ACM Sigplan LISP Pointers 3,2-4 (April-June 1990),8-22.

Efficient Implementation of Bit-vector Operations in Common Lisp

Henry G. Baker

Nimble Computer Corporation
16231 Meadow Ridge Way
Encino, CA 91436
(818) 986-1436
(818) 986-1360 FAX

This work was supported in part by the U.S. Department of Energy
Contract No. DE-AC03-88ER80663

In this paper we show various techniques for the efficient implementation of the various functions of Common Lisp involving bit-vectors and bit-arrays. Bit-vectors are extremely useful for computing everything from the Sieve of Eratosthenes for finding prime numbers, to the representation of sets and relations, to the implementation of natural language parsers, to the performance of *flow analysis* in an optimizing compiler, to the manipulation of complex communication codes like those used in facsimile machines. However, the efficient manipulation of bit-vectors on modern computers represents a curious point on the spectrum of data processing tasks. On the one hand, the possibility of packing many bits within a single computer word and operating on all of them in parallel offers a potential for speedup not usually available for other types of tasks. On the other hand, the lack of the ability to efficiently manipulate single bits because of addressing schemes tuned for larger objects can actually *reduce* the speed of operating on bits. As a result of these observations, it should be obvious that no simple, automatic techniques such as "in-lining" (procedure integration) or "loop unrolling" of the obvious serial algorithms will produce the kinds of efficiency we are seeking. For these reasons, the efficient implementation of bit-vector operations requires special-case code, and is an interesting challenge in ingenuity and engineering

We show how to organize the wide variety of bit-vector operations in Common Lisp into a few generic types of processing, and show how word-wide parallelism can be utilized for almost all of these operations to achieve speedups over similar character operations of approximately the size of the word. The major exception to our presentation will be Common Lisp's "SEARCH" function for bit-vectors, which can also be speeded up [Baker89], but the subtlety and complexity of its implementation is outside the scope of this paper.

Introduction

The manipulation of bit-vectors and bit-arrays is an important implementation technique of symbolic processing, because bit-vectors allow for the efficient implementation of very complex and relatively unstructured *relations*. Bit-vectors and bit-arrays therefore offer an efficient method for representing and manipulating certain complex relationships that are so unstructured that they cannot be further decomposed, and yet are also so *dense* that the alternative pointer techniques would require more space and time. For example, the LINGOL natural language understanding system [Pratt73] utilizes bit-vectors in the implementation of the Cocke-Kasami-Younger parsing algorithm, and [Baker92] shows the use of bit-vectors for effectively determining "subtypep" for the relatively unstructured type system of Common Lisp.

Symbolic processing can be roughly compared with numeric processing, where there are *dense* matrix techniques and *sparse* matrix techniques. Dense matrix techniques are best suited for relatively small matrices, where the underlying physical model is very closely coupled. Sparse matrix techniques are best suited for larger matrices where the underlying physical model is very loosely coupled. Dense matrix techniques assume that most matrix elements are non-zero, and hence that most operations involving those matrix elements are non-trivial. Sparse matrix techniques try to take advantage of the fact that a large fraction of the matrix elements are zero, and hence operations involving these elements are trivial. However, sparse matrices may still have relatively complex structure, even though a large portion of their elements are zero, so flexible methods must be used to represent and manipulate these sparse matrices. Most sparse matrix techniques, therefore, make use of *pointers* and linked structures which are much more reminiscent of symbolic than of numeric processing.

In symbolic processing, the underlying relationships are almost always sparse, hence the predominance of pointers and linked structures. Nevertheless, there are occasions where the relationships are so unstructured or so dense that pointer techniques are not optimal for symbolic processing. For example, the *transitive closure* of a binary relation is often dense, even if the original relation is not. In cases such as these, bit-vectors and bit-arrays can often be useful, if their implementation is efficient enough. ([Aït-Kaci89] performs transitive closures of object-oriented hierarchies using bit-vector techniques.)

Below is a straight-forward implementation in Common Lisp of the "or-and" multiplication of a bit-matrix by a bit-vector, which is useful in computing the *image set* resulting from the application of a binary relation to a *domain set*.

```
(defun mult-mat-vec (a v)
  "Post-multiply bit-matrix a by bit-vector v."
  (declare (type (array bit 2) a) (bit-vector v))
  (assert (= (array-dimension a 1) (length v))
  (let* ((m (array-dimension a 0))
         (result (make-sequence 'bit-vector m :initial-element 0)))
    (dotimes (i m)
      (when (intersection-test (nmatrix-row a i) v)
        (setf (elt result i) 1)))
    result))

(defun nmatrix-row (a i)
  "Returns a vector displaced to the i'th row of a matrix a."
  (declare (type (array * 2) a))
  (let ((n (array-dimension a 1)))
    (make-array n :displaced-to a :displaced-index-offset (* i n))))

(defun intersection-test (v1 v2)
  "Returns true if bit-vectors intersect one another."
  (declare (bit-vector v1 v2))
  (assert (= (length v1) (length v2))
  (some #'logtest v1 v2))
```

ACM Sigplan LISP Pointers 3,2-4 (April-June 1990),8-22.

Below is an implementation of Warshall's famous "in-place" algorithm for computing the transitive closure of a bit-matrix in Common Lisp. (Unfortunately, due to bugs in the implementation of displaced bit-arrays, this elegant program will not run on some Common Lisp implementations.)

```
(defun nwarshall (a)
  "Implements Warshall's algorithm for the transitive closure."
  "Transitively closes the square bit matrix a in place."
  "[Baase, p.223]"
  (declare (type (array bit 2) a))
  (assert (= (array-dimension a 0) (array-dimension a 1)))
  (let ((n (array-dimension a 0)))
    (dotimes (k n)
      (let ((ak (nmatrix-row a k)))
        (dotimes (i n)
          (let ((ai (nmatrix-row a i)))
            (unless (zerop (bit a i k))
              (bit-ior ai ak t)))))))
  a)
```

Common Lisp provides a number of representations and operations for bit vectors and arrays. The most obvious is the "BIT-VECTOR" datatype, which is a subtype of "ARRAY" in which the elements are restricted to "BIT"'s, i.e., the integers zero and one, and in which the array rank (number of dimensions) is one. Bit-vectors of this type (which we will call "array" bit-vectors) are also Common Lisp "sequences" because they are one-dimensional arrays, so they inherit all of the operations of the Common Lisp sequence functions. However, there are a number of additional Common Lisp functions which operate on all "bit-arrays"—i.e., arrays of bits. Curiously, even though Common Lisp has an abbreviation for "(ARRAY BIT (*))" — "BIT-VECTOR" — and an abbreviation for "(SIMPLE-ARRAY BIT (*))" — "SIMPLE-BIT-VECTOR" — it does *not* provide a standard abbreviation for "(ARRAY BIT *)", for which the obvious abbreviation would be "BIT-ARRAY". Finally, there is an extensive treatment of integers as bit-vectors (which we will call "integer" bit-vectors) through their 2's-complement notation, including "logand", "integer-length", etc. (Logical operations on Maclisp *bignums* were first implemented to support LINGOL [Pratt73] through the BBOOLE package [Baker75], and eventually found their way into the MIT Lisp Machine and Common Lisp.)

Unfortunately for a potential user of bit-vectors in a Common Lisp program, these different representations and function suites have not been rationalized, so it may be difficult to decide *a priori* which representation to use in an application. To make things worse, some Common Lisp vendors provide such inefficient implementations of some of the bit-vector functions that they are useless for real programming.

Before delving into the efficient implementation of the various bit-vector operations, we first give some perspective on the relationships among these different operations through the following chart, which compares the operations available for "integer" bit-vectors with those available for "array" bit-vectors. (See [Eliot89] for a discussion of various representations for sets in Common Lisp; note particularly the chart in his Figure 3.)

Operation	Integer Version	Array Version	Sequence Version
Copy v	unnecessary	make-array :initial-contents v	copy-seq
Make-all-zeros	0	make-array :initial-element 0	make-sequence :initial-element 0
Make-all-ones	-1	make-array :initial-element 1	make-sequence :initial-element 1
Test all zeros	zerop, not ldb-test	N/A	not find 1
Test all ones	eql -1	N/A	not find 0
Size	integer-length	array-dimension(s)	length
Access i'th bit	logbitp	bit, sbit	elt
Boolean functions	boole	N/A	N/A
	logand	bit-and	map #'logand
	logior	bit-ior	map #'logior
	logxor	bit-xor	map #'logxor
	logeqv	bit-eqv	map #'logeqv
	lognand	bit-nand	map #'lognand
	lognor	bit-nor	map #'lognor
	logandc1	bit-andc1	map #'logandc1
	logandc2	bit-andc2	map #'logandc2
	logorc1	bit-orc1	map #'logorc1
	logorc2	bit-orc2	map #'logorc2
	lognot	bit-not	map #'lognot
Intersect test	logtest	N/A	some #'logtest
Subset test	N/A	N/A	every #'<=
Select portion	ldb,ash	make-array :displaced	subseq
Count bits	logcount	N/A	count
Leftmost 1	integer-length	N/A	position
Mask a field	mask-field	N/A	:start, :end
Reverse a bit-vector	N/A	N/A	reverse, nreverse
Concatenate bit-vectors	logior+ash	N/A	concatenate
Compare bit-vectors	=, equal, equalp	equal, equalp	mismatch
Search for subsequence	N/A	N/A	search
Block initialize	N/A	:initial-element ?	fill
"BLT" block transfer	dpb	:initial-contents ?	replace, setf-subseq

Chart of analogous operations for integers, bit-arrays and bit sequences.

We can see by this chart that there is a great deal of overlap between the types of operations available for the various forms of bit-vectors. However, there are also some obvious holes, such as the lack of a quick intersection test for "array" bit-vectors, the lack of a subset test, and the lack of a reverse operation on integers. (A reverse operation on integers would come in handy for the implementation of FFT's, and would also aid in certain other recursive decomposition tasks.)

The largest difference, however, between array-type bit vectors and integer-type bit vectors is in the fact that integer bit-vectors are *functional*, while array bit-vectors allow for *side-effects*. Thus, if `n` different pointers exist to an integer bit-vector, and one applies the "lognot" function to that bit-vector, then this operation only affects the *result*, and not any of the `n` existing pointers to the integer. On the other hand, "bit-not" with a second argument of "t" will flip all of the bits in the bit-vector given as its first argument—not only for the result bit-vector, but also for every other holder of a pointer to that argument bit-vector. Therefore, integer bit-vectors are better for functional programming, while array bit-vectors are better for state-oriented programming.

(Those in the functional programming camp may argue that the functional integer bit-vectors are good enough for all purposes. However, I am not aware of any Lisp compilers which are smart enough to solve the "aggregate update problem" for integer bit-vectors [Schwartz75][Hudak86], and therefore the manipulation of integer bit-vectors results in quite a bit more *cons*'ing than might be used when programming in a more imperative style utilizing array bit-vectors. Without a generational garbage collector [Moon84], the excellent efficiency of the integer bit-vector operations is nullified by the large amount of garbage collection that results from their use.)

Given that we would like a complete complement of bit-vector operations for either style of programming, there are two ways to fill the holes in our chart—add additional operations for both types of bit-vectors and/or add a conversion function which offers the efficient conversion of one type into the other. Since we want to allow for the widest possible flexibility in the use of the Common Lisp language, we suggest doing both—filling the holes in the chart with new functions and providing for the interconvertability of integers and array bit-vectors (see also [Eliot89] for a similar proposal).

(There are often other differences between integer bit-vectors and array bit-vectors due to the vagaries of an implementation. E.g., because Coral Common Lisp on the Macintosh allocates temporary space from the run-time stack for all integer operations, its operations are very fast, but its integers are also limited to 32K bytes, which is quite large enough for most *integer* calculations, but not large enough for many multi-dimensional *bit-array* applications.)

Some languages other than Common Lisp offer bit-vector facilities for bit-vectors which exceed the word length of the underlying hardware. The original Pascal implementation offered "sets" up to size 60, thanks to Control Data. The Ada language [AdaLRM] offers the equivalent of Common Lisp's *bit-not*, *bit-and*, *bit-ior*, *bit-xor*, *equal*, *subseq*, *copy*, *concatenate*, *fill*, and *replace* as builtin capabilities for arbitrary-length bit-vectors. Interestingly enough, efficient Ada compilers must deal with the full complexity of *packed* bit-vectors which can occur with any bit-offset to a word boundary, just as we describe below. Ada also offers one builtin capability not available in Common Lisp—a lexicographic comparison, although such a comparison can easily be emulated using *mismatch*. Of course, the APL language—as the inspiration for many Common Lisp functions—offers many interesting and useful functions for manipulating bit-vectors, some of which are missing in Common Lisp—*scan* and *bit-vector-reduce*—which can often be quite useful.

Efficient Implementation of Sequence and Array Functions on Bit-vectors

Because "array" bit-vectors have a wider variety of operations, are more complex to implement, and are usually implemented with less efficiency than "integer" bit-vectors, we will focus on the efficient implementation of the various operations on array-type bit-vectors. The implementation of "integer" bit-vector operations can then utilize the same machinery, except that automatically extending and contracting the length of these bit-vectors requires some care (see [White86] for a discussion of the efficient implementation of the various algebraic *ring* operations on bignums).

By efficient, we mean that we would like to take advantage of the SIMD-type "word parallelism" that exists on all modern serial computers. In other words, if we can operate on 16-bit words at a time (called "Whiz-Along-By-Words" in [White86]) instead of single bits, we would like to achieve a 16-fold speedup over the fully serial operation operating on only a single bit at a time. Clearly, the larger the word size that we can utilize, the faster these operations should go.

There are some complications, however. Very few computers offer addressing capabilities down to the individual bit, and demand that words be addressed on "word boundaries", so there are the possibilities that incomplete words need to be processed, and that two bits which are to be combined may reside at different bit locations within a word. Both of these complications will require additional processing time, and add substantial complexity to the task of efficient implementation. Yet the payoff of 16-64 X speedups is far too important to ignore.

Virtually all processing of bit-vectors in Common Lisp is stream-like, involving one or more input streams of bits, and zero or one output streams of bits. (One can easily conceive of interesting operations involving multiple output streams of bits, such as a dual complement/reverse complement operation which treats the bit-vectors as sets, but none of the standard Common Lisp bit-vector operations requires more than *one* output bit stream.) Depending upon the operation, these bit-streams are derived from the bit-vector by processing from the start of the vector to the end, or from the end of the vector to the start. The bit-stream may also need to be inverted in the process of being read or written.

The key to the efficient processing of bit-vectors is in the ability to process more than one bit at a time. Therefore, these bit streams need to be able to read or write more than one bit at a time (White calls these chunks "bigits" [White86]). Ideally, one would like to specify the number of bits to be processed—or "byte size"—perhaps 8 bits at a time for one operation, or 32 bits at a time for another operation. The complications of word boundaries occurs when processing multiple bits—sometimes one must process less than the full word width, and sometimes the word to be processed straddles a word boundary.

(By *word*, we mean the smallest unit which can be *efficiently* read and written to memory. This does not necessarily mean the smallest addressable unit, as many modern "CISC" chips can address a 32-bit word occurring on any 8-bit boundary, but the maximum efficiency is reached when 32-bit words are read and written on 32-bit boundaries. By *byte*, we mean some number of bits which has been chosen as convenient for a particular type of processing; our *bytes* do not necessarily have 8 bits.)

Vanilla Common Lisp already has an excellent model for bit streams which could theoretically be used for these purposes—the "binary" file. Binary files in Common Lisp are homogeneous sequences of "bytes", but since the byte size need not remain the same for all uses of the file, the only portable implementation is as a sequence of bits. One could conceive of setting up a binary file associated with each bit-vector which needed to be processed, which would in turn set up a "byte-stream" which could be read or written, queried for end-of-file, etc. Such a stream could easily be set up to read either from the beginning or the end, and to perform on-the-fly complementation.

A true Common Lisp "stream" implementation of the bit-vector operations would be very easy to write code for, and the code would be quite readable. However, such an implementation would be even more inefficient than a straight-forward serial implementation of the same functions. This is because most implementations of streams in Common Lisp systems implement stream operations as function calls, and expect to perform quite a bit of processing per byte read or written, so the relative overhead of the stream operations is not normally objectionable. However, in the implementation of bit-vector operations, such overhead would be intolerable, since the cost of a bit-operation is essentially *free* compared with the cost of reading and writing the data.

However, rather than "throw out the baby with the bath water", we can keep the "byte stream" model of processing, but implement it with some extremely efficient *macros* instead of true Common Lisp streams, and thereby achieve intellectual economy together with high execution speed.

(A problem occurs in trying to implement these byte stream macros in *portable* Common Lisp—there is no efficient way to access the representation of a bit-vector where the actual bits are stored. Coral Common Lisp offers an escape—some open-coded "sub-primitives" which enable us to access 8-bit, 16-bit and 32-bit "bytes" at any offset within an object—irrespective of the type of the object. While using these operations involves delving unnecessarily deeply into the gory details of the representation of bit-vectors, Common Lisp provides no other intermediate level access to the actual bits. The basic nature of these routines is roughly equivalent to *ldb* for a particular subset of sizes and positions, except that we operate on bit-vectors instead of integers. Perhaps a future Common Lisp revision can incorporate such an operation.)

At the same time that we implement a new set of specialized and efficient byte-stream macros, we can also handle one of the complications of word addressing at the same time. While it is obvious that the *last* byte of a sequence may sometimes not be a *full* byte, but only a *partial* byte, there are times when it is more efficient to also process the *first* byte as only a partial byte. This is for "synchronization" reasons, when the processing of the middle (neither first nor last) bytes is thereby speeded up. Since we expect that bit-vectors will sometimes be quite long, it is important that the middle bytes be processed in the most efficient manner. Therefore, in the setting up of such a byte stream, we must separately specify both the *offset* within a byte of the beginning of the first byte as well as the *length* of the first byte.

What different capabilities will be required of our "byte-streams"? We will obviously need to read and write them, but less obvious is the need to "update" them in place. Due to the possibility of the "t" argument in the "bit-xxx" bit-array logical operations, it will be necessary to be able to read, then write back into the same location, the result of an operation. We will also have to read backwards (although we will keep the order of the bits within the byte the same as the forwards order), and we may also want to be able to complement a byte after reading or before writing.

Due to the fact that during any builtin Common Lisp bit-vector operation, at most *one* bit stream will be written (or updated), we can *always* "synchronize" all of the bit streams being read to the one being written. Once this synchronization has been performed, we can always *write* words on word boundaries, and thereby avoid a large amount of complexity and inefficiency, because writing across word boundaries entails more inefficiency than reading across word boundaries. Therefore, only the first and last byte on a write (or update) stream will be less than a full byte, and even that byte will never cross a word boundary. Thus, for implementing just the standard Common Lisp bit-vector functions, we need not implement the most general possible write/update stream.

(Note that even if a computer offered hardware addressing to the single bit, and allowed the hardware reading and writing of words at any bit boundary, it would still be much more efficient for the software to operate on full-word boundaries. This is because while the hardware reduces the software complexity in terms of the number of instructions executed, it is still slower, due to the larger number of cache misses and memory accesses.)

Synchronization of different byte streams is the process of choosing a stream which will be processed (except for its first and last bytes) on a word boundary, and then computing the appropriate offsets for the other streams so that the corresponding bits become aligned for processing. With a single stream, we can always synchronize the stream to itself, while if there are two streams, we can synchronize one to the other. (As we pointed out above, we will always synchronize to the write stream because writing unaligned streams is more expensive than reading unaligned streams.) With the possibility of up to two streams for reading and one stream for writing required to implement some Common Lisp bit-vector operations, we have the possibility that all three streams will have different alignments, so we force both read streams to synchronize to the write stream. We call the case of similar alignments "aligned" in our benchmarks, and the case of different alignments "unaligned".

Many implementors who first approach Common Lisp bit-arrays assume that because the definition of the "bit-xxx" functions (e.g., "bit-and", "bit-ior") require the same rank and the same dimensions, that they will always be "synchronized" in the sense that corresponding bits from the different arrays will occur at the same location within a word. However, this is not the case when one or more of the arrays is "displaced", as the "index-offset" of a displaced array can be any non-negative integer, which can therefore position the first element of the array at any bit within a word. In addition, these functions must also be careful not to disturb bits earlier in the word before the first element or later in the last word, because the underlying simple-array may also be accessible to the user, and bits which are not participating in the "bit-xxx" function should not be affected. The "nwarshall" algorithm shown above depends critically upon the correct implementation of "bit-xxx" functions on displaced arrays.

(We note that most modern workstations already have extremely efficient implementations of the "bit-xxx-t" functions under the guise of the "bitblt" function used to manipulate the two-dimensional bitmap of the screen. In fact, Symbolics Common Lisp utilizes these functions to implement the "bit-xxx" functions, and they are very fast; curiously, Symbolics sequence functions on bit-vectors are abysmally slow (ca.Rel. 7.0). However, *bitblt* (at least as originally envisioned by the Xerox Alto) is a *two*-dimensional operation involving only two arguments, and translating its use to a single dimension and three arguments for use within Common Lisp can require some care. Coral Common Lisp, for example, does not handle the case of *un*aligned arguments correctly. When special purpose *bitblt* hardware is available for use within normal memory, its could provide the effect of an array processor for these Common Lisp functions. One must be careful, however, of potentially long *startup overheads* for these 2-dimensional operations; Common Lisp bit-vectors which are short also want to be efficiently manipulated.)

We note that several proposals [Moon] have been made to modify the nature of the standard Common Lisp bit-vector functions. We feel that our techniques will not be affected in any major way by these proposals, so we have retained the original 1984 Common Lisp semantics [Steele].

Efficient implementation—general comments

We implement the Common Lisp operations on a substrate made up of primitives in which only simple-bit-vectors are supported. We strip out displaced arrays and higher-order dimensions in the upper levels of the implementation. Furthermore, all of the primitives come with explicit "start" and "end" parameters so that arbitrary bit offsets can be represented. While simple-bit-vector operations (without the "start/end" parameters) could be done, and would be slightly simpler, the general case needs to be as efficient as possible, since it occurs more often than one might expect—due to programming styles exhibited by our "nmatrix-row" routine at the start of this paper.

We will now go through the Common Lisp bit-vector functions in turn and indicate how they can be efficiently implemented. All times given are for Coral Common Lisp 1.2 running on a Macintosh Plus with a 16MHz Radius 68020 Accelerator card and 4Mbytes of main memory.

Find, position

Find returns the item being sought if it occurs in the stream, while *position* returns the bit-position of the item found. Both functions return nil if the item is not found within the bit-vector. Both functions are essentially the same process, and differ only in the information returned to the caller. The implementation involves one "read" stream, synchronized to itself. The basic operation of "find1" or "position1" is to find the first non-zero byte within the stream (after appropriate masking on first and last bytes). "Find0" and "position0" utilize a complemented byte stream. Once a non-zero byte is found, *position* determines the first bit within the byte by using a lookup table (on 8-bit bytes). The "with-end" versions read the stream in reverse.

Our implementation of *find/position* in Coral Common Lisp utilizes a byte-size of 16 bits (hence processes the bit-vector 16 bits at a time), and operates in about .5µsec/bit processed.

Our implementation of "find1" in Coral Common Lisp utilizes a byte-size of 16 bits (hence processes the bit-vector 16 bits at a time), and operates in .46µsec/bit processed, while "find0" requires .62µsec/bit processed.

Equal, mismatch

Equal and *mismatch* both compare two bit-vectors, except that *equal* returns t or nil, while *mismatch* returns the first position of the mismatch, or nil. Since the comparison can proceed either from the start or the end of the bit-vector, we require both normal and "from-end" versions of this process.

Equal/mismatch requires two read streams, with the second being synchronized to the first, so that at least one stream is processed on word boundaries. When a non-equality is found, then *mismatch* calls "position1" on the logxor of the two words to find the position of mismatch.

Our implementation of *equal/mismatch* requires .82µsec/bit for aligned streams, and 1.41µsec/bit for unaligned streams.

Fill, :initial-element

Fill fills a subsequence of a given bit-vector with either all 0's or all 1's. Presumably, *make-array* and *make-sequence* call upon fill when the keyword ":initial-element" is present. *Fill* utilizes a single write-stream which is synchronized to itself, and always operates from the start to the end (since every element must be processed, there is no reason for a "from-end" version).

Our implementation of *fill* requires .45µsec/bit processed.

Replace, :initial-contents

Replace replaces a subsequence of one bit-vector with a subsequence of another bit-vector. However, *replace* is defined such that it "still works" even if the two bit-vectors are the same, and the source and destination subsequences overlap. In this case, *replace* acts as though the bits were first transferred from the source to a "scratch" temporary bit-vector, and then transferred to the destination.

Replace is the closest operation Common Lisp has to a "bitblt" or "bit-block-transfer" operation. Since Common Lisp requires that replace "work correctly" even when source and destination overlap, we will require both a normal and a "from-end" version of *replace*, since when we need to move a subsequence of a bit-vector *up* within the same vector, and the destination overlaps the source, we will lose information if we transfer information from the start rather than from the end.

Whether replace-ing from the start or from the end, we always synchronize the source to the destination.

Both versions of *replace* take the same amount of time in our implementation: .78µsec/bit when source and destination are aligned, and 1.3µsec/bit when unaligned.

Bit-not

Bit-not comes in three flavors: *bit-not-t* inverts a bit-vector in-place; *bit-not-2* inverts one bit-vector into another, and *bit-not-new* is *bit-not-2* into a new bit-vector. *Bit-not-t* utilizes a single update stream synchronized to itself, which can always process upwards because all bits must be processed.

Bit-not-2, on the other hand, can have the same kinds of overlap as in replace, and therefore requires both a from-start and a from-end version, with the second being chosen if the vector is being shifted upwards in-place. *Bit-not-2* utilizes both a read and a write stream, with the read stream synchronized to the write-stream. On both *bit-not-t* and *bit-not-2*, the read stream is complemented.

Bit-not-t takes .89µsec/bit, while bit-not-2 takes .99µsec/bit aligned and 1.57µsec/bit unaligned.

Bit-xxx

The *bit-xxx* functions (*bit-and*, *bit-ior*, etc.) come in several forms, due to the possible overlaps of sources and destination. The *bit-xxx-t* functions always put the result back into the first argument, and therefore use both an update stream (for the first argument) and a read stream (for the second argument); the read stream is always synchronized to the update stream. Due to the possibility of overlaps between the first and second arguments, we must provide for both from-start and from-end versions of the *bit-xxx-t* functions, with the from-end versions being used when the destination overlaps with the second argument, and is higher than the second argument.

The *bit-xxx-3* functions have two read streams and a write stream, and come in both from-start and from-end versions. If a source overlaps with the destination, and both sources are *higher* than the destination, then we use the from-end version. The worst case occurs when the destination overlaps both sources, and yet lies in between them. In this case, we can neither process from the beginning or from the end without getting into trouble. Therefore, we either allocate a temporary on the stack or on the heap (depending upon the size), and move the offending source to the temporary location (using replace) and then utilize the *bit-xxx-3* function on the now-tractable situation.

(We note that it is theoretically possible to perform bit-xxx-3 in-place without using additional storage even in the worst case of overlap of both sources and the destination when the destination is between the two sources. However, this algorithm requires a group-theoretic approach to performing the updates in certain cycles, in a manner similar to algorithms which have been proposed for transposing non-square matrices in place without additional storage [Brenner]. Such a method is far too complicated (and slow, since it tends to operate on a single element at a time) for use in a real Common Lisp implementation.)

Bit-and-t takes 1.04µsec/bit for aligned sources and 1.55µsec/bit for unaligned sources. *Bit-and-3* takes 1.1µsec/bit for aligned sources and destination and 2.16µsec/bit for all three unaligned. Add an additional .78µsec/bit for the additional copy for the worst case.

Reverse, nreverse

Reverse returns a *new* bit-vector which is a reversed copy of the given bit-vector, while *nreverse* returns the *same* bit-vector, except that the bits have been reversed in-place.

Despite their similar names, a different technique is required in order to perform *reverse* and *nreverse*. We actually denote the functions "reverse-2" and "reverse-t". *Reverse-2* is called to move and reverse a bit-vector to a destination known not to overlap with the source. One "from-end" stream is used to read, while a normal "from-start" stream is used to write the destination. The read stream is synchronized to the write stream. A 256-element table is used to reverse the bits within an 8-bit byte, and this table is used twice to reverse the bits in a 16-bit word.

Reverse-t requires a subsidiary function which reverses a *byte*-vector in place. We first save the non-participating bits from the first and the last bytes, then reverse the entire byte-vector in place. However, the bits in this vector may not be properly aligned, so we then call *replace* to shift the appropriate bits up or down, if required. Finally, the non-participating bits are restored.

Reverse-2 takes 1.8µsec/bit for word-aligned bit-vectors and 2.4µsec/bit for non-word-aligned bit-vectors. *Reverse-t* takes 1.7µsec/bit for byte-aligned bit-vectors and 3.1µsec/bit for non-byte-aligned bit-vectors.

Intersection Test (proposed name "BIT-DISJOINTP")

An intersection test determines whether the logical "and" of two bit-vectors is non-zero. In other words, whether "(some #'logtest bv1 bv2)" is true. While there is no primitive Common Lisp sequence function to perform an intersection test quickly on array bit-vectors, there *should* be, because it is such a common and important operation. There *is* such an operation for bit-vectors represented as integers called "logtest". (The array language APL, for example, offers the "inner product" capability to express such things.)

We utilize two read streams similar to *equal* or *mismatch*, with the second stream synchronized to the first. Whenever the logical and of the two bytes read is non-zero, then we immediately stop. In the case where the intersection is likely to occur towards the end of the vectors, there may be a requirement for a "from-end" version.

Our test-bit-vector requires .95µsec/bit for aligned bit-vectors, and 1.41µsec/bit for unaligned bit-vectors.

Subset Test (proposed name "BIT-SUBSETP")

This operation is the same as the intersection test, except that the second stream is complemented. Its speed is similar.

Count

Count of a bit-vector is the number of items that occur within the bit-vector, where an item is either 0 or 1. Clearly, *count0* is equal to length *minus count1*, so we can make do with only *count1* without any loss of speed. *Count* is easily implemented using a single bit-stream synchronized to itself. Our implementation uses 16-bit bytes together with two lookups of a 256-byte table which indicates the number of bits within an 8-bit byte.

Our implementation of count in Coral Common Lisp requires .467-1.53 µsec/bit processed.

Remove, remove-duplicates, substitute

These functions produce modified copies of their input sequences. *Remove* returns a copy of the given bit-vector, except that certain elements are left out. The elements removed must all lie within the subsequence delimited by the ":start" and ":end" keywords, and only ":count" of these elements will have been deleted. Furthermore, the ":from-end" parameter indicates whether the elements are to be deleted from the right or the left of the subsequence. Similarly, *remove-duplicates* and *substitute* return copies of their input bit-vectors, except that the subsequence indicated has been modified in the appropriate way. Note that in all three cases, the portions of the input bit-vector prior to ":start" and after ":end" are copied without change. These considerations complicate the descriptions of the operations, but are easy to accommodate, so we ignore the situation with ":start/:end" in our presentation.

Remove-duplicates

Remove-duplicates of a bit-vector first determines whether the bit-vector is all zeros or all ones or mixed; if all zeros, the result is #*0, if all ones the result is #*1, if mixed and the first element is 1 then the result is #*10, otherwise the result is #*01. Therefore, no new low-level bit-vector functions are required for remove-duplicates.

Remove

Performing *remove* on a bit-vector requires counting the bits not to remove, followed by a *make-sequence* to create a new vector of the appropriate length with :initial-element set of the element not removed. *Remove* therefore requires no new low-level bit-vector functions.

Delete, delete-duplicates

We have not implemented these because they are not well-defined. These functions are the "in-place" versions of *remove* and *remove-duplicates*, but since the result may be shorter than the original bit-vector, there is a problem about how this shortening is to be accomplished. If the argument bit-vector is a "simple" bit-vector, then by definition it cannot be displaced, it cannot have a "fill pointer", and it is not "adjustable". Since the only two methods for changing the size of an array in place in Common Lisp are setting its fill pointer or calling "adjust-array", one could conclude that only those sorts of bit-vectors can participate in "delete". However, even when an array has a fill pointer or can be adjusted, another array may share the same "base" array, because another array may have used it as a displacement base, and therefore some thought has to be given to what happens to the non-participating bits when such a vector is shortened—are the other elements shifted down?

For these reasons, we do not perform in-place deletions, and "delete" and "delete-duplicates" are just additional names for "remove" and "remove-duplicates", respectively.

Substitute, nsubstitute

Substitute either changes 1's to 0's or 0's to 1's. In the first case the result is all 0's; in the second, the result is all 1's. Therefore, we can ignore the operation and simply make a new vector of the appropriate length with the appropriate initial-element.

Nsubstitute is easier than substitute, as we need only call "fill" on the affected portion of the bit-vector to make the affected portion either all 0's or all 1's.

Handling :count parameters

The occurrence of a :count parameter in one of the sequence functions *remove*, *substitute*, and *nsubstitute* substantially affects our ability to perform these operations at high speed, since we do not know in advance exactly what subsequence of the original bit-vector will be affected, because we don't know where the count'th item is located. However, by defining a new subsidiary sequence function which is a kind of "inverse count" function, which tells us the location of the n'th item, we can precompute the subsequence affected by the :count parameter. We call this new function "*position+1-count*" because if the n'th item is found, the position+1 of that item is returned as the value. However, if the n'th item is not found (because there were fewer than n items within the sequence), then the *negative* of the count of the sequence is returned. While "position+1-count" appears to be a peculiar function, it provides exactly the information we need. If we wanted to save space in our implementation, we could use this function to subsume the *count* function, since this function provides the same information. However, we keep a separate function for *count*, because the simpler *count* function is also faster than "position+1-count".

Sort, stable-sort, merge

Sorting is like *remove* of 1's concatenated with *remove* of 0's if sorting in ascending order, and the reverse of that if sorting in descending order. In any case, *sort* is trivially and efficiently computed using the other primitives.

Merge of two bit-vectors is slightly more efficient, since we need only find the *first* 1 in each sequence rather than count the 1's before constructing the answer.

Search

The efficient implementation of this function on bit-vectors is covered in [Baker89].

Timing Tests

Our tests were performed on Coral Common Lisp v.1.2 on a 4 Mbyte Macintosh Plus with a 16 MHz Radius 68020 accelerator card, using vectors of 100,000 to 4,000,000 bits. While this hardware configuration is not common, the relative performance should hold for more common configurations.

Operation	CCL simple	CCL displaced	Nimble aligned	Nimble unaligned	Speedup Factor (X faster)
subseq	102	112	—uses replace—		78-144
copy-seq	86	94.2	—uses replace—		66-121
reverse	3.4	84	1.8	2.4	1.4-47
nreverse	2.3	93	1.7	3.1	.74-55
make-seq :initial	45	—	—uses fill—		100
fill	46.8	56	.45	.45	104-124
replace	95.2	117	.78	1.3	73-150
remove nothing	133	142	.933-1.983		67-152
remove everything	94	106	.467-1.533		61-227
remove-duplicates	635	665	.5-.733		866-1330
substitute nothing	135	142	—uses replace—		
substitute everything	140	147	—uses fill—		
find1	90.2	100.3	.46	.46	196-218
find0	90.2	100.3	.62	.62	145-162
position	89.5	98.3	.5	.5	179-197
count	90.8	100.3	.467-1.53	.467-1.53	59-215
mismatch	154	175	.917	1.43	107-191
equal	70.3	93	.85	1.43	49-109
sort	approx. $94 \cdot n \cdot \log_2 n$		$.92 \cdot n$ to $2.0 \cdot n$		103-infinite
intersection test	260	—	.95	1.4	186-274
bit-not-t	.94	bad ans.	.89	.89	1.06
bit-not-2	.94	bad ans.	.99	1.57	.95
bit-and-t	.99	bad ans.	1.04	1.55	.95
bit-and-3	1.0	bad ans.	1.1	2.16	.91
mult-mat-vec $10^3 \times 10^3$	approx. 5 minutes		1 second worst case		
nwarshall 100x100	bad ans.		1.6 seconds worst case		

Timing Comparison of Coral Common Lisp (CCL 1.2) with Nimble bit-vector operations
(All times are in µsec/bit processed on 4Mbyte Mac+ w/16MHz 68020 accelerator)

Summary

We have shown how to efficiently implement the bit-vector operations found in standard Common Lisp. Using the techniques outlined here, we have achieved speedups over existing techniques of upwards of a *factor of one hundred*. Normal techniques such as procedure integration and loop unrolling account for a factor of 5-8, while changing the algorithm to handle multiple bits in parallel accounted for the rest.

This implementation is not the fastest possible for the Macintosh hardware: perhaps an additional factor of 5-8 could be accomplished through a combination of moving from a byte-size of 16 to a byte-size of 32 and recoding in assembly language. On the new pipelined RISC processors, additional factors can be gained through the proper scheduling of operations within the pipeline and defeating the cache (which is no help when accessing FIFO streams). We estimate that the new generation of 25+ mips serial processors now coming on stream should achieve times which are 100-1000 times as fast as these shown for the Macintosh — i.e., flirting with 1 GBOP (giga-bit-operations/second). Finally, bit-vector operations on true SIMD architectures such as the "DAP" and the "Connection Machine" can gain addition factors of 100 on bit-vector operation speeds.

Acknowledgements

We are indebted to Jon White for pointing out the [Eliot89] reference.

ACM Sigplan LISP Pointers 3,2-4 (April-June 1990),8-22.

References

AdaLRM. *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, American National Standards Inst., New York, 1983.

Aït-Kaci, Hassan; Boyer, Robert; Lincoln, Patrick, and Nasr, Roger. "Efficient Implementation of Lattice Operations". *ACM TOPLAS 11*,1 (Jan. 1989),115-146.

Baase, Sara. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Reading, Mass., 1978.

Baker, Henry. ML:HGB;BBOOLE >. MIT AI Lab., 1975.

Baker, Henry. "The Efficient Implementation of Common Lisp's SEARCH Function on Bit-vectors". Internal Memorandum, Nimble Computer Corporation, 1989.

Baker, Henry. "A Decision Procedure for Common Lisp's SUBTYPEP Predicate". *J. Lisp and Symbolic Computation 5* (1992), 157-190.

Brenner, Norman. "Algorithm 467: Matrix Transposition in Place". *CACM 16*, 11 (Nov. 1973),692-694.

Eliot, Christopher R. "Manipulating Sets in Common Lisp". *Lisp Pointers 2*, 3&4 (Jan.-June 1989),5-14.

Hudak, Paul. "A Semantic Model of Reference Counting and its Abstraction". *Proc. 1986 ACM Conf. on Lisp and Funct. Prog.*, August,351-363.

Moon, David. "Garbage collection in a large Lisp system". *Proc. 1984 ACM Conf. on Lisp and Funct. Prog.*,235-246.

Moon, David. "BIT-ARRAY-FUNCTIONS", proposal for modifying the Common Lisp standard. June, 1989.

Pratt, Vaughan R. "A Linguistics Oriented Programming Language". *Proc. IJCAI 3*, (Aug. 1973),372-380.

Schwartz, J.T. "Optimization of very high level languages, Part II: Deducing relationships of inclusion and membership". *Computer Languages 1*,3 (1975),197-218.

Steele, Jr., Guy L. *Common Lisp: The Language*. Digital Press, Burlington, Mass., 1984.

White, Jon L. "Reconfigurable, Retargetable Bignums: A Case Study in Efficient, Portable Lisp System Building". *Proc. 1986 ACM Conf. on Lisp and Funct. Prog.*, August,174-191.