory, and can represent dates having years in the range $[-32768, 32767]$. In this case, the range of valid years for a date is determined by the contract of the type, rather than by its storage limitations.

```
class HH_SERIAL_4 {
//...
    uint32_t x_   : 26;  // serial date
    uint32_t n_   :  3;  // date value control field
    uint32_t dow_ :  3;  // day of the week
//...
};
```

2.4 Raw Serial Type

In order to observe the different performance characteristics of the logic supplied by Howard Hinnant for transforming between serial and field representations of a date, we introduced a new “raw” type, in addition to the types provided by Howard Hinnant himself. In particular the logic supplied by Howard Hinnant makes use of the proleptic Gregorian calendar, in contrast with the Gregorian calendar used by Bloomberg’s date type. This implementation may be considered to be the ‘trivial’ serial-based implementation as an analogue for the trivial field-based implementation, YMD_12.

2.4.1 HH_SERIAL_RAW_4

The class HH_SERIAL_RAW_4 stores the serial representation of the date in an unsigned 4-byte integer data member, as do HH_SERIAL_4 and HH_YMD_SERIAL_8 (see sections 2.3.3 and 2.3.2). However, unlike HH_SERIAL_4 and HH_YMD_SERIAL_8, this type does not store n_, nor dow_.

```
class HH_SERIAL_RAW_4 {
//...
     unsigned int d_serialDate;
//...
};
```

2.5 Bloomberg Date Types

The following two date types are based on the production `date` type currently used at Bloomberg LP. These types are each prefixed with `BDET` (Bloomberg Development Environment Type), and each stores a 4-byte integer value that represents the number of days since the *epoch* date *1/1/1*.

2.5.1 Cache Optimization

Serial-based date implementations can be optimized using a cache of pre-calculated values, which allow fast bi-directional conversions between a serial date representation and its corresponding year, month, and day values, within a given range of dates of interest. The static cache used by the `BDET` cached types consists of two lookup tables: one mapping a serial date to the corresponding year, month, and day triplet; the other mapping a month and year pair to a serial date. The Bloomberg date class is optimized with a 64 kB cache for dates in the range *[1984/1/1, 2040/12/31]*. In addition, a modified version of the Bloomberg date class that does not support any caching mechanism, `BDET_UNCACHED_4`, was included in this report, both to illustrate the benefits of the static cache, and to provide a point of comparison between the Gregorian calendar used by the `BDET` types, and the proleptic Gregorian calendar used by Howard Hinnant's types. In particular, the only difference in implementation between the `HH_SERIAL_RAW_4` class and the `BDET_UNCACHED_4` class is the calendar used, and the results for just these two types should be compared to evaluate the merits of each calendar.

2.5.2 `BDET_CACHED_4`

The class `BDET_CACHED_4` is the standard date class used at Bloomberg. `BDET_CACHED_4` stores the number of days since 1/1/1 in a 4-byte integer, and makes use of a static cache to optimize conversions.

```
class BDET_CACHED_4 {
//...
    int  d_date;                // absolute (serial) date
//...
};
```

2.5.3 BDET_UNCACHED_4

The class BDET_UNCACHED_4 is a modified version of BDET_CACHED_4 (see section 2.5.2), whose interface and implementation are the same as BDET_CACHED_4 except that the conversions to and from the serial representation are performed without using a static cache.

Chapter 3

Benchmarks

A *benchmark* is a particular sequence of operation invocations designed to measure certain performance characteristics.

In this chapter we describe various benchmarks used to estimate the relative performance characteristics of the proposed `date` implementations. In particular, we will be looking to observe the effective cost of mandated checking in the class interface, the relative merits of field-based implementations vs. serial-based implementations, and whether there is a measurable performance benefit to the simple proleptic Gregorian calendar.

3.1 Benchmarks Overview

We devised seven benchmarks that were exercised for each `date` implementation:

1. Create a `date` object from trusted input. 3.3.1
2. Extract the date fields (year, month, day) from a `date` object. 3.3.2
3. Advance a `date` object by one day. 3.3.3
4. Advance a `date` object by one month. 3.3.4
5. Sort an array of `date` objects. 3.3.5
6. Execute the `modifiedFollowing` algorithm to find settlement dates. 3.3.6
7. Run a packed-calendar workload. 3.3.7

Each benchmark estimates the time to perform the considered operation by calculating the average execution time over a given (large) number of iterations.

3.2 Timing

Each measure presented in this report is the average of the times reported by several runs of a single test program (one for each benchmark). Each test program consists of a simple `main` that serves as a test launcher, and a test function template that is parametrized on the `date` type to be benchmarked. The launcher calls a particular test function template instantiation according to a `date` type specified as part of the input of the program. The timings used in the generation of the charts in this report are the average of the user time reported by the operating system, cf. `man time`.

3.3 Benchmark Scenarios

The scenarios described in this section can be broadly broken into two groups: *micro-benchmarks* and *composite-benchmarks*. The micro-benchmarks attempt to measure the efficiency of a single primitive operation on `date`. The composite-benchmarks perform a sequence of operations intended to be more representative of a real-world use-case for the `date` class. In order to avoid compiler optimizations that would render the benchmark meaningless (particularly for the micro-benchmarks), extra `volatile` accumulator variables were used inside the test-loops to record the results of the operation under test.

3.3.1 Creators

This benchmark creates `date` objects in a loop, repeatedly creating three fixed date values: *2000/1/1*, *2000/12/31*, and *2009/5/1*.

The objective of this benchmark is to measure the performance of the value constructor of each `date` implementation.

3.3.2 Accessors

This benchmark repeatedly extracts the year, month, and day from a constant `date` object having the value 2001/1/1, by invoking the `year`, `month`, and `day` accessors. The date 2001/1/1 is in the range of values that are preloaded in the Bloomberg cached implementation.

The objective of this benchmark is to measure the performance of the year, month, and day accessors of each `date` implementation.

3.3.3 Increase Day

This benchmark repeatedly advances a `date` object by one day, starting from date 1/1/1 and incrementing 3652061 times.

The objective of this benchmark is to measure the performance of `date` arithmetic, specifically `operator++`, for each `date` implementation.

3.3.4 Increase Month

This benchmark repeatedly advances a `date` object by one month, starting from date 1/1/1 up to the year 9000. When the month advances, there are some cases that would form an invalid `date`, such as advancing from 30th January into February. Our policy in such circumstances is to clip to the end of the new month, and this clipped day-of-month remains for all future advances.

The objective of this benchmark is to measure the performance of an operation that advances the month of a `date` by one unit, which is a non-primitive operation. However, each of the 'Hinnant' implementations provide a primitive method to increment the month in their class interface; this method is used by those types that provide it, so as not to unfairly penalize those designs.

3.3.5 Array Sorting

This benchmark sorts a `vector` of `date` objects. More specifically, the `vector` consists of repeated sequences of `date` objects arranged in an adverse order for popular sort algorithms (decreasing from date 9999/12/31 to 01/01/01).

The objective of this benchmark is to analyze a larger set of operations on the `date` types – in particular `operator<` and copy operations.

3.3.6 Modified Following

This benchmark invokes the function `modifiedFollowing` 360 times (once per month for 30 years) specifying the same fixed day of the month, and using a predefined calendar (an array of bits that indicates, from a given starting date, whether or not a date at a specified offset from the starting date is a business day).

The objective of this benchmark is to provide a real-world usage example of the `date` classes for a mission-critical application that demands maximal runtime performance. This function is used in financial domains to calculate settlement dates for financial obligations, an operation that can have very demanding performance requirements, as it is routinely applied in multi-dimensional analysis.

The implementation of the *modified following* algorithm is outlined below:

```
bool isNonBusinessDay(const Date&  targetDate,
                      const Date&  startDate,
                      const int   *calendar)
{
    int offset = targetDate - startDate;

    int wordIndex = offset / 8 / sizeof(int);
    int bitIndex = offset - wordIndex * sizeof(int);

    return (1 == (calendar[wordIndex] & (1 << bitIndex)));
}

Date modifiedFollowing(const Date&  targetDate,
                       const Date&  startDate,
                       const int   *calendar)
    // Return the date of the first business day on or after the specified
    // 'targetDate' according to the specified 'calendar', where 'calendar'
    // starts on the specified 'startDate'; if the resulting date does not fall
    // within the same month as 'targetDate', return the date of the last
    // business day of that month (prior to 'targetDate').  The behavior is
    // undefined unless the month of 'targetDate' contains at least one
    // business day.
{
    Date date(targetDate);

    while (isNonBusinessDay(date, startDate, calendar)) {
        ++date;
    }

    if (targetDate.month() == date.month()) {
        return date;
    }

    date = targetDate;

    do {
        --date;
    } while (isNonBusinessDay(date, startDate, calendar));

    return date;
}
```

3.3.7 Packed Calendar

This benchmark populates a ‘packed calendar’, and then reads back those dates. A packed calendar is a space efficient means of storing a set of date values – for example one could construct a calendar representing the set of holidays for a range of years. Such calendars are used frequently in financial applications. The packed calendar implementation used in this benchmark stores an epoch `date`, and an `unsigned short` offset from that epoch to each date in the calendar.

For this benchmark a `vector` of `date`s is populated with a sequence of consecutive dates spanning a decade. Then, we iterate over the `vector`, inserting every date that corresponds to a weekend day into a ‘packed calendar’. Finally we iterate over all the dates in the calendar, converting back to regular `date` values in order to query individual fields.

The objective of this benchmark is to exercise a set of operations that are expected to be relatively common in real-world use of a `date` class.

Chapter 4

Platforms

In this chapter, we identify and characterize the various platforms on which we have obtained the empirical results presented in this report. Each platform is characterized in terms of a processor type, the operating system it runs, the C++ compiler used to compile the benchmark code, and the compiler optimization options.

Linux

- **CPU:** Intel Xeon, X5670 @ 2.93GHz
- **OS:** Linux 2.6.18, x86-64
- **Compiler:** GCC 4.6.1
- **Compiler options:** -Wall -O3 -m32 -DNDEBUG -march=pentium2 -mtune=opteron

Sun

- **CPU:** UltraSPARC-T2+, 1582 MHz
- **OS:** SunOS 5.10 sun4v
- **Compiler:** Sun C++ 5.10 (128228-10)
- **Compiler options:** -O3 -m32 -DNDEBUG -template=no%extdef

AIX

- **CPU:** POWER5, 1.9 GHz
- **OS:** AIX 6.1
- **Compiler:** IBM XL C/C++ for AIX, V10.1 (10.01.0000.0004)
- **Compiler options:** -O2 -DNDEBUG

Chapter 5

Results

In this chapter, we present the results that were obtained in our experiments. For each test, we present a chart that illustrates the performance of each `date` class on all the considered platforms. The charts shown in this section are obtained by executing the tests described in Chapter 3: The average time of execution estimated by the test program was computed and averaged over 10 iterations on each platform. In the first chart of each result section, the bars represent the inverse of the average time (i.e., frequency, higher is better) estimated for the operation being considered, normalized to the performance of the unchecked version of the class `YMD_12` (see section 2.2.1) on each platform. Normalization was chosen in order to make the measures independent of the actual speed (and load) of the machine, as well as to make the measures comparable across different platforms. The second chart shows, for each implementation on each platform, the relative (percentage) overhead of running a checked vs. an unchecked implementation.

There are several examples in the results of the micro-benchmarks where adding the checking code to the implementation caused perturbations in the generated assembly code that result in slightly improved performance for the code compiled with checking enabled. Upon analysis, we found the surprising results could be attributed to several factors, including the compiler making different decisions on which functions to inline, the generation of slightly different instruction sequences, and the fortuitous alignment of code on cache-line boundaries. These effects were generally small, unpredictable, and independent of what we are trying to measure, and is one of the reasons we will later focus our analysis on the composite benchmarks (see section 6.2 for an analysis of the "figure of merit").

5.1 Creators

Figure 5.1 shows the performance of construction from valid date values.

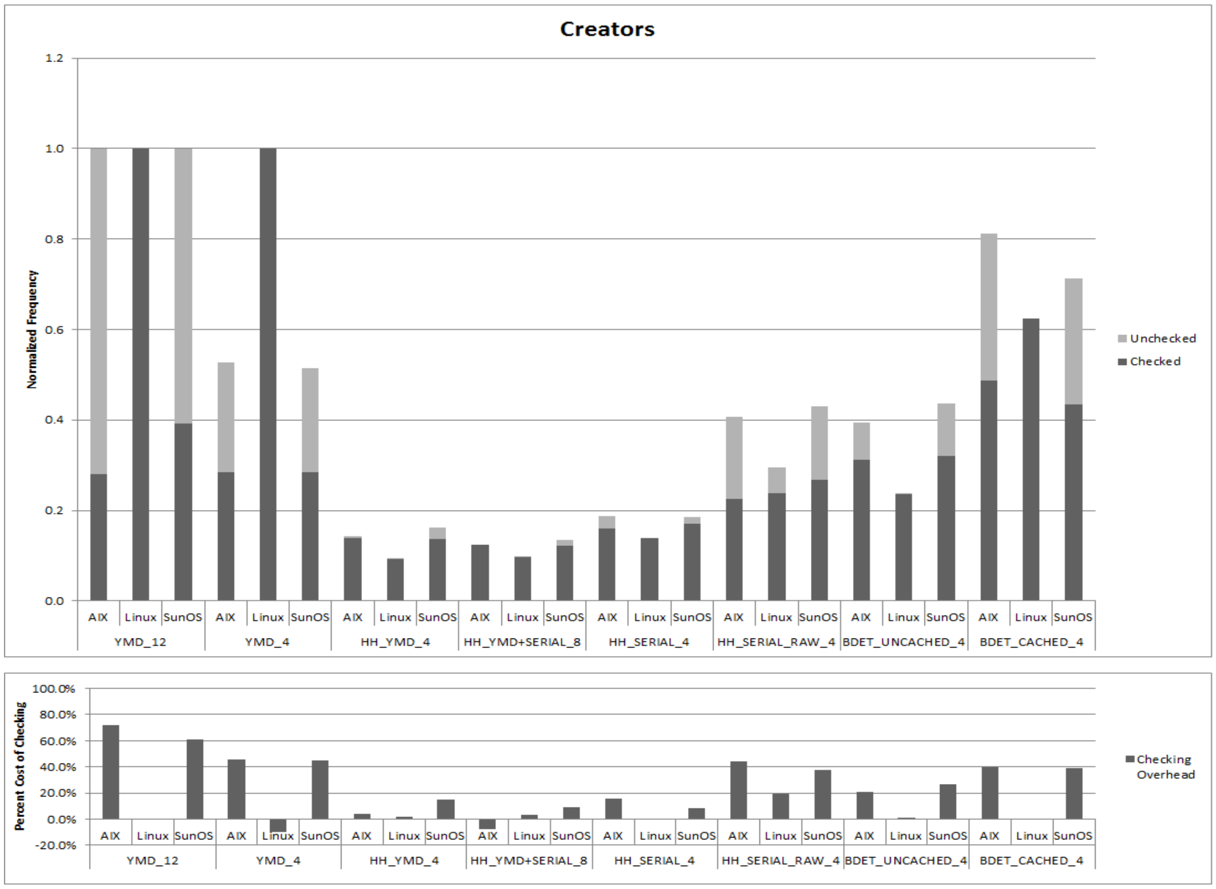

Figure 5.1: Creators Benchmark

The results show that the trivial implementation of the 'simple' field-based date types are more efficient for constructing `date` objects. Conversely, the relative cost of checking with such an interface is high, typically doubling the cost of the operation (or worse).

There is a clear cost to checking for many benchmarked implementations of the `date` interface. Our analysis suggests the cost is effectively fixed, so the demonstrated relative cost is higher on the faster/more-efficient implementations.

It should be noted that the cost of checking on the Linux platform is essentially free. This is a pattern to watch for in all benchmarks. Our research suggests that the hardware on which we are running the Linux benchmarks is particularly good at branch prediction and speculative execution, greatly reducing the cost of checks on this machine when profiling the primitive operations, and is not typical of our production platforms.

Another result we investigated further is the relatively low performance of the 4-byte field-based implementations. This is due to the promotion of smaller internal fields to integers for the result, an effect that is magnified for the bit-field implementations. One notable consequence is that the cache-optimized serial-based implementation outperforms the (packed) field-based implementations on two of our three platforms, on a benchmark where we expected field-based implementations to excel.

Finally, for two of the tests, `YMD_4` on Linux, and `HH_YMD_SERIAL_8` on AIX, the checked implementation performed faster than the unchecked one by a small, but measurable, amount. As noted earlier, this was not the effect of varying load on the system, but rather the result of small, unpredictable, changes in the generated code.

5.2 Accessors

Figure 5.2 shows the performance of pure “const” access, with no manipulation.

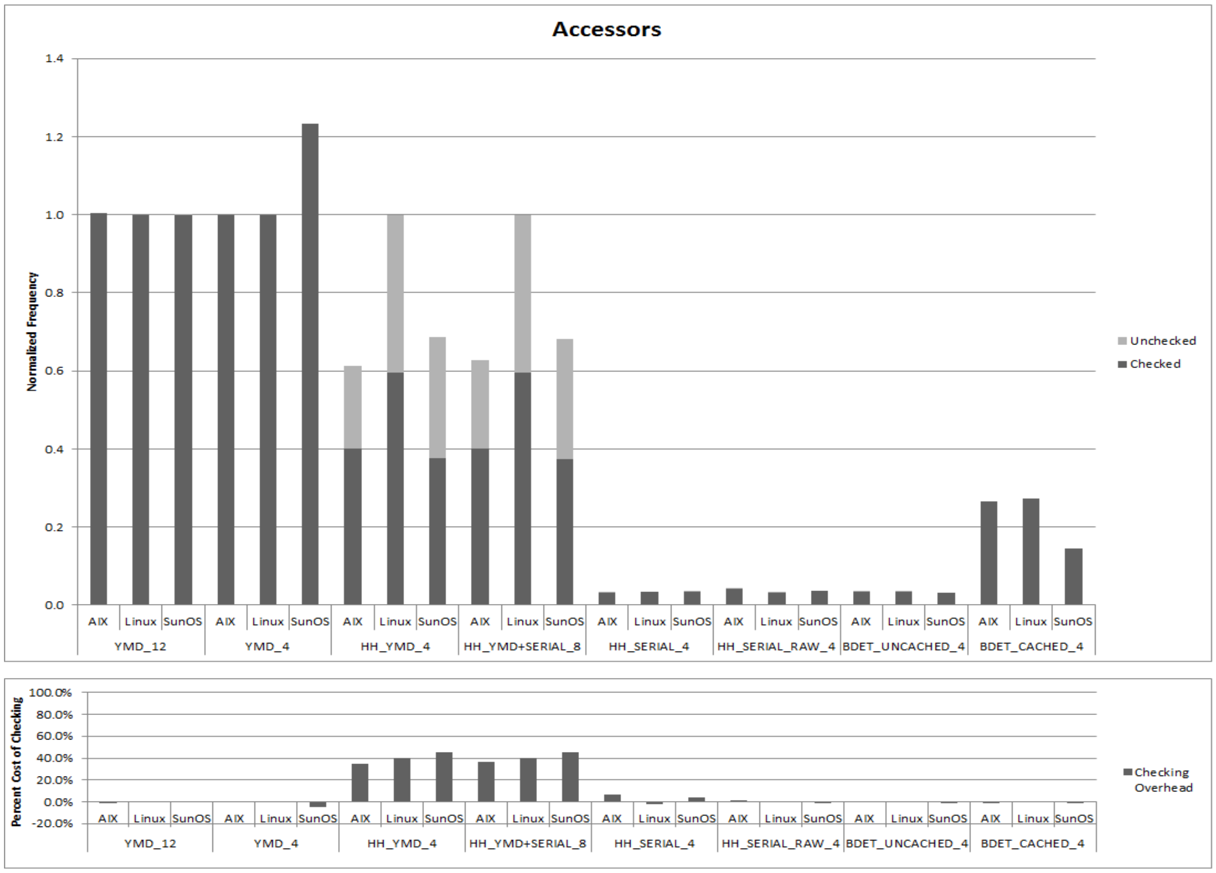

Figure 5.2: Accessors Benchmark

Field-based implementations are typically faster than serial-based implementations due to the trivial implementation of their accessor methods. The caching optimization for serial-based implementations proves vital to maintaining acceptable performance for serial-based implementations relative to the field-based ones.

While we did not expect this read-only operation to show any impact from checks in the interface, a surprising result is that the checked implementation of the accessors of the `HH_YMD_4` and `HH_YMD_SERIAL_8` types run at roughly half the speed of their unchecked implementations. This unexpected behavior turns out to be due to the checking built into each of the `chrono::year`, `chrono::month`, and `chrono::day` types returned by these methods, so just the act of creating the return value from an `int` induces a check in the constructor. This unnecessary overhead might be avoided by providing a (possibly private) unchecked interface to these types. We choose to allow the result to stand, rather than investigate such an optimization, as it highlights the exact concern we are investigating: the cost of checked vs. unchecked interfaces.

5.3 Increase Day

Figure 5.3 shows the performance of advancing the date by one day.

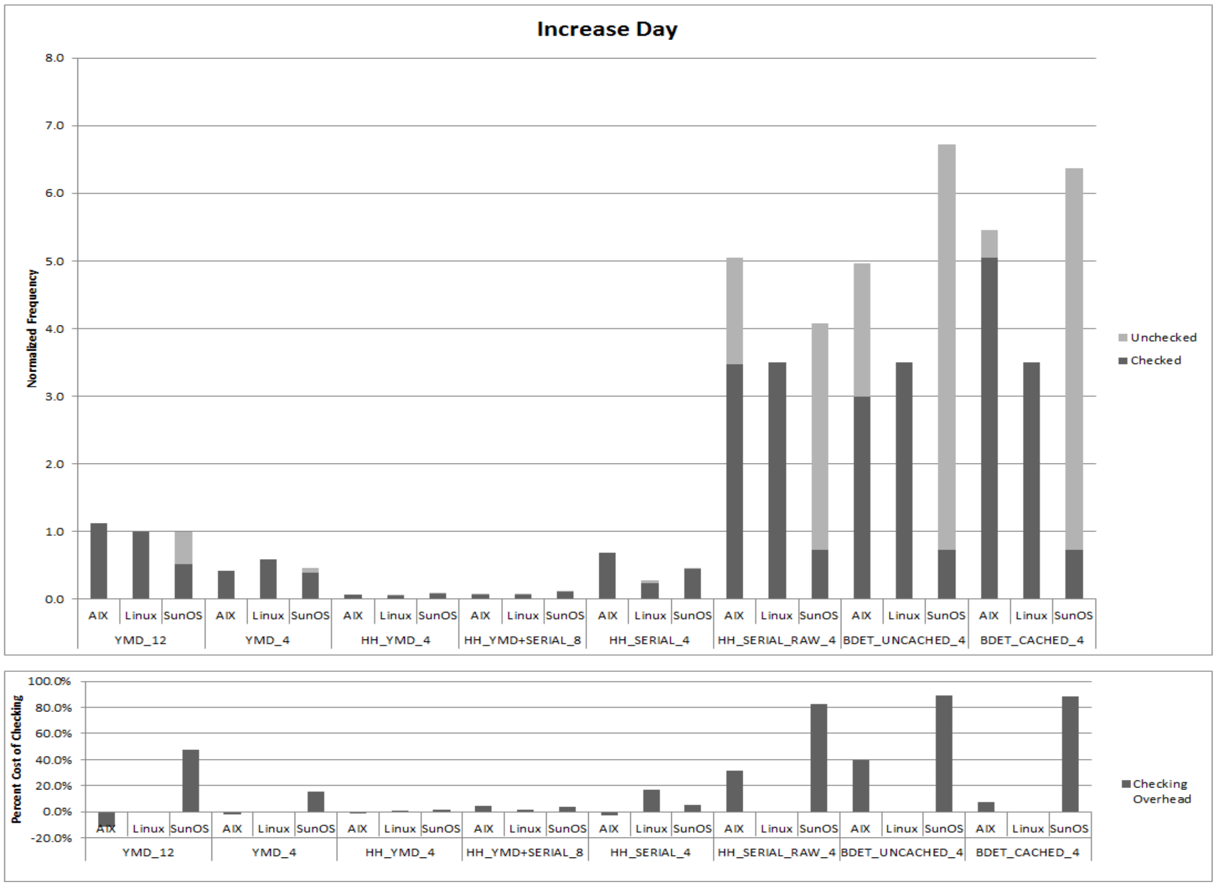

Figure 5.3: Increase Day Benchmark

The Increase Day benchmark greatly favors serial-based implementations, where the operation distills to a single integer increment. The relative cost of checking for overflow on each increment is therefore high.

Field-based implementations clearly demonstrate a disadvantage on this benchmark. However, the overhead of checking appears to be minimal in such implementations, not merely due to the relative cost, but as overflow can occur (and need be checked) only when the date represents December 31st, or just under 0.3% of possible values.

5.4 Increase Month

Figure 5.4 shows the performance of advancing the date by one month.

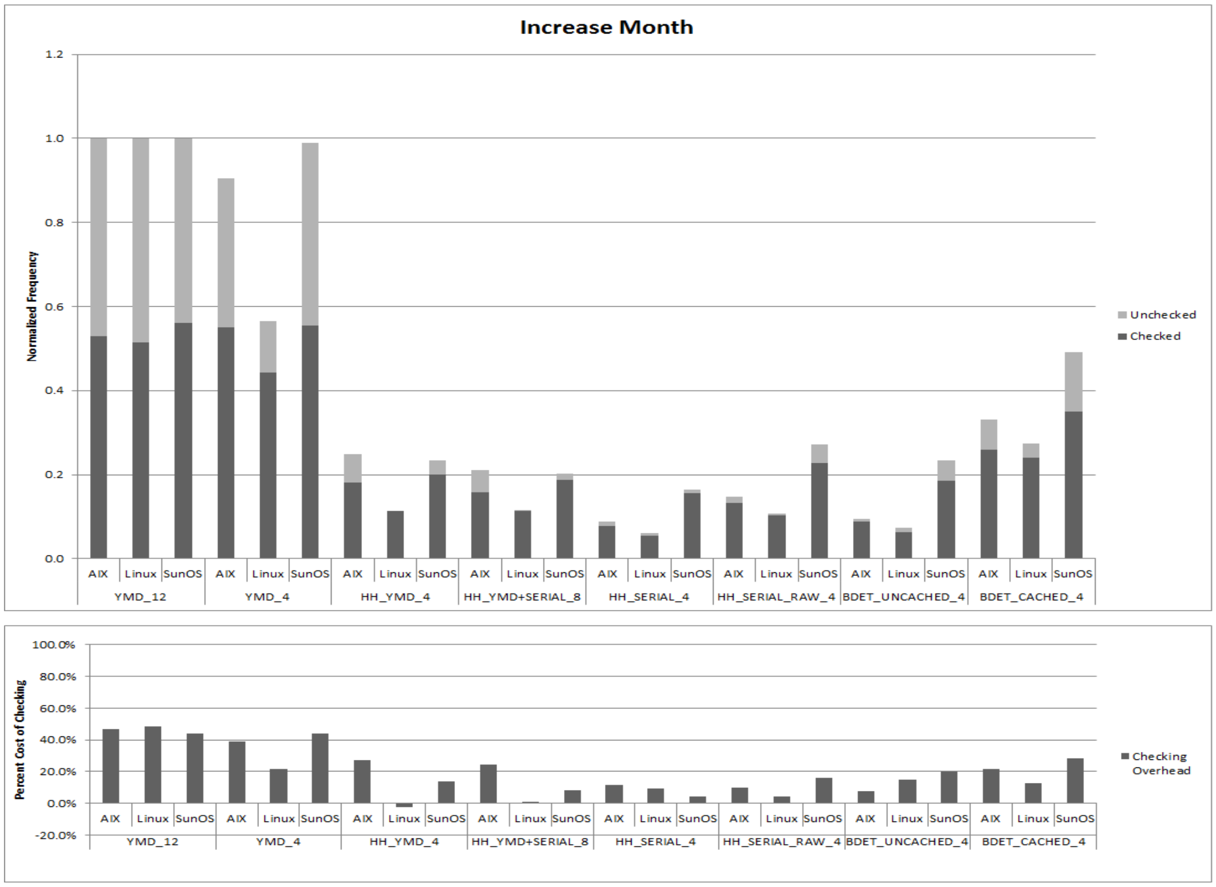

Figure 5.4: Increase Month Benchmark

The “simple” field-based implementations show a clear advantage on this benchmark. The caching optimization for serial-based implementations eliminates some of the advantage, generally running 2-3 times as fast as an uncached implementation.

The better performance of field-based implementations occurs because this benchmark performs most of its operations directly on the individual stored fields, rather than on a single combined value – i.e., the (encoded) serial representation.

Note that this is the first benchmark to highlight the difference between the Gregorian calendar used by the `BDET_UNCACHED_4` implementation, and the proleptic Gregorian calendar used by the `HH_SERIAL_RAW_4` implementation. It demonstrates a performance benefit of roughly 10% for the proleptic calendar.

The lower graph clearly shows a non-negligible cost for checking on almost every implementation. This is a trend to look out for on all the composite benchmarks, which better reflect real-world use than peak throughput.

5.5 Array Sort

Figure 5.5 shows the performance of sorting arrays of dates. It highlights the use of comparison and copy operations. The benchmark is a composite of operations intended to reflect real-world usage.

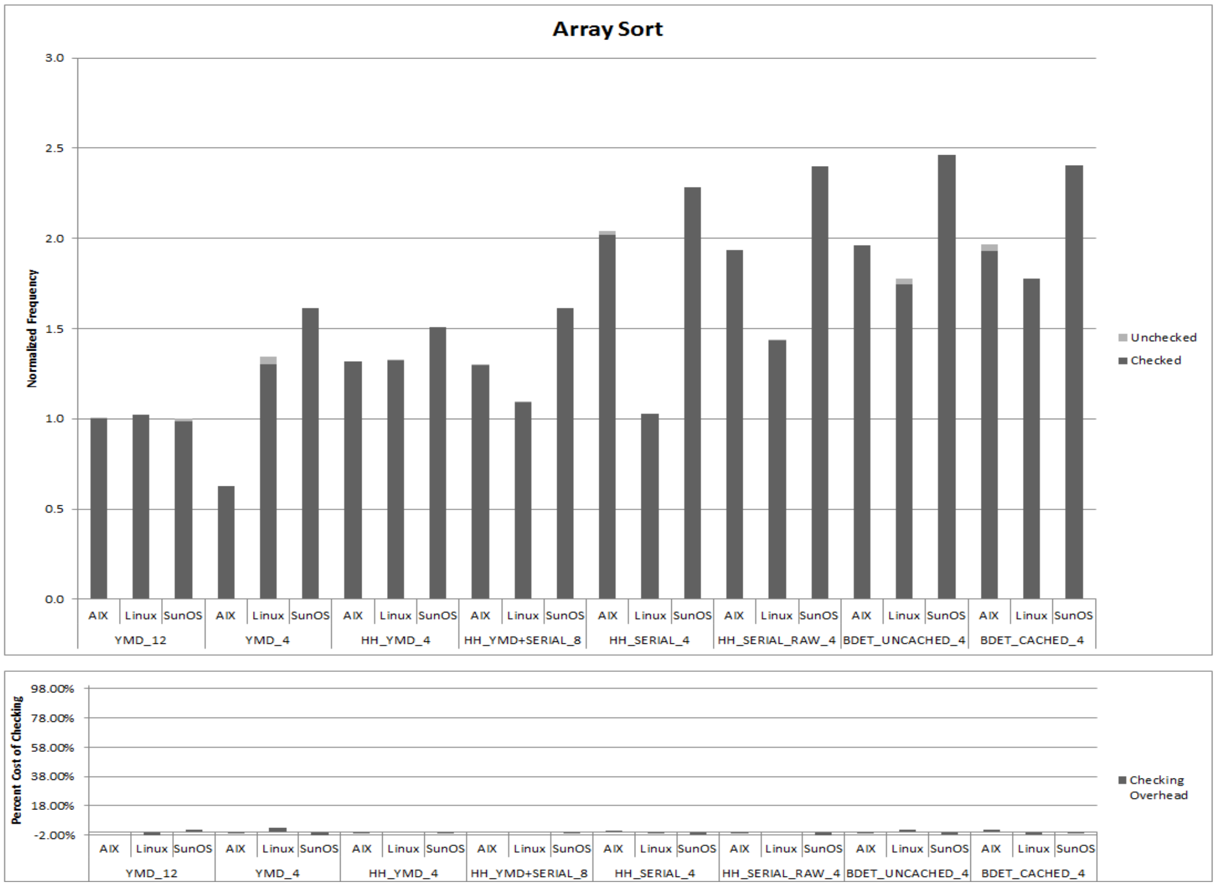

Figure 5.5: Array Sort Benchmark

The key elements to performing well on this benchmark are a small footprint (so less memory is copied, and locality of reference preserved) and an efficient `operator<`, which is demonstrated by all serial-based implementations.

There is no observed benefit to cached information, as it is not relevant to the operations stressed by this benchmark. Likewise, as the operations are either read-only, or are copying whole (valid) values, there is no significant observable effect from checked interfaces, with the tiny observed differences attributed to statistical noise from running the benchmarks.

5.6 Modified Following

Figure 5.6 shows the performance of executing the `modifiedFollowing` algorithm. The benchmark is an example of a real-world algorithm used in performance-sensitive contexts.

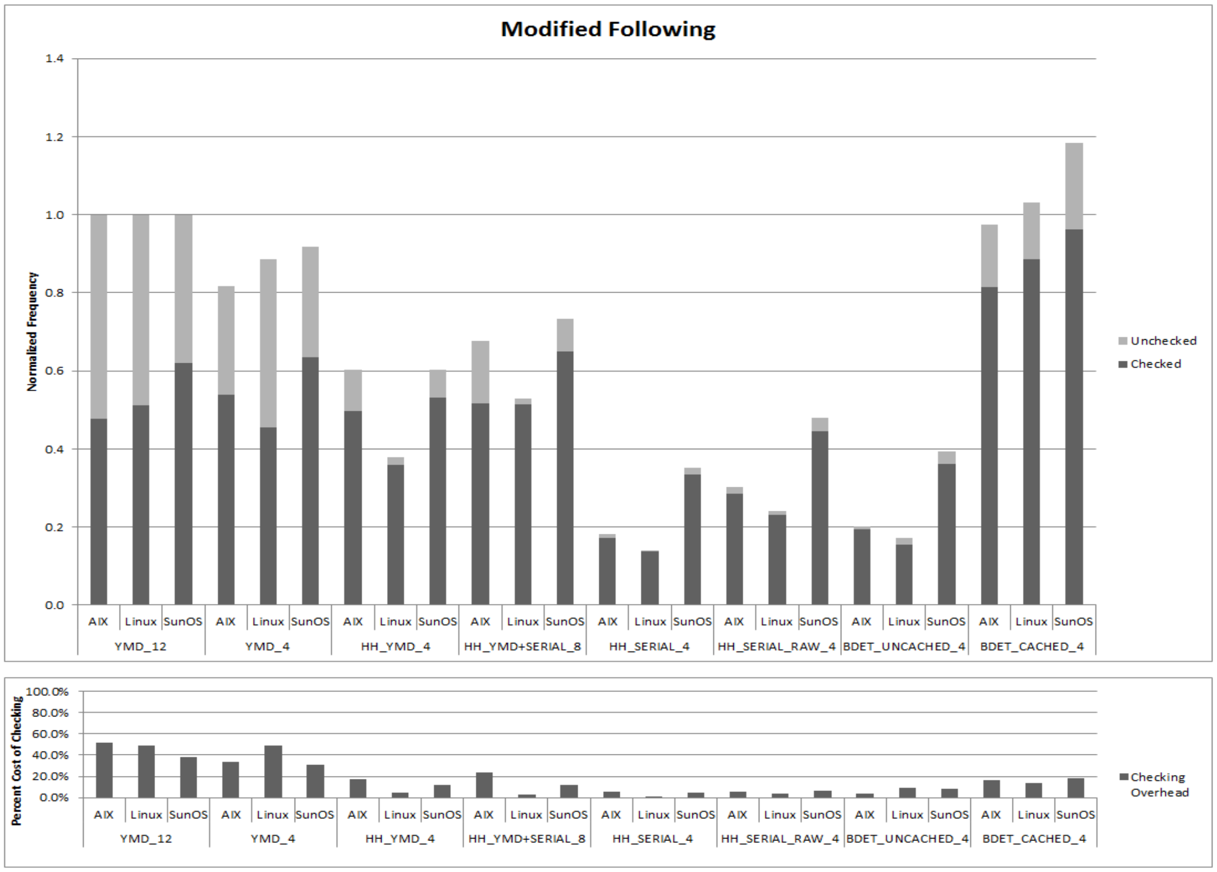

Figure 5.6: Modified Following Benchmark

The lower graph shows a non-negligible cost for checking on most implementations.

The field-based implementations are clearly faster than the uncached serial-based implementations. With the caching optimization, serial-based implementations offer equivalent performance on this benchmark.

The difference in performance between `HH_SERIAL_RAW_4` and `BDET_UNCACHED_4` again demonstrates the superiority of Howard Hinnant logic for serial date generation, including the benefits of the proleptic Gregorian calendar.

5.7 Packed Calendar

Figure 5.7 shows the performance of executing the “Packed Calendar” benchmark, as described in Chapter 3. This benchmark is a composite of operations intended to reflect real-world usage.

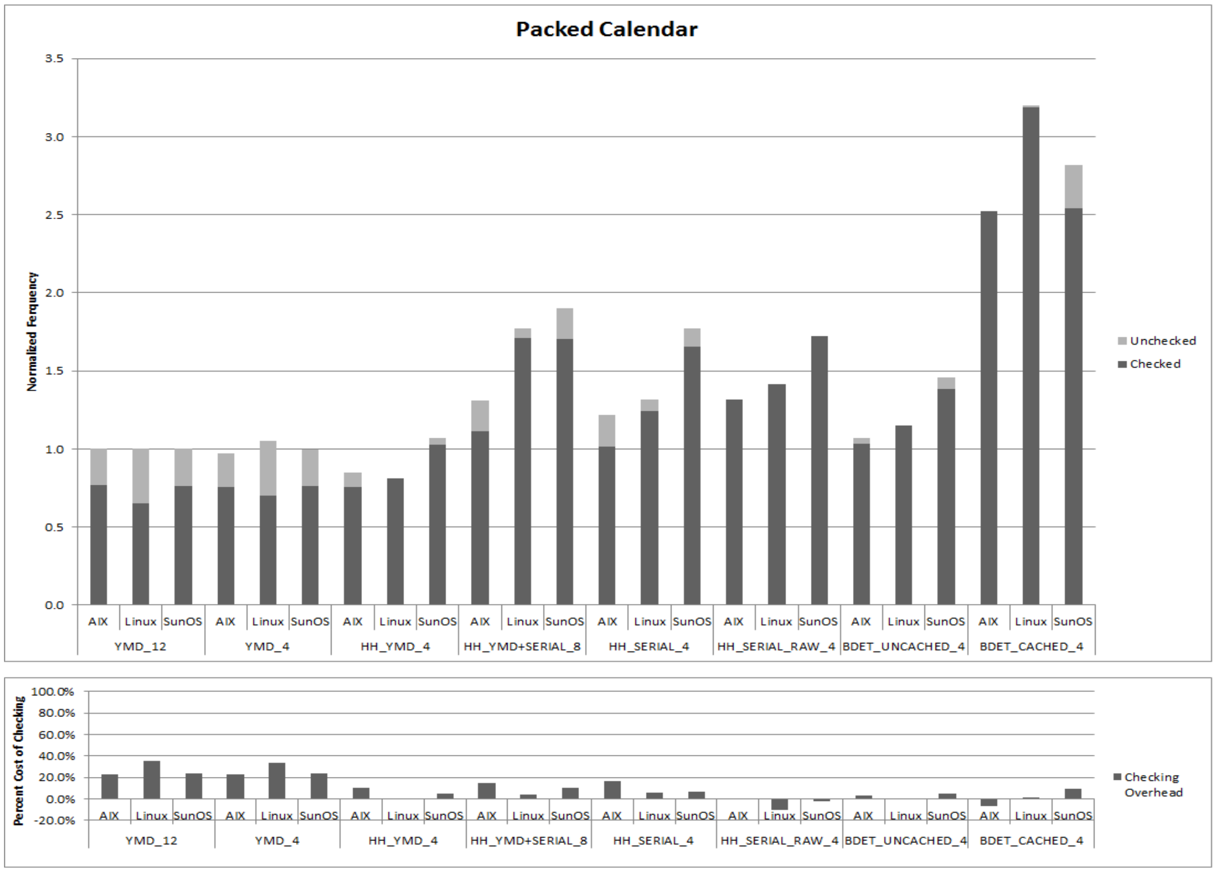

Figure 5.7: Packed Calendar Benchmark

Serial-based implementations are demonstrably quicker than purely field-based implementations, and the caching optimization for serial-based implementations offers additional benefit over the other serial-based implementations.

The lower graph shows a non-negligible cost for checking on most implementations.

Chapter 6

Analysis and Discussion

In the previous chapter, we presented results for the seven benchmark programs described in Chapter 3 for each of the 16 implementations of the `date` class discussed in Chapter 2 (the eight distinct implementations each having a checked and an unchecked version) on each of the three platforms described in Chapter 4.

In this chapter, we discuss general trends among the 336 individual results and then focus our analysis on the most interesting benchmarks and implementations. In particular, we reduce the data by taking simple averages over the three platforms, dismiss several "uninteresting" and otherwise unremarkable implementations, and introduce a single "Figure of Merit" (FOM) that is an equally-weighted average of the four "most interesting and representative" benchmarks (based on our technical and business experience).

6.1 Overview

All results given in this chapter are averaged over the three platforms described in Chapter 4. There are several interesting trends among platforms that can be seen in the data in Chapter 5 (e.g., in some cases, Linux has markedly lower cost for its checked implementations), but a discussion of such platform-specific results is outside the scope of this report.

For reasons that will be discussed explicitly in section 6.5, all comparisons are among *unchecked* implementations unless explicitly stated otherwise.

As will be discussed in the individual sections below, there are few surprises in the data analysis.

- Serial-based implementations have significant performance advantages for some operations and pay performance penalties for other operations.
- A cached serial-based implementation is dramatically faster than the corresponding uncached one on all operations but sorting.
- The hybrid serial-plus-fields-based implementation offers performance that is typically intermediate between pure serial and pure field representations.
- Input checking has a cost, typically on the order of 10-30%, which some might consider "modest" and others might consider "expensive".
- Input checking is relatively more expensive for higher-performance implementations.
- A particular weighted average of the benchmark chosen to represent use cases typical of our business operations strongly favors (by about 80%) the cached serial-based implementation.

6.2 Benchmarks and a Figure of Merit

Our set of seven benchmarks can be reasonably divided into four "primitive" benchmarks (Accessors, Creators, Increase Day, and Increase Month), and three "composite" benchmarks (Array Sort, Modified Following, and Packed Calendar).

We have chosen the three composite benchmarks as well as increase-month to serve as the basis for the "Figure of Merit" (FOM). We feel these benchmarks fairly represent typical use cases in our financial-data business.

$$FOM = \frac{F_{ModifiedFollowing} + F_{ArraySort} + F_{PackedCalendar} + F_{IncreaseMonth}}{4} \tag{6.1}$$

where F_i is the platform-averaged frequency result for the i^{th} benchmark, and the subscript names the benchmark.

We recognize that other industries and application areas may have very different use cases, but to presage a key recommendation, our point in advancing this FOM is to advocate that the Standard interface not put any one implementation or class of implementations at a disadvantage.

The FOM results for the unchecked versions of the various implementations are:

YMD-12	YMD-4	HH-YMD-4	HH-S+F	HH-SER	HH-RAW	BDET-Un	BDET-Ca
1.00	0.98	0.76	0.96	0.89	0.97	0.92	1.57

For the checked versions (with values still normalized to the unchecked `YMD_12` implementation) the results are:

YMD-12	YMD-4	HH-YMD-4	HH-S+F	HH-SER	HH-RAW	BDET-Un	BDET-Ca
0.54	0.73	0.73	0.88	0.85	0.96	0.74	1.37

Comparing the unchecked and checked FOM values allows us to estimate the cost of checking for each implementation:

YMD-12	YMD-4	HH-YMD-4	HH-S+F	HH-SER	HH-RAW	BDET-Un	BDET-Ca
46%	26%	5%	8%	5%	1%	20%	13%

6.3 Serial-Based vs. Field-Based Implementations

The most significant differences in performance occur when comparing the best field-based implementation to the best serial-based implementation. Not surprisingly, each has both dramatic advantages and dramatic disadvantages, depending on the benchmark. For example, the (cached) serial-based implementation has a tenfold *advantage* over the field-based implementation for the "Increase Day" benchmark, but a fivefold *disadvantage* for the "Accessors" benchmark. Averaged over both the four "primitive" benchmarks (Accessors, Creators, Increase Day, and Increase Month) and the FOM of section 6.2, the cached, serial-based implementation has about a twofold advantage.

6.4 The Benefits of Caching for Serial-Based Implementations

We have not explored the use of caching for field-based implementations, but are hard pressed to think of an obvious advantage that can be obtained from caching for field-based implementations. Caching for the serial-based implementations, however, has a profound effect on performance. The performance for Increase Day and Array Sort is substantially the same for the cached and uncached implementations, since neither benchmark makes use of the cache. For all other benchmarks, the cached implementation runs 2-6 times faster. In particular, the cached implementation is about 2.5 times faster when averaged over our FOM benchmarks. The main point is that caching offers a significant benefit for serial-based implementations.

6.5 The Cost of Checking

We recognize that there are many strong views on the relative merits of having individual classes validate all state-related input; in this section we simply show that checking has a cost that is burdensome for at least some users, even while it might seem modest and be gladly paid by others.

In the most extreme cases, which are inherently efficient operations where the compiler can't always optimize away the cost of checking, checked implementations can run two to four times slower (or, have only 25-50% of the performance of the unchecked operation). Averaged over all benchmarks and implementation strategies in this report, the unchecked interface is about 36% faster than its checked counterpart. A specific result of interest for our own business needs is that checking imposes a 40% performance penalty for the `BDET_CACHED_4` implementation for the Modified Following benchmark. In our experience, performance costs of this general magnitude have caused many applications developers to abandon "standard" types in favor of handwritten code — a situation that we seek to avoid.

6.6 An Observation on the Underlying Calendar

There is an interesting observation to be made when comparing the `HH_SERIAL_RAW_4` and `BDET_UNCACHED_4` implementations. Howard Hinnant's implementation is the clear winner – by 10-20%. We attribute the success of Hinnant's implementation to two distinct advantages: the use of the more regular proleptic Gregorian calendar, and a more efficient basic engine for converting to and from the serial representation. Although we have not yet merged the two implementations, we are confident that Howard Hinnant's approach, when combined with the cache, will add another 10% to the performance advantage of the cached serial-based implementation discussed in section 6.3 for operations outside the range of static cache.

Chapter 7

Conclusions

1. There are at least two very different approaches to implementing a date class: One involves storing the 3 date fields separately as year, month, and day; the other employes a single integer offset from an arbitrary epoch date.

    - Any interface we choose should admit either of these basic implementations.

2. Each of these two basic implementation approaches affords advantages for certain operations. Field-based implementations are typically faster for operations using the individual fields, such as constructing and setting values, querying individual fields, etc. Serial-based implementations, on the other hand, tend to favor operations on days, such as adding or subtracting a specific number of days, or operations using whole date values, such as finding the number of days between two dates, ordering two dates, etc. Serial-based implementations can also make better use of available storage to support a wider range of years.

    - There is no single implementation that provides optimal performance in all scenarios.

3. Maintaining a static cache for converting between field and serial representations can have a significant impact, making the runtime performance of serial-based implementations behave comparably to field-based implementations – even for operations in which field-based implementations excel. By contrast, when serial-based implementations excel, the performance increase can be by an order of magnitude.[1]

    - Static caches significantly enhance overall performance of serial-based date implementations.

4. Maintaining both representations, as in `HH_YMD_SERIAL_8`, appears beneficial for workloads dominated by read operations, as the most performant representation is always available. However, doubling the footprint is likely to be an unacceptable cost for workloads that process large volumes of data using dates.

   - Having both field and serial representations might perform well in specific circumstances, but the larger footprint is impractical for a general-purpose vocabulary type.

5. Incorporating additional information in bit fields is also counter-indicated from a performance perspective. There are, however, even more compelling reasons to avoid embedding "meta data" (i.e., relevant state other than that used to represent the valid value of a date) within the date object: Value semantics, the meaning of value, and the notion of substitutability of date objects would all be compromised.[2]

   - A date object should represent a valid date value, nothing more.

6. The proleptic Gregorian calendar, supported by Howard Hinnant's date types, provides a simpler contract than the Gregorian calendar (consistent with the Unix `cal` function) currently supported by Bloomberg's date class. The modified version of Hinnant's serial-based implementation, `HH_SERIAL_4`, is about 10% faster than the uncached version of Bloomberg's implementation, `BDET_UNCACHED_4`.

   - Adopting the proleptic Gregorian calendar admits the fastest implementations.

7. Finally, we assume that every `date` object always represents a valid date value. The runtime cost of checking preconditions varies depending on the interface, operation, and platform, but is generally significant, on the order of 10-30%. Any interface that mandates runtime checking for validity, and provides for specific remedies should an invalid value be supplied could be prohibitively expensive for operations such as the prefix increment operator (`operator++`) that (in our experience) would be unlikely to overflow or underflow the valid contiguous range of date values in practice.

[1]Note that static caching is unlikely to provide any performance boost for field-based implementations, as the individual field values are already available directly within the date object, and common operations are more efficiently implemented without the overhead of bi-directional lookup, leaving any remaining operations prone to L1 and L2 cache misses.

[2]Special flags to indicate, for example, that the date is the last day of the month in order to support month operations – in our view – have no place in a standard date class. Such inherently non-primitive operations are far better located at a higher level, as functions in a separate utility component.

Chapter 8

Recommendations

Based on what we now know, we propose the following specific recommendations for whatever standard date class is adopted.

1. Any date facility adopted for a C++ standard should allow for a maximally-efficient unchecked interface, to avoid the risk of requiring performance-constrained customers to create additional date types, fragmenting our vocabulary. Such an interface would be *narrow*, where users are responsible for providing valid date values, or else they get undefined behavior.

2. Use the proleptic Gregorian calendar as the basis for our standard date class.

   - It is the simplest and most general.
   - It admits the fastest implementations.

3. Provide a basic value-semantic `date` class to represent dates, having a minimal, but complete, primitive interface that admits either a field-based or a serial-based implementation (with or without cache), that does not specify what happens when an invalid date value is passed in (narrow contracts).

4. The definition of value is entirely specified by the salient attributes *year*, *month*, and *day*, as defined by the equality comparison `operator==`, which is the necessary and sufficient post condition for both copy construction and assignment: No other state is stored within the date that contributes to this value.

5. The `date` type should satisfy the following library requirements:

- `DefaultConstructible`
- `CopyConstructible`
- `CopyAssignable`
- `Destructible`
- `Swappable`
- `EqualityComparable`
- `LessThanComparable`

6. `std::hash` should supply a suitable overload.

7. Make an invariant of the `date` type that each `date` object represents a valid date value according to the proleptic Gregorian calendar. Checking a value is done at most once (on input), as opposed to each time it is used.

8. A primitive interface would further supply three basic accessors, `year`, `month` and `day`, and a combined setter, `set_yearmonthday`. We note that individual setters would have complex contracts, making it difficult to change `date` values without transitioning through invalid states.

9. To avoid penalizing serial-based implementations, add an additional accessor, `void get_yearmonthday(int *, int *, int *)` [1].

10. We should provide methods that check for validity:

```
    static bool is_valid_yearmonthday(int year, int month, int day);
    int set_yearmonthday_if_valid(int year, int month, int day);
```

11. Any additional "checked" functionality that is specified must be entirely redundant, and must not increase the footprint, nor degrade the performance of any of the core functionality described above for either a field-based or serial-based implementation.

12. Any functionality having to do with advancing the month or year of a `date` is implemented in a higher-level component, separate from `date`. In particular, what to do in the event that advancing the month leads to a target day that is invalid is specified as an argument to a utility

[1] Note that while Bloomberg convention is always to return multiple values from functions through pointers, an alternative declaration might be: `tuple<int, int, int> date::get_yearmonthday() const;` at this stage, the important aspect is that a function returning all three values exists, and not how we spell it. We leave that bicycle-shed discussion for the final proposal.

function, and specifically is not part of meta data stored within the `date` object — whether or not that data is considered to contribute to value as defined by `operator==`.

Appendix A

For illustration, we show the class definition for a minimal `date` type that follows the recommendations in this paper.

```
class date {
  public:
    constexpr date() noexcept;
    date(const date&) = default;
    date(date&&) = default;
    constexpr date(int year, int month, int day);

    date& operator=(const date&) = default;
    date& operator=(date&&) = default;

    date& operator+=(int days);
    date& operator-=(int days);
    date& operator++();
    date& operator--();

    void set_yearmonthday(int year, int month, int day);
    int set_yearmonthday_ifvalid(int year, int month, int day) noexcept;

    void get_yearmonthday(int *year, int *month, int *day) const;
    constexpr int year() noexcept;
    constexpr int month() noexcept;
    constexpr int day() noexcept;

    static constexpr bool is_valid_yearmonthday(
                              int year, int month, int day) noexcept;
};
```

```
constexpr bool operator==(const date&, const date&) noexcept;
constexpr bool operator!=(const date&, const date&) noexcept;
constexpr bool operator<(const date&, const date&) noexcept;
constexpr bool operator<=(const date&, const date&) noexcept;
constexpr bool operator>(const date&, const date&) noexcept;
constexpr bool operator>=(const date&, const date&) noexcept;

date operator++(date&, int);
date operator--(date&, int);

constexpr date operator+(int days, const date&);
constexpr date operator+(const date&, int days);
constexpr date operator-(const date&, int days);
constexpr int operator-(const date&, const date&);

ostream& operator<<(ostream& stream, const date&);

template<>
struct hash<date> {
    size_t operator()(const date&) const noexcept;
};
```