A Study in Memcmp*

Richard T. Saunders
Rincon Research Corporation
IRAD Division
Tucson, Arizona 85711 USA
rts@rincon.com

ABSTRACT

The memcmp function is a pivotal function that comes as part of the C/C++ library suite; The default implementation, however, can be surprisingly slow on many platforms. The Python Programming language uses the memcmp function in several deep function calls (most notably in string and dictionary processing) and would benefit greatly from a faster implementation of memcmp. We survey the Python codebase to discover where and how mempcmp is used, and investigate different implementations of memcmp from glibc and other platforms. Using the specialization optimization and the granularity optimization, we introduce a simple, portable version of memcmp that can be 3x faster than the standard memcmp implementation. For completeness, we compare many different memcmp implementations on many different platforms. Plugging these changes into the Python codebase, We conclude that even simple memcmp related changes to the CPython baseline can boost Python's performance.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Scalability

General Terms

Systems Programming, Python

Keywords

Python, SSE, MMX, granularity

1. INTRODUCTION

The memcpy and memcmp functions are two standard functions out of the C/C++ standard libraries that most systems programmers rely on heavily. The memcpy routine has been heavily optimized on most platforms: it is a workhorse of systems programmers. Surprisingly, its brother function, memcmp, *seems* to have been given much less attention. We present benchmarks demostrating the lagging performance later in the paper. Although not used as frequently as memcpy, memcmp is an important tool to the systems programmer and we present a portable memcmp that can be up to three times faster.

This paper explores the usage of memcmp in the Python Programming Language: how it used, how often it is used, how it affects performance, how memcmp is compiled into the baseline. Although this paper has a Python-centric view, the lessons of a portable memcmp are applicable to other languages and system libraries.

Although this may seem like very specialized and perhaps esoteric study of a single function, it was discovered that the default memcmp was surprisingly slow during optimization attempts of dictionaries and pickling. A simple change to memcmp made a significant performance increase to some specific test cases. The purpose of the study is to explore the impact of simple memcmp changes, with an eye towards the Python baseline.

The purpose of this paper is to present some findings to the Python community: the hope is these will lead to some consensus. In the name of science, the testcases are all available so other users may reproduce and/or confirm the results on other platforms.
Python dev list 2011 Tucson, Arizona
With apologies to Sir Arthur Conan Doyle.

2. USAGE WITHIN PYTHON

To discover more about how Python uses the memcmp, we surveyed the C baseline of Python: version 2.7 of CPython, downloaded from the Python repository in Sept. 2011. We found all places memcmp gets called: there were 30 total. Figure 1 lists all the references throughout the baseline. There are two things we note in the analysis: (1) whether the memcmp call is a *deep* or *simple* call and (2) whether the memcmp call is an *equality check* (checking for either `==0` or `!=0`) or *generic* call.

Note that this is a static analysis of the baseline: we aren't counting how many times memcmp gets called by actual code, because that depends tremendously on the code. A numeric simulation may never call memcmp more than a handful of times (as it does floating point computation), but a string-processing Python program may call memcmp millions of times (as it compares strings): it depends on the testcase.

2.1 Deep vs. Simple

We want to look at how memcmp is used: A few of the calls are just used for trivial tests during configure or test time: their uses are *simple* and will never impact a real user as they will never really get called more than a few times by a real program. For any objects or modules (string, buffer, memory, stdop), the use of memcmp is potentially in a *deep* loop: by this, we mean that the memcmp may be the hotspot of a program where that Object/Module is used significantly. The obvious example are the string libraries and unicode

```
    configure scripts:  3 simple uses: ==0
    Include/unicodeobject.c:    1 deep use:    ==0
    Modules/_ctypes/libffi/testsuite/libffi.call/closure_loc_fn0.c: 1 simple use: ==0
    Modules/_ctypes/_ctypes.c: 1 simple use:  generic memcmp
    Modules/datetimemodule.c:  3 deep use:    generic, converts to bool
    Modules/_bsddb.c:          1 deep use:    generic, converts to normalized
    Modules/stropmodule.c:     5 deep uses:   ==0 (common string operations)
    Modules/zlib/deflate.c:    1 deep use:    ==0 (actually, !=0)
                /zutil.h:      1 "declaration" use: (doesn't count)

    Modules/expat/xmlparse.c:  1 deep use: !=0
    Objects/floatobject.c:     4 simple uses: ==0
    Objects/memoryobject.c:    1 deep use: ==0 (wrapped and converted)

    Objects/bytearrayobject.c: 1 deep use:  generic, converts via Py_LT, GT, etc.
                               1 deep use: ==0

    Objects/bufferobject.c:    1 deep use: generic, converts to normalized memcmp
    Python/dynload_win.c:      1 simple use: ==0

    Objects/stringobject.c:
                 string_richcompare:
                     1 deep use : optimized for ==0 case
                     1 deep use : generic case, converts via Py_LT, GT, normalize
                 _PyString_eq:
                     1 deep use: ==0 case
                 _string_tailmatch:
                     1 deep use: ==0 case
```

Figure 1: Uses of memcmp throughout the Python baseline

objects: if you are doing signicant string processing (unicode or plain), memcmp performance will affect you.

The number of deep calls appears to be 20 out of the 30 calls: so about 66.6% are potential deep loops. If these objects are used significantly, any memcmp optimizations will be significant.

This question of deep vs. simple is really a feasibility analysis: Would making memcmp faster have any performance impact? The answer: potentially. Where? In string, unicode, datatime, memoryobject, byteobject, zlib, and buffer calls. Only systems programmers would care about the latter objects, but many Python programmers would care about string (which includes the strop module) and unicode operations being faster.

2.2 Traditional vs. Normalized

The memcmp function has a potentially non-intuitive interface: most C/C++ programmers recognize 0 as false and "non-zero" as true. However, memcmp counter-intuitively returns 0 if two regions of memory are equal.

```
/* memcmp: return <0 if s<t,0 if s==t,>0 if s>t */
int memcmp (const void* src1, const void* src2,
            size_t length);
```

The idea, of course, is that memcmp returns a negative number if `src1<src2`, 0 if `src1==src2` and a postive number if `src1>src2`, thereby *ordering* memory regions. If the value is negative or positive, the return value is the difference of the first two characters that don't match. At least, this is the *traditional* implementation described by K&R C:

```
int memcmp(cs,ct,n)
  compares the first characters of cs with ct,
  return as with strcmp
```

This implies an implementation very similar to the traditional implementation of strcmp,[1], found in K&R, second edition (p. 106):

```
/* strcmp: return <0 if s<t,0 if s==t,>0 if s>t */
int strcmp(char *s, char *t)
{
  for ( ; *s==*t; s++, t++) {
      if (*s == '\0')
          return 0;
   return *s - *t;  /* returns the diff */
}
```

However, many recent implementations *normalize* the result to -1, 0, and +1. In more recent years, more memcmps support the normalized version. The Python baseline doesn't appear to assume either (for the most part: the zlib library makes an interesting assumption that the result of (something==somethingelse) is a particular value)) but tends to immediately normalize results of memcmp to either -1, 0, +1 or to `Py_LT`, `Py_GT`, etc.

The notion of whether memcmp uses the traditional vs. normalized interface doesn't affect our analysis too much here (the Python baseline goes out of the way to avoid assuming either a traditional and normalized versions of memcmp), but other codebases should be aware of the difference.

[1]A discussion of memcmp's brother *strcmp* in the Python baseline is included in the Appendix

2.3 Equality vs. Ordered

Although memcmp supports the notion of memory regions being less-than or greater-than, most use cases *only* compare region of memory for equality or inequality: they don't care about the ordering. Most calls from the Python baseline look like some variation of:

```
if (memcmp(a,b,len)==0) /* strings are equal */
if (!memcmp(a,b,len))   /* strings are equal */
if (memcmp(a,b,len))    /* strings are not equal */
```

By our static analysis:

- 22/30 : 73.3% of memcmps only check for equality or inequality
- 8/30 : 26.6% of memcmps use generic test

One could argue these percentages are representative of a "typical" program and that about 73.3% of memcmp calls only check equality.

Another perspective is that we should really only count the deep calls (the memcmps that could potentially be in deep loops): Recall there are 20 deep memcmp calls, and of those:

- 13/20: 65% check for equality or inequality
- 7/20: 35% use generic comparison test

So 13/30=43.3% of the calls are deep, equality-only testing calls.

Depending on how you look at it, either 73.3% (best case) or 43.3% (worst case) of the time, we only test for equality. What this means is that about half of time, we are carrying the entire mechanism of memcmp (i.e., allowing full ordered comparisions) when all we really need is the equality notion. Said another way, sometimes we compute too much when we just want the simple notion of equality (not the more computational intense notion of ordering).

3. OPTIMIZATION

We discuss two major optimizations for a portable, fast memcmp: specializing memcmp code for equality testing and changing the granularity of items compared.

3.1 The Specialization Optimization

The specialization optimization is simply noticing that most uses of memcmp are for equality. Unfortunately, memcmp itself can't differentiate between equality checking and order checking : that is done at the call site, not inside memcmp.

```
/* memcmp can't notice that the call ONLY
/* checks for equality */
if (memcmp(a,b,len)==0) {
}
```

So, we have to split the memcmp into a family of two functions: *fast_memeq* and *fast_memcmp*: The first (*fast_memeq*) is specialized solely for equality checking and the second (*fast_memcmp*) is the more general memcmp (with the granularity optimization, see below).

The *fast_memeq* interface (below) preserves the non-intuitive memcmp interface (0 for equality, 1 for inequality) to enable direct substitution of *fast_memeq* for memcmp.

```
/* Returns 0 if the regions are equal
**  non-zero if they are not equal */
int fast_memeq (const void* src1,
                const void* src2,
                size_t len);
```

With a simple macro, we can go back and forth between the standard implementation and an alternative implementation:

```
#if defined(SYSTEM_MEMCMP_IS_FAST)
#  define fast_memeq memcmp
#  define fast_memcmp memcmp
#endif
```

Inside the implementation of *fast_memeq*, we can completely ignore the notion of ordering and just concentrate on checking if two memory regions are equal or not. We'll see the implementation in a second, after we introduce the granularity optimization.

3.2 The Granularity Optimization

The basis for the granularity optimization is very simple: the traditional *memcmp* compares memory regions byte-by-byte, but computers can operate on much larger quantities of memory at the same speed. Typical modern computers have 32-bit, 64-bit or even larger datapaths to memory; that means they they can grab memory from the bus in one large chunk (32-bytes) at about the same speed as 1 byte (8 bits). Rather than operate on one-byte at a time, its just as fast to operate on 4 (or even 8 bytes) at a time.

For *fast_memeq*, we don't want to compare one byte at a time, we want to compare (for a 32-bit machine) 4 bytes at a time. This simple piece of code of Figure 2 is the reason for the whole paper: we can tweak it slightly and get slightly better performance, but in the pristine state above, it is simple and it works surprisingly well. The simplicity allows the optimizer more opportunities: we have tried hand-unrolling the major loop, the optimizers (at least for modern optimizers) seem to do as good or better.

For *fast_memcmp*, we use the same of idea, but we have to be more careful: when things don't compare as equal, we have to take the last few bytes and decide the ordering. See Figure 8. Some interesting notes: On a big-endian machine, we can use the integer compare to get the immediate sense of order, rather than break out the bytes one at time: that's because on a big-endian machine, the upper bytes of an `uint_64` dominate the quantity and cause an int compare to be ordered in the same way a byte-by-byte compare would proceed. This means that the major loop can immediately return if there's an order issue. On a little-endian machine, when the major loop fails, we have to back-up 8 bytes, and run the final minor loop as if it were the last 8 bytes. There may better ways to do the last few bytes (some unravelling or bit-shifting) for little-endian code, but that breaks the simple mantra of this code.

4. TESTING

To see how well the *fast_memeq*/*fast_memcmp* suite works, we had to test across multiple platforms.

The platforms:

1. *RH6, 32-bit*: A RedHat 6.0 machine, 32-bit OS with an Intel Core 2 6600 at 2.4GHz with SSE/SSE2/SSE3. Two CPUs. Uses glibc 2.12 and gcc 4.4.5

```
#include <stdint.h>   /* So we have sized integers */

int fast_memeq (const void* src1, const void* src2, int len)
{
  /* simple optimization */
  if (src1==src2) return 0;

  /* Convert char pointers to 4 byte integers */
  int32_t *src1_as_int = (int32_t*)src1;
  int32_t *src2_as_int = (int32_t*)src2;

  int major_passes = len>>2;  /* Number of passes at 4 bytes at a time */
  int minor_passes = len&0x3; /* last 0..3 bytes leftover at the end */
  for (int ii=0; ii<major_passes; ii++) {
   if (*src1_as_int++ != *src2_as_int++) return 1; /* compare as ints */
  }

  /* Handle last few bytes, but has to be as characters */
  char* src1_as_char = (char*)src1_as_int;
  char* src2_as_char = (char*)src2_as_int;
  switch (minor_pass) {
  case 3: if (*src1_as_char++ != *src2_as_char++) return 1;
  case 2: if (*src1_as_char++ != *src2_as_char++) return 1;
  case 1: if (*src1_as_char++ != *src2_as_char++) return 1;
  }

  /* If we make it here, all compares succeeded */
  return 0;
}
```

Figure 2: Basic code illustrating ideas of a fast memcmp

2. *FC15, 32-bit*: A Fedora Core 15 machine, 32-bit OS with an Intel Xeon E3503 at 2.4GHz with SSE/SSE2 and SSE3 and SSE4. Two CPUs. Uses glibc 2.14 and gcc 4.6.1
3. *FC14, 64-bit*: A Fedora Code 14 machine, with a 64-bit OS with an Intel Xeon CPU W3520 at 2.67GHz with SSE/SSE2/SSE3/SSE4_1/SSE4_2. Eight CPUs. Uses glibc 2.13 and gcc 4.5.1
4. *Ubuntu, 32-bit* An Ubuntu 11.04 desktop machine with 32-bit OS with an Intel Penium 4 at 3GHz with SSE. One CPU. Uses glibc.
5. *SunOS 5.6, 32-bit* A SunOS machine, 32-bit OS with SPARC Ultra-4. Two CPUs. BIG ENDIAN machine.
6. *RH 5.0, 32-bit:* A RedHat 5.0 server machime, 32-bit OS with an older Intel Xeon CPU 3.2GHz with SSE/SSE2. Two CPUs. Uses glibc 2.5 and gcc 4.1.1
7. *Tru-64, 64-bit:* A Tru-64 machine, 64-bit OS (5.1.B) with Alpha EV6. Four CPUs. Uses own machine language memcmp and Tru64 C compiler.

Unless stated otherwise, the code is compiled with `-O2`, much like Python is with *configure*.

This list spans a list of very new machines (FC15 is actually quite new) to some very old (the Ubuntu machine is at least 6 years old and the Tru-64 machine is at least 10 years old). We wanted to make sure the *fast_memeq*/*fast_memcmp* suite works well across many different platforms.

The test, called *memtest* is written in C and it is used to time multiple implementations of memcmp with multiple options. The options are below:

1. **Implementation:** What implementation of memcmp do we use?
   (a) *memcmp:* Standard memcmp that comes with the system
   (b) *glibcmecmp:* This is the C version of the code (more discussion on this later) of memcmp from glibc 2.12. This has been grabbed from glibc and processed so its symbols won't interefere.
   (c) *fastmemcmp:* This is the fast version of the memcmp using the granularity optimization: this has the generic memcmp interface.
   (d) *fastmemeq:* This version is specialized for equality testing and uses the granularity optimization
   (e) *imemcmp:* This also is a call of memcmp, but called through a function pointer, rather than the hardcoded call to memcmp of above
2. **Size of Strings:** How big are the strings we compare?
   (a) *smallstrings:* Process strings only of length 1 to 8 bytes
   (b) *bigstrings:* Process strings of 8 to 80 bytes (increments of 8)
   (c) *allstrings:* Do both smallstrings and bigstrings (above)

```
  int fast_memcmp (const void* src1, const void* src2, size_t len)
  {
#if defined(INTS_NEED_ALIGNED)
    /* Check to make sure both addresses are aligned:  convert pointer
    ** to an appropriate sized integer (from stdint.h) and do quick
    ** alignment checking */
    uintptr_t ptr1 = (intptr_t)src1;
    uintptr_t ptr2 = (intptr_t)src2;
    if ( (ptr1 | ptr2) & 0x3) {
       /* Nope: on one of the two pointers, some bottom bit was non-zero,
          meaning one of the pointers is not aligned:  SIGH: defer to default */
       return memcmp(src1, src2, len);
    }
#endif
    /* ... rest of code ... */
}
```

Figure 3: Conditionally added alignment checks at top of fast memeq

3. **Comparison Type:** How do we compare strings?
   (a) *equal:* The strings memcmp uses are both equal (but at different addresses) so that memcmp has to do a full compare. The **Size of Strings** determines how big these strings are.
   (b) *different:* The strings memcmp uses are different in the last byte: memcmp still has to do a full compare to get to the end, but coupled with the different sizes of strings, this tests memcmp at many different places.
4. **Alignment of Strings:** How are the strings being compared aligned?
   (a) *alignedstrings:* The strings are allocated with malloc and aligned on a 4 or 8 byte boundary (whatever malloc for that platform guarantees)
   (b) *unalignedstrings:* The strings are referenced at offsets of 0 to 4 to simulate different offsets.
5. **Granularity:** At what granularity of integers do we compare strings? I.e., what are the number of bytes compared at a time? *This only applies to fastmemeq and fastmemcmp.*
   (a) *1:* Compares are done byte-by-byte, like a naive version of memcmp
   (b) *2:* Compares 2 bytes at a time using `uint16_t`
   (c) *4:* Compares 4 bytes at a time using `uint32_t`
   (d) *8:* Compares 8 bytes at a time using `uint64_t`

The test suite is relatively broad so that we are hitting all different types of strings: short, big, aligned, unaligned, different and equal.

Take a look at Figure 7 (it's at the end of the paper): This has all the timing numbers for all the different options and platforms. The chart is presented so the platforms are on the left axis and the different options are on the top axis. The options are discussed below (although we can differentiate between small and large strings, we only list the all-strings test cases in Figure 7 as it seems to be representative enough).

4.1 Alignment Issues

4.1.1 Correctness

Alignment of integers may be an issue on some platforms. On some platforms, if an integer is NOT at an address of memory that is a multiple of 4 or 8, attempting to access an integer through a non-aligned pointer will result in a alignment/segmentation fault causing the program to halt. Two older architectures that have this restriction are SPARC and ALPHA. Certain SSE instructions on Intel platforms have even stricter alignments (16 byte boundaries).

The granularity optimization won't work as well on those architectures because we can't always guarantee that strings will be aligned on 4-byte boundaries (especially for substrings). We can, however, add a simple check for alignment and defer to the original memcmp for those: See Figure 3. Many strings in Python will be aligned: see discussion below.

This alignment restriction of a platform can actually be caught with *configure* (a tool used on Linux platforms to configure and build Python): inside of *configure*, we can detect the restrictions of the architecture with a small test, and simply define or not define INTS_NEED_ALIGNED properly. This is something that Python can do to produce a generic baseline (as *configure* is the canonical way Python is built on most Linux platforms). Not all projects use configure and may have to come up with a more ad-hoc way of determining this.

It's interesting to note that most strings in Python will be aligned: the `PyObject` for strings looks something like Figure 4. When a *PyStringObject* is allocated, the `&ob_sval[0]` will almost certainally be aligned to some 4 or 8 byte boundary. Again, substrings (either from external libraries using C-style strings that aren't allocated on boundaries or internal string buffers) may be an issue. A large majority of Python strings will be allocated on proper boundaries, as both strings and string slices are allocated on proper boundaries.

4.1.2 Performance

Although some processors can access integers on integer boundaries, there can still be performance problems. Histor-

```
typedef struct {
    PyObject_VAR_HEAD
    long ob_shash;
    int ob_sstate;
    char ob_sval[1];

/* Invariants:
*     ob_sval contains space for 'ob_size+1' elements.
*     ob_sval[ob_size] == 0.
*     ob_shash is the hash of the string or -1 if not computed yet.
*     ob_sstate != 0 iff the string object is in stringobject.c's
*       'interned' dictionary; in this case the two references
*       from 'interned' to this object are *not counted* in ob_refcnt.
*/
} PyStringObject;
```

Figure 4: Python PyStringObject Definition

ically, processor documentation cliams that such operations can cause memory accesses to stall and take significantly more time. It is interesting, at least on modern Intel processors, that this doesn't seem to be as big a problem as it used to be. Basically, caching, prefetching and better hardware tends to make this a non-issue.

Modern processors fill cache-lines from memory in 32, 64, or even 128 byte chunks: Initiating a memory operation for an unaligned 4 bytes tends to fill the cache anyway, so it just happens to grab the 64 or so bytes around it as part of the memory operation. Thus, when the integer access occurs, the memory is in cache and doesn't suffer a performance penalty. The tests from Figure 7 (at least for Intel), tends to confirm this. Notice that the unaligned strings performance is on par with the aligned strings on most newer Intel platforms: the major exception is the older P4 (Ununtu) platform. Performance is reduced on that platform, but not significantly enough to be a detracter.

The unaligned memory cost seems much less a burden on today's cache-heavy architectures.

4.2 Comparison with glibc

As part of our testing, we extract (from glibc 2.12) the portable C version of memcmp (note that glibc also has some specialized assembly for different platforms, see below) and call it *glibmemcmp* in our testing. This version of glibc unrolls loops, deals with disparate alignments agressively, and in general has many different codepaths. The glibcmemcmp code does also use the granularity optimization: It uses the many same ideas as our *fast_memcmp*, yet surprisingly, it doesn't do nearly as well on modern architectures, but it does better on older architectures (SPARC, Alpha). The conjecture here is that alignment issues really do dominate running time on older platforms, and this is why glibcmemcmp performs very well there.

On the other hand, the relative simplicity of the suite *fast_memeq/fast_memcmp* makes it easier to optimize than the rather complex glibc memcmp. The glibmemcmp code is very large due to manual loop unrolling and alignment sensitive code. The alignment issues, which tended to be much more problematic historically, are much less so in current processors (see the section on alignment), so all the extra code that the glibcmemcmp carries seems to be excessive on modern CPUs.

4.3 Granularity

For the *fast_memeq/fast_memcmp* suite, the numbers suggest what we expect: the larger the granularity (i.e., the larger the int we use to compare), the better the performance. On 32-bit platforms, the granularity of 4 tends to be the best performer, although in a few cases, the granularity of 8 is slightly better (this seems to simply be a function of the hardware: if the hardware supports larger ints, even in the face of a 32-bit OS, the granularity of 8 may work better). The granularity of 8 always seems to be the best choice for 64-bit OS's.

A simple way to capture whether the OS is 32-bit or 64-bit is to use `size_t` as the integer inside the major loop instead of `uint32_t` on 32-bit machines or `uint64_t` on 64-bit machines: `size_t` is the size of the largest well-supported integer (arguably, so is `long`). The *fast_memeq/fast_memcmp* suite can simply use `size_t` so it doesn't need to check what kind of machine its own. This leads to code like Figure 5.

Note that we don't include the granularity of 2 in Figure 7: in every case, the granularity of 2 was faster than a granularity of 1 and the granularity of 4 was faster than 2: it was exactly what we expected, and it added no new information.

4.4 Indirect memcmp: Surprising Results on Intel

Inside the testing suite (called *memtest*) C code, the calls to *memcmp* and *glibmemcmp* are hardcoded, direct calls. The C calls to *fast_memeq* and *fast_memcmp* go through a function pointer (for ease: we have to call either the 1,2,4, or 8 byte granularity code). All the numbers from the test case up to this point use the hardcoded memcmp.

A fairness argument dictates that the *memcmp* and *glibmemcmp* should be called through a function pointer (even though the code in the Python baseline calls *memcmp* directly) so that all implementations are called the same way: we want the implementation comparisons to be apples-to-apples. Doing that, a very surprising thing happens on most Intel platforms: *the system imemcmp becomes as fast, if not faster, than fast_memeq!* This is surprising because usually calling a function through a function pointer is just a little

```
#include <stdint.h>   /* So we have sized integers */

int fast_memeq (const void* src1, const void* src2, int len)
{
  /* simple optimization */
  if (src1==src2) return 1;

  /* Convert char pointers to largest integers */
  size_t *src1_as_int = (size_t*)src1;
  size_t *src2_as_int = (size_t*)src2;

  int major_passes = len/sizeof(size_t);  /* major passes at large granularity */
  int minor_passes = len & (sizeof(size_t)-1); /* minor pass with leftover */
  for (int ii=0; ii<major_passes; ii++) {
   if (*src1_as_int++ != *src2_as_int++) return 0; /* compare as ints */
  }

  /* Handle last few bytes, but has to be as characters */
  char* src1_as_char = (char*)src1_as_int;
  char* src2_as_char = (char*)src2_as_int;
  for (int ii=0; ii<minor_passes; ii++) {
    if (*src1_as_char++ != *src2_as_char++) return 0;
  }

  /* If we make it here, all compares succeeded */
  return 1;
}
```

Figure 5: Generic fast_memeq works for both 32-bit or 64-bit machine at the best granularity

slower (as there is one extra indirection). Why would calling *memcmp* through a function pointer speed it up by 2x? In most non-Intel cases, it is slightly more expensive to call through a function pointer. Take a look at the numbers in Figure 7.

The answer lies in the assembly. Taking a look at the x86 assembly code reveals the issue: Compiling with any optimization in gcc *compiles away* the call to memcmp and generates a very, very small machine language stub as a memcmp replacement. See Figure 6

The idea is that gcc optimizes away the call to memcmp to avoid the function call overhead by executing the single instruction: `rep cmpsb`. Strictly speaking, this is one instruction in memory (reducing instruction cache pressure), but multiple `cmpsb` instructions in the instruction stream as the instruction is repeated. This small, concise piece of code seems optimal. Unfortunately, the granularity optimization seems to trump this, as evidenced in two ways: First, the *fast_memeq/fast_memcmp* suite seems to outperform this code by 2x-3x. Second, the hardcoded glibc assembly (which is much more complex) seems to outperform this as well.

By jumping through an indirect function, we disable the compiler optimizing away the call to memcmp. In this case, the memcmp it links in is an assembly language stub (at least for sparc, intel, and powerpc). The later versions of glibc (2.12, 2.13, 2.14) have a piece of Intel assembly code for Intel platforms that's further optimized for SSE3 and SSE4 Intel architectures (the SSE instructions are op-codes that allow vector operations). They are called (cleverly enough) *memcmp.S, memcmp-sse3.S, and memcmp-sse4.S*: When you link against memcmp, glibc discovers what MMX instructions the CPU supports and links in the "optimal" version of memcmp, specialized for your CPU.

A reasonable question is how far does this gcc optimization extend? We tried this with Intel gcc 4.6.1 (sometimes, depending on how we called it or other volatile qualifiers), 4.5.1 and 4.4.5 and saw gcc do the same thing: elide the call to memcmp in favor of the `rep cmpsb`. It turns out there is a compiler option to disable this gcc optimization called `-fno-builtin-memcmp`.

Compiling with Intel 11.1 (or Intel 10.1.025) uses a specialized `_intel_fast_memcmp`, which doesn't suffer from the disabling memcmp: the performance is on par with the glibc assembly or *fast_memcmp*.

To summarize: if we directly call memcmp from your C code (and optimization is turned on on gcc), the compiler optimizes away the call to memcmp and replaces it with a tight machine language loop: `rep cmpsbb`. Unfortunately, this seems to be very slow. If we call memcmp through a function pointer, the compiler can't optimize away the call, and instead calls the machine language from glibc (which are optimized for the Intel processor of interest).

What should Python do to get a faster memcmp? Both the *assembly language* memcmp code in glibc and the suite of *fast_memeq/fast_memcmp* are about the same performance: There are some differences if the strings are aligned or not aligned, and/or if the strings are equal or different. In some cases, the portable code is faster, in some cases the glibc assembly is faster. In general, both suites are about 2x-3x faster than the gcc generated assembly.

We want to make the Python baseline faster: there seem

```
movq 0x8(%rsp), %rdx          # Set-up:
movl (%r14, %rbp,1), %ecx
movsxd %ebx, %rdi
mov %rdi, %rsi
addq (%r15, %rbp,2), %rdi
sub %ebx, %ecx                # Putting Length in cx register
movsxd %ecx, %rcx
cmp %rcx, %rcx                # Resets the zero-flag

# Memcmp loop right here (single instruction)
rep cmpsbb (%rdi), (%rsi)     # Repeats cmpsbb instruction
                              #  (1) until zero flag set
                              #  (2) decrements cx until cx=0
setz %al
movzx %al, %eax
movl %eax, 0x260(%rsp)
jmp 0x400e58 <block 44>       # Back to next iteration of the loop
```

Figure 6: GCC assembly stub replacing memcmp calls

to be three options.

1. Carry the *fast_memeq/fast_memcmp* suite as part of the Python baseline, and replace all calls of memcmp.
   (a) **Advantages:** Python owns the code and will get the performance increases on any platform, even if memcmp is poorly implemented. We also distinguish between equality calls and generic memcmp calls, which has its own benefits.
   (b) **Disadvantages** On platforms where memcmp is highly optimized, we may miss out on some extra speed. Alas, the fastmemcmp code is only fast when the baseline is compiled with optimization on.
2. Replace all calls of memcmp with a call through a function pointer.
   (a) **Advantages:** Very simple change and gets the extra performance right now and is portable.
   (b) **Disadvantages** May disable 'better gcc' optimizations in the future. This does seem a silly thing o do.
3. Change compiler flags to Python to disable that optimization.
   (a) **Advantages:** May fix it now, with no changes to the Python baseline. It is easy to do on gcc: `-fno-builtin-memcmp` causes this "feature" to go away.
   (b) **Disadvantages** Specific to gcc and may not be the right thing to do on some architectures.

Note that this is only majorly problem on the Intel gcc platforms, however, given the ubiquity of these platforms, it's probably worth fixing memcmp somehow *in the Python baseline*. One of the purposes of this paper is to discover what's happening and recommend some options, deferring to the Python community to decide on the right thing. The easiest thing to do for the Python community may just be to turn on the `-fno-builtin-memcmp` flag.

The basic conclusion is the optimized handcoded memcmp libraries are good, if you can get to them (to be fair, the Tru-64 hand-assembled libraries trump the fast memcmp we propose hands down: See Figure 7). The portable *fast_memeq* is better than the optimized hardcoded libraries in some cases (usually when strings are different), and worse in other (usuallly when strings are equal). Unfortunately, the portable *fast_memeq/fast_memcmp* suffers at least a 2x hit if compiled without optimization. It is, however, portable.

5. INTEGRATION WITH PYTHON

In this section, we incorporate some changes described in the previous section into the Python baseline. For test one, we integrate the *fast_memeq/fast_memcmp* suite into Python. For the test two, we assert `-fno-builtin-memcmp`. After initial testing of the plain C code with its test suite and verifying that these optimizations are viable, we need to integrate it with Python. For test one, integration with Python is a simple matter:

1. replace memcmp call with *fast_memeq* call in the 22 equality calls
2. replace memcmp call with *fast_memcmp* in the remaining 8 "generic" calls
3. add a check to *configure* so we can detect platforms with alignment issues: this will set and define the C macro INTS_NEED_ALIGNED properly
4. run some performance tests to verify that we see the expected speed improvements

For test two (in Intel platforms), we simply turn on the compiler option `-fno-builtin-memcmp`.

As a baseline, we include the original raw Python codebase (compiled with the standard configure options with -O2 and -O3) as a baseline of comparison.

5.1 String Comparisons

This test simply shows off *fast_memeq/fast_memcmp*: this test does massive string comparisons and shows that, indeed,

we are faster in many cases to the original implementation of Python with the default memcmp.

The Python test is simple:

```
a = []; b = []; c = []; d = []
for x in xrange(0,1000) :
    a.append("the quick brown fox"+str(x))
    b.append("the wuick brown fox"+str(x))
    c.append("the quick brown fox"+str(x))
    d.append("the wuick brown fox"+str(x))
count = 0
for x in xrange(0,200000) :
    if a==c : count += 1
    if a==c : count += 2
    if a==d : count += 3
    if b==c : count += 5
    if b==d : count += 7
    if c==d : count += 11
print count
```

The results:

Timings of the Python string test (in seconds)		
System	memcmp version	time
Python compiled on Fedora Core 14, 64-bit machine	standard memcmp	29.965
	fastmemcmp suite	14.671
	-fno-builtin-memcmp	12.29
Python compiled on Fedora Core 15, 32-bit machine	/usr/bin/python27	42.33
	standard memcmp	18.62
	fastmemcmp suite	18.17
	-fno-builtin-memcmp	18.61
Python compiled on RedHat 6.0, 32-bit machine	standard memcmp	46.497
	fastmemcmp suite	22.468
	-fno-builtin-memcmp	22.595

We don't expect the the exact same level of speed-up (2x-3x) as we saw for the staight C code because the Python code is still interpreted: some code still has to be dispatched through the interpreter. But the speed-up is still very close to what we saw with C—at least 2x.

Note that the FC15 machine actually had little change: this is because on that machine (see Figure 7) the standard memcmp performs well. Note that the standard Python 2.7 that comes on the same FC 15 machine still suffers from the slow memcmp problem. Although the FC15 machine gains no benefits from the fastmemcmp suite, it does lend credence to the fact that the fastmemp suite doesn't slow Python.

5.2 Dictionary-Based Tests

The Python dictionary uses strings very heavily, so a related question: how do the memcmp optimizations affect the performance? In the best case, we hope we can speed up performance of dictionaries, We hope, at least, not to cause any slowdown.

The only place that the `memcmp` optimization could cause any difference is during hash-collision: a Python dictionary is implemented as a hashtable, and hash collisions are when two keys hash to the same value. If two hash values are the same, we only know if the actual values are the same by memcmping the strings.

The first testcase continually builds a large dictionary from scratch, throws it away, then builds it again inside a loop. The dictionary is large enough that collisions do happen: we have instrumented the code and see *fast_memcmp* being called repeatedly.

Timings of the Python dict test (in seconds)		
System	memcmp version	time
Python compiled on Fedora Core 14, 64-bit machine	standard memcmp	24.797
	fastmemcmp suite	25.17
	-fno-builtin-memcmp	24.91
Python compiled on RedHat 6.0, 32-bit machine	standard memcmp	36.912
	fastmemcmp suite	37.151
	-fno-builtin-memcmp	37.284

The difference is negligible: the timings in this test have enough timing variation that it's not accurate enough to say one implementation is really faster than another. Part of this is because Python has a small optimization which goes a long way: in a hash collision, (i.e, the hash values of two objects are the same), Python only calls memcmp *after* it checks the equality of the very first character of both strings. Since the hash function in Python is very good, the odds of the first character distinguishing two colliding hash values is very high.

The second testcase tries even harder to force collision lookups: the test continually reinserts the same keys over and over into a table: this forces a re-lookup of the previous value so it can overwrite it. In this re-lookup, we *expect* the strings to be same when we hash because we are reinserting in the same table at the same key. The hash value will be the same, but we don't know if it's the same key until the full memcmp completes.

Timings of the Python dict test (in seconds)		
System	memcmp version	time
Python compiled on Fedora Core 14, 64-bit machine	standard memcmp	19.896
	fastmemcmp suite	18.024
	-fno-builtin-memcmp	18.137
Python compiled on RedHat 6.0, 32-bit machine	standard memcmp	31.707
	fastmemcmp suite	28.908
	-fno-builtin-memcmp	28.273

There is some difference (and as the size of the string grows, the effect of the faster memcmp is felt more and more). The test seems stable and gives about the same results each time, implying a modest 8-10% increase in speed. This indeed shows the faster memcmps having some effect. It's a bit specific for dictionaries, though, but it's interesting that we can improve dictionary performance (a very highly-optimized data structure in Python), even in this particular case.

A little mucking with either test case, though, and the memcmp difference becomes negligible. It's not clear that the memcmp is helping significantly in dictionaries: partly because the hash function in Python is so good that memcmp gets called rarely, and partly because of the Python micro-optimization of looking at the first chacracter of the strings.

6. CONCLUSION

The purpose of this paper is to explore what seems to be an "obvious" slowdown in some code using memcmp: On numerous Intel platforms, the system memcmp seemed to be jarringly slower than the portable implementation described herein. After much exploration, the slowdown was traced to gcc trying an dated optimization.

To be a useful optimization to the Python community, the *fast_memeq*/*fast_memcmp* has to be faster than *memcmp* in most (if not all) situtations and not NOT cause any slowdown. The portable *fast_memeq*/*fast_memcmp* suite com-

pares favorably to the handcoded assembly: in some cases the handcoded assembly is faster (usually when strings are equal) and in other cases the portable code is faster (usually when strings are different).

But in the end, it may just be easier and less risky to add `-fno-builtin-memcmp` to *configure* for Python. This may be the right solution for many applications/languages besides Python: the *fast_memeq/fast_memcmp* is another solution for non-glibc codebases.

APPENDIX

Any discussion of memcmp has to at least include some discussion of strcmp. There are quite a number of strcmp calls in the Python baseline: almost all calls to strcmp in the Python are of the form:

```
char* some_var = ...
if (strcmp(some_var, "hard-coded string")==0)
```

The calls they are almost all equality/inequality checks with the the second arg being a hard-coded C string.

Because C uses C-style string constants where a trailing `0` indicates the end-of-string rather than a length (`"some"` is 5 characters, the 4 of *some* followed by a `'\0'`), it's harder to implement the granularity optimization. In strcmp, each character has to be compared byte-by- byte as it looks for the ending `\0` marker: it's more difficult to equate regions of memory to an int: do you grab a larger granularity int and shift 8 bits at a time looking for 0s? Do you mask a large int looking for 0s? Do you do a table lookup? None of those seem particularly faster than a straight-forward implementation. The granularity optimization works so well in memcmp because it is so simple.

The strcmp call, however, could definitely benefit from the specialization optimization: most strcmp calls compare against 0. It may be worth changing some of those calls to specialized strcmpeq, if they were deep loops.

Timings for memtest (in seconds)					
Operating System/CPU	implementation	different/aligned	different/unaligned	equal/aligned	equal/unaligned
RedHat 5, 32-bit	memcmp	8.58	8.31	8.94	8.68
	glibcmemcmp	4.37	4.24	4.07	3.93
	fastmemcmp 1	5.36	5.13	5.53	5.26
	fastmemcmp 4	3.33	4.12	3.68	4.79
	fastmemcmp 8	4.3	4.58	4.32	4.7
	fastmemeq 1	5.07	5.02	5.27	5.25
	fastmemeq 4	3.58	3.77	3.57	4.74
	fastmemeq 8	3.66	3.77	4.04	4.51
	imemcmp	4.12	3.97	3.52	4.14
RedHat 6.1, 32-bit	memcmp	8.94	8.25	8.27	7.86
	glibcmemcmp	4.64	4.2	4.37	4.37
	fastmemcmp 1	4.87	4.47	5.26	5.04
	fastmemcmp 4	2.74	3.16	3.11	3.32
	fastmemcmp 8	4.15	3.86	3.55	3.31
	fastmemeq 1	4.77	4.31	5.2	4.88
	fastmemeq 4	2.65	3.17	3.05	3.32
	fastmemeq 8	3.31	2.98	3.75	3.04
	imemcmp	2.52	2.55	2.16	2.42
Fedora 14, 64-bit	memcmp	4.38	4.19	4.67	4.52
	glibcmemcmp	2.63	2.67	2.38	2.26
	fastmemcmp 1	4.38	4.16	4.69	4.41
	fastmemcmp 4	1.9	1.94	2.14	1.98
	fastmemcmp 8	2.1	2.1	2.0	1.46
	fastmemeq 1	4.43	4.34	4.95	4.57
	fastmemeq 4	1.95	2.0	2.68	2.4
	fastmemeq 8	1.72	1.51	2.08	1.97
	imemcmp	2.04	1.86	1.81	1.67
Fedora 15, 32-bit	memcmp	3.57	3.71	2.73	2.98
	glibcmemcmp	4.15	3.52	3.46	3.53
	fastmemcmp 1	5.42	5.1	5.41	5.0
	fastmemcmp 4	2.9	2.88	3.02	2.83
	fastmemcmp 8	3.74	2.84	3.05	2.36
	fastmemeq 1	5.21	5.32	5.43	4.94
	fastmemeq 4	2.64	2.57	3.11	3.06
	fastmemeq 8	2.67	2.37	3.22	2.56
	imemcmp	3.42	3.53	2.61	2.74
Ubu 32-bit	memcmp	7.4	7.12	7.82	7.58
	glibcmemcmp	5.26	4.74	4.96	4.51
	fastmemcmp 1	6.08	6.03	6.48	6.12
	fastmemcmp 4	3.58	4.37	3.94	5.19
	fastmemcmp 8	4.67	4.97	5.01	4.85
	fastmemeq 1	5.7	5.92	6.35	5.99
	fastmemeq 4	3.75	4.2	3.95	5.21
	fastmemeq 8	4.17	4.22	4.49	4.66
	imemcmp	3.96	4.12	3.89	4.59
SunOS SPARC, 32-bit	memcmp	39.5	39.1	43.1	41.8
	glibcmemcmp	36.8	38.3	34.7	35.5
	fastmemcmp 1	61.2	58.2	66.4	64.0
	fastmemcmp 4	33.1	40.9	37.6	39.7
	fastmemcmp 8	36.9	41.3	38.3	39.7
	fastmemeq 1	69.9	66.7	79.4	75.6
	fastmemeq 4	34.3	39.4	36.7	37.8
	fastmemeq 8	35.9	39.5	39.1	38.0
	imemcmp	45.7	44.8	49.9	48.1
Tru64, 64-bit	memcmp	11.0	11.0	9.4	9.8
	glibcmemcmp	14.4	16.1	15.2	16.3
	fastmemcmp 1	20.7	21.0	22.8	22.6
	fastmemcmp 4	13.9	16.5	14.5	17.1
	fastmemcmp 8	14.0	16.8	14.1	17.4
	fastmemeq 1	19.9	21.4	22.8	22.1
	fastmemeq 4	13.6	16.4	13.4	16.5
	fastmemeq 8	13.4	16.2	13.6	16.8
	imemcmp	11.3	11.6	10.4	10.9

Figure 7: Timing matrix of memcmp across all platforms and options

```
#include <stdint.h>   /* So we have sized integers */

int fast_memcmp8 (const void* s1, const void* s2, int len)
{
  if (s1==s2) return 0; /* optimization */

#if defined(INTS_NEED_ALIGNED)
  /* Check and see if we can do "memory aligned" instructions:
  ** This is where we get the speed: most architectures are MUCH
  ** faster when the do aligned memory accesses */
  intptr_t lp = (intptr_t)s1;
  intptr_t rp = (intptr_t)s2;
  if ((lp | rp) & 0x3 ) {
    /* Nope, bottom three (two) bits set on one of the two, SIGH: use memcmp */
    return glibcmemcmp(s1,s2,len);
  }
#endif

  /* Assertion: pointers are aligned, can do compares ACROSS 8 bytes */
  uint64_t *li = (uint64_t*)s1;
  uint64_t *ri = (uint64_t*)s2;
  int greater_len = len>>3;      // large granularity: 8 byte chunks
  int lesser_len  = len & (0x7); // small granularity: <4 bytes

  /* Major pass, 8 bytes at a time */
  int ii;
  for (ii=0; ii<greater_len; ii++) {
    uint64_t lc = *li++;
    uint64_t rc = *ri++;
    if (lc != rc) {
# if __BYTE_ORDER == __BIG_ENDIAN
      return lc < rc ? -1 : 1;
# else
      li-=1; ri-=1;   /* back up and address where the problem was */
      lesser_len = 8; /* Force to look at next 8 bytes to find problem */
      break;
#endif
    }
  }

  /* Minor pass, last 1..7 bytes */
  unsigned char *l = (unsigned char*)li;
  unsigned char *r = (unsigned char*)ri;
  for (ii=0; ii<lesser_len; ii++) {
    unsigned char lc = *l++;
    unsigned char rc = *r++;
    if (lc != rc) {
      return lc < rc ? -1 : 1;
    }
  }
  /* Made it all the way: yep, they are same */
  return 0;
}
```

Figure 8: 64-bit fast_memcmp