Towards Fast Incremental Hashing of Large Bit Vectors

Michael Wilder Clinton Jeffery
University of Idaho
{mwilder,jeffery}@uidaho.edu

Abstract

Many application domains require algorithmic support for collections of large bit vectors that are frequently referenced. In a compiler that performs type inferencing, this problem occurred in representing sets of possible types at each program point. This paper presents an incremental hashing technique that performs well in this domain.

1. Introduction

Hash tables are used in many application domains to map keys (usually small) onto (often much larger) values. In their degenerate form, hash tables can be used to store a set of keys, with no associated values. Such sets may be used for memory sharing, as in the Flyweight design pattern [3]. The traditional hash table $O(1)$ average case lookup and insert speeds break down when keys are very large (for example, hundreds of machine words) and heavily referenced (many millions or billions of lookups). In such circumstances, it is imperative to identify a fast hash function that performs well despite the large keys to be hashed.

This paper presents an algorithm to solve this problem in a particular real-world context: type inferencing in a compiler for a dynamically typed, object-oriented language. The techniques are relevant in contexts where a collection of large-state data is heavily referenced. The paper reports a real-world best effort and solicits ideas for improvement.

2. Type Inferencing

This work began with a fast brute-force type inferencer for the Icon programming language written by Walker that required $O(n^2)$ space on average and $O(n^3)$ in the worst case [6], where n corresponds to program size (as a number of expressions, not bytes or lines of code). The inferencer allocated large bit vectors at every (sub)expression point in a program where typed values are transmitted. The brute force algorithm runs acceptably for small and medium size programs, but fails for large programs. For example, in one of the test programs, the brute force algorithm attempted to `malloc()` 24GB of memory. By the time such large allocations become tractable, larger programs will be written for which much higher space would be needed.

The type inferencer whose incremental hashing techniques are described in this paper is being developed for an object-oriented successor of the Icon programming language, called Unicon [4]. Type inferencing space requirements are exacerbated in object-oriented programs which tend to utilize many hundreds of user-defined types for classes and method vectors.

3. Type Vector Hashing

Analysis revealed much redundancy among type vectors associated with type-mutable program points when inferring the types in programs. Hashing was introduced to eliminate type vector redundancy by employing copy-on-write semantics in a type vector pool. Using a hash table to implement a shared pool of bit vectors largely solves the space problem, reducing space needs to $O(n)$ on average and $O(n^2)$ in the worst case, at a potentially large cost in time. If the time constants involved increase the real-world time to an impractical extent, then the use of hashing fails despite its successful space reduction.

The hashing of type vectors in the type inferencer poses many challenges. The width of type vectors varies with the number of run-time types present in an input program. Type vectors can be quite large because the run-time type information produced is of very high quality. For example, in addition to bits for all the base types and declared record or class types, the bit vector also must contain bits for each field (or member variable) of each class or record. The type inferencer also uses a separate bit in the vector to differentiate each

1530-1605/08 $25.00 © 2008 IEEE

constructor site for composite types (such as lists). A list containing strings, for instance, is assigned a different type than a list containing records or a list containing strings and records. This fine granularity of type information contributes admirably to the speedup of targets generated by the Unicon Compiler, but causes type vectors to become very large during type inferencing.

The number of vectors required for type inferencing depends upon the number of type-mutable program points in the dataflow graph of the input program. Some programs have many possible types, but contain relatively few type-mutable program points; other programs contain relatively few run-time types but have many type-mutable program points. And, of course, some programs contain many type-mutable program points and many possible run-time types. These variables exacerbate the requirements imposed upon a type vector hashing scheme. To add further interest, analysis of the bit patterns held in non-redundant type vectors within the type inferencer at compile-time revealed no reliable lever of disparity that could be employed as a collision reductive agent in a type vector hashing scheme.

4. Implementation

Implementation began with a harness to permit the direct comparison of various hashing techniques within the specific confines of the Unicon Compiler type inferencer. Experimentation produced a fixed-address, multiplicative hashing scheme wherein the number of buckets in the hash table is a variant power of 2 greater than 255. This number varies depending upon characteristics of the input program ascertained before the start of type inferencing.

Type vectors are allocated from a pool of vectors that serves as the vector store during type inferencing. The size of this pool also varies depending upon characteristics of the input program ascertained before the start of type inferencing. The pool of vectors contained in the Unicon Compiler type inferencer is a discrete entity that is not accessed directly by the type vector hash table. This separation facilitates experimentation with different pooling algorithms and implementations without modifying the type vector hash table implementation. All pools in the Unicon Compiler type inferencer are implemented in this manner.

The vector hash table implementation in the Unicon Compiler does not ever migrate type vectors among buckets in the hash table. Once a type vector is inserted into a hash table bucket, it remains in situ for the duration of type inferencing. Reference counts are not maintained for type vectors contained within the hash table because any type vector in the system (except one) is created as the result of a transformation applied to another type vector that already exists within the type vector hash table.

A large space savings during type inferencing was immediately realized by eliminating type vector redundancy through hashing. The initial space reductions obtained were comparable to those indicated in Section 5. Although adding a hash table increases the complexity and likewise increases the number of instructions required for type vector generation, eliminating type vector redundancy provided a speedup of some vector manipulation operations because we were able to employ reference-based vector equality comparisons where value-based comparisons were previously required.

4.1 Bit Vector Hashing

An early version of the vector hashing function employed in the Unicon Compiler type inferencer is depicted in Figure 1. Subsequent analysis and refinement of our hashing scheme produced a much faster method than this straightforward approach.

```
00. static
01. inline
02. unsigned int
03. hash(bits)
04.     unsigned int * bits;
05. {
06.     int i;
07.     unsigned long long rawhash, xor;
08.     unsigned int cookedhash;
09.
10.     rawhash = xor = 0ULL;
11.     i = n_rttyp_ints - 1;
12.     while (i >= 0) {
13.        rawhash += xor;
14.        rawhash += bits[i];
15.        xor ^= bits[i];
16.        i--;
17.        }
18.     rawhash *= 0x20c49ba5e353f7cfULL;
19.     i = hash_shifts;
20.     cookedhash = rawhash & hash_mask;
21.     while (i) {
22.        rawhash >>= hash_upper_shr;
23.        cookedhash ^= rawhash;
24.        i--;
25.        }
26.     return (cookedhash & hash_mask);
27. }
```

Figure 1: Hashing before halving

In function hash(), the cumulative xor is used to perturb the raw hash value. The variable n_rttyp_ints is the size of array in words; hash_shifts is the word size divided by the bucket size, and is used in order to ensure that the entire hash number is involved in bucket selection. The hex constant was derived experimentally to maximize distribution on a wide range of input bit vectors.

Having achieved a dramatic space reduction, the next step was to refine the hashing and pooling algorithms to improve the speed of type vector creation and retrieval during type inferencing.

4.2 Hash Halving

It is not uncommon to encounter type vectors that are well over 100 machine words wide when inferring the types of large programs. Locality is a major performance impediment when hashing type vectors in programs of this nature. The techniques described in this section were devised to improve the speed of type inferencing in the Unicon Compiler.

The hashing routine depicted in Figure 1 was employed in early space-reduction experiments with the Unicon Compiler type inferencer. This hashing routine traverses the entire length of a type vector in order to compute its hash value. This conservative approach makes use of all the information contained within the bits of a type vector, but it becomes problematic when type vectors are hundreds of machine words wide and the hashing routine is invoked hundreds of millions of times during type inferencing. Approaches in which the hash function considers only a portion of the information in a type vector would hash faster, at the expense of more collisions. Instead, we conducted experiments to reduce the time required to compute the hash value of full type vectors. This experimentation produced a technique referred to as *hash halving*.

Hash halving permits the hashing of arbitrarily wide type vectors without traversing said vectors. To accomplish this, the hashing routine is split into a *top half* routine and *bottom half* routine. The top half routine performs a full traversal of a given type vector in order to produce a "raw" hash value for the vector. The bottom half routine is a scrambling function that transforms the raw hash value produced by the top half into a "cooked" hash value for the vector. Referring to the hashing routine in Figure 1, lines 10-17 constitute the top half and lines 18-26 constitute the bottom half of this routine. In a conventional hashing scheme, computing the hash value of a type vector would entail invoking the top half and bottom half hashing routines. The Unicon Compiler type inferencer does not do this.

It is immediately apparent that the bottom half of the hashing routine in Figure 1 is computationally much less expensive than the top half when type vectors are very wide. Lines 10-17 are executed in time proportional to the width of the type vector being hashed, which in the worst case is $O(n)$. Lines 18-26, on the other hand, are $O(1)$ because the value of hash_shifts will never be larger than the width of a "raw" hash value. If, for instance, a raw hash value is represented in 64 bits and the hash table contains 256 buckets, the value of hash_shifts is initialized to 7 at the start of type inferencing. The intuition behind hash halving is to circumvent the computational expense of the top half whenever possible, thereby reducing the $O(n)$ cost of the top half to an $O(1)$ incremental cost whenever a type vector is modified. In the Unicon Compiler type inferencer, this is facilitated by the fact that all type vectors instantiated during type inferencing are descendants of a single type vector called the *genesis vector*.

At the start of type inferencing in the Unicon Compiler, there exists a single type vector in the system. This genesis vector represents the union of all types permissible in the input program. All type-mutable program points in the dataflow graph of the input program initially refer to this genesis vector because reference-based semantics are used and no refinement of information about the types at said program points has yet been accomplished. When the genesis vector is created at the start of type inferencing, the raw (top half) hash value for this vector is computed and stored as a datum with the vector. Any subsequent vectors produced as a result of some transformation of this genesis vector use the raw hash value stored in the genesis vector to compute what the raw hash value of the potentially new type vector would be, based upon the transformation applied to the genesis vector. This computation occurs without any knowledge of the actual content of the genesis vector.

The raw hash value associated with a type vector in the Unicon Compiler type inferencer is a simple summation (see Figure 2). This summation facilitates incrementality by permitting fast updating of the raw hash value of a vector when attempting to determine the raw hash value for a potentially new type vector.

For example, assume that some type-mutable program point contains a reference to the genesis

vector V_G. In the event that the type inferencer wants to apply transform t_1 to V_G, it is desirable to determine whether the result of this transformation (V_1) is already a member of the vector hash table without performing a full hash of V_1. The Unicon Compiler type inferencer accomplishes this as follows.

The raw hash value of V_G is copied into a temporary vector. This raw hash value is then transformed *in place* based upon the nature of the transform t_1 to produce the raw hash value of the subject vector V_1. This circumvents a call to the top half hashing routine. The type inferencer then invokes the bottom half hashing routine to transform the raw hash value of V_1 into the cooked hash value of V_1. If the vector hash table bucket corresponding to the cooked hash value of V_1 is empty, V_1 of course does not exist in the hash table. The temporary vector is populated with the content of the vector V_G, the contents of the temporary vector are transformed via t_1, and the temporary vector is inserted into the vector hash table to become V_1.

In the event that the vector hash table bucket corresponding to the cooked hash value of the subject vector V_1 is not empty, the temporary vector is populated with the characteristics of V_1 and the hash table bucket is searched for the subject vector V_1.

```
static inline unsigned long long hash_th(bits)
   unsigned int * bits;
{
   int i;
   unsigned long long h;
   h = 0ULL;
   i = n_rttyp_ints - 1;
   while (i > 0) {
      h += bits[i] * i;
      i--;
      }
   h += bits[0];
   return h;
}
```

Figure 2: Top Half Vector Hashing Routine

It should be noted that if V_1 is indeed not a member of the vector hash table, its raw hash value has already been computed and any subsequent transformations of V_1 to produce some vector V_2 can be treated in the same manner as described above. To convey an idea of the useful nature of this technique, there are 12 type vector mutator routines in the Unicon Compiler type inferencer. In all of these mutators, type vector transformations occur without invoking the top half hashing routine.

An example of how the Unicon Compiler updates the raw hash value of a vector when setting a bit in a type vector appears in Figure 3. The simple arithmetic operations required to perform this update are the intentional result of choosing a top half hashing routine that maximizes the performance of incrementality. Thanks to the modulo-based ring semantics of unsigned integer arithmetic, the previous summation term can be subtracted out and the new term can be added in correctly even in the presence of integer overflow and underflow.

```
n = idx ? idx : 1;
vect->raw_hash -= vect->bits[idx] * n;
vect->bits[idx] |= bit_mask;
vect->raw_hash += vect->bits[idx] * n;
```

Figure 3: Updating a Raw Hash Value

The nature of the top half hashing routine is of paramount importance when performing hash halving. The top half hashing routine must be selected such that it contains logic simple enough to facilitate the in-place manipulation of the raw hash value of some vector V_n to produce the raw hash value of vector V_{n+1} based upon a vector transformation t applied to vector V_n with no a priori knowledge of the actual content of vector V_n. An implicit requirement in this pursuit is bi-directionality of operation: both the setting and clearing of bits in a vector must be computationally expedient.

The top half vector hashing routine currently used in the Unicon Compiler type inferencer is depicted in Figure 2. In this figure, the value of n_rttyp_ints is computed before the start of type inferencing and remains constant thereafter. The top half hashing routine depicted in this figure is extraordinarily simple. This simplicity is of the highest importance because the logic in this routine is inlined wherever a bit vector is mutated during type inferencing, and is executed hundreds of millions of times during the course of type inferencing.

The bottom half hashing routine must transform the raw hash value supplied by a top half into a cooked hash value that contributes to a smooth distribution of vectors over hash table buckets. The bottom half must also be as fast as possible because it will be invoked hundreds of millions of times during type inferencing. The bottom half vector hashing routine currently used in the Unicon Compiler type inferencer is depicted in Figure 4. In this figure, the values of hash_shifts, hash_mask, and hash_upper_shr are computed before the start of type inferencing and remain constant thereafter.

The bottom half hash routine depicted in Figure 4 runs in time $O(1)$.

```
static inline unsigned int hash_bh(raw)
   unsigned long long raw;
{
   int i;
   unsigned int u;
   raw *= 0x20c49ba5e353f7cfULL;
   i = hash_shifts;
   u = raw & hash_mask;
   while (i) {
      raw >>= hash_upper_shr;
      u ^= raw;
      i--;
      }
   return u & hash_mask;
}
```

Figure 4: Bottom Half Vector Hash Routine

5. Metrics

All measurements in this section were taken on a dual-processor, dual-core, 1.8GHz AMD64 machine with 4GB core running Fedora Core 3 Linux because the sponsor is running Fedora. A program called cls-gen was used to generate all the programs that are used to measure type inferencing space and time usage in the following metrics. The cls-gen program is a program that generates programs. This program takes a numeric parameter n and creates files containing n class declarations and a single main() procedure. The generated main() procedure creates a list that is populated with a single instance of each of the n classes. The main() procedure then enters a loop in which all methods contained in each of the n class instances is invoked.

Versions 076 and 0c5 of the Unicon Compiler are compared in the metrics contained in this section. Version 076 of the Unicon Compiler is the last version that does not use vector hashing. Version 0c5 of the Unicon Compiler is the first version that uses vector hashing, hash halving, and vector auras.

5.1 Space

The space consumed during type inferencing in versions 076 and 0c5 of the Unicon Compiler are compared in Figure 5.

# types	*V076*	*V0c5*
416	1.3 MB	26 KB
864	13.1 MB	129 KB
1,676	85 MB	454 KB
3,276	649 MB	1.7 MB
4,876	2.6 GB	3.9 MB
6,476	5.0 GB	6.9 MB

Figure 5: Type Inferencing Space Usage

The reader will note that in the type inferencing space comparison, v076 always consumes at least an order of magnitude more space than v0c5. A program containing 500 run-time types is considered a fairly small (but not tiny) program in this language.

5.2 Time

The time consumed during type inferencing in versions 076 and 0c5 of the Unicon Compiler are compared below.

# types	*V076*	*V0c5*
416	.040	.038
864	.588	.549
1,676	6.6	3.6
3,276	101.6	25.4
4,876	524.4	82.8
6,476	1512	198

Figure 6: Type Inferencing Times (seconds)

The type inferencing time comparison contained in Figure 6 reveals a couple noteworthy points. Version v0c5 takes about the same time to infer the types in small programs such as that containing 416 run-time types. This is despite the fact that v076 allocates all type vectors at the start of type inferencing, whereas v0c5 contains extra logic to pool type vectors and replenish the pool when it is exhausted. Tailoring of the vector pool allocation algorithm might further improve the performance reported in v0c5.

The second point of note regarding Figure 6 is that v076 of the Unicon Compiler type inferencer consumes more memory than is present in the system when inferring the types in a program containing 6,476 types. Some of the time

represented in the last entry is swapping. Enhancing the scalability of the type inferencer, and thus avoiding such swapping, was one of the primary motivations for implementing type vector hashing. A comparison of the inferencing times for v076 and v0c5 on programs containing more types than those depicted in Figure 6 would only serve to illustrate the grisly divergence provoked by swapping as physical memory is exhausted.

5.3 Measuring the Techniques

This subsection examines the contribution that each of the techniques described in this paper makes to reducing the overall time to infer types in large programs. Three instances of the compiler were created and labeled A, B, and C. Instance A is a basis point that contains only vector hashing. Instance B contains vector hashing and hash halving. Instance C contains vector hashing, hash halving, and vector auras. The time required to perform type inferencing on three programs was measured in each of these individual instances and appears in Figure 7.

# types	*A*	*B*	*C*
3,276	364.6 s	63.7 s	30.3 s
4,876	1847.4 s	287.0 s	107.2 s
6,476	6,319.9 s	800.8 s	246.0 s

Figure 7: Effect of Hash Halving and Vector Auras on Type Inferencing Time

Each row of Figure 7 shows the contribution of the hash halving and vector aura techniques to the speedup of type inferencing in the Unicon Compiler. This data shows that, when using a conventional hash table, the dramatic savings in space is achieved at an unacceptable cost in time, compared with the brute force, non-hashing, highly redundant implementation in v076. The reader will note that, when comparing the A column in Figure 7 with the v076 column in Figure 6, the A instance of the Unicon Compiler takes a factor of 3 or more time to perform type inferencing than the v076 version of the Unicon Compiler in the programs indicated in these figures. Type vector hashing using conventional techniques is clearly insufficient in terms of time.

The data in Figure 7 indicate that hash halving (B) contributes admirably to type inferencing speedups in the Unicon Compiler. The smallest speedup gained by adding hash halving to the Unicon Compiler type inferencer in this set of measurements is roughly 82%. Eliminating type vector redundancy by hashing type vectors provides a significant space improvement, but the hash halving technique makes type vector hashing practical in terms of time.

The vector aura technique (C) in these measurements provides at least a 60% reduction in type inferencing time. Vector auras are checksums stored in hash table entries to avoid comparing long data values in the event of collisions; we have experimented with several kinds of auras. If the hash function is good the hash number itself makes a good aura. Especially fast hash functions may benefit from different auras.

5.4 Collision Rates

In this subsection we examine the expected rate of collisions in the vector hash table contained in the Unicon Compiler type inferencer. The expected value rate of hashing collisions is shown for the Unicon Compiler v0c5, the version that employs vector hashing, hash halving, and vector auras.

# types	*M*	*N*	*collision rate*
416	512	255	0.003922
864	1024	507	0.001972
1,676	2048	1,152	0.000868
3,276	8192	4,377	0.000228
4,876	16384	6,527	0.000153
6,476	16384	8,677	0.000115

Figure 8: Vector Hash Table Collisions

In Figure 8 the rate of collisions within the vector hash table is shown for each of the programs depicted in Subsection 5.1. As mentioned previously, the number of buckets in the vector hash table (M) is calculated before the start of type inferencing based upon information gleaned from the input program. The number of vectors contained within the hash table is represented by N. Type vectors are never removed from the vector hash table once they are added, so N in Figure 8 represents the number of type vectors in the hash table at the termination of type inferencing.

Figure 8 indicates that the expected rate of collisions in the type vector hash table decreases as the number of types in an input program increases. Rows 5 and 6 of this figure are particularly interesting in that the number of buckets in the

vector hash table remains the same for each of the programs, but the collision expectation rate decreases as the number of the vectors present in the hash table increases. One hypothesis for this behavior is that the type vectors contained within the hash table for the program run in row 6 are less sparse. The bottom half hash routine in the Unicon Compiler type inferencer (see Figure 4) employs xor as its primary operator, and this form of perturbation tends to favor vectors that are more densely populated with bits that are set.

Measurements of the expected rate of collisions when using the "full" hashing routine appearing in Figure 1 were also taken. The collision rate produced by this routine is statistically insignificant from the expected value of collisions appearing in Figure 8.

To illustrate the scalability of the Unicon Compiler type inferencer, the time and space consumed by the Unicon Compiler v0c5 while inferring the types in a very large program containing 9,676 run-time types was measured, and the time and space consumed by gcc when compiling the C code generated by the Unicon Compiler for this same program was measured. The Unicon Compiler v0c5 consumed 1GB of memory to infer the types in this program. At the apex of its memory consumption, gcc used roughly 4GB. The Unicon Compiler consumed 13.9 minutes to infer the types in the program, and gcc consumed roughly 23.1 minutes to compile the resulting C code (not including assembly and linking). In large programs, type inferencing no longer dominates total compile time. Prior to the start of this work, the Unicon Compiler v076 attempted to allocate 24GB of core at the start of type inferencing when presented with this same program. On the test machine, this results in a kernel panic before completing a single iteration of type inferencing.

6. Related Work

After designing and implementing hash halving in the context of type inferencing, it was not surprising to discover that related techniques have been developed in other domains. High-speed hashing for large binary data is likely to recur in numerous unrelated applications. Edelkamp and Mehler [2] use *incremental state hashing* to compute hash values in constant time for depth-first search traversals of state space graphs, in applications such as Model Checking. In state space graphs, the hash table saves repetitive search work and not just space. The approach is similar, but the hash functions used appear to be slower than the simple ones presented in this paper.

Bellare and Micciancio [1] devised a similar scheme in the domain of cryptology. The scheme of Bellare and Micciancio, referred to as *incremental hashing*, employs hash halves that they call the "randomize-then-combine paradigm." The top hash half (randomizer) in their scheme is derived from a standard hash function that is considered ideal and must be collision-free. This randomizer is applied to individual blocks of a message and combined (by the bottom half) to produce a hash value for the message. Their scheme does not differentiate between a partial (raw) and full (cooked) hash value. They experimented with three combinators: exclusive or, multiplication, and addition-with-modulus. They concluded that exclusive or was an insecure combinator.

The approach devised for the Unicon Compiler type inferencer is similar in that it, too, is incremental. The speedup offered by incrementality seems to have been the primary motivator in both efforts. The approaches differ in that the hash halving scheme employed in the Unicon Compiler type inferencer was not designed with security in mind. The only adversary of the Unicon Compiler is an input program. Instead of using a collision-free random oracle as a hashing function, the Unicon Compiler type inferencer substitutes fast, simple arithmetic operations to augment the speedup realized through the use of incrementality. The Unicon Compiler type inferencer uses xor as the primary operation in the bottom hash half for one reason: speed. It is irrelevant that the bottom hash half is insecure and one way, because we do not achieve incrementality of the full hash, but rather only of the "raw" hash value produced by the top half hashing routine. This approach does not restrict the domain of operations that we can employ in a bottom half hashing routine. It is interesting to note that Bellare and Micciancio wanted to use xor as the primary operator in their bottom half hashing routine (combinator), but security constraints prevented them from doing so.

Profiling compilations armed us with the fact that the bottom half hashing routine is invoked hundreds of millions of times when compiling large programs. It is hard to beat the speed of a bottom half hashing routine that performs *at most* 7 shr and xor operations to scramble the raw hash value when a raw hash value is represented in 64 bits. This approach is, of course, not collision-free. The collisions that ensue are filtered by type vector auras in the Unicon Compiler type inferencer.

Phan and Wagner [5] describe security issues related to incremental hashing and show how collisions can result from incremental hashing routines that are based upon pair-block chaining. They detail design and implementation criteria for incremental hashing when employed in security-sensitive capacities. The hash halving incremental hashing scheme employed in the Unicon Compiler type inferencer assumes that hash collisions occasionally occur and mitigates the effect of collisions through the use of type vector auras.

7. Conclusions and Future Work

The hashing of type vectors in the Unicon Compiler type inferencer has proven to be an effective technique for decreasing the space required to infer types used in all programs. We have shown in this paper that vector hashing results in space savings of multiple orders of magnitude when inferring the types in large input programs. Given the urgent need for vector hashing, the hash halving technique provides a significant speedup of the type inferencer by eliminating the need to traverse a type vector in order to compute its hash value. This speedup is particularly evident when type vectors are very wide. The speedup realized by providing the aura of a type vector to act as a filtering mechanism when searching a hash table bucket for a given vector is also meaningful, given that implementing type vector auras is straightforward.

Despite the results described in this paper, global type inferencing remains a space challenge. When type inferencing alone still requires ~1GB or more for some programs, total compilation memory requirements still exceed available memory on most machines. This motivates further reductions in type inferencing space requirements.

Despite substantial time gains, the time required to infer the types in programs containing thousands of run-time types motivates further time-directed research. Additional type inferencing speedups may be obtainable by devising better hash halves, thereby improving the distribution of type vectors over hash table buckets, but other approaches are also needed. A trie (with edges based on type vector manipulations) superimposed on the vector hash table might reduce bit manipulation and hash function calls enough to overcome its additional costs in terms of space and code complexity.

Experiments were conducted using sparse vectors to represent type vectors in very early space-directed experiments in the Unicon Compiler. The 67% space savings produced by sparse vector representation is largely subsumed by the greater space savings and space-complexity improvement provided by vector hashing. However, the techniques are not mutually exclusive and it may be possible to couple a sufficiently clever sparse representation of type vectors with the copy-on-write, reference-based semantics of type vector hashing (and hash halving) to further reduce the space and time consumed during type inferencing in the Unicon Compiler.

The techniques described in this paper were devised by exploiting strengths and attacking weaknesses specific to the Unicon Compiler type inferencer. However, these techniques can be applied to other domains where large data is derived incrementally and must be hashed.

8. Acknowledgments

This research was supported in part by an appointment to the National Library of Medicine Research Participation Program. This program is administered by the Oak Ridge Institute for Science and Education for the National Library of Medicine.

9. References

[1] M. Bellare and D. Micciancio. A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost. In *Advances in Cryptology, EUROCRYPT '97*, pages 163-192, 1997.

[2] S. Edelkamp and T. Mehler. Incremental Hashing in State Space Search. 18th Workshop on New Results in Planning, Scheduling, and Design (PUK), Ulm, 2004.

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns. Addison-Wesley, Reading MA, 1995.

[4] Clinton Jeffery, Shamim Mohamed, Ramon Pereda and Robert Parlett, *Programming with Unicon*, book and software downloads from from unicon.org, 2007.

[5] R. C. W. Phan and D. Wagner. Security Considerations for Incremental Hash Functions Based on Pair-Block Chaining. *Elsevier Computers and Security*, 25:131-136, 2006.

[6] K. W. Walker and R. E. Griswold. Type Inference in the Icon Programming Language. *University of Arizona Technical Report*, 93(32a), March 1993.