12-1992

Hardware-Assisted Garbage Collection for the Icon Programming Language

Ramakrishna Chinta
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Systems Architecture Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/32

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Hardware-Assisted Garbage Collection for the Icon Programming Language

Abstract
Hardware-Assisted garbage collection for Icon guarantees real-time response and consistent performance. This report describes the changes made to Icon’s run-time system in porting Icon to the hardware-assisted garbage collection environment and compares the performance of fifty Icon programs using the traditional and hardware-assisted garbage collection implementations. Throughput for the hardware-assisted system ranges from 45% slower to 28% faster than the traditional system. Suggestions for improving the performance of the hardware-assisted implementation are also discussed.

Disciplines
Systems Architecture
Hardware-Assisted Garbage Collection
for the Icon Programming Language

TR 92-39
Ramakrishna Chinta

December, 1992

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011
Hardware-Assisted Garbage Collection
for the
Icon Programming Language

Ramakrishna Chinta

December 17, 1992
## Contents

1 Introduction ........................................... 1
   1.1 Icon Language ....................................... 1
   1.2 Icon For Real-Time Programming ................. 1
   1.3 Hardware-Assisted Garbage Collection .......... 2
   1.4 Objective ........................................... 2

2 Hardware-Assisted Garbage Collection ........... 3
   2.1 Architecture ......................................... 3
   2.2 Supported Data Types ............................. 3
   2.3 Algorithm ............................................ 5

3 Implementation of Icon ............................... 7
   3.1 Icon Descriptors .................................... 7
   3.2 Interpreter Stack ................................... 9
   3.3 Memory Management ............................... 9

4 Overview of Implementation Changes .......... 11
   4.1 Background Work ................................... 11
   4.2 Traditional memory vs GC memory ............... 11
   4.3 Mapping Icon Structures to Data Objects supported by GC Hardware ............. 12
   4.4 Descriptor Tags ................................... 15
   4.5 Predictive-Need .................................... 17
   4.6 Source Descriptors ............................... 18
   4.7 Summary ............................................ 19

5 Performance Analysis ................................. 20
   5.1 Instructions .................................... 20
   5.2 Icache Hit Rate .................................... 25
   5.3 Dcache Hit Rate .................................... 27
   5.4 Bus Utilization .................................... 27
   5.5 Machine Cycles .................................... 29
   5.6 Number of Collections ........................... 31

6 Conclusions and Future Work ....................... 32

Acknowledgements ........................................ 33

References ................................................. 34
1 Introduction

1.1 Icon Language

Icon is a very high-level language, with a syntax very similar to that of C. It is an imperative, procedural language with variables, operations, functions, and conventional data types, and is conventional in many respects. However, its semantics and high-level constructs to handle strings and structures set it apart from other high-level languages. Its semantics offer expression-based, goal-directed evaluation. It offers various features that free the programmer from low-level details and help reduce software development time.

Some of features offered are listed below. The language is described in detail in reference [7].

- No type declarations. Variables can have values of any type.
- Several high-level data structures such as lists, sets, tables. These structures can hold heterogeneous values.
- Variable-sized strings and structures.
- A rich repertoire of string and structure manipulation functions.
- Several polymorphic operations, which perform different operations depending on the types of their arguments.
- Automatic allocation and reclamation of memory.

Because of the need to support untyped variables and heterogeneous aggregates of unbounded sizes, it is not possible for the compiler to set aside memory for variable values at compile time. Rather, memory for strings and structures is allocated at run time. Programmers need not be concerned with the details of memory management because memory allocation takes place implicitly when a new data value is created. Garbage collection reclaims unused memory automatically.

1.2 Icon For Real-Time Programming

Since Icon eliminates many low-level programming concerns, reducing the likelihood of programming errors, it is desirable to use Icon in a real-time environment. Unfortunately, it is observed that garbage collection can have a measurable effect on the overall performance of the language. It can account for up to 76% of total execution time of some Icon programs [9]. Of even
greater concern are the delays that occur at unpredictable times to garbage collect memory.

In a real-time environment, it is critical to have a tightly bounded response time. Nearly all existing Icon implementations use a traditional mark-and-sweep garbage collector that imposes unpredictable delays at arbitrary times during execution of Icon programs. Thus, Icon programs cannot guarantee strict bounds on the time required to execute particular segments of code. It is possible that an Icon program under execution may suspend at arbitrary times during allocation of new memory. Icon, or any other language that performs garbage collection using traditional techniques, is incompatible with the requirements of real-time programming due to this memory management overhead. Thus real-time programmers are unable to utilize the power of very high-level languages like Icon.

1.3 Hardware-Assisted Garbage Collection

The above-mentioned problem has been addressed by researchers working on real-time support for high level languages [1, 5, 6, 11, 12, 10]. Reference [1] describes a hardware architecture that enables hardware-assisted garbage collection. This architecture offers high average-case allocation rates and memory bandwidths, with a tight bound on worst-case allocations, fetches, and stores. It provides real-time response by interleaving incremental garbage collection activities with ongoing execution of application software.

1.4 Objective

This work aims at providing an Icon implementation that is more conducive to a real-time environment and investigating the utility of the garbage collection architecture proposed in reference [1].

The traditional run-time implementation of Icon is changed to use the special hardware to perform hardware-assisted garbage collection. The work also attempts to characterize the efforts required to port existing garbage-collected languages to the hardware-assisted garbage-collection environment. Performance results on sample test cases are provided to give an estimate of the overhead introduced by the garbage collection protocol on the overall performance.
2 Hardware-Assisted Garbage Collection

This section gives an overview of the hardware architecture and the garbage-collecting algorithm. More thorough descriptions are provided in [1, 2, 3].

2.1 Architecture

![Diagram of a computer system with CPU, Cache, Conventional System Bus, ROM, RAM, and Garbage-Collected Memory Module]

The garbage-collected memory module is configured into the traditional bus architecture as shown above. The GC module appears to be an extension to normal memory and occupies part of the real-memory address space. The module contains special circuitry to implement a garbage-collection algorithm, and to respond to memory stores and fetches. Application tasks, such as the Icon implementation, run on the CPU. They interact with the GC module by reading and writing to the I/O ports of the module and by fetching and storing the contents of garbage-collection memory cells. The module carries on garbage collecting activities as a background activity. Garbage collection is interrupted whenever the CPU requires memory access.

2.2 Supported Data Types

The GC module provides support to operate on data objects. An object is simply a contiguous region of memory that shares a particular attribute.
Each word of the GC memory is either a descriptor word, that points to another GC data object, or a terminal word, that does not hold a pointer to another GC object. The garbage collector distinguishes between memory representing descriptors and memory representing terminal data by accompanying each word with a 1-bit descriptor tag.

Records: Records are fixed-size objects containing both terminal and descriptor words. Operations are provided for initializing and modifying the descriptor tags that accompany each word contained within the record. It is the application task’s responsibility to update the descriptor tags to reflect the current contents of each word.

Slice Objects and Regions: A slice object consists of two words. One word is a pointer to a segment of contiguous data called the slice region data. The other word contains the length of the segment. Once allocated, a slice object is considered to be read only. Only the slice region data referenced by the slice object is writable by the application process.

The garbage-collection module provides two different mechanisms for allocating slice objects. The first allocates the slice object and its accompanying slice region data. The second, which allocates only the slice object, initializes the slice object to refer to previously allocated slice region data.

Slice regions can hold terminal or descriptor data. If it is known at the time a slice object is created that the referenced slice region data will hold only terminal data, the slice object can be marked to so indicate. Garbage collection of terminal-only slice objects is much more efficient than garbage collection of slice objects that may refer to descriptor data. A typical interaction between slice objects, slice regions, non-slice objects, and arbitrary descriptors is shown below:
In the figure above, a single slice region data is shared by three slice objects. The GC module identifies a slice region by by its title SD, which stands for SliceData. The slice region contains either terminal values or pointers to other data objects. Pointers are represented by arrows to the objects they point. Two of the slices are titled as descriptor slices, indicating that the referenced slice region data may contain descriptors. The third slice is marked as a terminal slice so that the garbage collector can collect the region efficiently. The figure also shows that arbitrary descriptors may point directly into the slice region.

During garbage collection, if any location within a record is referenced by any live descriptor, then the entire record is considered alive. However, slice region data is treated as live only if it is referenced by a live slice object. Furthermore, if portions of a slice region data object are found to be unreferenced by live slice objects, the garbage collector shrinks the slice data region in order to reclaim the unreferenced portion as garbage.

### 2.3 Algorithm

The GC module divides the GC memory into to-space and from-space. New memory is allocated from to-space until garbage collection is triggered by an allocation request that cannot be satisfied. Garbage collection begins with
a flip of regions. The old \textit{from-space} becomes the new \textit{to-space}, and the old \textit{to-space} becomes the new \textit{from-space}. The garbage collector then copies all \textit{live} objects from \textit{from-space} to \textit{to-space}.

The application task on the CPU is called the \textit{mutator} since, as far as the garbage-collecting module is concerned, its role is to mutate (or change) the contents of garbage-collected memory cells. To bootstrap the garbage collection process, the mutator must inform the garbage collector of each of the pointers into the garbage-collected heap that it maintains. These descriptors are called the \textit{source descriptors} and the process of updating a descriptor so that it points to \textit{to-space}, rather than \textit{from-space}, is called \textit{tending}.

On a flip, the mutator tends each of the source descriptors by communicating with the garbage collector, and then proceeds with its usual activities. A call to tend a descriptor, \texttt{tendDesc}, returns in at most 2 memory cycles. In response to a \texttt{tendDesc} invocation, the garbage collector reserves space in \textit{to-space} for the object referenced by the source descriptor, and places the object in a FIFO queue of objects waiting to be copied. It defers actual copying to a later stage. As far as the mutator is concerned, garbage collection completes immediately. The garbage-collecting hardware takes responsibility for maintaining the illusion that all live memory is copied instantaneously into the \textit{to-space} at the time of a flip. It accomplishes this job by copying the queued live objects incrementally in parallel with the CPU's activities.

The pointers contained within record and slice objects are tended as they are copied. Furthermore, whenever the mutator requests to read an untended descriptor or to read data residing on the copy queue, the requested word is fetched out of \textit{from-space} if necessary, and, depending on its descriptor tag, tended before its value is returned to the mutator. The worst-case fetch time is shown to be 6 traditional memory cycles [2]. For both fetch and store operations, the average cost is nearly the same as traditional memory.
3 Implementation of Icon

The Icon system for version 8.0 of Icon consists mainly of two parts: a translator and a run-time system. The translator converts an Icon program into a binary code called the icode. The icode is then interpreted by the run-time system during execution. The run-time system consists of an interpreter for icode and a library of support routines to carry out the various operations required by the interpreter.

Reference [8] describes the implementation of the run-time system in detail. This section introduces implementation issues that are relevant to memory management.

3.1 Icon Descriptors

Every Icon value is represented by a descriptor comprised of 2 words. The first word is called the d-word and the latter, v-word.

```
  d-word
   -------
    v-word
```

The two words together contain enough information to determine the type of the value it represents and to locate the actual data. The d-words of Icon descriptors contain a type code in their least significant bits and a set of flags in their most significant bits. The type code denotes the type of the value. Twelve of the type codes correspond to source-language data types: null, integer, real, cset, file, procedure, list, set, table, record, co-expression. Other data type codes represent internal objects that are not visible at the source-language level. The flags characterize the value stored in the v-word as described below.

The v-word either contains the value represented by the Icon descriptor if the value is small enough to fit into a 32-bit word, or is a pointer to another data object which contains the value. The flags in the d-word are useful in interpreting the contents of the v-word.

There are 3 different descriptor layouts for Icon values. The first layout is for Icon string descriptors, which do not have any type code in their d-words. Instead, the d-words contain the length of the strings referenced by the descriptor. The v-word points to the first character of the string. A descriptor
for string “A String” is shown below:

```
  8
“A String”
```

Icon descriptors for all other values contain type codes in their v-words. In order to differentiate string descriptors from other descriptors with type codes that might resemble a string length, descriptors that do not represent strings have their n flag set. Hence a string descriptor is identified by the lack of an n flag.

The second descriptor layout contains the actual value in the v-word itself. The null value and integers use this layout. They have type codes in their d-words and the values in their v-words. Descriptors for the null value, and integer 426 are illustrated below:

```
  n  null
     0
```

```
  n  integer
     426
```

All other data types use the third layout for Icon descriptors. Values of these data types do not fit within the v-word and hence they are stored in separate blocks of data. Descriptors for such data values contain a type code in their d-words but have pointers to blocks of data in their v-words. These descriptors have their p flag set to indicate that the value in the v-word is a pointer to a block. For example, a descriptor for a table looks like:
3.2 Interpreter Stack

The run-time system executes code instructions using stack-based evaluation. Code instructions typically push and pop descriptors onto the interpreter stack and perform operations on elements of the stack. The interpreter stack is an array of words that is distinct from the ‘C’ stack.

3.3 Memory Management

During execution of an Icon program, memory is divided into several regions. The layout of the regions is typically of the following form:

The run-time system contains the executable code for the interpreter, built-in operators and functions, and other support routines. It also contains
static space allocated by the C compiler for the Icon strings and blocks that are referenced by the C-implementations of Icon’s built-in library functions and operators.

The icode region contains the icode for the Icon program that is to be executed. Remember that the icode is generated by a translator before executing the program. Apart from the instructions, the icode region also contains string, real, and cset literals, procedure blocks for procedures defined in the Icon program, and the values and names of global identifiers. Data in the icode region is never moved, although some values may change during run time.

The space for data objects that are constructed during run time is in the allocated storage region. This portion of memory is divided into static, string and block regions. The static region contains space for co-expression blocks and interpreter stacks for each co-expression. Data in the static region is never moved. The space is however freed when the data is no longer of use. The remaining part of the allocated storage region is divided into string and block regions. Space for strings is allocated at run time from the string region. All other structures are dynamically allocated in the block region. The string region contains only characters whereas the block region may contain pointers in addition to terminal data. The Icon implementation uses different garbage-collection techniques in each of these regions.

The run-time system uses a mark-and-sweep strategy for garbage collection of the block region. This involves two phases. In the marking phase, all live objects and the pointers to them are marked. The marking starts from a root set of pointers known as the basis, and traces through pointers to mark all live objects. In the second phase, known as the sweep phase, all the marked live objects are moved into consecutive memory locations and all pointers to relocated live objects are adjusted to point to the objects’ new locations.
4 Overview of Implementation Changes

This section gives an overview of the changes made to the traditional run-time implementation to incorporate hardware-assisted garbage collection. The modified Icon implementation that uses hardware-assisted garbage collection will henceforth be called GC-Icon.

4.1 Background Work

The GC module has been simulated on the DLX architecture [13]. In order to incorporate hardware-assisted garbage collection we need an Icon implementation for DLX architecture. The implementation of Icon’s run-time system is written in C and contains about 50 source code files, containing an average of 600 lines per file. The first step of this project was to port the traditional Icon run-time system to DLX. The implementation has been successfully ported and tested. Support for co-expressions is not incorporated at this stage. All other functionalities of the run-time implementation have been retained.

4.2 Traditional memory vs GC memory

Hardware-assisted garbage collection requires that each of the source descriptors pointing into the GC memory be tended on a flip. In order to reduce the overhead involved in tending the source descriptors, it is desirable to limit the number of source descriptors. This can be done by keeping most of the dynamically allocated structures under the GC’s purview. However the following overheads are directly proportional to the amount of data stored in the GC area.

- overhead to keep track and update the descriptor tags of GC memory cells. Other studies have shown this overhead to be very significant [4].

- overhead to copy the live objects from from-space to to-space on a flip.

We need to balance the tradeoffs by properly choosing between traditional and garbage-collected memory for all dynamically allocated data. For each of Icon’s run-time data structures, it was necessary to decide whether to store the structure in traditional memory or garbage-collected memory.

In GC-Icon, all structures that are frequently allocated and reclaimed are stored in the GC memory. Structures that are never freed, are stored in traditional memory to reduce the above-mentioned overheads.
Traditional Memory: Except for the string and block regions, all the memory regions discussed in the previous section continue to be stored in traditional memory. All these regions, except for the allocated static memory region, rarely change during run time and are therefore justified to remain in traditional memory. The static region, which contains the interpreter stack, is the only region whose size and content keeps changing and is yet not stored in GC memory. The following arguments go in favor of this decision:

- changes to the interpreter stack do not involve heap memory allocation and freeing. By storing it in traditional memory, we are reducing the overhead of moving the stack between the to-space and from-space regions.

- the interpreter stack is the most frequently accessed structure. If it were stored in GC memory, any change to it would need an update of the associated descriptor tags, and this could introduce tremendous overhead [4].

- co-expression blocks contain internal pointers and hence are difficult to relocate to another place. So, the decision holds good even when GC-Icon is adapted to support co-expressions.

It may be noted that by not storing the above regions in GC memory, we have taken up the responsibility of determining the pointers into GC memory from these regions in traditional memory, and tend each such pointer value on a flip.

GC Memory: Both Icon strings, and all of the objects that are allocated in the Block region in the traditional Icon system, are stored in the GC area of the GC-Icon system.

4.3 Mapping Icon Structures to Data Objects supported by GC Hardware

Strings

*Terminal Slice Objects* provide a convenient way to represent strings. A straight-forward method to allocate storage for a string would be to allocate a slice object and return a *pointer to the slice region* it points to. An Icon string descriptor would ideally look like
However, slice region data referenced by an arbitrary descriptor is only treated as live if it is also referenced by a slice object [1]. This means that on a flip, a pointer to the slice object has to be tended, rather than a pointer to the slice region, for the string to be treated as live data. Hence, for all the allocated strings, a pointer to the slice object is stored instead of a pointer to the slice region itself. The routine that allocates space for a string is therefore modified to return a pointer to the slice object. An Icon string descriptor in GC-Icon is shown below:

The additional level of indirection thus introduced for strings allocated in GC area introduces a lot of changes to the implementation.
The traditional implementation defines an Icon descriptor as

```c
struct descr {            /* descriptor */
    word dword;        /* type field */

    union {
        word integr;     /* integer value */
        char *sptr;      /* pointer to character string */
        union block *bptr; /* pointer to a block */
        dptr descptr;    /* pointer to a descriptor */
    }

    vword;
};
```

In GC-Icon the v-word of string descriptors is defined to contain the address a word containing a pointer to the string. An Icon descriptor is now redefined as

```c
struct descr {            /* descriptor */
    word dword;        /* type field */

    union {
        word integr;     /* integer value */
        char **sptr;     /* pointer to address of character string */
        union block *bptr; /* pointer to a block */
        dptr descptr;    /* pointer to a descriptor */
    }

    vword;
};
```

If \( d \) is a string descriptor pointing to a string allocated during run time, the actual string is accessed as \( *(d.vword.sptr) \) rather than \( (d.vword.sptr) \). So `StrLoc`, the macro to identify the string location given an Icon string descriptor, is redefined as

```c
#define StrLoc(d)    (*((d).vword.sptr))
```

But if the string for \( d \) has been statically allocated, the v-word of the Icon descriptor would directly point to the string rather than its address. In order to treat both statically and dynamically allocated strings uniformly, an additional level of indirection is introduced for all statically allocated string descriptors.

A few instances where statically allocated Icon strings are converted to the required format are shown below:

1. For each string literal in an Icon program, the translator allocates space in the icode for the string literal. During run time, an additional word
is created for each string literal. The address of the string literal is stored in the allocated word and the the address if the allocated word is assigned to the v-word of the string descriptor.

2. In the C code for the traditional implementation, C string literals are directly assigned to the v-words of Icon string descriptors. For example, when creating a descriptor d for the type image of the null value, we come across

```c
    d.dword = 4;
    d.vword.sptr = "null";
```

Such an instance is replaced by

```c
    static char *Xnull = "null";
    d.dword = 4;
    d.vword.sptr = &Xnull;
```

**Blocks**
The sizes of Icon blocks do not change once allocated. Record objects supported by the GC module provide a convenient way to store Icon blocks.

### 4.4 Descriptor Tags

Having decided on the C structures and objects to be stored in GC memory, we need to identify the descriptor tags for each of the GC-allocated objects. This is an important step which needs lot of attention. Improper descriptor tags will mislead the GC module in distinguishing descriptor words from terminal words and can lead to unexpected results. Usually, most of the objects allocated in the GC memory hold either pointer data or terminal data, but not both, within a particular word of the same object. A C structure is a good example where a field is of fixed type. The tags of such objects can be set immediately after allocation. Once set, the descriptor tags of the object do not change during its lifetime in a program.

However, some objects can contain fields that may hold either terminal or descriptor data or both at different times during execution. A C union which can either hold a pointer or an integer is an example of one such data object. The descriptor bits of such data objects must be properly updated to reflect their content. Finding all the places where the descriptor bits of GC objects must be modified poses a real challenge. The size of the Icon run-time system (30,000 lines of C code) made this task especially challenging.
Most of the ‘C’ structures allocated by the run-time system of Icon do not contain unions and hence their descriptor tags remain unchanged once set immediately after allocation. However an Icon descriptor is a union, whose v-word can either hold a terminal value or a pointer value depending upon the type of value it represents. Since Icon allows heterogeneous aggregates, quite a few dynamically allocated structures contain Icon descriptors as elements. For example, an Icon list can hold elements of any type. A list-element block that holds an element of a list is declared as

```c
struct b_lelem {
    word title;           /* T_Lelem */
    word blksize;         /* size of block */
    union block *listprev; /* previous list-element block */
    union block *listnext; /* next list-element block */
    word nslots;          /* total number of slots */
    word first;           /* index of first used slot */
    word nused;           /* number of used slots */
    struct descrip lslots[NSLOTS]; /* array of slots */
};
```

The tags for `title`, `blksize`, `nslots`, `first`, and `nused` fields are 0 as they contain integer values. The tags for `listprev` and `listnext` words are 1 as they point to other list-element blocks allocated in the GC memory. The list elements are represented by an array of Icon descriptors `lslots[]`. The d-word of each of the Icon descriptors will have a tag of 0. But the tag for the v-word of an `lslot[]` element depends on the Icon value it represents. Initially a slot may hold an Icon descriptor for an integer in which case the descriptor tag for its v-word is a 0. But the list element represented by the slot could be changed to an Icon set during execution, which demands that the tag be changed to a 1. Hence, all locations within the run-time system where any Icon descriptor may be modified are identified and the descriptor tags appropriately set.

The implementation of GC-Icon keeps track of possible changes to all structures that hold Icon descriptors and updates the descriptor tags appropriately. The following macros are used to check if the v-word of an Icon descriptor is a GC descriptor.

```c
#define Pointer(d) ((d).dword & F_Ptr)  /* is p flag set ? */
#define Qual(d) (!((d).dword & F_Qual))  /* is n flag not set ? */
#define GcDescr(d) (Qual(d) || Pointer(d))
```
4.5 Predictive-Need

The traditional implementation of Icon uses predictive-need strategy for memory allocation. Before a request for storage allocation is made, an estimate of the maximum possible memory that can be requested, $n$, is made and intimated to the garbage collector. The garbage collector ensures availability of $n$ bytes of memory and performs garbage collection if required to make sure that the memory is available. This ensures that no garbage collection is performed when the actual allocation request is made. A typical code sequence where predictive-need strategy helps is shown below:

```plaintext
manipulate pointers
allocate $n_1$ words of memory
manipulate pointers
allocate $n_2$ words of memory
  ...
manipulate pointers
allocate $n_r$ words of memory
```

In the absence of predictive-need strategy, garbage collection might be triggered by any of the allocations, requiring that the run-time system be prepared to tend all pointers following each allocation attempt. This adds considerably to the burden of allocation. With predictive-need strategy, a request to set apart $n > n_1 + n_2 + \cdots + n_r$ words is made before any pointers are manipulated at the beginning of the above code segment. Thereafter, pointers can be freely manipulated as it is assured that garbage collection will not occur prior to allocations of $n$ bytes.

Predictive-need strategy lets the traditional implementation make the following assumptions for future allocations up to $n$ bytes:

1. Memory is available immediately upon request and garbage collection doesn’t take place. Hence, pointers may be freely manipulated without fear of any inconsistencies introduced by garbage collection.

2. Memory allocated by subsequent allocations is contiguous. Hence, a string of $n$ bytes, can be allocated in parts by more than one allocation requests and still have a contiguous space for the string.

The GC module does not currently support predictive-need strategy. But, since much of Icon’s run-time implementation assumes predictive-need strategy, it is essential to assure support for the above two assumptions in the modified run-time system also. The GC-Icon implementation does this as follows. It actually `allocates` a region of $n$ words of memory when a request
to set apart $n$ words is made. Support is provided to ensure that space for future memory requests up to $n$ words is granted from this region. This ensures that no garbage collection is required for obtaining the next $n$ words. The support routines also see to it that the space for subsequent memory requests is granted contiguously from the region.

The traditional implementation takes advantage of predictive-need strategy to optimize some space allocations. Consider for example the function \texttt{entab}, which creates a new string by replacing the tabs in the original string with spaces. This routine reserves space for a string of maximum possible size. It then copies characters from the old string space to the new string space, replacing tabs with spaces. At the end of the transformation, it compares the old and new strings to determine if there has been any change. In the absence of any changes it decides not to use the reserved space. The routine just returns the old string. The reserved memory is still available for future allocations.

The GC-Icon implementation is unable to make this optimization as it actually allocates memory when predictive-need request is made. Since memory has been allocated, the memory can be reused only after garbage collection.

### 4.6 Source Descriptors

When the GC module cannot satisfy an allocation request, it indicates the need for a flip. The run-time system then initiates garbage collection by tending each of the the source descriptors pointing into the GC memory. It is crucial that all source pointers are properly identified. If any descriptors are accidently not tended, the referenced data may be treated as garbage and the descriptors would retain invalid pointers to \textit{from-space}.

The set of source descriptors for GC-Icon is identified to be

- values of global variables in the source program
- values of static variables in the source program
- the keyword variable \&subject
- temporary variables used to hold pointers to live data
- all pointers from the interpreter stack

All the pointers from the interpreter stack are tended by scanning each of the frames on the stack. Starting at the stack pointer, all the frames are reached by walking through the pointers to previous frames until the top-most frame is reached.
4.7 Summary

In general, the steps required to transform Icon from conventional garbage collection to hardware-assisted garbage collection can be summarized as:

1. Obtain a thorough understanding of the existing data structures and the garbage collecting mechanism.

2. Divide the programming-language structures into structures to be stored in traditional memory and structures stored in garbage-collected memory.

3. Map the programming language structures to be stored in GC memory to the data objects supported by the GC module.

4. Determine the initial descriptor tags for each of the GC allocated objects.

5. Identify the objects whose descriptor tags may change during run-time. Make sure that the descriptor tags are properly updated.

6. Identify the set of source descriptors and provide support for tending of descriptors.

Similar steps would be required to port other garbage-collected languages to the hardware-assisted garbage collection environment.
5 Performance Analysis

To evaluate the Performance of the GC-Icon implementation, we have run a variety of Icon programs through the modified interpreter. The test suite consists of a collection of 50 Icon programs that exercise various features of Icon. The traditional Icon implementation and GC-Icon implementation both allocate space for code within traditional memory, and use an interpreter stack of 40,000 bytes.

Table 1 reports a number of statistics for programs executed with GC-Icon system. The table lists the results in a non-decreasing order of execution times. Table 2 reports the statistics for the traditional Icon implementation. The meanings of each statistic accompanied by a brief explanation of its significance are provided in the following subsections.

5.1 Instructions

The ‘machine instructions’ columns of Table 1 and Table 2 represent the number of DLX instructions required to run the program.

Compared to the traditional Icon system, GC-Icon system requires fewer instructions for garbage collection as it does not have to implement garbage-collection in software. It only has to tend the source descriptors. The GC module performs the garbage collection in parallel with GC-Icon activities.

However, GC-Icon has to execute additional instructions due to the following overheads:

1. conversion of all Icon strings that appear in the icode to the required format as described in section 4.3. This overhead is directly proportional to the number of procedures, records, global variables, and string literals declared in an Icon program.

2. overhead required to access strings due to the two-level indirection.

3. overhead for polling the status word GC_Status after requesting a service from the GC module.

4. overhead to initialize and update the descriptor tags of GC memory cells.

5. function call overhead to allocate memory and manage descriptors. GC-Icon uses functions in several places where traditional Icon uses in-line code.
<table>
<thead>
<tr>
<th>program number</th>
<th>program name</th>
<th>machine instructions</th>
<th>machine cycles</th>
<th>icache hit rate</th>
<th>dcache hit rate</th>
<th>bus utilization</th>
<th>garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>hello1.icn</td>
<td>14654</td>
<td>83240</td>
<td>67.414</td>
<td>74.318</td>
<td>71.448</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>save.icn</td>
<td>27423</td>
<td>109150</td>
<td>81.168</td>
<td>77.066</td>
<td>67.847</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>transmit.icn</td>
<td>42035</td>
<td>128199</td>
<td>88.146</td>
<td>80.613</td>
<td>63.870</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>hello.icn</td>
<td>42929</td>
<td>139946</td>
<td>86.495</td>
<td>90.020</td>
<td>61.599</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>var.icn</td>
<td>40671</td>
<td>148265</td>
<td>83.945</td>
<td>87.869</td>
<td>64.604</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>versionS.icn</td>
<td>45424</td>
<td>172936</td>
<td>82.608</td>
<td>84.929</td>
<td>65.507</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>proto.icn</td>
<td>97414</td>
<td>192101</td>
<td>96.268</td>
<td>77.883</td>
<td>56.916</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>key.icn</td>
<td>86091</td>
<td>265275</td>
<td>88.230</td>
<td>93.818</td>
<td>61.111</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>over.icn</td>
<td>85369</td>
<td>284208</td>
<td>85.854</td>
<td>92.061</td>
<td>62.010</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>io.icn</td>
<td>11164</td>
<td>290948</td>
<td>91.647</td>
<td>84.057</td>
<td>59.957</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>pdco.icn</td>
<td>165734</td>
<td>299647</td>
<td>96.997</td>
<td>83.645</td>
<td>52.616</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>recogn.icn</td>
<td>128024</td>
<td>350120</td>
<td>90.874</td>
<td>92.400</td>
<td>60.703</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>endtab.icn</td>
<td>175248</td>
<td>389295</td>
<td>94.867</td>
<td>91.710</td>
<td>55.314</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>roman.icn</td>
<td>163904</td>
<td>479801</td>
<td>89.410</td>
<td>93.598</td>
<td>60.969</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>tracer.icn</td>
<td>216436</td>
<td>572532</td>
<td>90.874</td>
<td>95.291</td>
<td>56.772</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>btrees.icn</td>
<td>300090</td>
<td>635051</td>
<td>95.792</td>
<td>95.201</td>
<td>53.337</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>name.icn</td>
<td>367837</td>
<td>753366</td>
<td>96.047</td>
<td>97.253</td>
<td>51.511</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>diffwrds.icn</td>
<td>467358</td>
<td>919871</td>
<td>96.973</td>
<td>97.604</td>
<td>48.518</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>prefix.icn</td>
<td>569164</td>
<td>1224435</td>
<td>95.239</td>
<td>96.928</td>
<td>54.930</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>sieve.icn</td>
<td>648849</td>
<td>1384090</td>
<td>95.815</td>
<td>98.038</td>
<td>49.855</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>model.icn</td>
<td>701987</td>
<td>1594688</td>
<td>94.746</td>
<td>95.488</td>
<td>52.651</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>numeric.icn</td>
<td>763008</td>
<td>1674389</td>
<td>95.047</td>
<td>94.019</td>
<td>53.087</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>kross.icn</td>
<td>746276</td>
<td>1704371</td>
<td>94.524</td>
<td>97.180</td>
<td>52.605</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>scan.icn</td>
<td>867389</td>
<td>1831742</td>
<td>95.698</td>
<td>95.714</td>
<td>51.419</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>check.icn</td>
<td>979293</td>
<td>1996710</td>
<td>96.022</td>
<td>94.700</td>
<td>49.932</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>wordcnt.icn</td>
<td>1125220</td>
<td>2479749</td>
<td>94.953</td>
<td>98.073</td>
<td>51.988</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>mem02.icn</td>
<td>1757571</td>
<td>2848455</td>
<td>99.585</td>
<td>99.692</td>
<td>42.719</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>eval.icn</td>
<td>1498911</td>
<td>3112234</td>
<td>95.799</td>
<td>95.607</td>
<td>50.479</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>string.icn</td>
<td>1517038</td>
<td>3140387</td>
<td>95.950</td>
<td>96.839</td>
<td>50.064</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>augment.icn</td>
<td>1676043</td>
<td>3459990</td>
<td>95.862</td>
<td>95.164</td>
<td>50.773</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>collate.icn</td>
<td>2301791</td>
<td>3593080</td>
<td>99.156</td>
<td>96.521</td>
<td>41.876</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>meander.icn</td>
<td>1923412</td>
<td>3759233</td>
<td>96.779</td>
<td>97.380</td>
<td>47.279</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>errors.icn</td>
<td>2181660</td>
<td>4219039</td>
<td>96.657</td>
<td>97.478</td>
<td>47.294</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>struct.icn</td>
<td>2406252</td>
<td>5353959</td>
<td>94.704</td>
<td>96.137</td>
<td>51.693</td>
<td>0</td>
</tr>
<tr>
<td>35</td>
<td>fnos.icn</td>
<td>2788953</td>
<td>5525642</td>
<td>96.532</td>
<td>96.880</td>
<td>47.601</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 1: GC-Icon (continued)

<table>
<thead>
<tr>
<th>program number</th>
<th>program name</th>
<th>machine instructions</th>
<th>machine cycles</th>
<th>icache hit-rate</th>
<th>dcache hit-rate</th>
<th>bus utilization</th>
<th>garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>math.icn</td>
<td>3176450</td>
<td>941477</td>
<td>88.581</td>
<td>99.346</td>
<td>59.058</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>concord.icn</td>
<td>5089380</td>
<td>11329929</td>
<td>94.641</td>
<td>97.507</td>
<td>52.272</td>
<td>0</td>
</tr>
<tr>
<td>38</td>
<td>permute.icn</td>
<td>6511134</td>
<td>11868092</td>
<td>97.917</td>
<td>97.293</td>
<td>48.170</td>
<td>0</td>
</tr>
<tr>
<td>39</td>
<td>checkfp.icn</td>
<td>4592142</td>
<td>12264340</td>
<td>90.976</td>
<td>98.229</td>
<td>56.985</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>mfsol.icn</td>
<td>5178069</td>
<td>12667975</td>
<td>93.638</td>
<td>98.017</td>
<td>55.036</td>
<td>0</td>
</tr>
<tr>
<td>41</td>
<td>speller.icn</td>
<td>7035087</td>
<td>15044527</td>
<td>95.377</td>
<td>98.761</td>
<td>52.227</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>version7.icn</td>
<td>7910241</td>
<td>16218795</td>
<td>96.124</td>
<td>98.950</td>
<td>51.064</td>
<td>0</td>
</tr>
<tr>
<td>43</td>
<td>mem01.icn</td>
<td>10034402</td>
<td>17723737</td>
<td>99.810</td>
<td>96.416</td>
<td>71.799</td>
<td>6</td>
</tr>
<tr>
<td>44</td>
<td>large.icn</td>
<td>9749887</td>
<td>17920878</td>
<td>97.001</td>
<td>99.353</td>
<td>39.894</td>
<td>0</td>
</tr>
<tr>
<td>45</td>
<td>endtab1.icn</td>
<td>16318626</td>
<td>30866258</td>
<td>96.862</td>
<td>98.612</td>
<td>49.372</td>
<td>1</td>
</tr>
<tr>
<td>46</td>
<td>gc2.icn</td>
<td>19271734</td>
<td>36620714</td>
<td>97.451</td>
<td>97.622</td>
<td>48.236</td>
<td>2</td>
</tr>
<tr>
<td>47</td>
<td>mindfa.icn</td>
<td>24687260</td>
<td>50264045</td>
<td>96.261</td>
<td>97.108</td>
<td>50.259</td>
<td>4</td>
</tr>
<tr>
<td>48</td>
<td>rsg.icn</td>
<td>36186117</td>
<td>75274308</td>
<td>95.800</td>
<td>97.528</td>
<td>50.322</td>
<td>4</td>
</tr>
<tr>
<td>49</td>
<td>gc1.icn</td>
<td>265506240</td>
<td>374976014</td>
<td>99.851</td>
<td>86.796</td>
<td>54.117</td>
<td>722</td>
</tr>
<tr>
<td>50</td>
<td>typsum.icn</td>
<td>349152911</td>
<td>672319030</td>
<td>97.396</td>
<td>97.375</td>
<td>48.269</td>
<td>385</td>
</tr>
</tbody>
</table>

Table 2: Traditional Icon

<table>
<thead>
<tr>
<th>program number</th>
<th>program name</th>
<th>machine instructions</th>
<th>machine cycles</th>
<th>icache hit-rate</th>
<th>dcache hit-rate</th>
<th>bus utilization</th>
<th>garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>hello1.icn</td>
<td>10900</td>
<td>76176</td>
<td>57.170</td>
<td>75.250</td>
<td>72.986</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>save.icn</td>
<td>21879</td>
<td>96803</td>
<td>77.507</td>
<td>76.502</td>
<td>68.828</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>transmit.icn</td>
<td>35616</td>
<td>115168</td>
<td>86.497</td>
<td>80.649</td>
<td>64.356</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>hello.icn</td>
<td>39308</td>
<td>144398</td>
<td>83.427</td>
<td>90.777</td>
<td>63.706</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>var.icn</td>
<td>34817</td>
<td>133568</td>
<td>82.264</td>
<td>88.186</td>
<td>64.887</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>versionS.icn</td>
<td>40146</td>
<td>166094</td>
<td>80.087</td>
<td>85.280</td>
<td>66.556</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>proto.icn</td>
<td>84416</td>
<td>167847</td>
<td>95.939</td>
<td>75.692</td>
<td>56.819</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>key.icn</td>
<td>75674</td>
<td>270704</td>
<td>84.552</td>
<td>94.796</td>
<td>64.288</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>over.icn</td>
<td>80133</td>
<td>280276</td>
<td>84.845</td>
<td>92.666</td>
<td>63.084</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>io.icn</td>
<td>102450</td>
<td>274336</td>
<td>91.080</td>
<td>84.226</td>
<td>60.402</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>pdc0.icn</td>
<td>151488</td>
<td>273309</td>
<td>96.837</td>
<td>83.114</td>
<td>52.133</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>recogn.icn</td>
<td>120206</td>
<td>333977</td>
<td>90.430</td>
<td>93.199</td>
<td>61.011</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>endtab.icn</td>
<td>150379</td>
<td>357963</td>
<td>93.758</td>
<td>93.522</td>
<td>57.224</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>roman.icn</td>
<td>148376</td>
<td>455078</td>
<td>88.192</td>
<td>95.275</td>
<td>62.165</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>tracer.icn</td>
<td>210165</td>
<td>570512</td>
<td>90.322</td>
<td>95.537</td>
<td>57.368</td>
<td>0</td>
</tr>
<tr>
<td>program number</td>
<td>program name</td>
<td>machine instructions</td>
<td>machine cycles</td>
<td>icache hit-rate</td>
<td>dcache hit-rate</td>
<td>bus utilization</td>
<td>garbage collections</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------</td>
<td>----------------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>16</td>
<td>btrees.icn</td>
<td>279506</td>
<td>613729</td>
<td>95.214</td>
<td>95.666</td>
<td>54.681</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>name.icn</td>
<td>358437</td>
<td>776212</td>
<td>95.268</td>
<td>97.833</td>
<td>53.480</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>diffwrds.icn</td>
<td>441088</td>
<td>911106</td>
<td>96.207</td>
<td>98.476</td>
<td>50.727</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>prefix.icn</td>
<td>546279</td>
<td>1114469</td>
<td>96.132</td>
<td>97.666</td>
<td>53.832</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>sieve.icn</td>
<td>623254</td>
<td>1459061</td>
<td>94.264</td>
<td>98.709</td>
<td>53.231</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>model.icn</td>
<td>607819</td>
<td>1578610</td>
<td>92.349</td>
<td>96.666</td>
<td>57.434</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>numeric.icn</td>
<td>702459</td>
<td>1647613</td>
<td>93.777</td>
<td>94.642</td>
<td>55.029</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>kross.icn</td>
<td>695573</td>
<td>2068597</td>
<td>89.145</td>
<td>98.480</td>
<td>60.001</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>scan.icn</td>
<td>814762</td>
<td>1822132</td>
<td>94.726</td>
<td>96.521</td>
<td>53.722</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>check.icn</td>
<td>940580</td>
<td>2035754</td>
<td>95.184</td>
<td>95.383</td>
<td>52.101</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>wordcnt.icn</td>
<td>1065680</td>
<td>2357511</td>
<td>94.972</td>
<td>98.788</td>
<td>52.383</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>mem02.icn</td>
<td>1718303</td>
<td>2793381</td>
<td>99.576</td>
<td>99.772</td>
<td>43.032</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>eval.icn</td>
<td>1367837</td>
<td>3120143</td>
<td>91.165</td>
<td>96.124</td>
<td>53.660</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>string.icn</td>
<td>1419128</td>
<td>3192125</td>
<td>94.542</td>
<td>97.482</td>
<td>53.008</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>augment.icn</td>
<td>1579119</td>
<td>3546815</td>
<td>94.356</td>
<td>95.953</td>
<td>53.641</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>collate.icn</td>
<td>2204963</td>
<td>3615870</td>
<td>99.011</td>
<td>96.362</td>
<td>43.923</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>meander.icn</td>
<td>1770587</td>
<td>3197708</td>
<td>97.791</td>
<td>98.743</td>
<td>46.092</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>errors.icn</td>
<td>2158105</td>
<td>4872645</td>
<td>94.215</td>
<td>97.884</td>
<td>52.817</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>struct.icn</td>
<td>2200581</td>
<td>5213947</td>
<td>93.727</td>
<td>97.257</td>
<td>54.284</td>
<td>0</td>
</tr>
<tr>
<td>35</td>
<td>fnos.icn</td>
<td>2690511</td>
<td>5846453</td>
<td>95.179</td>
<td>97.720</td>
<td>51.258</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>math.icn</td>
<td>3041813</td>
<td>9572471</td>
<td>87.229</td>
<td>99.641</td>
<td>60.602</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>concord.icn</td>
<td>4810912</td>
<td>10827207</td>
<td>94.671</td>
<td>98.776</td>
<td>53.875</td>
<td>0</td>
</tr>
<tr>
<td>38</td>
<td>permute.icn</td>
<td>5992172</td>
<td>11635889</td>
<td>96.938</td>
<td>99.252</td>
<td>51.497</td>
<td>0</td>
</tr>
<tr>
<td>39</td>
<td>checkfp.icn</td>
<td>4437184</td>
<td>12636345</td>
<td>89.607</td>
<td>98.760</td>
<td>58.821</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>mfsol.icn</td>
<td>4831124</td>
<td>11736967</td>
<td>93.771</td>
<td>98.982</td>
<td>55.641</td>
<td>0</td>
</tr>
<tr>
<td>41</td>
<td>speller.icn</td>
<td>6712316</td>
<td>14334782</td>
<td>95.311</td>
<td>99.541</td>
<td>52.486</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>version7.icn</td>
<td>7360957</td>
<td>14476729</td>
<td>96.767</td>
<td>99.641</td>
<td>50.230</td>
<td>0</td>
</tr>
<tr>
<td>43</td>
<td>mem01.icn</td>
<td>9835941</td>
<td>12289459</td>
<td>99.657</td>
<td>99.923</td>
<td>33.218</td>
<td>12</td>
</tr>
<tr>
<td>44</td>
<td>large.icn</td>
<td>9410397</td>
<td>17876219</td>
<td>97.054</td>
<td>99.549</td>
<td>43.011</td>
<td>0</td>
</tr>
<tr>
<td>45</td>
<td>endetab1.icn</td>
<td>15676336</td>
<td>28768528</td>
<td>97.373</td>
<td>99.516</td>
<td>49.068</td>
<td>0</td>
</tr>
<tr>
<td>46</td>
<td>gc2.icn</td>
<td>18268804</td>
<td>35640604</td>
<td>97.053</td>
<td>98.861</td>
<td>50.593</td>
<td>5</td>
</tr>
<tr>
<td>47</td>
<td>mindfa.icn</td>
<td>23913959</td>
<td>48279129</td>
<td>96.271</td>
<td>98.547</td>
<td>50.381</td>
<td>3</td>
</tr>
<tr>
<td>48</td>
<td>rsg.icn</td>
<td>34463162</td>
<td>71487400</td>
<td>95.996</td>
<td>98.579</td>
<td>51.927</td>
<td>6</td>
</tr>
<tr>
<td>49</td>
<td>gc1.icn</td>
<td>389279615</td>
<td>514892143</td>
<td>99.883</td>
<td>99.963</td>
<td>36.145</td>
<td>1258</td>
</tr>
<tr>
<td>50</td>
<td>typsum.icn</td>
<td>466317214</td>
<td>843761917</td>
<td>98.149</td>
<td>96.033</td>
<td>45.563</td>
<td>59</td>
</tr>
</tbody>
</table>
Graph 1

% increase

program number

instructions

cycles
Because of the above overheads, we can expect GC-Icon to execute more instructions for programs that do not garbage collect. But as programs do more garbage collections, the gain due to the hardware-assisted garbage-collecting protocol may overcome the above overheads. These expectations are reflected in the measurements for the test suite. Graph 1 shows how many more instructions and cycles the GC-Icon system executes when compared to the traditional Icon system for each of the Icon programs. The above-mentioned overheads have greater impact on smaller programs. The impact reduces as the program size increases and as the programs perform garbage collections. The results show that all programs which perform no garbage collection execute 5% to 15% more instructions. Programs gc1.icn and typsum.icn trigger a lot of garbage collections resulting in the dramatic decrease in instructions executed. Further, gc1.icn has no string operations and hence overhead 2 mentioned above does not apply to this program. Programs reg.icn, mindfa.icn and endetab1.icn also perform some garbage collections, but much less than gc1.icn and typsum.icn. These programs involve string operations. Hence the gain due to a simpler garbage collecting protocol is unable to offset the additional load-time and run-time costs. GC-Icon executes about 5% more instructions for these programs when compared to the traditional system.

An estimate of the load-time overhead on the instruction count is given by programs pdco.icn and proto.icn. The two programs define a good number of procedures and variables but just exit without executing any operations. The run-time systems loads the code, performs the initial setup operations and then exits. Hence, the initial overhead entirely accounts for the increase in the number of instructions.

### 5.2 Icache Hit Rate

The ‘icache hit rate’ columns of Table 1 and Table 2 report the percentage of instruction fetches that hit the instruction cache. It is observed that the icache hit rates for GC-Icon system are higher than the hit rates for the traditional system. Graph 2 gives the percentage increase in icache hit rates for each of the 50 programs in the hardware-assisted implementation. A decrease in the hit rates by at most 1% is seen for very few programs. The increased icache hit rates for GC-Icon can be explained by the fact that the increase in instructions, as described earlier, is partly due to polling of the status word after a service request to the garbage-collection module. The instructions that poll the status word are likely to be hits as they are executed repeatedly in a loop. Another reason to expect GC-Icon to generally exhibit a higher icache hit rate is that GC-Icon uses several functions to replace
Graph 2
in-line code in the traditional system. In cases where several occurrences of the in-line code are replaced with calls to the same function, code locality is improved. Furthermore, it is important to understand that icache hits occur only when the current instruction was expected previously. Because of increases in the number of instructions required to perform certain tasks, the bodies of many loops contain more instructions in GC-Icon than the traditional system. This results in greater opportunities to reuse previously fetched instructions.

The icache hit rates for pdco.icn and proto.icn support the above conclusions. These programs do not have any additional instructions for polling or accessing strings. As all additional instructions are for the initial setup only, they have no scope for improving locality of reference. So the icache hit rates for these programs are only slightly affected.

Other programs such as wordcnt.icn, mem02.icn, prefix.icn and meander.icn show slightly poorer icache performance in the GC-Icon implementation. This presumably results from less localized behaviour of the extra instructions required to implement GC-Icon.

### 5.3 Dcache Hit Rate

The ‘dcache hit rate’ of Table 1 and Table 2 specify the percentage of data fetches that are found in the data cache. The dcache hit rates do not take the write operations into account. Graph 3 gives the percentage increase in dcache hit rates for GC-Icon implementation when compared to the traditional implementation.

GC-Icon is observed to have poorer dcache hit rates in general. This difference in dcache hit rates could result from the extra-level of indirection required for access to strings. GC-Icon holds an additional word in the cache for each Icon string descriptor. This causes an extra cache miss each time a string is referenced, and increases the likelihood of collisions which will eventually result in additional cache misses.

The garbage-collection hardware completely invalidates the data cache on a flip. This could be another possible reason for the lower dcache hit rates for GC-Icon Implementation.

### 5.4 Bus Utilization

The ‘bus utilization’ columns of the tables report the percentage of total bus cycles during which the bus is serving to pass information between modules connected to the bus. Traffic on the bus depends on the cache hit rates and also on the amount of data exchanged between the mutator and the
Graph 3

**dcache hit rate**

- Program number
- % increase

![Graph showing dcache hit rate with program number on the x-axis and % increase on the y-axis.](image-url)
garbage-collection module. With greater cache hit rates, we have lower bus utilization as fewer memory operations are required. Graph 4 shows the percentage increase in bus utilization for GC-Icon implementation. GC-Icon has a lower utilization of the bus for almost all Icon programs in the test suite. For most programs, the number of instructions fetched is over 10 times greater than the number of data words fetched. So icache performance is much more important than dcache performance. Hence, better icache hit rates observed for GC-Icon result in a lower utilization of the bus in spite of the lower dcache hit rates and the data exchanged between the mutator and the GC module.

mem01.icn and gc1.icn are specific cases where the bus utilization increases by nearly 110% and 50% respectively. These two programs are artificial ones written to exercise the garbage collector. mem01.icn repeatedly creates new Icon strings while gc1.icn repeatedly creates large Icon lists. These programs do little other work. This unnatural system load places a large burden on the system bus, which must take part in allocating and initializing each object. The decrease in dcache hit rates for these programs, as seen in Graph 4, is also partially responsible for the large increase in bus utilization.

5.5 Machine Cycles

The columns titled ‘machine cycles’ in Table 1 and Table 2 report how many DLX machine cycles are required to implement each of the test programs.

With an increase in the instruction count, we can expect an increase in the execution cycles also. However, due to the changes in memory management protocols for the two Icon implementations, the number of machine cycles for the execution of an Icon program is influenced by the changes in icache hit rate, dcache hit rate and the bus utilization also.

Graph 1 shows that the cycles for overall execution of Icon programs that do no garbage collection is usually more for the GC-Icon system. The additional instructions due to the earlier-mentioned overheads result in this increase of cycles. However the number of cycles do not increase in proportion with the increase in number of instructions. The increase in percentage of cycles is almost always less than the increase in number of instructions. In other words, the CPI for GC-Icon implementation is almost always less. This decrease in CPI may be attributed to the following:

1. The hardware-assisted system has higher icache hit rates and lower data cache hit rates. But since instruction fetches far outnumber data fetches, we have a better hit ratio for memory operations leading to
bus utilization

% increase

Graph 4
lower CPIs.

2. Bus utilization is higher in the traditional system. Hence there is more contention for the bus in the traditional system for these programs.

Programs `kross.icn` and `errors.icn` take nearly 15% fewer cycles on the hardware implementation. This relates to the higher icache hit rates and lower bus utilization for these two programs. It is similarly observed that programs such as `hello.icn`, `var.icn`, and `key.icn`, which have significantly better icache hit rates and lower utilizations, have much lower CPIs on GC-Icon system.

GC-Icon takes more cycles to execute programs such as `prefix.icn`, `meander.icn` and `mem01.icn`. The first two of these programs have a higher bus utilization and lower icache hit rates which gets reflected in higher execution cycles. GC-Icon takes nearly 45% more cycles for `mem01.icn`. This increase can be attributed to the huge increase in bus utilization.

### 5.6 Number of Collections

The number of garbage collections performed by the Icon implementations are listed under columns ‘garbage collections’ of Table 1 and Table 2. Except for `typsum.icn`, test-runs of programs with GC-Icon system use 256K bytes of GC memory and hence garbage collect every time 128K bytes are allocated (each semi-space is half as large as the total region). The traditional Icon system uses a total of 128K bytes of garbage-collected memory for these programs.

Program `gc1.icn` characterizes the best case for GC-Icon implementation in terms of the number of garbage collections. Though both the implementations have 128K bytes of memory for heap allocation, it is seen that the traditional implementation makes 1258 collections compared to 722 collections by the GC-Icon system. This is essentially because the traditional system divides the heap into string and block regions and allocates strings and blocks only from their respective regions. Since `gc1.icn` allocates only lists and never strings, the traditional implementation is effectively limited to only 64K of block region and the 64K of string region remains unused. In contrast, GC-Icon has the entire 128K available for allocating blocks. This strategic difference translates into the huge difference in the number of collections and part of the difference in execution times.

Similarly, the block region of `mem01.icn` goes unused since it allocates only strings. Hence, GC-Icon performs 50% fewer garbage collections than the traditional Icon system.
The significant difference between the number of collections for `typsum.icn` can be explained by the different memory sizes available for the two Icon implementations. The traditional implementations use a block region of 278K and a string region of 64K. The two regions together add to 342K of memory. The GC-Icon implementation works with a GC memory of size 512K and therefore has only 256K available for allocation during run time.

6 Conclusions and Future Work

The traditional implementation of Icon is successfully adapted to perform hardware-assisted garbage collection. We now have an Icon implementation that provides bounded-time response to all memory operations. The hardware protocol introduces some overhead on the overall run time of a program. This overhead is however observed to increase the run time of ‘typical’ programs by at most 15%. For programs that require a lot of garbage collection, the modified implementation out-performs the traditional implementation. The gain due to concurrent hardware-assisted garbage collection more than compensates for the performance loss due to the protocol overhead.

A few specific programs run very slow on the hardware-assisted Icon implementation. Such behaviour needs to be investigated further. Even those programs that conform to our expectations should be analyzed in greater detail to verify which factors influence their performance.

The current work focuses mainly on adapting the traditional version to the hardware protocol with minimal changes. No attempt is made to alter the basic design to reduce the garbage-collection protocol overheads. Improvements can be made to the design, with the garbage-collection module in view. A possible improvement would be to use a one-word Icon descriptor instead of a two-word Icon descriptor so that the descriptor-word always points to the value it represents. This simplifies initialization of newly allocated objects, reduces the sizes of all Icon objects, and eliminates the need to update descriptor tags. Given that Icon objects would occupy less memory, we would expect to see improved dcache performance. This proposed improvement to the GC-Icon implementation involves a lot of work since the software assumes two-word descriptors throughout. It may also require changes to the icode format so that proper Icon descriptors are created.
Acknowledgements

I wish to express my sincere appreciation for my major professor Dr. Kelvin D. Nilsen who has helped me throughout the course of my research with on-line help and motivating discussions. He has been friendly, flexible and understanding. I would like to thank Dr. Suresh Kothari for providing guidance throughout my graduate study. I would also like to thank Dr. Doug Jacobson for agreeing to be on my committee.
References


