Automated Locality Optimization Based on the Reuse Distance of String Operations

Silvius Rus
Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
Email: rus@google.com

Raksit Ashok
Google India Pvt. Ltd.
No. 3, RMZ Infinity - Tower E
Old Madras Road
Bangalore, 560 016, India
Email: raksit@google.com

David Xinliang Li
Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
Email: davidxl@google.com

***Abstract*—String operations such as memcpy, memset and memcmp account for a nontrivial amount of Google datacenter resources. String operations hurt processor cache efficiency when the data accessed is not reused shortly thereafter. Such cache pollution can be avoided by using nontemporal memory access to bypass L2/L3 caches. As reuse distance varies greatly across different memcpy static call contexts in the same program, an efficient solution needs to be call context sensitive. We propose a novel solution to this problem using the page protection mechanism to measure reuse distance and the GCC feedback directed optimization mechanism to generate nontemporal memory access instructions at the appropriate static code contexts. First, the compiler inserts instrumentation for calls to string operations. Then a run time library measures reuse distance using the page protection mechanism during a representative profiling run. The compiler finally generates calls to specialized string operations that use nontemporal operations for the arguments with large reuse distance. We present a full implementation and initial results including speedup on large datacenter applications.**

***Index Terms*—memcpy; nontemporal; reuse distance**

I. Introduction

A. Background

Standard library string operations such as *memcpy*, *memset* or *memcmp* and others are ubiquitous. They implement basic functions: memory copy, memory move, finding a character in a string or finding the length of a string. They are used in core libraries, such as dynamic memory allocators, or runtime systems of various languages, including C, C++, Java or Python. In a function profile across all Google datacenter applications string operations take 2 of the top 10 spots.

String operations can cause performance problems because they flush large portions of the processor caches. Consider the case in Fig. 1, where the *memcpy* source is read again immediately thereafter, but the destination is not reused for a while. If the destination memory is not in cache, this will have a doubly negative effect. First, writing requires bringing each line to cache. To do so, previously cached data will have to be written back. That makes already two cache-to-memory operations. Moreover, since the cache lines will not be reused for a while, this effectively reduces the usable size of the cache. At some point they will get written back, perhaps replaced by an access to the same data that they evicted in the first place unnecessarily. That makes two more cache-to-memory operations.

```
memcpy(dest, src, 65536);
// 'src' is reused immediately.
do_something(src);
work_on_something_else();
// 'dest' is reused much later.
do_something(dest);
```

Fig. 1. Example of a call to *memcpy* where the *read* memory operation has temporal locality, but the *write* does not, so it pollutes the cache.

Such cache pollution can be avoided by using nontemporal memory access to bypass L2/L3 caches. Then a *write* operation will take a single memory operation. Moreover, *writes* to contiguous memory locations may be aggregated in on-chip *write-combine* buffers and then written out more efficiently. When a write-combine buffer is completely full it can be written out without requiring a read-modify-write sequence. However, nontemporal operations are suboptimal when the data is in fact reused shortly thereafter because it will have to be fetched to cache. A regular *write* followed by a reuse of the data is faster because it has a useful prefetch effect. Nontemporal *writes* are also not beneficial when data is already cached In such cases regular *writes* are usually faster because they may modify only the cache and not the main memory. Moreover, nontemporal *writes* require explicit flushing of write buffers, which adds a fixed amount of overhead and places extra burden on the programmer or code generator.

Throughout this paper we will focus only on x86_64 processors, namely AMD Family10h [1] and Intel Nehalem [4] and Core2 [3]. While the techniques presented in this paper are generally applicable, their effectiveness depends on microarchitecture design.

B. Motivation

Unfortunately it is not easy to tell beforehand whether a *memcpy* target will be used or not. A heuristic is to use nontemporal operations for large operations. Such large operations are more likely to have a large reuse distance. That

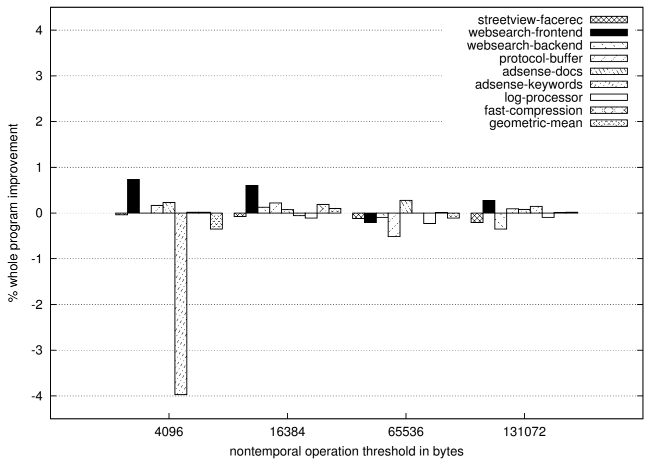


Fig. 2. Operation size alone is not a good indicator alone for substituting string operations with nontemporal counterparts.

is how *memset* works on a recent implementation of GNU *libc*. However, Fig. 2 shows that *size* alone is not a good discriminator for other string operations including *memcpy*. (*Size* is an important factor though, and is considered alongside reuse distance throughout the remainder of this paper.)

The fundamental problem is that *reuse distance may vary across different memcpy static call contexts in the same program*. An efficient decision to use nontemporal memory access for string operations may need to be made for each static call context, and possibly for different workload classes.

The practical problem is that measuring the reuse distance by *instrumenting each memory reference is expensive*, especially for large programs with large data sets. [12] reports a 200x time expansion even for fairly small data sets. This is unfortunate since these issues are most important on larger servers running applications with high memory usage on multiple processing cores and memory nodes. A light-weight reuse distance measurement mechanism is crucial to practical applicability.

C. Contribution

We propose a novel feedback-directed solution to this problem. First the compiler inserts instrumentation for calls to string operations. A run time library measures reuse distance using the page protection mechanism during a representative profiling run. The compiler then generates calls to specialized string operation versions that use nontemporal operations for the arguments that presented large reuse distance.

This paper makes the following original contributions:

- A novel pay-per-view method to measure the reuse distance of string operations. The overhead is proportional to the dynamic number of string operations instrumented and not to the number of dynamic memory references, since most work is piggy-backed on the hardware paging mechanism. No hardware extensions are needed.
- A feedback directed compiler optimization pass that replaces string operations with semantically equivalent but nontemporal operations at static call contexts with large reuse distance.

II. String Operations

String operations are an important component of about every program. Function profiles across all applications in Google datacenters show *memcpy*, *memset*, *memcmp* among the top cycle consumers globally. We will focus on these functions, and in particular on *memcpy* for the remainder of this article. Note that while we are providing an in-depth performance characterization of nontemporal memory access operations, the purpose of this article is not to explain how nontemporal memory access is implemented in specific processors. For implementation details please see the corresponding documentation [1], [4], [3].

A. Optimizing String Operations

Optimizing *memcpy* is a popular Internet topic. A web search showed 10 times more matches for *optimize memcpy* than for *optimize malloc*. Much of this *memcpy* optimization effort is short term, related to a project and a specific microprocessor architecture. Why not solve this once and forever?

First, *memcpy* performance depends on the memory hardware capability and configuration: bus/interconnect throughput and latency, cache hierarchy (cache capacity and associativity, cache coherence protocol). The main reason why a single most efficient *memcpy* does not exist is that the processor microarchitecture changes over time. Some of these changes have a profound impact on *memcpy* performance. The bus width or clock rate have a direct effect. A more subtle effect comes from cache behavior. Processors with *writeback* caches keep data in cache without sending it to memory until a cache conflict, capacity or coherence event requires it. In order to write a memory location, its whole cache line is brought to cache first. However, we will see shortly that this mechanism is not always optimal. The alternative is to use *nontemporal* memory access, which bypasses some levels in the processor cache hierarchy.

Second, *memcpy* performance is often affected or dominated by cache misses related to its interaction with the surrounding code. Moreover, the same code may show completely different *memcpy* behavior on fundamentally different input sets.

We argue that it is this combination of microarchitecture changes and dependence on context and input that make it hard, perhaps impossible, to come up with a one-size-fits-all memcpy. Thus we propose a feedback-directed solution. The remainder of this section explores the performance tradeoffs between *memcpy* implementations using temporal and nontemporal memory operations, which establishes our field of choices and tells us what factors are relevant in the feedback-directed decision process.

Nontemporal memory operations can speed up execution in two ways. First, nontemporal writes may not require a cache line to be fetched before the write, possibly resulting in a 2x

```
#define BLOCK_SIZE (32 * 1024)
#define ARRAY_SIZE (256 * 1024 * 1024 + BLOCK_SIZE)
#define REPS (8 * 1024)

void test(char* p1, char* p2, char* p3, char* p4) {
 size_t offset = 0;
 for (int i = 0; i < REPS; ++i) {
  start_timer();
  memcpy(p1 + offset, p2 + offset, BLOCK_SIZE);
  stop_timer();
  elapsed_memcpy += get_elapsed();

  start_timer();
  work(p3 + offset, p4 + offset);
  stop_timer();
  elapsed_work += get_elapsed();

  offset += BLOCK_SIZE * (random() % 100)
  offset = offset % ARRAY_SIZE;
 }
}

char a1[ARRAY_SIZE], a2[ARRAY_SIZE],
     a3[ARRAY_SIZE], a4[ARRAY_SIZE];

test(a1, a2, a3, a4);  // No reuse.
test(a1, a2, a3, a2);  // Read reuse.
test(a1, a2, a1, a4);  // Write reuse.
test(a1, a2, a1, a2);  // Read and write reuse.
```

Fig. 3. Code kernel to simulate 4 different reuse scenarios. In all cases the test consists of a *memcpy* of size 32 KB followed by a *work loop* that also touches 2 * 32 KB of memory. Unlike *memcpy*, it touches it in random order. The first argument to *work* is read and written and the second one only read).

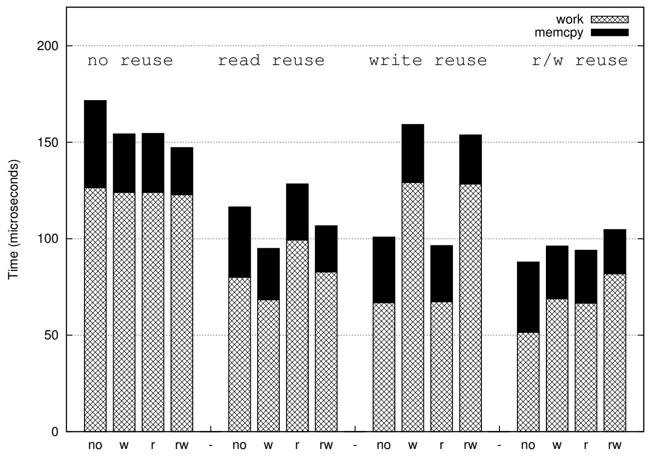
increase in throughput. Second, they do not replace existing lines in cache, thus a cache line referenced before and after a conflicting nontemporal *memcpy* will not cause a miss.

However, nontemporal operations are suboptimal when the data is in fact reused shortly thereafter because it will have to be fetched to cache. A regular *write* followed by a reuse of the data is faster because it has a useful prefetch effect. Moreover, at least on the processors we experimented with, nontemporal *writes* require explicit flushing of write-combine buffers, which adds a fixed amount of overhead and places extra burden on the programmer or code generator.

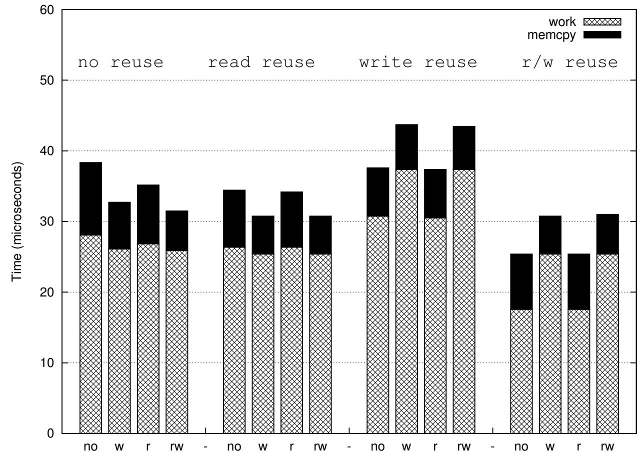
Fig. 3 shows a code snippet used to evaluate the effect of using nontemporal memory operations. We implemented four versions of *memcpy*, corresponding to all the combinations of temporal/nontemporal read/write. Nontemporal *reads* are implemented by issuing `prefetchnta` instructions in advance. Nontemporal *writes* were implemented using `movntq` instructions. Whether this is the best way to implement nontemporal operations is outside the scope of this paper. Each implementation is evaluated in four reuse scenarios.

Fig. 4 illustrates the effectiveness of nontemporal operations for each possible reuse pattern.

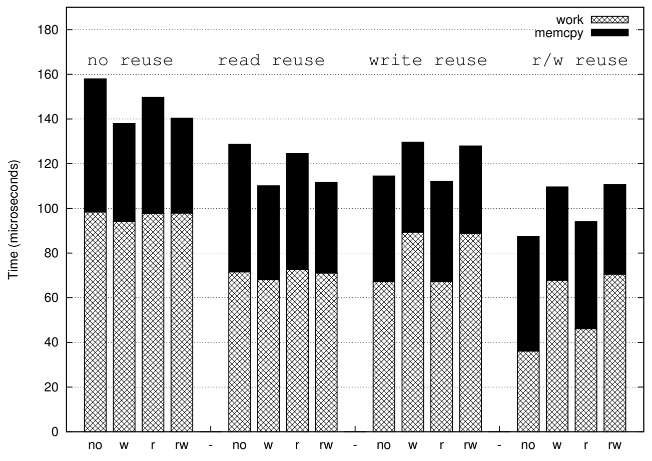
AMD Family10h (a) and Intel Core2 (c) show similar behavior patterns that fit our expectations. When there is no reuse between the *copy* loop and the *work* loop (first bar group in each graph), the best *memcpy* version is *rw*, with



(a) AMD Family10h



(b) Intel Nehalem



(c) Intel Core2

Fig. 4. Performance of the kernel in Fig. 3 under four reuse scenarios using four *memcpy* implementations (no = temporal read and write, r = nontemporal read, temporal write, w = temporal read, nontemporal write, rw = nontemporal read and write).

```
record range
  start: address value type
  len: size value type
  counter: reuse distance value type
  time: time value type

routine instrument(start, len, counter)
  mprotect(start, len, PROT_ALL)
  range = create_range(start, len, counter)
  range_tree.insert(range)
  range.time = get_time()

routine segfault(address)
  time = get_time()
  range = range_tree.find(address)
  mprotect(range.start, range.len, PROT_NONE)
  elapsed = time − range.time
  update_running_mean(range.counter, elapsed)
  range_tree.remove(range)
  destroy(range)
```

Fig. 5. Reuse distance measurement pseudocode.

both nontemporal *reads* and nontemporal *writes*. When only the *memcpy* source is reused (second group of bars), the best *memcpy* version is *w*, nontemporal *writes* but temporal *reads*. This make sense, since the destination is not reused but the source is. When only the *memcpy* destination is reused (third group of bars), the best *memcpy* version is *r*, temporal *writes* but nontemporal *reads*. When both the source and the destination are reused, the best version is *no*, the one using temporal *reads* and temporal *writes*.

All these graphs were recorded using a block size of 32 KB. Different block sizes (not illustrated here) produce different numbers but overall the nontemporal/reuse interaction is significant. When the size decreases below 4 KB, nontemporal *reads* start to become too expensive. At a block size of 1 KB the overhead of nontemporal *reads* becomes so high that overall the versions that use nontemporal *reads* are always slower than the others, regardless of the reuse pattern.

There are several conclusions from this study. First, performance numbers vary across microprocessor architectures. Second, there are at least two factors to be considered when designing nontemporal string operations: (1) the actual reuse distance of the memory touched by the string operation and (2) the size of the string operation.

III. Memory Reuse Distance Measurements

Measuring memory reuse distance is an established topic. However, the existent techniques rely on analyzing large traces produced by instrumentation of all memory references [17]. Since we are interested at this point only in string operations, it is wasteful to instrument every memory reference. Moreover, our benchmarks are large, thus instrumentation may be impractical due to memory or time overhead.

A. Measuring the Reuse Distance using Page Protection

We chose to measure reuse distance in a different way. Instead of instrumenting every memory reference or a random selection, we decided to instrument only the operations we care about. The instrumentation mechanism (Fig. 5) protects the memory pages from read/write access and records the current *time*. When one of these protected pages is accessed next, a custom page fault handler is invoked. This handler determines whether the fault occurred at an address in the original *memcpy* range. If so, it records the current *time*, then it records the time elapsed from the moment the page was last protected and then it unprotects the page and resumes execution. The elapsed time is accumulated as a running mean.

The cost of our instrumentation is thus pay-per-view. It it limited to one call to *mprotect* for each instrumentation point plus one insertion in a range tree, which is bounded in size. Each memory reference does get checked, but this is piggybacked on the hardware virtual-to-physical address translation mechanism. No special hardware is required. No more than a fault occurs for each dynamic instrumentation point. Each fault takes a lookup in the range tree, followed by a call to *mprotect* and one range tree deletion.

B. Reuse Distance Metrics

We are currently using a lightweight system clock as the metric for the reuse distance timer. While this is an imprecise estimation of memory activity, it has provided sufficient information that led to measurable gain in several benchmarks. In a different study, [14] showed that locality measures can be approximated using time with over 99% accuracy.

As work in progress, we are exploring other measures, hardware event counters such as L2 cache writebacks. The challenge is to implement a measure that is robust across threads and processing units. Many of our benchmarks are heavily multithreaded and it is not uncommon that the result of a *memcpy* executed on some processor core get reused first on a different core, and computing the difference between the counter values on different cores does not make sense. One way to achieve this would be to record counter values on all cores periodically via timer interrupts and to build a per-core database of counter values along a global time line (itself based on a synchronized clock). Then we can simply record the time and core where the protection occurred, and the same for the moment the page fault was triggered. Consulting the database will then allow us to differentiate between writebacks on the original core vs. on the core where the page fault is triggered. All current experimental results were obtained using a system clock.

C. Engineering

Although the idea is simple, we met several technical hurdles in implementing this mechanism.

First, string operations are ubiquitous. They may be called in application bootstrap code, such as memory allocator code in dynamic library initialization routines. This code may run before even the *pthread* library is functional, which means we could not use *malloc* or *pthread* everywhere in the instrumentation library.

Second, the data copied by *memcpy* could be anywhere in the process space. It could be on the same page as the global offset table or the procedure linkage table, which means we cannot always call shared library functions from the signal handler, as that would cause a second SEGFAULT which crashes the program. It could be on the same stack page as the instrumentation routine's stack frame, thus the SEGFAULT would be triggered before even returning, resulting in pure overhead with no reuse distance information gain.

Third, several threads may try to read a protected region at about the same time, which will produce several calls to the SEGFAULT signal handler for the same protected area. These calls get serialized. The first handler instance must unprotect the pages so execution can continue. However, the second signal on the same range will appear unrelated to the reuse distance measurement mechanism because this range is not registered anymore. To distinguish such duplicated signals from unrelated signals possibly caused by software bugs, we simply try to read one word at the SEGFAULT address. If this signal is a duplicate, nothing will happen because the pages were just unprotected. If the signal is unrelated then this *read* will result into a second SEGFAULT that will crash the program as it would do in a normal execution.

A more severe issue is that if a pointer to a protected page is passed into a system call, dereferencing it will result not in a call to our fault handler, but rather in an EFAULT system call return value.

Our solution is to isolate the instrumentation code and data from the application code and data. There are several ways to accomplish it. First, the program could run under *ptrace*, a Linux system utility that permits an observer program to intervene at each system call. This would give us the separation of data and code as the observer runs different code in a different process space. However, this approach has severe drawbacks. Running under *ptrace* may conflict with other monitoring processes, such as debuggers or sandboxes. It adds more slowdown and it makes it harder or impossible to run in production environments.

Our current approach is to run in the same process space and manage the separation explicitly. The SEGFAULT handler runs on its own stacks separate from the thread stacks. For global state we do not use global variables. Instead we use macro names that expand to offsets in a privately mapped memory pool. We use a custom dynamic memory allocator that uses privately mapped memory pages as well. Our library does not make system calls with a few controlled exceptions. There are a few locations that never get protected: the current stack frame, the global offset tables and procedure linking tables and a special page that contains a master mutex used to control updates to internal global data structures.

To avoid leaking protected pages through system calls, we wrap all *libc* library calls and walk all their pointer arguments that may be passed to system calls. For instance, for function *open* we walk the string that stores the file path. If this string lies on a protected page, the SEGFAULT will be triggered during this walk, *before* the system call. If protected memory *is* passed through the system call inadvertently, this does not lead to a SEGFAULT, but instead the system call returns with an error code, usually EFAULT, which may not crash the application but rather change its behavior–a hard to detect bug. Walking all the memory that could be referenced by a system call is unfortunately hard to do reliably for vaguely specified interfaces such as *ioctl*, where the argument list is variable and the argument types may depend on runtime flag values. However, the wrapping mechanism has worked in practice for several million lines of benchmark code.

D. Overhead Reduction

The instrumentation overhead comes from multiple sources.

First, a call to an instrumentation stub is made for every dynamic call to a string operation. Most calls return after a brief check because not all string operations are actually instrumented (controlled by profiling run environment variables). However, even these quick checks do lead to measurable overhead. One way to diminish this effect is to tell the compiler exactly what operations are to be instrumented and avoid inserting calls to instrumentation stubs altogether for functions deemed not interesting.

Second, most instrumentation calls would result in one or two *mprotect* calls, which are very expensive. There is also overhead in looking up and maintaining the reuse distance library internal structures (range tree, dynamic memory allocator free lists, result tables). In a small stress test we measured a 1,000x slowdown for very short and frequent *memcpy* calls. Most programs do not call *memcpy* nearly as much, but can still show significant overhead, up to around 2x in practice. We reduced this greatly by employing sampling uniformly at the call site of *mprotect*. The instrumentation library thus drops 100 of every 101 instrumentation requests.

Third, we are limiting the amount of extra memory used to store internal data structures such as custom memory allocator free lists or page trees. We do it primarily to avoid out-of-memory kills in production environments but it also reduces overhead by dropping instrumentation requests that would go over the memory budget. In practice this mechanism is useful mostly at very low sampling rates. It has the same drawback as sampling, potential decrease in profile quality.

Fourth, we are wrapping all *libc* calls that may result in system calls with access to user space memory. The wrapper code needs to walk the memory before calling the original function to avoid page faults within the system call. To reduce the overhead of these walks the wrapper code does not read every memory word but rather one word per page, which reduces this overhead by a factor for large strings. This is unfortunately not possible for strings with unknown length, where reading all the characters is needed in order to detect the terminating character.

Section V-A shows benchmark statistics including instrumentation overhead across many large benchmarks. As a geometric mean the overhead of reuse distance measurements for calls to *memcpy*, *memset*, *memcmp* and *bcmp* is about 2% to 3%.

Code	Lines	Description	memcpy	memset	memcmp	memchr
bigtable	≈ 1M	Bigtable server	1.72%	0.11%	0.40%	0.01%
streetview-facerec	≈ 1M	Face recognition	0.59%	0.06%	0.33%	0.00%
streetview-stitch	≈ 1M	Panorama stitcher	0.38%	2.19%	0.14%	0.00%
websearch-frontend	≈ 1M	Web server	2.96%	0.10%	1.09%	0.63%
websearch-backend	≈ 1M	Search engine	0.25%	0.12%	0.20%	0.00%
encryption-ssl	≈200K	Openssl	0.12%	0.32%	0.01%	0.31%
protocol-buffer	≈ 1M	Wire protocol	8.50%	0.04%	0.01%	0.00%
adsense-serving	≈ 1M	AI/Data mining	37.58%	0.80%	0.00%	0.00%
adsense-indexing	≈ 1M	AI/Data mining	1.20%	2.49%	0.03%	0.00%
log-processor	≈ 1M	Log mining	4.29%	0.18%	0.49%	0.42%
fast-compression	≈100K	(De)Compression	21.74%	1.47%	0.00%	0.00%

Fig. 6. Google datacenter benchmarks. The numbers represent the percentage of total CPU cycles spend in top string operations during benchmark execution.

IV. Compiler Automation

A. Feedback Directed Optimization

We decided to implement the automation of the substitution with nontemporal string operations in the GCC[2] feedback directed framework. GCC users are already used to it:

```
// Code instrumentation phase.
gcc -fprofile-generate test.c
// Profile collection in representative run.
./a.out
// Optimization using profile data.
gcc -fprofile-use test.c
```

The instrumentation phase is straightforward. The compiler inserts calls to the runtime library after every call to a string operation. The advantage of doing this in the compiler, as opposed to, e.g., using the LD_PRELOAD mechanism is that the compiler will insert this instrumentation even for *memcpy* calls that are transformed by other compiler mechanism, e.g., as an inline loop. Also, over time we hope to make use of compiler information to enhance and extend this mechanism beyond string operations.

We are using the GCOV mechanism and the GCC profile-generate infrastructure. Each static call context is allocated a GCOV set of counters. The address of the first counter will identify this context throughout the runtime library. The library is responsible to update the counter values continuously as more information about the same static context accumulates during execution. Currently there are four counters for each pointer argument of each string operation: the average operation footprint, the average reuse distance, the dynamic number of calls and the product $size * reuse\ distance$ aggregated over all the calls.

In the profile-use phase the compiler reads back the counter values for each call context and decides whether to substitute the call with an equivalent nontemporal one. The decision is controlled by compiler parameters that set thresholds for the operation footprint and for the reuse distance.

In addition to sensitivity to static call context, making these substitutions in the compiler makes sense because the compiler has access to the memory reference pattern before the string operation. Thus the compiler can improve a decision if it knows, for instance, that a *memcpy* is immediately preceded by a loop that references its arguments. In such cases using nontemporal memory access would not be beneficial because data may be already cached. The compiler could also help reduce the number of instrumentation points by not instrumenting string operation call sites follow shortly by reuse of their arguments. We have not explored this avenue yet.

Section V presents profile-generation and profile-use data and discusses quantitative choices.

B. User Advisory Tool

Is user advice to switch to nontemporal operations appropriate and feasible? This question turned out more complex than it appeared initially. On one hand a fully automated compiler mechanism has several advantages. First, decisions may change over time. A suggestion to replace a call to *memcpy* with a call to a nontemporal version may not be given anymore after the input or the program code changes significantly. The compiler mechanism would simply not make the substitution anymore, whereas a programmer would have to re-edit the code in order to revert the original substitution. Another advantage on the compiler side is that *memcpy* calls from inlined functions will get analyzed and optimized differently at each inline site. The advice to the user would then be to perform inlining by hand and then specialization at only some sites. That would be likely to have little practical success.

On the other hand, giving advice to the programmer may point to design mistakes at a level higher than the compiler can fix. Replacing with a nontemporal *memcpy* may simply offer a partial fix of the effect, which will hide the root cause. It may be better to let the programmer figure out the root cause. Consider the following code snippet.

```
Item item;  // sizeof(item) == 64 KB
pool.get(item, location);  // memcpy(item, ...)
item.set_field(key, val);  // Touch only 1 KB.
pool.put(item, location);  // memcpy(..., item)
```

The first call to *memcpy* will have a short reuse distance (although only a small part of the item is touched). The second call to *memcpy* may have a very large reuse distance for the *write* operation. The compiler solution would be to use nontemporal *writes* for the second call to *memcpy*. While that leads to speedup, possibly significant, this is hiding a design choice that leads to severe performance issues. The problem is that the design considers *item* as an atomic quantum, and this quantum is much larger (64 KB) than the state (1 KB)

```
void add_int_vector(int* dest, int* src1, int* src2,
                    size_t len) {
  size_t i;
  for (i = 0; i < len; ++i)
    dest[i] = src1[i] + src2[i];
}
```

Fig. 7. Streaming kernel that is not a string operation.

that needs to be changed by the operation. This is a hard to detect high level performance bug. The programmer may not be aware of it because the ratio between the copy size and the modified state size may increase over time even though it was initially close to 1.

We use the compiler feedback-directed mechanism as a user advisory tool as well. The reuse distance and footprint at each call context are printed as compiler notes for all the sites where they were measured. They are extremely valuable in giving application programmers a better understanding of how their programs reference memory. The compiler notes are given in a fixed format, which makes it trivial to postprocess them and figure out quickly the top, say, 10 sites which in many cases account for over 50% of all time spent in string operations.

C. Extension to Arbitrary Streaming Kernels

Although this paper focuses on string operations, the same technique can be used with other streaming kernels. In the example in Fig. 7 two arrays are read and one is written. Depending on the reuse distance of each of (*dest*, *src1* or *src2*), the compiler can generate one of eight combinations of temporal/nontemporal choices. There are a few additional issues when transforming arbitrary streaming kernels.

First, the compiler must be able to recognize the streaming pattern, or the library programmer must be able to tell the compiler how to instrument important functions. We have not yet explored automatic pattern recognition but believe there is strong potential in this direction. We did design the compiler framework to make it easy for programmers to define instrumentation methods for arbitrary functions.

Second, the substitution decision must be aware of hardware capabilities. Issuing nontemporal memory operations for several streams concurrently might not help even when the memory reuse distance is large, because the hardware may impose limitations on the number of in-flight nontemporal memory access operations.

Third, the compiler must be able to generate efficient code. Unlike the canned string routines, arbitrary kernels have to be reconstructed and specific memory access instructions have to be substituted. Instructions must be inserted to flush write-combine buffers.

A practical alternative to this relatively complex compiler mechanism is to ask developers to recognize and instrument such kernels by hand using the reuse distance library we provide. The feedback-directed optimization compiler pass can be extended to handle functions other than string operations. We have designed but not yet implemented the compiler support for this semi-automated process.

D. Limitations

Our approach has some limitations, some technical and some fundamental.

First, we are not taking into account the reverse reuse distance for string operations, which is the distance from the previous reference on the same memory region to the string operation. This could be important because a nontemporal *write* on cached data in exclusive of modified state is slower than a temporal *write*. While the nontemporal one modifies RAM, the temporal one will likely modify just the cache, which takes much less time.

Second, we are using an artificial measure for the reuse distance. We are investigating ways to integrate hardware event counter readings in this measure, such as writeback events.

Third, we have not looked at interaction with hardware prefetch mechanisms. We are planning to study this interaction and to research possible correlations between hardware events related to hardware prefetch and the opportunity of nontemporal operations.

Fourth, we are not detecting dynamic phases for a single static context. The same static context may change reuse distance over time. Our approach misses such opportunities.

Finally, we are only instrumenting operations larger than 4,096 bytes, so we are missing a large number of opportunities. Instrumenting only objects larger than a memory page reduces the overhead. Instrumenting smaller pages may result in useless instrumentation, in the case where a page fault is generated by a memory access to data on a protected page, but not within the actual string operation range. This happens with our current scheme as well, but with lower probability.

Also, we acknowledge that feedback-directed optimization is not a feasible solution for all programs, but it is for a large class of applications, such as many datacenter servers or repetitive tasks such as stitching images into StreetView panoramas or recognizing characters on scanned books.

V. Evaluation

A. Benchmarks and Setup

We used a set of benchmarks largely representative of Google datacenter workloads. Fig. 6 presents a few characteristics of these benchmarks. Note that while benchmarks *adsense-serving* and *adsense-indexing* use the same binary, they are run on different classes of input. This leads to very different importance of string operations. 37.58% of *adsense-serving* is spent in *memcpy* vs. 1.20% for *adsense-indexing*. Note that the performance behavior is stable within a single input set class (serving or indexing). For instance, the percentage spent in *memcpy* is consistent across serving different workloads. This supports our hypothesis that feedback-directed optimization is both needed and effective. In order to deploy this optimization with the *adsense* team, they built two binaries, one optimized for *serving* and the other for *indexing*. It was not a problem because these are two completely different processes.

Benchmark	Slowdown	Instrumentation Points
bigtable	4.0%	15,743
streetview-facerec	0.0%	19,775
streetview-stitch	0.8%	14,516
websearch-frontend	1.8%	31,282
websearch-backend	1.1%	27,022
encryption-ssl	0.0%	1,225
protocol-buffer	15.5%	12,390
adsense-serving	1.7%	11,449
adsense-indexing	7.5%	11,449
log-processor	4.3%	26,958
fast-compression	7.4%	2,438

Fig. 8. Instrumentation statistics. The last column shows the number of static instrumentation points. The baseline for the overhead computation was taken with a binary build with the same options except *-fprofile-reusedist*. We used the default sampling rate of 101.

Benchmark	Coverage	Peak	Peak/Coverage
bigtable	2.24%	0.00%	0.00%
streetview-facerec	0.98%	0.65%	66.32%
streetview-stitch	2.71%	0.74%	27.30%
websearch-frontend	4.78%	1.55%	20.08%
websearch-backend	0.57%	0.31%	54.39%
encryption-ssl	0.76%	0.00%	0.00%
protocol-buffer	8.55%	0.00%	0.00%
adsense-serving	38.38%	31.95%	83.24%
adsense-indexing	3.72%	0.22%	5.9%
log-processor	5.38%	1.99%	36.99%
fast-compression	23.21%	1.84%	7.9%
geometric mean	5.46%	3.23%	

Fig. 9. Optimization statistics. The *coverage* column shows the time spent in string operations as percentage of whole program execution. The *peak* column shows the peak speedup across the whole experiment set. The *peak/coverage* column shows the ratio between the peak speedup and the *coverage*.

Large hot functions are rare in highly tuned datacenter applications. The *adsense-doc* case is clearly an outlier. However, string operations do account for about 3 to 4% of the total application time in many cases.

All experiments were run on *AMD Family10h*, *Intel Nehalem* and *Intel Core2* processors. The number of trials varies with the benchmark. The measurement error margin for speedup is below 0.5% in all cases and around 0.1% in many. Since reviewers expressed doubt about the possibility of measuring large applications with such high accuracy, we feel obliged to shed some light on the methodology. Although the benchmarks generally use the exact same code as production binaries, they were run in isolation, thus they do not access remote file systems or make remote procedure calls during the part of the program execution that gets measured. We tried to reduce the amount of local disk access as well whenever possible. For experiments run on NUMA machines we locked benchmarks to a particular CPU and memory node, when the workload fit within a node. For larger benchmarks we used an interleaved page allocation policy to spread pages across memory nodes evenly thus ensuring similar CPU to memory node mapping across different runs. Each benchmark is run several times and the trimmed mean is reported. The actual run count depends on the natural variance of each benchmark and guarantees error margin below 0.5% with 95% confidence. Benchmarks generally produce a performance metric internally, so performance is usually reported as throughput, server latency or phase run time, and not as an external observation. The test machines are fully dedicated to one experiment at a time and are brought to a clean state before running each experiment. This includes killing rogue processes and dropping system file caches. Not least, benchmarks get better over time and flaky ones tend to get discarded.

Fig. 8 shows that the overhead of the instrumentation is tolerable in most cases. As discussed in Section III-D, overhead is reduced significantly by sampling.

B. Performance Improvement

Fig 9 shows statistics of the compiler optimization phase and peak speedup across all test platforms. Fig. 10 shows speedup graphs for each platform for an operation size threshold of 16 KB (only operations of average size 16 KB or higher are substituted with nontemporal operations).

The *peak/coverage* column shows how well (or not) we are exploiting the opportunities. For *adsense-serving* we appear to be speeding up more than expected. The application speeds up 31.95% while the initial *memcpy* coverage is 38.38%. This high speedup was measured on AMD Family10h, which is most sensitive to nontemporal operations (see Fig. 4. On Intel Core2, the speedup for this application was 16.84% and on Intel Nehalem 9.77%. It turns out that a large part of the execution is taken by *memcpy* with large reuse distance (around 40 million cycles) and fairly large footprint, around 55 KB each.

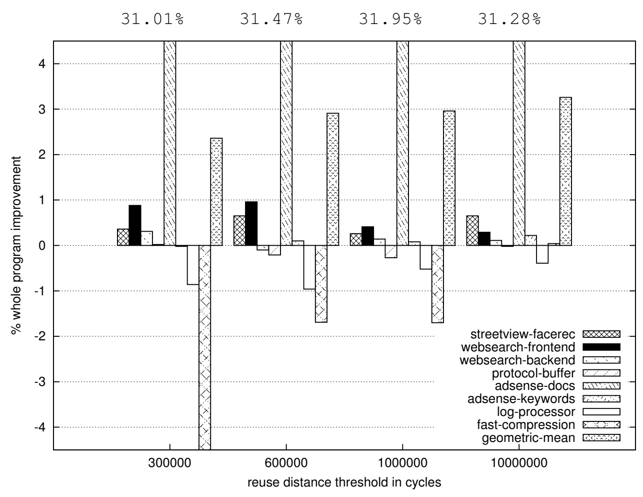
We are approaching theoretical improvement limit in *streetview-facerec* and *websearch-backend*, but the overall speedup is modest because string operations take little of the total execution time to start with.

The *fast-compression*, *adsense-indexing* and *log-processor* benchmarks speed up moderately. Although they spend much time in *memcpy*, the reuse distance is relatively short. A naive replacement with nontemporal operations at all sites in *fast-compression* led to significant slowdown. Note that *adsense-indexing* and *adsense-serving* use the same code, but are run on different input set classes.

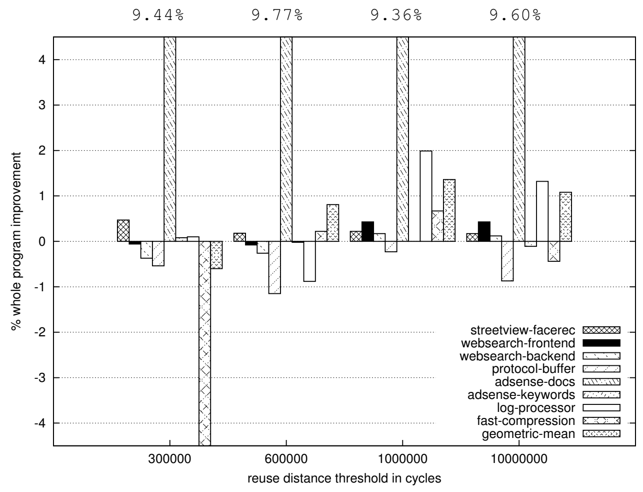
In the *protocol-buffer* and *bigtable* benchmarks many or most string operations are shorter than 4,096 bytes, so our instrumentation misses them. We are planning to investigate whether it is worth reducing our instrumentation threshold to record the reuse distance for shorter strings. This may lead to a large increase in instrumentation overhead. In *encryption-ssl* there are almost no string operations.

Fig. 10 shows that using reuse distance as a discriminator for switching to nontemporal operations makes sense. The best threshold across all three test platforms was 1,000,000 cycles. At this level the geometric mean is 3% on AMD Family10h, 1.36% on Intel Nehalem and 1.90% on Ilium Core2.

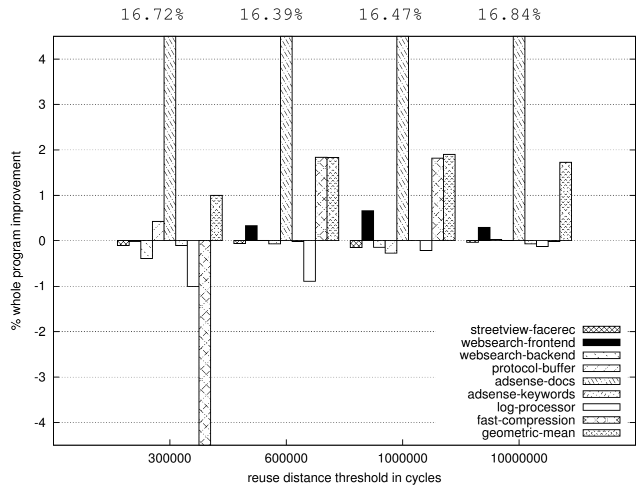
The results are certainly not earth-shattering, but they do prove that string operations can be optimized through fully automated reuse distance analysis, and that this analysis is practical. Note that on average string operations account for only 5.46% of the whole program execution time, so a



(a) AMD Family10h



(b) Intel Nehalem



(c) Intel Core2

Fig. 10. Speedup comparison. String operations are substituted with nontemporal ones when the mean reused distance for a given call context is above the threshold (X axis labels).

reduction of the total execution time by roughly 2% means a 36% improvement in string operation. This is significant especially given that not all initial string operations had large reuse distance.

VI. Related Work and Acknowledgments

Reuse distance measurement. The concept of *reuse distance* was introduced by [11]. [9] observed that patterns of hot data streams do not change much across input sets, which supports our decision to use offline feedback-directed decisions. Similar conclusions appear in [17], where locality in the SPEC benchmark *gcc* is estimated to depend 70% on code and only 30% on input. [17] presents a fairly comprehensive report of recent reuse distance approximation techniques. Approximation is needed as traces of all memory references are too large to analyze with 100% accuracy. [12] documents a 200x increase factor for instrumentation based methods. Some of these papers offer ways to reduce instrumentation overhead, mostly sampling or reducing input size. For us reducing input size is not an option. For instance, the *streetview-stitcher* benchmark stitches images to create streetview panoramas. All the input images are of a certain size. Program behavior does not change significantly when changing input sets, as long as the image stay at that same size. Using smaller images would change the memory access pattern and the ratio between the resident set and cache capacity, which would affect the behavior of the benchmark. We are thus left with sampling, which works well for us due to the pay-per-view feature (overhead is directly proportional with the number of dynamic instrumentation calls and not with the number of all memory references).

Focusing only on string operations misses other opportunities, such as hand coded *memcpy* versions. Previous approaches are more general, in the sense that they instrument every memory operation. However, the focus of previous research was to compute a reuse distance histogram for the whole program, whereas we want to be more precise and compute the reuse distance for all static calls to string operations. We hope that what we learn from this experience can be generalized later through compiler analysis to other similar loops.

We are not aware of any previous implementation that used the paging mechanism to measure reuse distance. This is a central feature of our measurement mechanism. It's what allows us to control the level of overhead simply by adjusting the amount of instrumentation.

One of our current limitations is that we use time as an approximation for reuse distance. However, [14] showed that temporal locality indicators can be approximated using cycle counters with over 99% accuracy. We still plan to explore more useful measures, of which [13] offers a good discussion.

We acknowledge the contribution of anonymous reviewers who helped improve this article. They also helped improve the proposed technique by suggesting sampling to reduce overhead and extending the applicability by targeting functions other than string operations.

Hardware Optimization. [10] discusses memory disambiguation using store distance measured using small buffer in hardware. [13] present a new cache line replacement policy using predictions based on history, using an extra 20 KB in L2. [15] proposing not caching references that miss all the time. The idea is to use 2 prediction bits to measure nontemporality analogous to the branch prediction mechanism.

Compiler Optimization. [8] introduced some of the first key ideas about using the compiler to control cache bypass. [5] presents context based optimization. Unlike our approach, theirs records the reuse distance for each instruction and adds nontemporal hints if over 90% of the instances suggest nontemporal behavior (the actual decision depends on cache size and associativity). The paper does not discuss instrumentation overhead or trace size. [6] make a distinction between uses of reuse distance information. *source hints* are passed to the compiler to define cache dependence information such as *instruction B's operand is prefetched by instruction A, thus its latency will be that of a cache hit*. Target hints are encoded in the instruction and must be supported by the hardware. They propose dynamic hints that take into account program phases. This is more than what we support currently. However, their dynamic hints only work for polyhedral patterns, which is a strong limitation. (They could work for other patterns as well, but it may cost too much to produce the hints for access patterns that are not polyhedra.) [16] describe how to generate nontemporal hints for Itanium and mentions that instead of bypassing the cache, Itanium implements a modified LRU in which nontemporal hints result in going directly to the first position to be evicted. This is better than bypassing the cache because it does not hurt spatial locality. [7] presents static reuse distance. It is elegant and precise, but only works for matlab (regular matrices, no pointers, no irregular data structures).

User Advisory Tools. [12] documents 200x reuse distance instrumentation factor and suggests sampling and/or reducing input size. They document data size between 1 MB to 80 MB. They present a user study where a research program (2,200 lines of code) is improved 7% through a 6 line change based on temporal locality analysis based on reuse distance measurement. Our compiler pass gives advice as well. To put things in perspective, most of our benchmarks have over 1 million lines of code each and the data size is generally over 1 GB. We cannot reduce our data set size for fundamental and practical reasons.

VII. Conclusions

This paper addresses a fundamental, hard problem. The problem is to predict which string operations have large reuse distance. The goal is to replace such operations with nontemporal counterparts, thus reducing cache pollution and gaining performance.

Our solution to the prediction problem is to actually measure reuse distance during a run on representative input. We acknowledge that this is not a feasible solution for all programs, but it is for a large class of applications, such as many datacenter servers or repetitive tasks such as stitching images into StreetView panoramas or optical character recognition. Our solution is call context sensitive in that we make an entirely new decision for each static call to a string operation.

More work is needed. On the fundamental side, we need to go deeper into hardware details and understand the regression behavior on particular processor microarchitectures. On the technical side, we need to improve the analysis precision while keeping the overhead low enough not to perturb measurements, and to explore the optimization parameter space to produce the best results.

Both the reuse distance measurement library and the feedback directed compiler optimization pass are fully implemented on a private branch of the GCC compiler suite. Results on large datacenter applications show that the implementation is robust and that initial improvement numbers, while not earth-shattering, are substantial.

References

[1] AMD family 10h. http://en.wikipedia.org/wiki/AMD_K10.

[2] The gcc compiler collection. http://gcc.gnu.org.

[3] Intel Core processors. http://en.wikipedia.org/wiki/Intel_Core_2.

[4] Intel Nehalem. http://en.wikipedia.org/wiki/Nehalem_(microarchitecture).

[5] K. Beyls and E. H. D'Hollander. Reuse distance-based cache hint selection. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 265–274, London, UK, 2002. Springer-Verlag.

[6] K. Beyls and E. H. D'Hollander. Generating cache hints for improved program efficiency. *J. Syst. Archit.*, 51(4):223–250, 2005.

[7] A. Chauhan and C.-Y. Shei. Static reuse distances for locality-based optimizations in matlab. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 295–304, New York, NY, USA, 2010. ACM.

[8] C.-H. Chi and H. Dietz. Unified management of registers and cache using liveness and cache bypass. *SIGPLAN Not.*, 24(7):344–353, 1989.

[9] T. M. Chilimbi. On the stability of temporal data reference profiles. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, page 151, Washington, DC, USA, 2001. IEEE Computer Society.

[10] C. Fang, S. Carr, S. Önder, and Z. Wang. Feedback-directed memory disambiguation through store distance analysis. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 278–287, New York, NY, USA, 2006. ACM.

[11] J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, 1970.

[12] X. Gu, I. Christopher, T. Bai, C. Zhang, and C. Ding. A component model of spatial locality. In *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, pages 99–108, New York, NY, USA, 2009. ACM.

[13] P. Petoumenos, G. Keramidas, and S. Kaxiras. Instruction-based reuse-distance prediction for effective cache management. In *SAMOS'09: Proceedings of the 9th international conference on Systems, architectures, modeling and simulation*, pages 49–58, Piscataway, NJ, USA, 2009. IEEE Press.

[14] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–61, New York, NY, USA, 2007. ACM.

[15] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, pages 93–103, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[16] H. Yang, R. Govindarajan, G. R. Gao, and Z. Hu. Compiler-assisted cache replacement: Problem formulation and performance evaluation. In *16th International Workshop on Languages and Compilers for Parallel Computing(LCPC03)*, pages 131–139, 2003.

[17] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst.*, 31(6):1–39, 2009.