Map Library Design Notes

David Mosberger

January 1996

Abstract

This document describes the current implementation of the *x*-kernel map library. The focus is on its underlying data structures and algorithms. This document does not describe the map library's interface or how it is used. Please refer to the *x*-kernel Programmer's Manual [1] and the *x*-kernel Tutorial [2] for that purpose.

1 Overview

The map tool is the *x*-kernel's central hash-table manager. It provides three kinds of operations: mutating operations (insertion and removal of an element), a lookup operations (lookup and membership test), and an iterator (visit all elements in a hash-table).

The map tool is used mainly to translate protocol-specific or session-specific identifiers of incoming packets into session objects. Thus, on the inbound-side, hash-table lookups are on the critical path of protocol processing and performance is crucial. Typically, each protocol performs one lookup per incoming packet. Locality of network traffic has it that given a session s has received a packet, it is highly probably that s will be the session receiving the following packet. During connection establishment, two lookups are usually performed, the first to detect that no session exists yet and the second to find the so-called enable object needed to create the new session. It is therefore important to keep in mind that testing a hash-table for the presence of an element ("membership test") is also a quite common operation.

Session creation/destruction usually involves inserting/removing an element to/from the hash-table. Thus, any protocol expecting frequent changes in the number of sessions will also cause frequent changes to hash-tables. While not as performance sensitive as lookup operations, it is still important to maintain "reasonable" performance for hash-table modifying operations.

Iterating over the hash-table is seldom on the critical path of protocol processing. As a matter of fact, the *x*-kernel traditionally used map-iteration only during exceptional events, such as system startup or shutdown. TCP, however, is an interesting and important exception: it is commonly believed that maintaining timeouts on a per-session basis would cause excessive overheads. TCP therefore maintains a timer *per-protocol* and internally updates all sessions by traversing the active-session hash-table[1]. In a narrow sense, hash-table iteration is not on the critical path of TCP either. But the fact that it is performed repeatedly implies that it should at least be as little disruptive as possible to other processing. This is particularly important because a properly used hash-table is sparsely populated.

2 Design

Given the above considerations, it is clear that a pure hash-table will not suffice. As a matter of fact, the *x*-kernel map tool consists of four interwoven data-structures:

1. A table (array) of hash-table buckets.

[1] That is, TCP essentially has a rate-controlled time-out mechanism. Independent of the number of sessions, it guarantees that there won't be more than R timeouts per second for some constant value of R.

2. A linked list of map elements (one per hash-table bucket).
3. A one-entry cache remembering the last successful hash-table lookup.
4. A linked-list of potentially non-empty hash-table buckets.

2.1 Hash Table

The actual hash-table is an array of fixed size whose dimension is user-specified at map-creation time. To allow for an efficient computation of the modulo operation, the hash-table size is rounded up to a power of two. Due to the hash algorithm, a hash-table cannot be bigger than 1024 elements. The elements in the hash-table are called hash-table buckets.

2.2 Hash Table Buckets

A linked list of map elements is used to accommodate the case where different map elements hash to the same hash-table bucket. Using a linked list to has the advantage that there is no a priori limit on the number of map elements a map can hold and that at most one hash computation is needed for any map operation. For best performance, these linked lists should never be longer than a few elements, though.

2.3 Lookup Cache

After a successful lookup, the map element that was found is stored in a one-entry cache. The next lookup will check this cached element before doing a regular hash-table lookup. In the common case of one session receiving multiple packets back-to-back, this avoids a hash computation and a potential bucket-list traversal for all except the first lookup.

2.4 Non-Empty Buckets List

A naive implementation of hash-table iteration would simply visit all buckets in a hash-table. Because hash-tables are usually sparse, this can have an adverse effect on cache-behavior because the whole hash-table is brought into the cache without doing any significant computation on the data. To avoid this situation, a linked list of potentially non-empty hash-table buckets is maintained. For efficiency, this list is maintained lazily: whenever a hash-table bucket becomes non-empty, it is ensured that the bucket is on this list. However, when a bucket becomes empty, nothing is done. Removing empty buckets from the list is instead deferred until the next map iteration. This has the desirable effect that in the common case of the absence of any map iterations, the performance penalty of maintaining the linked list is essentially zero (the map will converge to the point where each bucket is on the list so the only overhead is a test in the insertion routine that finds that the bucket is on the list already). Also, lazy evaluation enables the use of a singly linked rather than a doubly linked list. This reduces the size of hash-table buckets from three to two pointers, thus not only reducing space requirements but also allowing for efficient hash-table indexing. Also, insertion and removal from a singly linked list has about half the cost of that of a doubly linked list. In summary, it is essential to avoid a doubly linked list in this case.

3 Implementation Tricks

Implementing the above design is rather straight-forward. There is, however, one point that deserves more attention. A map lookup that results in a cache-lookup should ideally cost no more than a few integer loads and comparisons. To achieve this on a modern RISC processor, this would imply that the keys used to look up a map element would have to be properly aligned. For example, on a 64-bit architecture, a key that is 16 bytes long would have to be 8 byte aligned in order to achieve the ideal of two integer loads and comparisons to check for a cache-hit. However, the map tool cannot make such alignment assumptions: a key that is 16 bytes long could contain 16 characters (guaranteeing 1 byte alignment only) or it could contain two 64-bit integers (guaranteeing an 8 byte alignment). Ideally, the latter case should be recognized at compiled time, leading to the optimal code while the latter case should fall back to a generic (conservative) lookup routine that checks the key-alignment explicitly.

The following pseudo-code illustrates how this can be achieved through an inlined function (*alignof()* is assumed to return the alignment of its argument):

```
static inline void*
lookup(Map m, void *key, int key_size)
{
    MapElement *el = m->cache;

    if (alignof(key) >= MIN(sizeof(long), key_size)) {
        if (el) {                       /* cache is not empty */
            switch (key_size) {
              case 4:
                if (*((int*)key) == *((int*)el->key)) {
                    return el->value;   /* cache hit */
                } /* if */
                break;                  /* cache miss */
              case 8:   ... similar code as above ...
              case 12:  ... similar code as above ...
              case 16:  ... similar code as above ...
              default:  break;
            } /* switch */
        } /* if */
    } /* if */
    /* regular lookup in case of misalignment or cache-miss: */
    return regular_lookup(m, key);
} /* lookup */
```

Provided a compiler can determine the alignment and key-size of a call to the *lookup()* function at compile time, this expands into very efficient and short code. For example, for a key-size of 4 bytes and an alignment of 4, the resulting code would be:

```
static inline void*
lookup(Map m, void *key, int key_size)
{
    MapElement *el = m->cache;

    if (el && *((int*)key) == *((int*)el->key)) {
        return el->value;            /* cache hit */
    } /* if */
    return regular_lookup(m, key);   /* cache-miss: do regular lookup */
} /* lookup */
```

On the other hand, if the compiler cannot determine the alignment of the key or its size at compile time, the function will be expanded fully resulting in a significant code explosion. It is likely that the this code explosion will negate all potential benefits of the code.

What is really desired is *conditional inlining*: function *lookup()* should be expanded inline *only* if the key alignment and size can be determined at compile time. Fortunately this can be accomplished easily provided that the compiler allows to test whether an expression evaluates to a (compile-time) constant. The following pseudo-code shows how this could be done, assuming the compiler provides a builtin function *isconst()* that returns true if and only if its argument evaluates to a compile-time constant:

```
static inline void*
lookup(Map m, void *key, int key_size)
{
    MapElement *el = m->cache;
#   define is_true_const(e)     (isconst(e) && (e) != 0)

    if (is_true_const((alignof(key) >= MIN(sizeof(long), key_size))
                      && key_size % 4 == 0))
    {
        if (el) {                       /* cache is not empty */
            switch (key_size) {
              case 4:
                if (*((int*)key) == *((int*)el->key)) {
                    return el->value;   /* cache hit */
                } /* if */
                break;                  /* cache miss */
              case 8:   ... similar code as above ...
              case 12:  ... similar code as above ...
              case 16:  ... similar code as above ...
              default:  break;
            } /* switch */
        } /* if */
    } /* if */
    /* regular lookup in case of misalignment or cache-miss: */
    return regular_lookup(m, key);
} /* lookup */
```

In the case where the compiler cannot statically determine key-size and alignment, the above function will reduce to the simple call to function *regular_lookup()* as desired.

```
static inline void*
lookup(Map m, void *key, int key_size)
{
    return regular_lookup(m, key);   /* cache-miss: do regular lookup */
} /* lookup */
```

The GNU C compiler provides a builtin function *__builtin_constant_p()* that allows to test whether an expression is a compile-time constant. For technical reasons, this predicate cannot be used in an inlined function, however[2]. It is therefore necessary to express the above as a pre-processor macro, which is somewhat awkward. It does, however, work as expected.

4 Performance

The performance of the *x*-kernel map tool has been measured on a SPARC 4/40 workstation and compared to an older version of the tool that did not support efficient map iteration (i.e., it's implementation of map iteration was not optimized and did not make use of a non-empty bucket list).

The map tool was also measured on an Alpha AXP 3000/600 workstation to illustrate performance effects on a modern, super-scalar, high-clockrate CPU. The older version of the map tool did not accommodate 64-bit workstation, so no comparison with the old two version can be made.

Five different performance parameters where measured. The first four were measured as a function of key-size, while the last one was measured as a function of the number of elements in a map.

1. Time to do a lookup (*mapResolve()*) operation when there is a cache hit.
2. Time to do a lookup (*mapResolve()*) operation when there is a cache miss. This was measured by looking up eight different keys in a hash-table. The reported number is the average execution time per lookup.
3. Average time to insert an element in an otherwise empty bucket and to remove that element (*mapBind()*/*mapRemoveBinding()*). The reported time is the average time of the insert and remove divided by 2.
4. Average time to insert two elements into the same, otherwise empty, bucket and to remove those elements (*mapBind()*/*mapRemoveBinding()*). The reported time is the average time of the two insert and two remove operations divided by 4.
5. Average time to iterate over all elements in a hash-table.

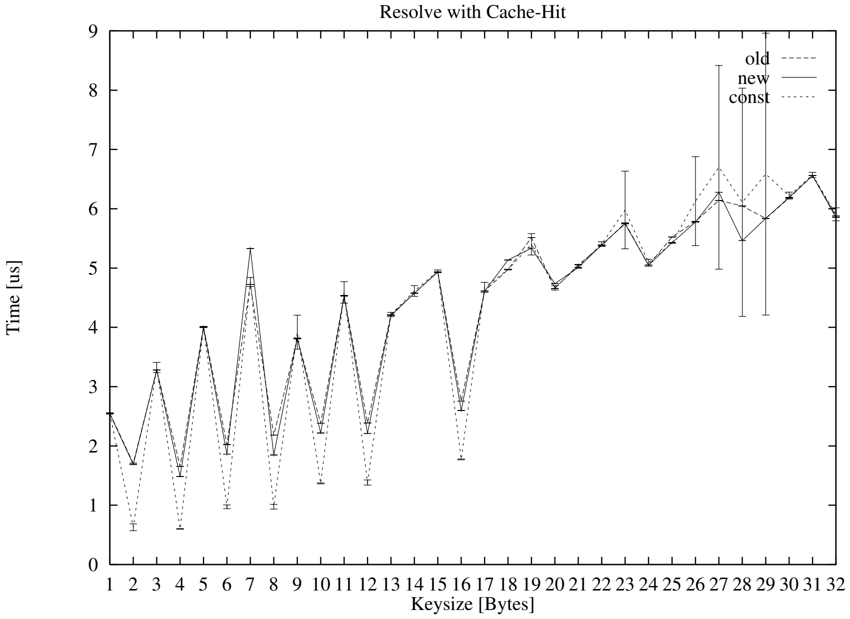
Each test was run for about 0.5 seconds (10^3–10^6 iterations). To get an estimate of variance, this was repeated 10 times and the reported number is the sample mean of the 10 executions. The graphs also show error-bars of ± 1 standard-deviation height.

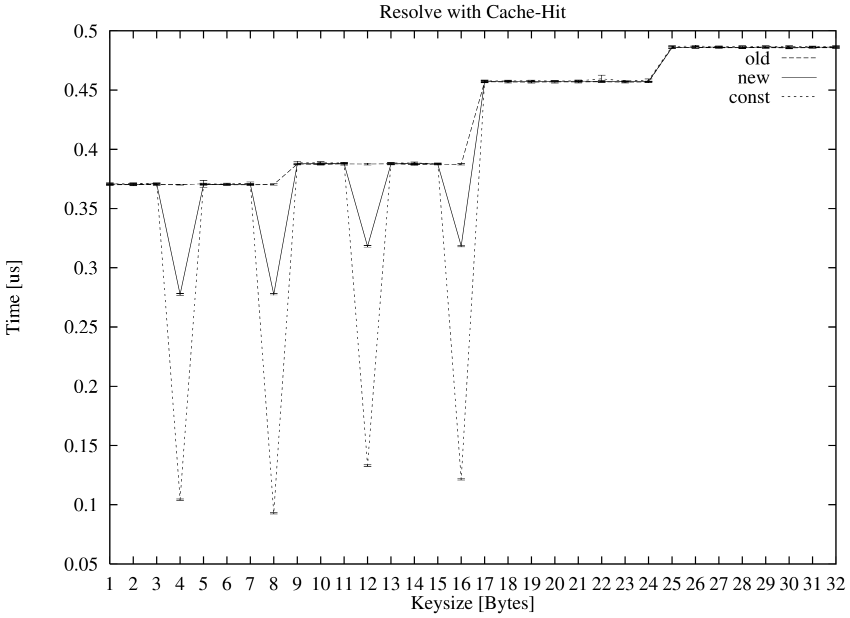
The following table is a description of the labels used in the graphs below:

Label	SPARC 4/40 (left column)	Alpha AXP 3000/600 (right column)
old	*x*-kernel v3.2 map manager	map manager without key-size specific functions
new	new map manager with specialized functions for 2, 4, 6, 8, 10, 12, and 16 byte keys	new map manager with specialized functions for 4, 8, 12, and 16 byte keys
const	same as "new" but using conditional "inlining" for *mapResolve()*	same as "new" but using conditional "inlining" for *mapResolve()*

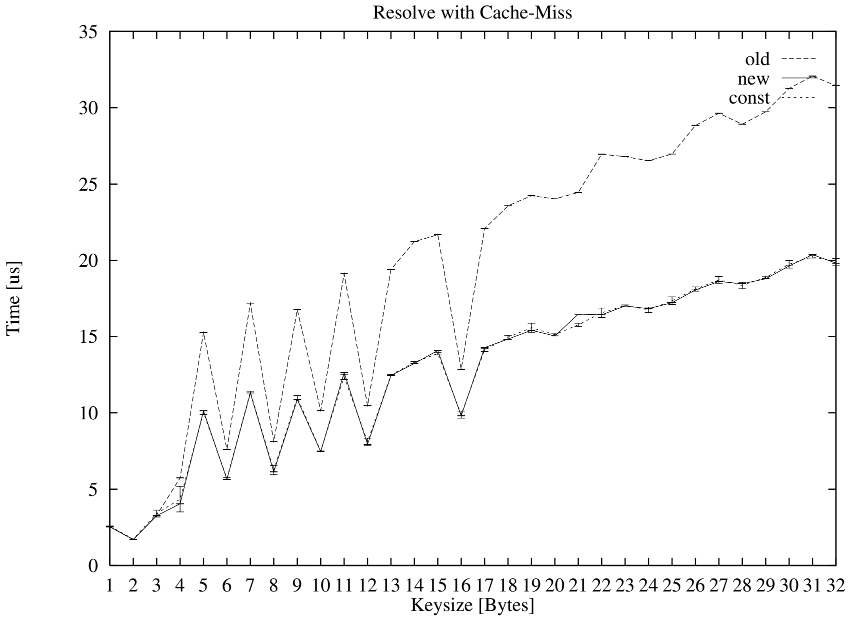
[2] The predicate is evaluated when *translating* the function, not when *expanding* it.

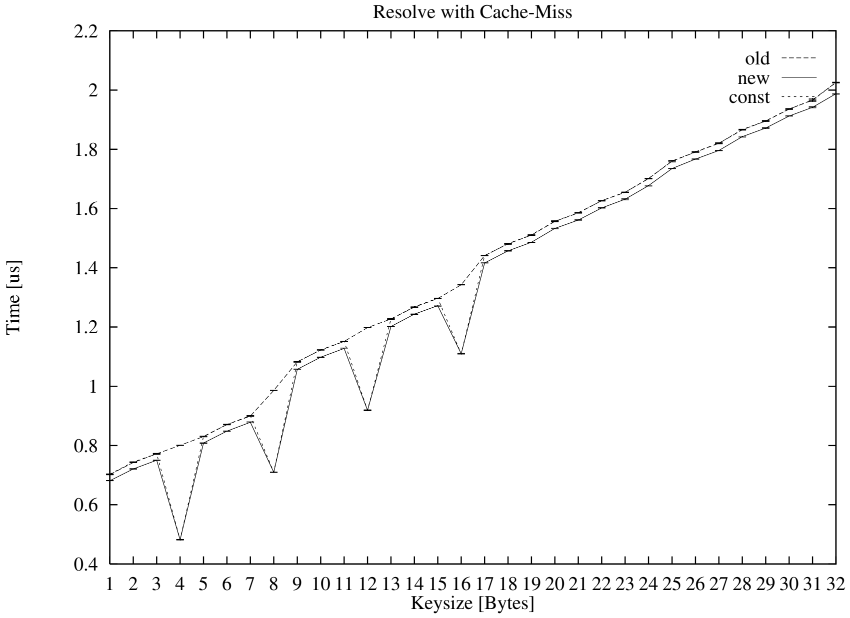
The graph below shows lookup performance with a cache hit. Clearly, key-size specific functions improve performance on both machines. On the Alpha, the use of conditional "inlining" reduces the resolve time by about a factor of three. It is also interesting to compare the performance of the generic (not key-size specific) lookup function of the SPARC to that of the Alpha: the lookup time on the Alpha increases with the number of *longword* accesses, not with the number of *bytes* as on the SPARC. The large jump when going from 16 to 17 bytes is due to an additional data-cache miss.



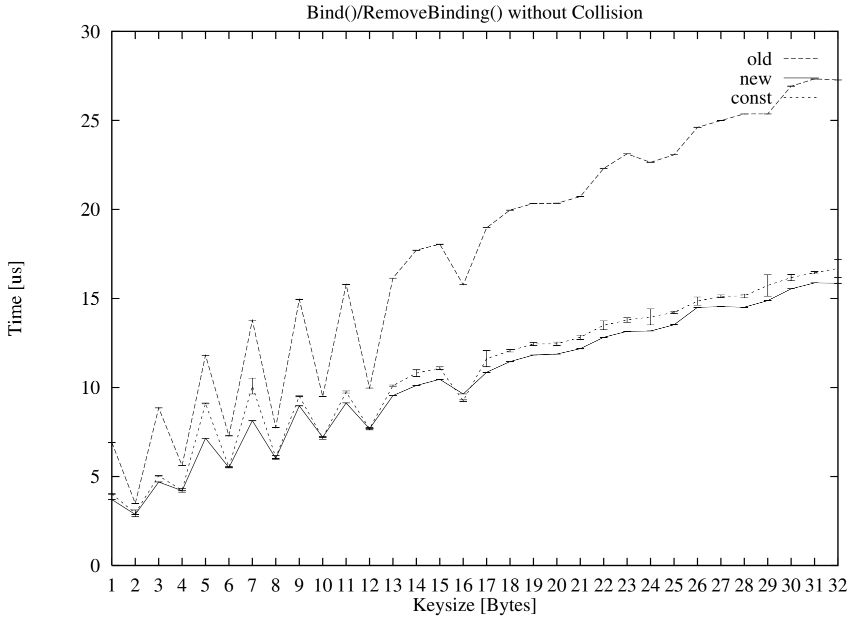


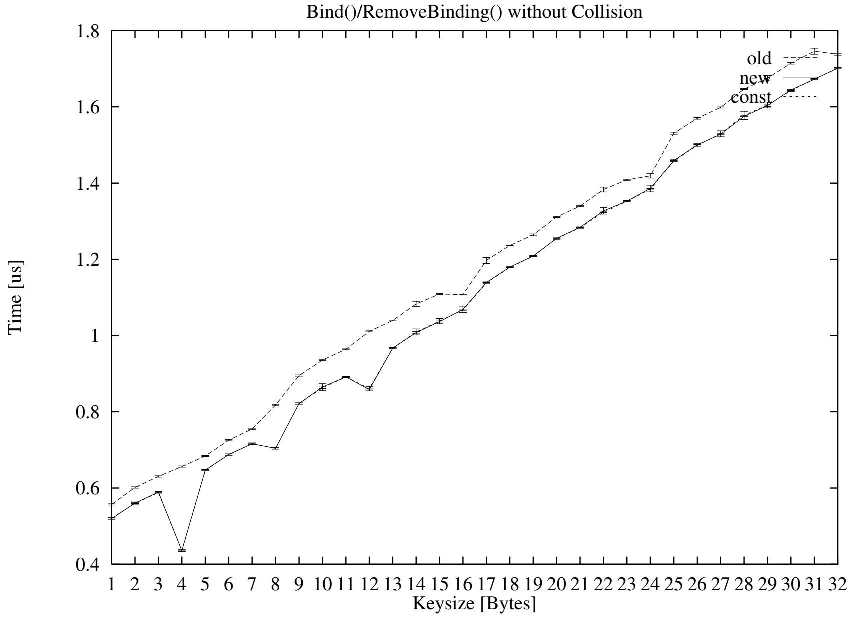
The graph below gives the execution for lookups that miss the cache. There are few surprises other than that, in the SPARC 4/40 graph, the slope of the new version is smaller than the one of the old map tool. There were a number of minor changes between the two versions and it is not immediately clear which one caused the reduction in the slope.



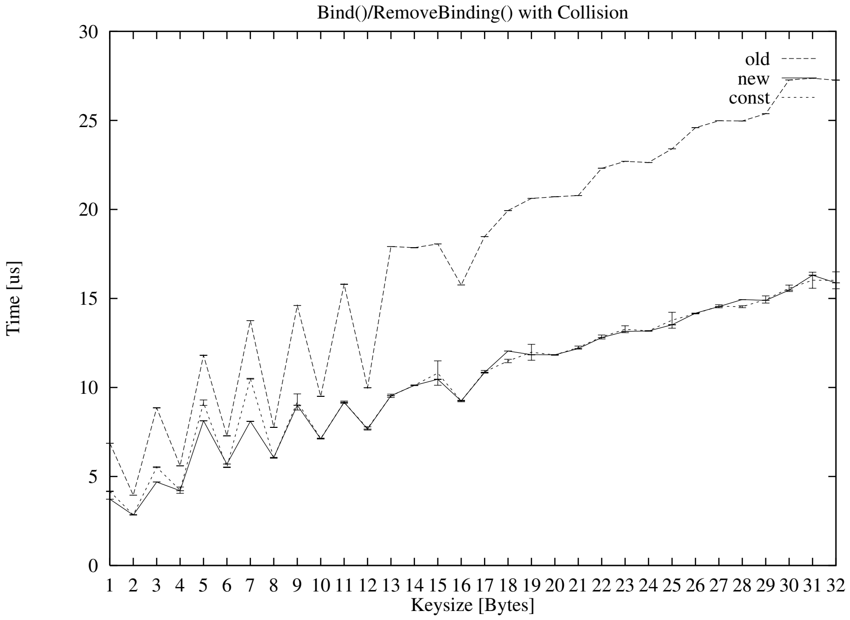


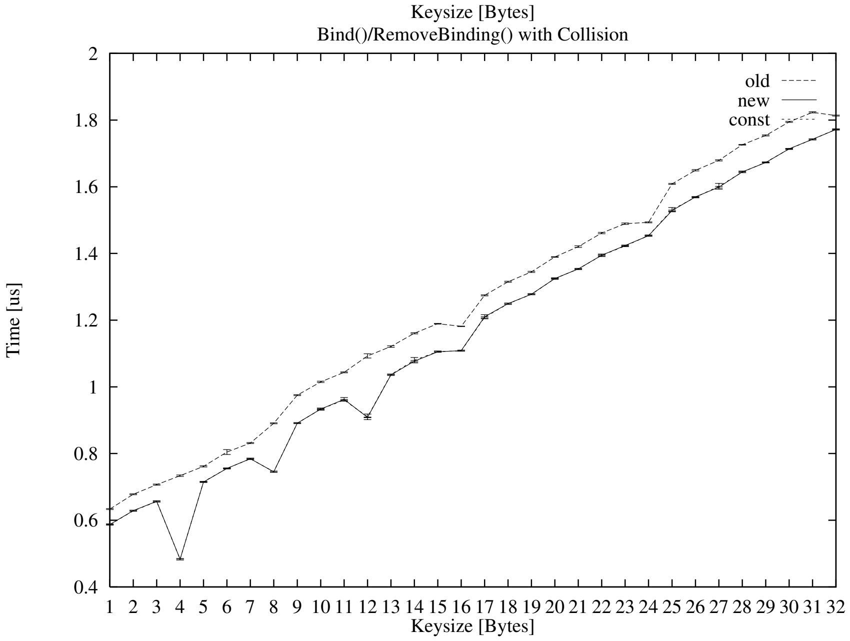
The following graph shows the insert/remove performance. Comparing the old and new/const curves in the left graph, it is apparent that maintaining the linked list of potentially non-empty buckets does not adversely affect performance. As a matter of fact, performance is slightly better, probably because the other minor improvements mentioned before outweigh the overhead due to maintaining a linked list. Key-size specific routines are only slightly faster. In particular on the Alpha it is questionable whether it is worthwhile to have specialized routines for 8, 12, and 16 byte keys.



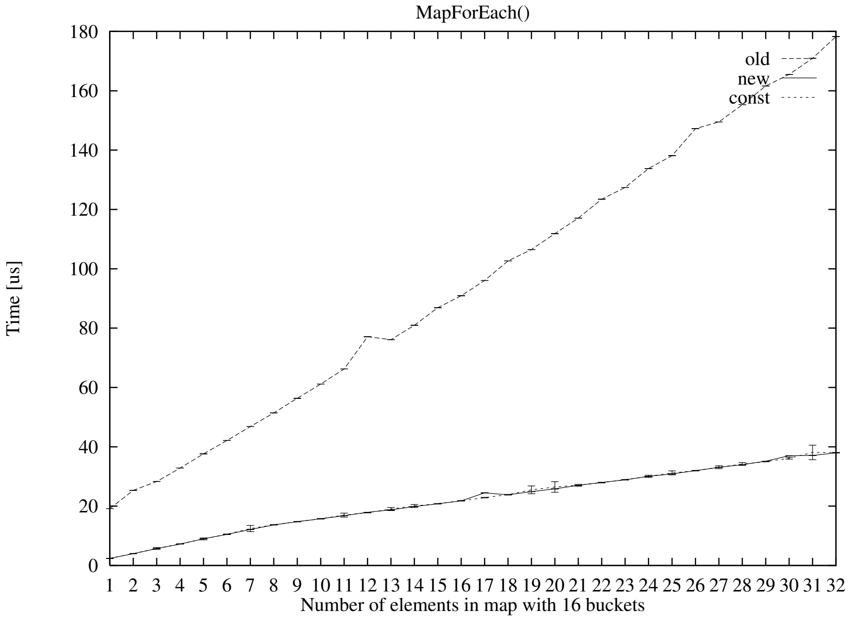


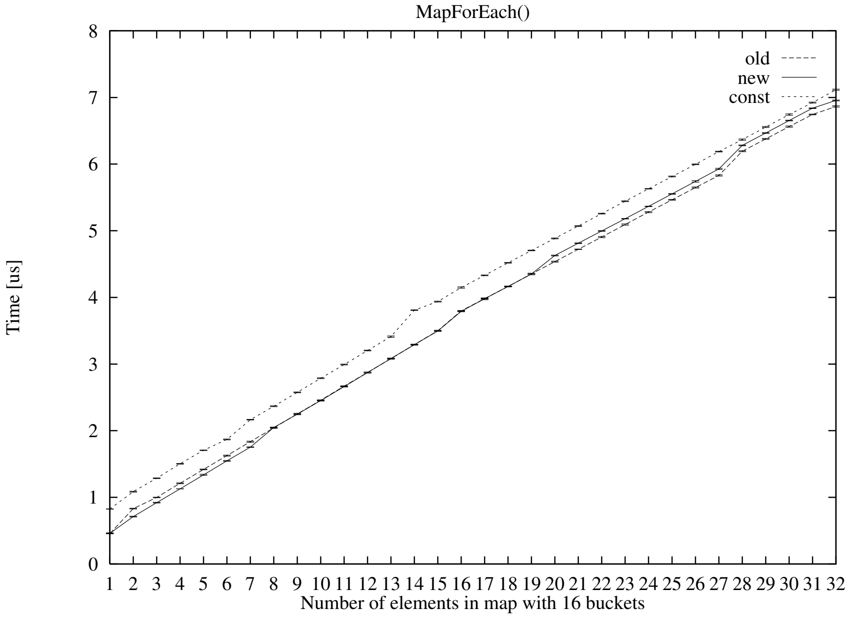
Comparing the following graphs with the ones above, it becomes clear that collisions during insertion/removal have only a minor impact on performance.





The last graph below shows map iteration performance. The significantly smaller intercept of the new/const curve on the SPARC 4/40 graph clearly demonstrates that maintaining the potentially non-empty bucket-list is advantageous. The smaller slope is due to an additional optimization that avoids copying the keys during map iterations.





The above graphs can be misleading because all measurements were done with a warm cache. In particular, key-size specific functions increase code-size dramatically. For example, the Alpha version of the map tool is about 3 KB big without key-size specific functions. When specializing for 4, 8, and 12 byte keys, this code size increases to about 6 KB. To quantify these effects, we also ran tests with a cold instruction cache when iterating across the different key-sizes in the inner loop. In this test, the program with the smaller size should perform better, because there are fewer instruction cache misses. On the Alpha, it was found that the version with key-size specific functions ran about 15% slower. However, this loss of 15% when running with a cold cache compares to a roughly 70% *improvement* when running with a warm cache. Thus, the additional code-size due to key-size specific functions appears to be warranted.

As a summary, the following table gives the approximate slope and intercept of the new map tool's generic operations for both the SPARC and the Alpha. These numbers are rough estimates and not least-squares approximations, but they should be sufficient to give a good idea of the cost of each operation:

Operation	SPARC 4/40 Intercept	SPARC 4/40 Slope	Alpha AXP 3000/600 Intercept	Alpha AXP 3000/600 Slope
lookup (cache hit)	2500 ns	140 ns/B	360 ns	4 ns/B
lookup (cache miss)	9000 ns	360 ns/B	700 ns	42 ns/B
insert/remove	6000 ns	320 ns/B	500 ns	39 ns/B
insert/remove w/collision	6000 ns	320 ns/B	600 ns	39 ns/B
iteration	3000 ns	1290 ns/E	500 ns	210 ns/E

5 Conclusions

The measurements demonstrate that efficient map iteration can be supported without adversely affecting performance of other operations. The use of a lazily maintained linked-list is crucial to this end. With a doubly linked list, efficiency would have been significantly worse, both in terms of execution time and space use.

Fully exploiting compile-time knowledge can improve performance dramatically. On the Alpha, the use of conditional "inlining" reduces time for a lookup that is a cache hit by a factor of three, without affecting code-size or execution time in the case where alignment or key-size information is not available at compile time. However, while conditional inlining is possible with the GNU C compiler, it is cumbersome to do so. A cleaner mechanism would be highly desirable.

References

[1] Network Systems Research Group, Department of Computer Science, University of Arizona. *x-kernel Programmer's Manual (Version 3.3)*, Jan. 1996.

[2] L. L. Peterson, B. S. Davie, and A. C. Bavier. *x-kernel Tutorial*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.