# An Efficient Representation for Sparse Sets

PRESTON BRIGGS and LINDA TORCZON
Rice University

Sets are a fundamental abstraction widely used in programming. Many representations are possible, each offering different advantages. We describe a representation that supports constant-time implementations of *clear-set*, *add-member*, and *delete-member*. Additionally, it supports an efficient *forall* iterator, allowing enumeration of all the members of a set in time proportional to the cardinality of the set.

We present detailed comparisons of the costs of operations on our representation and on a bit vector representation. Additionally, we give experimental results showing the effectiveness of our representation in a practical application: construction of an interference graph for use during graph-coloring register allocation.

While this representation was developed to solve a specific problem arising in register allocation, we have found it useful throughout our work, especially when implementing efficient analysis techniques for large programs. However, the new representation is not a panacea. The operations required for a particular set should be carefully considered before this representation, or any other representation, is chosen.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures; E.2 [**Data**]: Data Storage Representations

General Terms: Algorithms

Additional Key Words and Phrases: Compiler implementation, register allocation, set representations, set operations

## 1. INTRODUCTION

Sets are a fundamental abstraction widely used in programming. Many representations are possible, each offering different advantages. The choice of a "best" representation for a given set depends on the operations required, their cost in both time and space, and the relative frequency of those operations.

As a part of our exploration of register allocation via graph coloring [Briggs 1992; Chaitin et al. 1981], we looked for good implementations for each phase of the allocator. To quickly construct the interference graph, we needed a set representation that supported efficient implementations of the operations *clear-set*, *add-member*, and *delete-member*, as well as an iterator, *forall*, that enumerated the members.

This work has been supported by ARPA through ONR grant N00014-91-J-1989 and by the IBM Corporation.
Authors' address: Department of Computer Science, Rice University, Houston, TX 77251-1892.

Bit vector representations, a traditional choice for dataflow analysis, are not efficient in this case. They require $O(u)$ time to clear and $O(u)$ time to iterate over all the members, where $u$ represents the size of the universe. These requirements are especially distressing in applications like ours, where the number of elements in the set is small relative to the size of the universe.

Inspired by a memorable homework problem of Aho et al. [1974, Problem 2.12], we developed a *sparse* set representation that supports all the required operations efficiently. We have since implemented several versions of the new representation and find it widely useful in our work, particularly in the implementation of efficient analysis techniques for large routines [Cytron et al. 1991].

In the next section, we introduce the sparse representation and discuss its implementation. In Section 3, we consider the asymptotic complexity of the sparse representation and compare the costs of using it versus a bit vector representation. In Section 4, we discuss applications of the sparse representation in the context of our optimizer. We conclude with a review of related work and a brief summary.

## 2. THE SET REPRESENTATION

In our problems, we usually manipulate sets with a fixed-size universe $U$, where $u$ will represent the number of elements in the universe (e.g., the variables in a program or the compiler temporaries in a routine). For convenience, we map elements in $U$ to the integers 0 through $u - 1$. Note that the restriction to a fixed-size universe is significant—it restricts the use of sparse sets to *off-line* algorithms [Aho et al. 1994, p. 109]. While more flexible alternatives are available for use with on-line algorithms, they seem to be necessarily less efficient. Of course, bit vectors also require a fixed-size universe.

Our sparse-set representation has three components: two vectors, each $u$ elements long, and a scalar that records the number of members in the set. Figure 1 illustrates an example set with a single member 3. The scalar members delimits the initialized portion of the dense vector. Initialized elements in dense point to members in the sparse vector, which point back into dense. The values of other elements in dense and sparse are unimportant; *they are never initialized.* If a number $k$ is a member of a set, it must satisfy two conditions:

$$0 \leq \text{sparse}[k] < \text{members}$$

and

$$\text{dense}[\text{sparse}[k]] = k.$$

Therefore, the C code for a membership test might look like this:

```
int member(Set *s, unsigned int k)
{
  unsigned int a = s → sparse[k];
  return a < s → members && s → dense[a] == k;
}
```

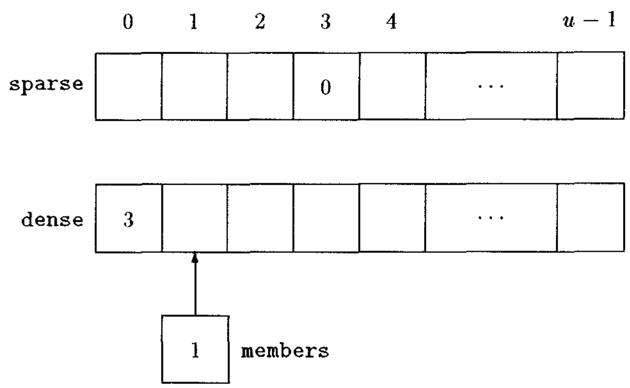


Fig. 1. A set with one member.

For simplicity, we assume that k will always be less then $u$; therefore, the initial access to s → sparse[k] will never provoke a bounds violation. Furthermore, because sparse is a vector of unsigned integers, no comparison is required to prove that a $\geq$ 0. Many variations are possible (e.g., using pointers instead of integer indices); however, they will all have the same asymptotic complexity.

Since all members of the set must appear between 0 and members in dense, clearing the set requires only setting members to 0. Enumeration of all the members (*forall*) is accomplished by iterating over the elements of dense. To apply a function foo to every member of the set s, we would use the following loop:

```
for (i = 0; i < s -> members; i + + ) {
  unsigned int member = s -> dense[i];
  foo(member);
}
```

Adding a member involves first checking for membership, then adding the new element to dense. The corresponding entry in sparse is made to point at the new dense entry.

```
void add_member(Set *s, unsigned int k)
{
  unsigned int a = s -> sparse[k];
  unsigned int n = s -> members;
  if (a >=  n | | s -> dense[a]  != k) {
    s -> sparse[k] = n;
    s -> dense[n] = k;
    s -> members = n + 1;
  }
}
```

Again, we are assuming that k $< u$. As a final example, consider the code for deleting a member:

```
void delete_member(Set *s, unsigned int k)
{
  unsigned int a = s → sparse[k];
  unsigned int n = s → members − 1;
  if (a <= n && s → dense[a] == k) {
    unsigned int e = s → dense[n];
    s → members = n;
    s → dense[a] = e;
    s → sparse[e] = a;
  }
}
```

In this case, we first check for membership, then use an element e from the end of dense to overwrite the deleted member in dense[a]. Finally, the link from sparse[e] is updated to point at dense[a].

## 3. COSTS

In this section, we discuss the asymptotic space and time complexity of sparse sets. Additionally, we compare the actual costs of common set operations in the context of sparse sets and a bit vector representation.

### 3.1 Asymptotic Complexity

A sparse set requires $O(u)$ space, regardless of the number of members in the set. In our implementations, we allocate $4u + 2$ bytes per set (storing 16-bit indices in the vectors).

We have shown constant-time implementations for *member*, *add-member*, and *delete-member*. Additionally, *clear-set*, *cardinality*, and *choose-one* have simple, constant-time implementations. The following operations have obvious implementations based on *forall* that run in $O(n)$ time, where $n$ is the number of members: *set-union*, *set-intersection*, *set-difference*, *set-copy*, and *set-equality*. *Set-complement* requires $O(u)$ time, but is rarely necessary given the existence of an efficient *set-difference*.

### 3.2 Comparisons with a Bit Vector Representation

Table I compares the asymptotic time complexities of several set operations on a bit vector representation and a sparse-set representation. Notice that for every operation, sparse sets are at least as good as bit vectors; therefore, if we only considered asymptotic costs, we could safely choose a sparse set over a bit vector representation for every application. Of course, the relative costs of different operations are also important, particularly when there is not a clear difference in the asymptotic complexities. In this section, we explore the actual costs of operations on the sparse set and on a bit vector representation.

The costs of most bit vector operations are straightforward to determine since they do not depend on the data. For sparse sets, the costs of many operations depend on both the data and the state of the uninitialized portion of sparse. For example, on the IBM RS/6000, the cost of performing a

Table I. Asymptotic Time Complexities

| *Operation* | *Bit Vector* | *Sparse* |
|---|---|---|
| *member* | O(1) | O(1) |
| *add-member* | O(1) | O(1) |
| *delete-member* | O(1) | O(1) |
| *clear-set* | O($u$) | O(1) |
| *choose-one* | O($u$) | O(1) |
| *cardinality* | O($u$) | O(1) |
| *forall* | O($u$) | O($n$) |
| *copy* | O($u$) | O($n$) |
| *compare* | O($u$) | O($n$) |
| *union* | O($u$) | O($n$) |
| *intersect* | O($u$) | O($n$) |
| *difference* | O($u$) | O($n$) |
| *complement* | O($u$) | O($u$) |

*member* operation for a bit vector is always 6 cycles.[1] On the other hand, the cost of performing a *member* for a sparse set is either 10, 16, or 17 cycles, depending on the exact path taken through the conditional branches. Cycle counts for other simple operations are:

—*add-member* requires 7 cycles for bit vectors and 14, 15, or 20 cycles for sparse sets;

—*delete-member* requires 7 cycles for bit vectors and 9, 15, or 22 cycles for sparse sets.

The sparse set's remaining constant-time operations (*choose-one*, *cardinality*, and *clear-set*) are inexpensive compared to the corresponding operations for a bit vector representation.

To verify the calculated costs, we performed an experiment comparing the performance of operations on the two representations. For each representation, we measured the time required for one million *add-member*, *delete-member*, and *member* operations. Since the time required for operations on a sparse set are data dependent, we performed the operations in sequence with random data. The test framework is shown here:

```
set = create_set(size);
for (i = 0; i < 1000000; i++) {
  add_member(set, rand( ) % size);
  delete_member(set, rand( ) % size);
  (void) member(set, rand( ) % size);
}
```

where rand( ) is a Unix system call returning integers in the range 0 to $2^{15} - 1$. To factor out overhead, we subtracted the time required for the same

[1]Cycle counts were determined by examining the output from IBM's xlc compiler for the RS/6000, ignoring the possibility of cache misses.

Table II. Relative Operation Timings

| | *Bit Vector* | | *Sparse* | | |
|---|---|---|---|---|---|
| *size* | *total time* | *average cost* | *total time* | *average cost* | *ratio* |
| 500 | 0.75 seconds | 7.5 cycles | 1.86 seconds | 18.6 cycles | 2.48 |
| 5,000 | 0.74 seconds | 7.4 cycles | 1.85 seconds | 18.5 cycles | 2.50 |
| 50,000 | 0.76 seconds | 7.6 cycles | 2.76 seconds | 27.6 cycles | 3.63 |

loop with calls to the dummy set routines. To help show the effect of cache misses, we performed the experiment with three different universe sizes: 500, 5000, and 50,000 elements. The tests were run on an IBM RS/6000, Model 540, with a 64KB data cache, a 30MHz clock, and a 100Hz timer. Each test was repeated 10 times, and the results were averaged.

Table II summarizes the results of the tests. Our earlier comparisons suggested that simple operations on the bit vector representation would be 2 to 3 times faster than on the sparse representation. The experimental results confirm these comparisons.[2] Not surprisingly, the results also show that operations on the sparse representation are more severely affected by cache misses, particularly for large universes.

We also compared the costs of complex operations (e.g., *set-union* and *set-intersection*). The bit vector operations require $O(u)$ time; the corresponding sparse-set operations require $O(n)$ time. We would expect the tradeoff between the representations to depend on the density of the sets (the value of $n$ versus $u$) and the constant factors implied by the implementations. We considered two operations in detail:

—*set-copy*: Copying an entire set requires $12 + 3\lceil u/32 \rceil$ cycles for bit vectors and $8 + 6n$ cycles for the sparse-set representation.

—*set-union*: A two-address union ($A \leftarrow A \cup B$) requires $7 + 5\lceil u/32 \rceil$ cycles for bit vectors. A sparse set requires 10 cycles startup, plus 10 or 13 cycles per member of $B$.

Comparing the requirements for *set-copy* suggests that a sparse is faster if $n < u/64$. For *set-union*, we would expect the tradeoff point somewhere in the range $u/83 < n < u/64$. Of course, these costs are all machine dependent; in particular, they depend heavily on the 32-bit word of the RS/6000. Longer word lengths will favor bit vector representations.

We performed tests to verify these comparisons. First, we measured the time to perform each operation using bit vectors with a universe size of 5000 elements. Since these times are independent of the data in the set, we made no effort to initialize the operands. Second, we measured the time required to

[2] There is no obvious reason why the bit vector operations require an average of 7.5 cycles instead of the expected 6.67 cycles. The RS/6000 has a complex superscalar implementation, and we rely on compiler estimates of structural interlocks. It seems likely that the compiler's estimates are flawed due to lack of interprocedural information.

perform the same operations using sparse sets of the same universe size. The sparse sets were initialized with random elements, where the number of elements varied between 0 and 100.

—*set-copy*: Copying the bit vector required 16.6$\mu$s versus our prediction of 16.1$\mu$s (a 15-cycle difference). The measured cost of copying a sparse set varied linearly from 0.6$\mu$s for an empty set up to 20.5$\mu$s for a set with 100 members (versus a predicted cost of 20.3$\mu$s, a difference of 7 cycles). The breakeven point occurred with a set containing 79 members (compared with the predicted $5000/64 = 78.125$).

—*set-union*: The bit vector union required 27.0$\mu$s versus our prediction of 26.4$\mu$s (a difference of 18 cycles). The time required for the sparse union varied linearly from 1.0$\mu$s for an empty set up to 40.8$\mu$s for a set with 100 members, suggesting a cost of approximately 12 cycles per member. The breakeven point occurred with a set containing 65 members (corresponding to a density of 1/77).

## 4. APPLICATIONS

We have found the sparse-set representation to be widely applicable. In this section, we study our motivating example in greater depth and present an experimental comparison of two implementations. We also describe briefly several other successful applications of sparse sets.

### 4.1 Constructing an Interference Graph

Our original motivation was to efficiently construct the interference graph for use during graph-coloring register allocation [Briggs 1992; Chaitin 1982]. The interference graph construction algorithm is sketched below. In this setting, we are concerned with the operations involving the set live.

```
for each block b in the flow graph {
  clear_set(live)                                      (1)
  for each live range lr in b → liveOut
    add_member(live, find(lr))                         (2)
  for each instruction i in b (in reverse order) {
    if i is a copy instruction (dst ← src)
      delete_member(live, find(dst))                   (3)
    for each defined live range def in i
      for each live range lr in live                   (4)
        add_edge(graph, lr, find(def))
    for each defined live range def in i
      delete_member(live, find(def))                   (5)
    for each referenced live range use in i
      add_member(live, find(use))                      (6)
  }
}
```

Considering only asymptotic complexities, the sparse representation seems well suited for this application. However, the measured superiority of bit vectors for the *add-member* and *delete-member* operations suggest that a bit vector representation for live might prove competitive. To test this possibility,

Table III. Interference Graph Construction

| | program | doduc | | | | tomcatv | | fpppp | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | routine | repvid | | iniset | | tomcatv | | twldrv | | fpppp | |
| before spilling | live ranges | | 440 | | 2,113 | | 699 | | 4,911 | | 5,439 |
| | *clear-set* | | 71 | | 493 | | 99 | | 310 | | 1 |
| | *add-member* | | 4,953 | | 15,176 | | 9,651 | | 88,129 | | 8,600 |
| | *delete-member* | | 592 | | 2,934 | | 894 | | 6,310 | | 6,446 |
| | *forall* | | 427 | | 1,976 | | 680 | | 4,726 | | 5,456 |
| | avg. length | | 51.8 | | 24.9 | | 71.4 | | 297.4 | | 118.3 |
| | density | | 11.8% | | 1.2% | | 10.2% | | 6.0% | | 2.2% |
| | time | *0.04* | 0.02 | *0.16* | 0.06 | *0.08* | 0.05 | *2.14* | 1.08 | *1.77* | 0.66 |
| after spilling | live ranges | | 378 | | 1,343 | | 652 | | 3,968 | | 5,584 |
| | *clear-set* | | 71 | | 493 | | 99 | | 310 | | 1 |
| | *add-member* | | 2,141 | | 13,850 | | 2,359 | | 10,974 | | 9,312 |
| | *delete-member* | | 417 | | 1,393 | | 723 | | 4,285 | | 5,635 |
| | *forall* | | 366 | | 1,206 | | 657 | | 4,103 | | 5,601 |
| | avg. length | | 20.6 | | 24.0 | | 16.5 | | 18.4 | | 22.5 |
| | density | | 5.5% | | 1.1% | | 2.5% | | 0.5% | | 0.4% |
| | time | *0.02* | 0.01 | *0.08* | 0.04 | *0.03* | 0.01 | *0.40* | 0.08 | *0.57* | 0.13 |
| total construction time | | *0.24* | 0.15 | *0.70* | 0.29 | *0.68* | 0.33 | *15.75* | 7.86 | *10.49* | 3.76 |
| total allocation time | | *0.48* | 0.39 | *1.38* | 0.97 | *1.22* | 0.87 | *20.55* | 12.66 | *16.32* | 9.59 |

we built two versions of Chaitin's allocator: one version using a bit vector representation for live and a second version using our sparse representation for live.

Table III summarizes the results of our experiment. We tested the allocators on five routines collected from three programs in the SPEC benchmark suite [Standards Performance Evaluation Corp. 1990]. Two of the routines, twldrv and fpppp, were chosen because they are well known for the difficulties provoked by their size. The routine iniset was chosen for its relatively high ratio of basic blocks to instructions (fpppp represents the other extreme, with a single basic block). The remaining routines were chosen as representative of smaller examples.

We made two sets of measurements on each routine: one set for the initial interference graph and one set for the interference graph constructed immediately after the first round of spilling.[3] We also measured the total time spent constructing interference graphs and the total time spent in register allocation. All tests were conducted on an IBM RS/6000, Model 540, with a 64KB data cache, a 30MHz clock, and a 100Hz timer. Each test was repeated 10 times, and the results were averaged. All times reported in Table III are in seconds. Times for the bit vector version are reported in *italics*; times for the sparse version are shown in sans serif.

The timing results are quite conclusive; for this application, the sparse-set representation is much faster than a bit vector representation. For the phases

[3]Note that the interference graph is constructed in two passes and then repeatedly refined via coalescing—we measured only the first pass.

shown and analyzed in detail, the improvement is usually at least a factor of two; in the extreme case, we see a factor of five. This produces a factor of two improvement in the total amount of time spent constructing interference graphs for each routine. The change in total allocation time, while always less than a factor of two, is still significant.

In addition to the timings, Table III also contains a variety of other data to help illustrate the behavior of the graph construction algorithm. The number of live ranges indicates the size of the universe for the set live. The entries for *clear-set*, *add-member*, and *delete-member* indicate the number of times each operation was invoked during the graph construction. Since the construction process invokes *clear-set* once for each basic block, the entries for *clear-set* correspond to the number of basic blocks in each routine. The entries marked *forall* indicate how many definition points were encountered; that is, they indicate how many times line (4) was executed. The entries for average length and density show the average number of members in live. The low average density shows why the bit vector version was not competitive.[4] Naturally, the average length is reduced by spilling.[5]

## 4.2 Other Successful Applications

Since adding an implementation of the sparse-set representation to our toolbox, we have discovered several additional opportunities for its use. In some cases, sparse sets provide an asymptotically superior implementation choice; in other cases, they simply offer convenient reuse without unnecessary run-time costs. Three typical applications are described here.

—In Chaitin's [1982] register allocator, the computation of spill costs, while straightforward at the high level, is tricky to implement. In our version [Briggs 1992, Section 8.7], we walk backward over each basic block, maintaining a set needLoad of all live ranges referenced since the last death. The set requires the operations *member*, *add-member*, *delete-member*, *clear-set*, and *forall*. The operation *clear-set* is especially important, since we must empty the set at each death (potentially at each instruction).

—During the actual coloring phase of Chaitin's allocator, we divide nodes among two sets, high and low, depending on their degree. As a node n and its edges are removed from the graph, the neighbors of n may migrate from high to low. While low could be efficiently implemented as a singly linked list and high as a doubly linked list, the sparse-set representation is just as fast. Indeed, given an existing implementation, sparse sets are the simpler implementation choice.

[4]Our *forall* for bit vectors considered each 32-bit word in the vector. If the word was nonzero, it shifted through the bits until the word was empty.

[5]The allocator was allowed 16 integer registers and 16 floating-point registers. The average length, after spilling, suggests an interesting way to compare the effectiveness of two register allocators: a greater average number of members in live would indicate more efficient utilization of the machine's register set.

—When placing $\phi$-functions during construction of minimal SSA [Cytron et al. 1989, Figure 4], the boolean arrays *Work* and *DomFronPlus* are cleared for each variable. By using sparse sets instead of simple arrays, we are able to clear the sets in constant time.[6] As a matter of convenience, we also represent the worklist $W$ with a sparse set.

While these applications of sparse sets were discovered during our work on register allocation [Briggs 1992; Briggs et al. 1992], we have used similar ideas in other passes of our optimizer, including dead-code elimination, value numbering, constant propagation, and partial-redundancy elimination.

## 5. RELATED WORK

The problem of designing specific set representations is covered in several textbooks on algorithms (e.g., Aho et al. [1974]). Papers analyzing special representations for specific problems are also common. For example, Westbrook and Tarjan [1989] analyze algorithms for set union with backtracking, and Yellin [1989] considers the problem of providing a constant-time test for set equality.

The approach we use is based on the suggested solution to a problem posed by Aho et al. [1974]. A similar problem and solution is given by Bentley [1986, p. 9]. This idea is also used by Boehm and Weiser [1988] to support pointer identification in their conservative garbage collector.

An interesting alternative to our approach is used in the implementation of SETL [Dewar et al. 1979]. The default SETL representation uses hashing to achieve constant expected time for *member*, *add-member*, and *delete-member* with links to support an efficient *forall*. However, a hashed representation requires time to initialize the hash table for both *set-clear* and *set-copy*.

## 6. SUMMARY

Programs should be designed in terms of well-understood abstractions, e.g., sequences, trees, and sets. When the design is complete, the programmer should decide on a representation for each abstract object, where the choice of representation is guided by the relative frequency of the different operations, perhaps tempered by space considerations.

Unfortunately, programmers tend to rely on intuition developed by experience rather than consider each new problem with the care it deserves. Of course, part of the difficulty lies in recognizing when a particular problem *is* new. In the case of interference graph construction, it was three years before we carefully considered the requirements for live. Part of the difficulty lay in lack of adequate profiling; it is difficult to measure the overhead of a for loop in a C program, and we were distracted by the cost of adding edges to the graph. Since we had computed liveOut using traditional bit vector techniques,

[6] In a later presentation [Cytron et al. 1991, Fig. 11], the two boolean arrays are replaced with integer arrays, *Work* and *HasAlready*, and a counter is maintained to avoid the need to reinitialize for every variable.

it was natural (and wrong) to use the same bit vector routines to implement live.

In this article, we have described a representation suitable for sets with a fixed-size universe. The representation supports constant-time implementations of *clear-set*, *member*, *add-member*, *delete-member*, *cardinality*, and *choose-one*. Based on the efficiency of these operations, the new representation will often be superior to alternatives such as bit vectors, balanced binary trees, hash tables, linked lists, etc. Additionally, the new representation supports enumeration of the members in $O(n)$ time, making it a competitive choice for relatively sparse sets requiring operations like *forall*, *set-copy*, *set-union*, and *set-difference*.

ACKNOWLEDGMENTS

Rob Shillingsburg and Brian West suggested improvements to our implementation. The editor, the referees, and Keith Cooper suggested many ways to improve our presentation. Keith Cooper and Ken Kennedy have supported our work for many years. We thank them all for their help and interest.

REFERENCES

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

BENTLEY, J. 1986. *Programming Pearls*. Addison-Wesley, Reading, Mass.

BOEHM, H.-J., AND WEISER, M. 1988. Garbage collection in an uncooperative environment. *Softw. Prac. Exp. 18*, 9 (Sept.), 807–820.

BRIGGS, P. 1992. Register allocation via graph coloring. Ph.D. thesis, Rice Univ., Houston, Tex.

BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1992. Rematerialization. *SIGPLAN Not. 27*, 7 (July), 311–321. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. *SIGPLAN Not. 17*, 6 (June), 98–105. Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.

CHAITIN, G. J., AUSLANDER, M. A., CHANDRA, A. K., COCKE, J., HOPKINS, M. E., AND MARKSTEIN, P. W. 1981. Register allocation via coloring. *Comput. Lang. 6*, 1 (Jan.), 47–57.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct.), 451–490.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1989. An efficient method of computing static single assignment form. In *Conference Record of the 16th Annual ACM Symposium on the Principles of Programming Languages*. ACM, New York, 25–35.

DEWAR, R. B. K., GRAND, A., LIU, S.-C., SCHWARTZ, J. T., AND SCHONBERG, E. 1979. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Trans. Program. Lang. Syst. 1*, 1 (July), 27–49.

STANDARDS PERFORMANCE EVALUATION CORP. 1990. SPEC Release 1.2. SPEC Corp., Freemont, Calif.

WESTBROOK, J. AND TARJAN, R. E. 1989. Amortized analysis of algorithms for set union with backtracking. *SIAM J. Comput. 18*, 1 (Feb.), 1–11.

YELLIN, D. M. 1989. Representing sets with constant time equality testing. Tech. Rep. RC 14539, IBM, Yorktown Heights, N.Y.

Received April 1993; revised August 1993; accepted October 1993