ACM *Lisp Pointers* **V**,3 (July-Sept. 1992), 18-21.

Speeding up the 'Puzzle' Benchmark a 'Bit'

Henry G. Baker

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

We show how Baskett's "Puzzle" benchmark can be speeded up at least an order of magnitude by utilizing *bit-vectors*. Unlike many optimization techniques, the use of bit-vectors enhances the readability and understandability of the code. Since bit-vectors already utilize word-wide parallelism, it is unlikely that parallel processors will be able to solve the problem much faster.

A. INTRODUCTION

Baskett's "Puzzle" benchmark has been used for almost a decade for the evaluation of hardware architectures, and was included in the "Gabriel" suite of Lisp benchmarks [Gabriel85]. "Puzzle" solves a 3-dimensional packing problem by attempting to pack pieces of 4 different types into a 5x5x5 cube. The class of such packing problems is closely related to the "bin-packing" and "knapsack" problems of complexity theory, which are known to be NP-complete [Baase78]. Thus, while it is unlikely that clever programming will allow us to escape the asymptotic exponential behavior of these problems, it can gain us some very real performance improvements.

It is worth studying packing problems because of their ubiquity in the real world. In addition to the obvious examples from business—e.g., freight loading—there are similarities with real problems in biochemical bonding.

The standard version of Puzzle found in the Gabriel benchmark suite is an embarrassment for the Lisp language because its implementation prefers hacking Fortran-like arrays instead of exploiting Common Lisp's rich set of datatypes and functions [Steele90] to solve the problem in a natural and efficient manner. In particular, the standard Gabriel Puzzle does not take advantage of Common Lisp's excellent bit-vector capabilities [Baker90].

We show how the use of bit-vectors in Common Lisp can speed up "Puzzle" by at least an order of magnitude, and these techniques allow us to achieve on a workstation (80860-based OKIstation™) a speed of 8.3 times the Cray-1 on the old benchmark.

B. THE STANDARD PUZZLE BENCHMARK

The standard Gabriel code for Puzzle solves the problem by "pre-rotating" all of the different puzzle pieces, so that these rotations do not have to be performed during the actual combinatorial search. Thus, each piece "class" has a number of different piece "types" which result from different rotations of the pieces. For example, the 4x2x1 piece has 6 distinct rotations, while the 2x2x2 piece has only 1. The 5x5x5 puzzle cube itself is represented as a section of an 8x8x8 cube of Boolean values, while the various piece types (rotations) are represented as a vector of Boolean values which is in correspondence with the representation of the puzzle itself. By embedding a 5x5x5 cube within an 8x8x8 cube, a "border" is created which makes sure that the pieces stay within the 5x5x5 boundaries.

After initialization, the standard Puzzle code linearly searches the puzzle representation for the smallest-numbered empty location. It then tries all of the remaining pieces to see if they can be fit into the puzzle in such a way that this empty location will be filled. If a piece can be fitted, then the code performs a depth-first search for the next empty location and the next piece to be fitted. In many instances, the code will find that it has pieces which cannot be fitted, and initiates a backtrack to remove previously fitted pieces.

The standard code investigates 2,005 placements of the 18 pieces. The speed of the standard code is highly dependent upon the ordering of the pieces, which affects the ordering of the search; a different ordering investigated 10 times as many placements, for example. Interestingly enough, of the 2,005 partially-completed puzzles investigated, 1,565 of them are distinct, meaning that there is little hope of speedup from the "memoization" techniques which have been found effective for other puzzles and games [Bird80] [Baker92]. (The standard implementation of Puzzle investigates surprisingly few configurations, making the ordering of the puzzle pieces appear to have been tampered with to produce shorter searches. See [Beeler84] for deeper analysis of the puzzle itself.)

The standard Gabriel code for Puzzle does not have any errors, but it does show evidence of a hasty conversion from a non-Lisp language. It cannot decide, for example, whether to consistently use 0-origin or 1-origin indexing. The standard code prefers to use the more complex `do` instead of the simpler `dotimes`, and does not utilize macros like `incf` and `decf`. None of these stylistic issues should affect performance, however.

The one obvious stylistic change which could significantly improve performance occurs at the end of the `place` routine where the puzzle is searched for the smallest-numbered empty location. The Common Lisp `position` "sequence" function could be used for this purpose, and it could conceivably improve performance due to its presumably high level of optimization.

C. SPEEDING UP PUZZLE A "BIT"

Since Puzzle operates on arrays whose elements are always Boolean values of `t` or `nil`, Puzzle cries out for a bit-vector implementation. The `puzzle` vector itself is an excellent candidate, as are the rows of the `p` array used in `fit`. In fact, `fit` itself is essentially a verbose version of the Common Lisp `logtest` function which operates on bit-vectors represented by large integers. Furthermore, the search for an empty bit location at the end of `place` is analogous to the search in the Common Lisp `integer-length` function for the highest numbered bit.

The decision to use bit-vectors in Common Lisp is complicated by the fact that there are at least 3 different bit-vector models—bit-vectors represented by bit-arrays, bit-vectors represented by bit "sequences", and bit-vectors represented by large binary integers. Bit-vectors represented as large binary integers are *functional*, in that such a bit-vector cannot have a single bit changed, but the whole bit-vector must be copied. Bit-vectors represented as bit-arrays can be manipulated in a destructive (imperative) manner, and *may* therefore have an advantage in reducing garbage collector overhead, but have some less obvious defects. Bit-vectors represented by binary integers are only as large as they need to be to represent the highest-numbered bit, while bit-vectors represented by bit-arrays always occupy their full allocated length; this difference in sizes can result in higher performance for binary integers if the integers are often much smaller than the maximum size.

We here discuss only a binary integer version of Puzzle. We suspect that a bit-array version of Puzzle can be more efficient than one utilizing binary integer bit-vectors, but the lack of a quick intersection test (e.g., `logtest` for binary integers) may scuttle this hope.

Our representation of the puzzle itself is a single binary integer, while the puzzle piece types are also binary integers. A straight-forward translation would convert the `puzzle` global vector into a global variable holding a large binary integer, and convert the global 2-dimensional `p` array into a global vector of large binary integers. Such a translation would also place and remove pieces from the puzzle at a high rate; a better solution is to remember the previous state of the puzzle, which is trivially done by making `puzzle` into a parameter of the `trial` function. The function `fit` disappears entirely, to be replaced by `logtest`, while `place` is accomplished by `logxor`. If we move the empty location search from the non-existent `place` function to the top of the recursively called `trial` function, then we no longer have to pass the address of this empty location as a parameter to `trial`. This search can be accomplished by the following code, but we will rearrange things so that a less complex solution can be obtained.

```
(defun find-lowest-0-bit (bv)
  (declare (type (integer 0 *) bv))
  (1- (integer-length (logandc2 (1+ bv) bv))))
```

The problem with this solution is that we must construct 2 temporary binary integers before using `integer-length` to do the search, instead of simply searching the bit-vector directly. If we simply reversed `puzzle` end-for-end, then we could search from the *end* of the bit-vector for an empty position instead of searching from the beginning. The Common Lisp function `integer-length` can be used to search for the highest "0" bit, *so long as the integer is negative*. We can thus use the following code:

```
(defun find-highest-0-bit (bv)
  (declare (type (integer * (0)) bv))
  (1- (integer-length bv)))
```

It turns out that we don't have to actually reverse the puzzle end-for-end, since the original choice to search for the lowest-numbered empty position instead of the highest-numbered was arbitrary, and the same combinatorial search is performed either way. The advantage of the new ordering is that the number of integer constructions is reduced.

D. PRECOMPUTING SHIFTS

The standard code for Puzzle precomputes rotations for the pieces, but not shifts. This is probably because there is little cost to a shift using the standard algorithm. When utilizing bit-vectors, however, the cost of shifting can easily exceed any savings from parallel operations on multiple bits. We therefore extended `definepiece` to precompute shifts as well as rotations; this change has the additional benefit that the "border" is no longer needed, so we can represent `puzzle` with length-125 instead of length-512 bit-vectors. However, there are far more shifts than rotations; a single rotation of a 4x2x1 piece can be shifted into 2x4x5=40 different positions, and there are 240 different shifts for all rotations of the 4x2x1 piece "class". The number of piece "types" must therefore be increased from 13 to 769.

With so many different piece "types" to consider, our algorithm should run far more slowly than the standard code. (Indeed, a preliminary version of this kind had to be cut off before it finished.) However, since we are trying to fill the highest-numbered empty position, it is obvious that we should index these different piece "types" by their largest bit-position, so that `trial` will consider only the piece types that can actually fill the empty position. Our index is thus a vector of lists of piece types, which vector is indexed by the 125 positions in the puzzle; the elements in its last list, for example, give the piece types which can be used to fill the last puzzle position. It so happens that the maximum number of elements in any of these lists is 13, so we could have arranged this information as a 125x13 array. But we are programming in Lisp! Hence, we will keep the vector-of-lists representation for our index. At the cost of one additional vector location, we can use a 1-origin for our index and thereby eliminate the decrement which would otherwise surround every call to `integer-length`.

```
(defun trial (puzzle)
  (incf *kount*)
  (if (eql puzzle -1) t
    (let* ((j (integer-length puzzle)))
      (dolist (i (aref index j) nil)
        (let* ((classi (aref class i)) (ocnt (aref piececount classi)))
          (unless (zerop ocnt)
            (let* ((pi (aref p i)))
              (setf (aref piececount classi) (1- ocnt))
              (unless (logtest puzzle pi)
                (when (trial (logxor puzzle pi))
                  (format t "Piece ~4D." i) (return-from trial t)))
              (setf (aref piececount classi) ocnt))))))))
```

Given the large increase in types from 13 to 769, it is not obvious whether the 769 piece types and their index can be built fast enough. If we use code similar to that in the standard `definepiece`, our approach would founder on the large amount of large integer construction ("bignum consing") required, because the standard code builds up the bit-vectors one bit at a time.

The proper way to build the bit-vector patterns is to build them up recursively on their 3 dimensions. Since Puzzle utilizes Fortran-like indexing (fastest-varying-first), we first build the patterns along the `i` dimension, then the `j` dimension, and finally the `k` dimension. Once they are built, they can then be shifted into position *in toto*.

```
(defun definepiece (iclass ii jj kk)                  ;uses 0-origin indexing.
  (let* ((iimask (1- (ash 1 ii))) (jjmask 0) (kkmask 0))
    (dotimes (j jj) (setf (ldb (byte 5 (* 5 j)) jjmask) iimask))
    (dotimes (k kk) (setf (ldb (byte 25 (* 25 k)) kkmask) jjmask))
    (dotimes (ioff (- 6 ii))
      (dotimes (joff (- 6 jj))
        (dotimes (koff (- 6 kk))
          (let* ((mask (ash kkmask (+ (* (+ (* koff 5) joff) 5) ioff))))
            (push *iii* (aref index (integer-length mask)))
            (setf (aref p *iii*) mask)
            (setf (aref class *iii*) iclass)
            (incf *iii*)))))))
```

E. RESULTS

Our changes have improved the performance of Puzzle by more than an order of magnitude. On a 40MHz 80860-based OKIstation, we achieve a Puzzle time of 0.12 second,[1] which is 8.3 X faster than the Cray-1 on the old benchmark. In fact, the new Puzzle runs on our Apple Mac+ w/68020 accelerator only 2.5 X slower than the Cray-1 on the original benchmark, and shows our Mac+ to be 200 X faster than the 750 NIL implementation [Gabriel85]! Yet the new Puzzle performs exactly the same number of placements as the original Puzzle, thus demonstrating that it explores the search space in the same way.

The new Puzzle can almost certainly be speeded up with better implementations of Common Lisp bit-vector operations on binary integers. In particular, mask construction using `ldb` and `dpb` must be highly optimized, and therefore our version should run well on machines like the Symbolics Ivory.

There are two types of potential parallelism in Puzzle—bit-parallelism during `logtest` and `logxor`, and parallelism due to multiple searches in parallel. We believe that all of the bit-parallelism has already been tapped by

[1]There are some additional minor optimizations, including additional declarations, in this version.

using bit-vectors, and it is not clear how multiple searches can be efficiently organized and still be faster than 0.12 second.

We have shown that efficiency and elegance in this algorithm are not unrelated.

F. REFERENCES

Anderson, J.Wayne, *et al*. "Implementing and Optimizing Lisp for the Cray". *IEEE Software* (July 1987),74-83.

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

Baker, H.G. "Efficient Implementation of Bit-vector Operations in Common Lisp". ACM *Lisp Pointers 3*,2-3-4 (April-June 1990), 8-22.

Baker, H.G. "The Gabriel 'Triangle' Benchmark at Warp Speed". ACM *Lisp Pointers V*,3 (Jul-Sep 1992), 15-17.

Beeler, M. "Beyond the Baskett Benchmark". ACM *Comput. Arch. News 12*,1 (Mar. 1984), 20-31.

Bird, R.S. "Tabulation Techniques for Recursive Programs". ACM *Comp. Surv. 12*,4 (Dec. 1980),403-417.

Gabriel, R.P. *Performance and Evaluation of Lisp Systems*. MIT Press, Camb., MA, 1985.

Steele, Guy L. *Common Lisp, The Language; 2nd Ed*. Digital Press, Bedford, MA, 1990,1029p.

