The Boyer Benchmark at Warp Speed

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 501-4956 (818) 986-1360 (FAX)

We show how to speed up the Boyer Benchmark by an order of magnitude (46 X faster than the Cray-1) on a Common Lisp system (80860-based OKIstation) using better programming techniques. The resulting program fits nicely within next generation on-chip caches and kills almost all potential parallelism, thus becoming worthless as a general-purpose Lisp benchmark.

A. INTRODUCTION

The Boyer benchmark is one of the most durable benchmarks in the Lisp landscape. It was already being used circa 1977 to compare various Lisp systems, and it continues to be one of the few Gabriel benchmarks [Gabriel85] whose performance correlates well with "general-purpose" Lisp programming. It has been used to study the capabilities of generational garbage collectors [Moon84], provide data for the evaluation of the Symbolics 3620/3630/3650 ("G") Machine architecture, and provide grist for a parallel Lisp mill [Osborne89].

Nevertheless, the Boyer benchmark is fatally flawed, primarily because the code given in [Gabriel85] is not nearly the best method to solve the problem, and hence any attempt to speed up the Gabriel version of the code speeds up an artificial problem.

We show that the standard technique of *memoizing* [Bird80] reduces the number of conses to 1% of those in the standard code, and with the additional technique of *hash consing* [Goto74], the total space required by all consing is satisfied by a total of 2219 cons cells.

B. DEBUGGING THE BOYER BENCHMARK

The code given in [Gabriel85] has several errors. First, the function `assq` is not a standard Common Lisp function, but must be replaced by the following macro:

```
(defmacro assq (x y) `(assoc ,x ,y :test #'eq))
```

Second, the definitions of `falsep` and `truep` are incorrect, because the MacLisp `member` uses `equal` as its testing predicate, while Common Lisp uses `eql`. (With this bug, `tautologyp` cannot prove that the given expression is a tautology.) Thus, we must utilize the following code:

```
(defun falsep (x lst)
  (or (equal x '(f)) (member x lst :test #'equal)))

(defun truep (x lst)
  (or (equal x '(t)) (member x lst :test #'equal)))
```

Third, some un-Common Lisp systems may not provide a property list for `nil`, which will cause the code to blow up when it attempts to manipulate `nil`'s property list. Fourth, the `prog` macro in `test` should be changed into a `let` special form in order to report the answer back to the user. (We note that none of these bug fixes has any significant effect on the execution time of the benchmark.)

C. MEMOIZING THE BOYER BENCHMARK

The `rewrite` function of the Boyer benchmark cries out for memoization [Bird80] [Keller86], because it continually rewrites the same argument. Although there are 91,024 calls to `rewrite` in the standard benchmark, only 205 of them are distinct, for a ratio of 1:444. Memoizing `rewrite` reduces the number of calls to `cons` from 226,464 to 2,219, for a ratio of 1:102. This change alone can produce an order of magnitude improvement. Below is the new code for `rewrite`:

```
(defparameter *memo-table* (make-hash-table :test #'equal))

(defun rewrite (term)
  (cond ((atom term) term)
        ((gethash term *memo-table*))
        (t (setf (gethash term *memo-table*)
             (rewrite-with-lemmas
              `(,(car term) ,@(rewrite-args (cdr term)))
              (get (car term) 'lemmas))))))
```

D. HASH-CONSING

Memoizing `rewrite` produces the largest performance improvement, but performance can be further improved by using *hash consing*. Hash consing was invented by Ershov in 1958 [Ershov58] to speed up the recognition of common subexpressions, and popularized by Goto [Goto74] for use in a symbolic algebra system.

Hash consing builds up an expression recursively from atoms by use of the `hcons` function, which is a true mathematical function in the sense that given the same pair of arguments, the result is always `eq`. Hash consing can be simulated in Common Lisp by using an `equal` hash table, but such an implementation is at least an order of magnitude less efficient than a direct implementation, because the hash table never needs to recurse beyond the `car` and `cdr` in order to determine equality, and the value is always the key itself. As a result, the effectiveness of the hash consing optimization of the Boyer benchmark is dependent upon the efficiency of the `hcons` function.

```
(defparameter *cons-table* (make-hash-table :test #'equal))

(defparameter *a-cons-cell* (list nil))          ;holding area for spare cons.

(defun hcons (x y)                          ;not very fast; only for illustration.
  (setf (car *a-cons-cell*) x (cdr *a-cons-cell*) y)         ;don't cons yet.
  (let ((z (gethash *a-cons-cell* *cons-table*)))
    (or z (prog1 (setf (gethash *a-cons-cell* *cons-table*) *a-cons-cell*)
            (setq *a-cons-cell* (list nil))))))           ;do next cons here.
```

Notice that hash consing saves both space and time. Space is saved, because a subtree is never duplicated in memory. Time is saved, because any `equal` tests reduce to `eq` tests, and the original Boyer benchmark performs 1,403 `equal` tests. More importantly, however, the memo hash table itself can now use the more efficient `eq` test instead of an `equal` test.

```
(defparameter *memo-table* (make-hash-table :test #'eq))
```

The effectiveness of hash consing for the Boyer benchmark can be seen by considering the result of `rewrite`. In the original Boyer benchmark, the answer consumes 48,139 cons cells, but if one copies the answer using `hcons`, the answer consumes only 146 distinct cons cells (!), for a ratio of 1:330. With hash consing in effect for both `setup` and `test` in the Boyer benchmark, the total number of cons cells consumed is 2,216, for a ratio of 1:102.

E. RESULTS

The memoization optimization improves Boyer by an order of magnitude, and the hash-consing optimization improves it by an additional order of magnitude. With but a day's work on the 40MHz 80860-based OKIstation, we achieved a Boyer time of 0.04 second, which is 46 X faster than the Cray-1 on the old benchmark. (We "deep-copy" the result of `apply-subst` inside `test` in order to eliminate sharing in the argument to `tautp` and keep within the spirit of the original benchmark.) Unfortunately, our reductions in consing and memory utilization have destroyed the usefulness of this benchmark, because the memoized Boyer benchmark will fit entirely within the on-chip cache of the next generation microprocessors. Even without memoization, the use of `hcons` dramatically improves the locality of reference so that the entire benchmark fits within the cache, and should produce a significant speedup.

(We note that Boyer himself achieves speedups of 2 orders of magnitude on rewriting, by using rule compilation techniques [Boyer86]; Peter Deutsch [Deutsch73] also utilized memoizing and hash consing for similar effect.)

The success of memoization shows that almost all of the parallel processes in the Multilisp implementation of Boyer [Osborne89] are performing identially the same computation, hence the Boyer benchmark is worthless for estimating parallelism in real (i.e., highly optimized) applications. In a sense, memoization is the most powerful of all parallel programming techniques, because one processor simulates the execution of many processes with a single execution!

F. REFERENCES

Bird, R.S. "Tabulation Techniques for Recursive Programs". *ACM Comp. Surv. 12*,4 (Dec. 1980),403-417.
Boyer, R. "Rewrite Rule Compilation". TR AI-194-86-P, M.C.C., Austin, TX, 1986.
Deutsch, L. Peter. "An Interactive Program Verifier". Xerox PARC TR CSL-73-1, 1973.
Ershov, A.P. "On Programming of Arithmetic Operations". *Doklady, AN USSR 118*,3 (1958),427-430, transl. Friedman, M.D., *CACM 1*,8 (Aug. 1958),3-6.
Gabriel, R.P. *Performance and Evaluation of Lisp Systems.* MIT Press, Camb., MA, 1985.
Goto, Eiichi. "Monocopy and Associative Algorithms in Extended Lisp". TR. 74-03, U. Tokyo, 1974.
Keller, R.M., and Sleep, M.R. "Applicative Caching". *ACM TOPLAS 8*,1 (Jan. 1986),88-108.
Osborne, R.B. *Speculative Computation in Multilisp.* MIT/LCS/TR-464, Dec. 1989.
Moon, D.A. "Garbage Collection in a Large Lisp System". *ACM Lisp & Funct. Prog. Conf.*, Aug., 1984.
Steele, Guy L. *Common Lisp, The Language; 2nd Ed.* Digital Press, Bedford, MA, 1990,1029p.