An Hybrid Model for Very High Level Threads

Jafar Al-Gharaibeh
University of Idaho
Department of Computer Science
Moscow, ID, USA
+1-208-301-0338
jafara@vandals.uidaho.edu

Clinton Jeffery
University of Idaho
Department of Computer Science
Moscow, ID, USA
+1-208-885-4789
jeffery@uidaho.edu

Kostas N. Oikonomou
AT&T Labs Research
200 Laurel Avenue
Middletown, NJ, USA
+1-732-420-5902
ko@research.att.com

ABSTRACT

Languages with multiple paradigms or other special-purpose features often are implemented in ways that make true concurrency difficult in the virtual machine or runtime system. Several popular languages feature a global interpreter lock that limits them to pseudo-concurrency. This paper presents lessons learned in developing true concurrency for a goal-directed, object-oriented language called Unicon. Parts of the work were anticipated, such as switching to thread-safe C library functions, while other parts were a surprise, such as eliminating race conditions in self-modifying virtual machine instructions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *concurrent programming structures.*

General Terms

Design, Languages.

Keywords

threads, concurrency, language design, garbage collection.

1. INTRODUCTION

Many rapid prototyping and scripting languages feature no concurrency or only user-level concurrency via a global interpreter lock [4]. These languages may have large code bases that are unable to take advantage of the advances in modern multi-core computer hardware. Such languages must acquire true concurrency or face a lingering decline as multi-core computing becomes more and more central in hardware design. This paper describes lessons learned from extending one such language, Unicon, to support true concurrency.

Adding concurrency to Unicon was motivated by a request from AT&T Labs Research, where the language has been used to implement a network performability ("performance + reliability") analysis tool called nperf [15]. nperf is computationally intensive, and seriously needed the benefits of parallelization.

2. BACKGROUND

Different programming languages employ different techniques to achieve parallelism. Haskell is an example functional language for which implicit concurrency has been implemented [9]. The nature of computations in functional languages, which are free of side effects and depend heavily on graphs, facilitates such implicit parallelism by assigning different parts of the graph to different threads [8][18].

Paalvast et al, described Booster [17], a high level parallel programming language that can be translated into lower level languages such as FORTRAN and C. The language features the idea of separation of algorithm description from algorithm decomposition and representation. After the algorithm and the data/code decomposition are described, a transformation can be done automatically to achieve concurrency [17].

ALLOY [13] is an example of a weakly-typed, statically-scoped parallel programming language based on the functional and object-oriented paradigms. ALLOY is similar in many aspects to Unicon and its predecessor Icon. It provides features to express parallel algorithms and their related control structures including synchronization and mutual exclusions. Mitsolides and Harrison [14] describe the concept of "replicators" in ALLOY, which are control structures that provide a new view of generator. Replicators help deal with problems related to generators in a concurrent environment.

Relatively popular modern concurrency-oriented, user-friendly languages include Java (java.com) and Erlang (erlang.org). Java features portable true concurrency and mitigates the pain of writing locking code for concurrency synchronization using monitor semantics. Erlang is arguably higher level, with a more esoteric functional syntax and a message passing model for communication and synchronization.

The most famous scripting language, Python, and its popular competitor Ruby feature a Global Interpreter Lock, or GIL. Experimental implementations such as Jython and Iron Python feature true concurrency, but users tend to stick with original interpreters such as CPython despite their use of a GIL. Recent work on CPython has made progress towards eliminating the GIL, but it is still present.

The SR (Synchronizing Resources) programming language is a language for developing parallel programs. The parallelism in SR is mainly exploited through message passing [1], even though it provides mechanisms for techniques like rendezvous and remote procedure call. SR has a highly expressive message passing

PMAM 2012, February 26, 2012, New Orleans LA, USA.

interface, which is useful in many applications, but it is considered by some to provide a low level abstraction in applications where other concurrency mechanisms can be used [3].

OpenMP (Open Multi-Processing) is a shared memory, multi-platform API. It achieves concurrency through a unique approach using compiler directives, and a combination of library routines and environment variables [5]. The directives enable the compiler and the runtime system to create several threads when appropriate to accomplish certain tasks, without direct intervention from the programmer. OpenMP provides a simple mechanism for parallelizing some parts of the program. Most of the work is handled automatically. The directives can be skipped by the compiler if a sequential version is required. Despite its simplicity, OpenMP requires compiler support, and in some cases can introduce race conditions that are hard to debug, especially with the little control that the programmer has over the threads. These issues and more are addressed in [7] [12].

3. LANGUAGE DESIGN

Compared with the very rich related work, the goal of this work is modest. Unicon is an object-oriented descendant of the Icon programming language [6] [10]. This section presents the design of the concurrency features added to Unicon, with the express goal of minimizing the additions required at the source code level, and providing a simpler higher-level concurrency mechanism.

Unicon's ancestor Icon integrates Prolog-like goal-direction and implicit backtracking within a conventional syntax and an imperative semantic core. Icon's traditional domain is string and file processing, and the rapid development of experimental algorithms and data structures. Unicon is a superset that extends Icon along two dimensions: features such as classes and packages for larger-scale projects, and extensive access to modern I/O capabilities such as graphics, networking, and databases.

3.1 Co-expressions

Because Icon features synchronous, ultra-lightweight threads called co-expressions, the easiest graft of concurrency onto Unicon was to extend that type. Co-expressions are similar to coroutines, except that they can be composed from arbitrary expressions. Co-expressions are finer-grained than coroutines, which are limited to the function call level. Co-expressions are created by a control structure (create expr) and control is transferred between co-expressions by either implicit means (producing a result or *failing*) or an explicit transmit-and-activate operator [sendvalue] @ co-expression. This primitive is always a *rendezvous* in Icon, since co-expressions are synchronous.

Although co-expressions can be implemented on top of POSIX threads, at the language level they feel far simpler. They require no locking and provide easy data sharing via global and heap variables. Co-expressions also have a native implementation in assembler code on popular processors that is much faster than the pthreads implementation, on the order of ~100x on modern processors. For example the x86_64 native co-expression switch consists of 7 instructions. But unlike this native implementation, the pthreads implementation can be scheduled by the operating system to make use of multiple cores. It was natural to build Unicon's concurrency facilities on top of the pthread library, but then run native co-expressions within each thread—an hybrid model.

3.2 Minimalist Concurrent Extension

From a language design standpoint, adding concurrency to Unicon consisted of: (1) adding a way to tell co-expressions to execute concurrently, and (2) extending the activation operator @ for thread synchronization and communication. These were achieved by introducing a control structure thread expr analogous to create expr, and extending the @ operator to maintain producer-consumer queues for asynchronous operation.

Unicon's version of the @ operator enables the following forms of communication:

Name	Example	Description
send/receive	x := y @ thread	If queues empty, rendezvous
send	y @thread	No wait; no answer needed
receive	x := @thread	No wait if message is already here
produce	y @	Send to any receiver
consume	x := @	Receive from any sender

The language was also extended with seven new built-in functions for high-level access to mutual exclusion primitives and condition variables. The built-in function wait() was extended to allow a co-expression argument to support the semantics of join() in thread programming. Also, new syntax was introduced for critical regions, to provide a higher-level alternative to the mutual exclusion functions. The syntax is

```
critical mtx: expr
```

meaning that expr will be serialized on the mutex mtx, which will be automatically locked before executing expr and unlocked after that. The remainder of this paper focuses on the lessons learned from the implementation of concurrency within the legacy virtual machine interpreter, known as *iconx.*

3.3 Code Example

Figure 1 presents a code example in Unicon for solving a simple problem; computing the summation of a list of elements. Two versions are presented in the figure. Lines 1 through 8 represent the sequential version, while the remaining code is the parallel version. The parallel version takes some extra steps to create the threads and divide the job to four threads and collect their results. For a huge list with millions of elements, the parallel code runs nearly four times faster than the sequential version.

This example is not representative of a typical real world application. The sequential version is trivial, making the addition of a few more lines of code very noticeable. In a more serious context (such as, nperf mentioned earlier in this paper), the code that is added to enable parallel execution may be negligible compared to the size of the original code. The extra code in the figure that is added to the parallel version is responsible for creating the threads and dividing the work among them (lines 14 through 18). listsum() in the sequential version can be compared to slicesum() in the parallel version. listsum() finds the summation of the list while the latter finds the summation of a subset of the list (elements start through start+length) allowing different threads to work on different slices of the same list. slicesum() also includes a mutex (region) to protect the global sum where all threads accumulate their results at the end (line 28).

```
1. # sequential version. : loop through the entire list
2. # simply summating them and return the result
3. procedure listsum(L)
4.    sum:=0
5.    every 1 to *L do
6.        sum+:=L[i]
7.    return sum
8. end
9. #----------------------------------------------------------------------#
10.   # parallel version:  divide the work between 4 threads.
11.   global sum
12.   procedure listsum(L)
13.      # create the threads, prepare their data and fire them
14.      thrds := list();   sum := 0; Q := *L/4
15.     every put(thrds,thread(create slicesum(L,(0 to 3)* Q, Q)))
16.      every wait(!thrds)      # wait for the threads to finish
17.      return sum
18.   end
19.   # find the summation of n to m elements in list L.
20.   procedure slicesum(L, start , length)
21.      static region
22.      initial region := mutex()
23.      tot := 0
24.      every tot +:= L[start to start+length]
25.      critical region:  sum+ := tot
26.   end
```

Figure 2. Computing the summation of an entire list elements, sequential [1:8] and parallel [10:26]

4. VIRTUAL MACHINE

The Unicon virtual machine is a modified version of the Icon virtual machine. The extensions made to support true concurrency would apply equally to the Icon implementation. Similar languages face analogous situations, where the lessons learned here may be applicable.

4.1 VM Registers

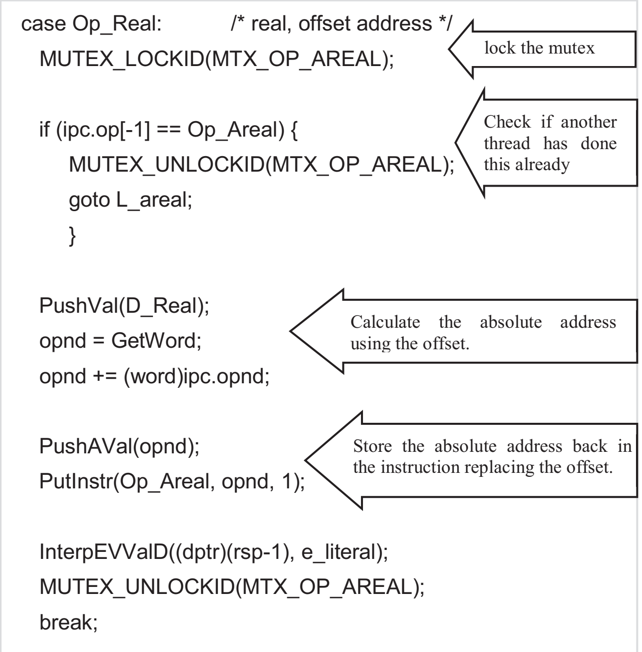
Virtual machine "registers", which were formerly global variables in the VM C code, include the program counter, stack pointer, the several frame pointers necessary to manage calls, returns, suspension and resumption on the stack, and other pieces of virtual machine state. To add concurrency, these elements of state are moved into a struct threadstate that is allocated for each thread in thread local storage. The threadstate structure is fairly large, under the assumption that memory is cheap whereas thread synchronization is expensive. POSIX threads API calls are used to obtain references to the threadstate structure where it is needed.

A key concern when using thread-local storage is performance. Initially, all references to VM registers were replaced, via macros, by references through a global threadstate pointer variable declared with the storage specifier __thread that is available in some compilers such as gcc and Sun's (now Oracle's) cc. This implementation was straightforward but incurred a substantial performance cost. In addition, it was not available in the pthreads implementation on OS X and the pthreads subset for Mingw gcc on Microsoft Windows. Switching to the more portable pthreads API for thread-local storage using pthread_getspecific() allowed a faster implementation than __thread. The straightforward implementation invoked pthread_getspecific() once at the top of each C function containing references to VM register and state variables. The many calls to this API wherever the VM state is referenced in the runtime system still entail an unattractive performance cost. Passing a pointer to the thread state as an extra parameter on the stack where it is needed as discussed in Section 7 will finally mitigate this cost.

4.2 Self-Modifying Instructions

The Unicon virtual machine has seven instructions with integer operands denoting offsets to other instructions, or to static data in the bytecode. They are used for literals, globals, static variables, and also for instructions that jump to addresses, such as Op_Goto. For performance reasons, when the opcode first executes, offsets are converted to pointers. The opcode is modified to indicate that the operand is a pointer, and the instruction proceeds at full speed on subsequent executions.

This implementation introduces a race condition when multiple threads execute the self-modifying instruction at the same time. The problem was solved by adding a mutex for each self-modifying instruction opcode. Figure 2 shows an example self-modifying virtual machine instruction protected with a mutex. Mutex contention could be reduced further by allocating a separate mutex for each instance of each opcode (requiring a number of mutexes proportional to the size of the program) instead of using just seven mutexes for the seven self-modifying opcodes; it is unlikely that this would be worth implementing, since each instance of a self-modifying instruction uses its mutex only once when it replaces itself. Some intermediate granularity such as allocating up to these 7 mutexes per procedure/method, class, or object file might be shown beneficial by further study.



```
case Op_Real:          /* real, offset address */
   MUTEX_LOCKID(MTX_OP_AREAL);

   if (ipc.op[-1] == Op_Areal) {
      MUTEX_UNLOCKID(MTX_OP_AREAL);
      goto L_areal;
      }

   PushVal(D_Real);
   opnd = GetWord;
   opnd += (word)ipc.opnd;

   PushAVal(opnd);
   PutInstr(Op_Areal, opnd, 1);

   InterpEVValD((dptr)(rsp-1), e_literal);
   MUTEX_UNLOCKID(MTX_OP_AREAL);
   break;
```

lock the mutex
Check if another thread has done this already
Calculate the absolute address using the offset.
Store the absolute address back in the instruction replacing the offset.

Figure 1. Self modifying instruction protected by a mutex

5. RUNTIME SYSTEM

Achieving thread-safety in the implementation without a global interpreter lock required extensive modifications to the runtime system, notably the garbage collector.

5.1 Allocation and Separate Heaps

Allocation is a frequent operation for which speed is a top priority. Unicon programs start with a heap consisting of one string and one block (structure) region, and allocate more regions as needed. At any moment, the string and block regions used for allocation are together referred to as the current heap. If the memory request is bigger than what is available in the current heap, other regions are checked. If enough memory is found, the current heap is switched to use the satisfying region. Garbage collection is triggered if there is not enough free memory in the heaps. If the request cannot be granted even after the garbage collection, a new region is allocated.

Using a shared heap with multiple threads, when a thread allocates memory and updates the heap's state variables, the operation must be protected by a mutex. The locks required in the shared heap strategy were slowing down the frequent task of memory allocation, especially when there was more than one thread running, limiting the scalability of this approach. It was decided then to switch to use private heaps to solve this problem. The terms "private heap" and "public heap" are introduced to describe allocation controls in the heaps that avoid the need for mutexes; the terms do not imply restrictions on memory accesses. Separate, per-thread heaps allow threads to allocate from their own private heaps at full speed. Heaps that are not owned by any thread are referred to as public heaps. Public heaps are a product of threads out-growing their private heaps. This happens when a thread requests more memory and that request can only be granted by allocating a new private heap. A program starts with no public heaps and one private heap for the main thread. Figure 3 shows the heap layout with regards to threads.

All heaps (public and private) are visible to all threads, so any thread can access variables and data structures in any heap at any time. The accesses can be protected at the application level, if needed. In other words, *a thread can allocate memory only in its private heap, but can access (read and write) variables in all heaps*.

When a memory request cannot be granted in the private heap, public heaps are checked to see if any of them has enough free memory to grant the request. If so, the private heap is swapped with the public heap, changing the private heap to public and vice-versa. The pool of public heaps is protected by a mutex. If public heaps do not have enough free memory, a garbage collection will take place. If the memory request cannot be granted after that, the thread allocates a fresh heap.

5.2 Garbage Collection

The frequency of garbage collection depends on heap size and application memory usage patterns. Historically, many Icon applications ran to completion without ever garbage collecting. However, modern object oriented event-driven applications run for long periods of time, and garbage collect proportionally often, despite increases in physical memory and heap size. When garbage collection does occur, it concerns every thread.

Since garbage collection does not happen often and its negative impact on performance can be greatly reduced using large heaps, garbage collection is kept as simple as possible under concurrent execution. To allow safe access to data in other threads' heaps, garbage collection suspends all running threads except the thread that triggered it, referred to as the *GC thread*. The GC thread performs a conventional garbage collection, during which the program runs sequentially without safety issues.

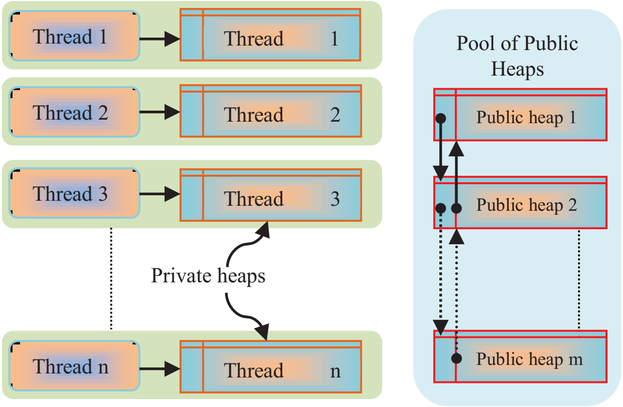
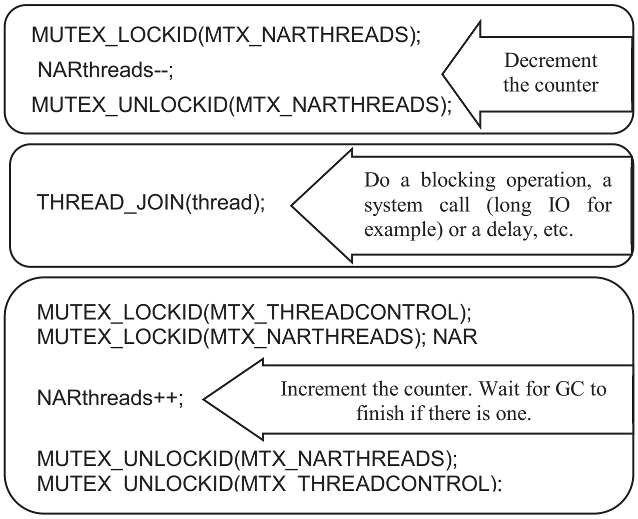


Figure 3. Threads, private heaps and public heaps

The GC thread first locks the garbage collection mutex MTX_THREADCONTROL, and then sets a global flag that announces the need to perform garbage collection. It directs all running threads to converge to a special routine called thread_control, the coordination point for thread suspension and resumption. All threads once per virtual machine instruction check the global flag. The runtime system keeps track of how many threads are running using a global counter NARthreads. Before a thread blocks on a mutex it decrements the counter, and increments it after unblocking. The counter itself is protected by a mutex and can be incremented only if there is no garbage collection request pending or taking place (Figure 4).

After a call for garbage collection causes a thread to enter the thread_control function, the thread decrements the running threads counter NARthreads and goes to sleep on the gc condition variable. The GC thread tests the running threads counter until its value decreases to 1, meaning that the GC thread



```
MUTEX_LOCKID(MTX_NARTHREADS);
 NARthreads--;
MUTEX_UNLOCKID(MTX_NARTHREADS);
```

```
THREAD_JOIN(thread);
```

Do a blocking operation, a system call (long IO for example) or a delay, etc.

```
MUTEX_LOCKID(MTX_THREADCONTROL);
MUTEX_LOCKID(MTX_NARTHREADS); NAR

NARthreads++;

MUTEX_UNLOCKID(MTX_NARTHREADS);
MUTEX UNLOCKID(MTX THREADCONTROL):
```

Increment the counter. Wait for GC to finish if there is one.

Figure 4. Tracking the number of running threads

is running by itself. Then the GC thread proceeds to perform the garbage collection. After finishing, the GC thread unlocks the garbage collection mutex, and does a broadcast to the gc condition variable. This wakes up all of the threads blocked on the gc variable, allowing them to resume their execution and return to where they called the thread_control function in the first place.

In garbage collection-intensive applications, it may happen that two (or more) threads trigger a garbage collection at the same moment. This situation can be handled in various ways. The simplest solution is to block all of the threads requesting garbage collection after the first one and let the first one proceed and finish as described above (block all of the other threads and then wake them up again). A better solution, the one adopted in this design, is to make the threads that are competing for garbage collection aware of each other. Instead of a sequence of "block all others" followed by a "wake up others" performed by each GC thread, a GC thread can hand the control over to the next GC thread in line, if there is one, and then go to sleep. Only the last GC thread in the line wakes up all of the blocked threads. This strategy eliminates intermediate block/wakeup operations, leaving only one "block others" operation by the first GC thread and one "wake up others" operation by the last GC thread. A counting semaphore (gc semaphore) controls the queuing and synchronizing of several threads to garbage collect. The first thread to request a garbage collection does not block on the semaphore, but subsequent requesters do. After finishing garbage collection, each thread signals the semaphore to wake up the next GC thread and goes to sleep. The last GC thread to wake up is responsible for waking up all of the threads after finishing.

An additional measure increases the effectiveness of the strategy just described: triggering a pre-emptive garbage collection if another thread has a garbage collection request pending. After a thread triggers a garbage collection and starts the protocol to suspend all other threads, each thread answers the call, but before going to sleep on the gc condition variable it checks if its heap is "nearly" full. The "nearly full" value depends on the heap size and on the nature of the application. Currently the implementation is set to force a thread to garbage collect if its heap is 92% full or more. The 92% figure is mainly an experimental value that was picked after conducting few tests. Thus if the thread's heap has only 8% of its total size free, the thread queues up to do garbage collection instead of going to sleep on the gc condition variable. This technique forces several garbage collections that might be separated by very short periods of time to cluster together, requiring only a *single* suspension for all of the threads to do the several garbage collections. Instead of relying on a hardwired number (92%), a more dynamic approach might take into account the heap size and the rate of allocation to decide whether to force a garbage collection. Figure 5 shows an abstract overview of garbage collection and concurrency control.

5.3 Input/Output Sub-system

The Unicon input output sub-system includes very high-level facilities for accessing files, networks and messaging, 2D and 3D graphics, databases, pipes and pseudo-ttys. The underlying C library functions implementing these capabilities are often thread-safe [22]. However, Unicon language-level IO operations usually involve several underlying library calls that must be atomic with respect to threads. Each IO handle is assigned a mutex when it is opened. Any IO operation that uses this handle locks the mutex to guarantee the atomicity of such operations.

5.4 C Library Thread Safety

In addition to IO, many C library functions called by the Unicon VM are documented in the POSIX standards as thread-unsafe. Most calls to thread-unsafe functions were replaced with the thread-safe alternatives available in modern C implementations. Other, infrequently called, thread-unsafe functions were protected by mutexes. A few of the thread-safe alternatives were not portable; new wrapper functions or implementations were developed in such cases.

6. DISCUSSION

Introducing concurrency to Unicon involved transforming parts of the virtual machine and runtime system to support the new feature. One of the biggest issues was to handle garbage collection correctly without imposing great overhead and allowing the threads to run at full speed most of the time (sec. 5.2).

Another issue was making the whole virtual machine and the runtime system thread-safe (sec. 4, 5). Many changes that had to be made were obvious and straightforward. Others needed more experimentation, debugging, and code analysis before catching possible race conditions and problem scenarios. Even after extensive analysis, code review, and bug fixes, tests revealed occasional race conditions, deadlocks, and crashes.

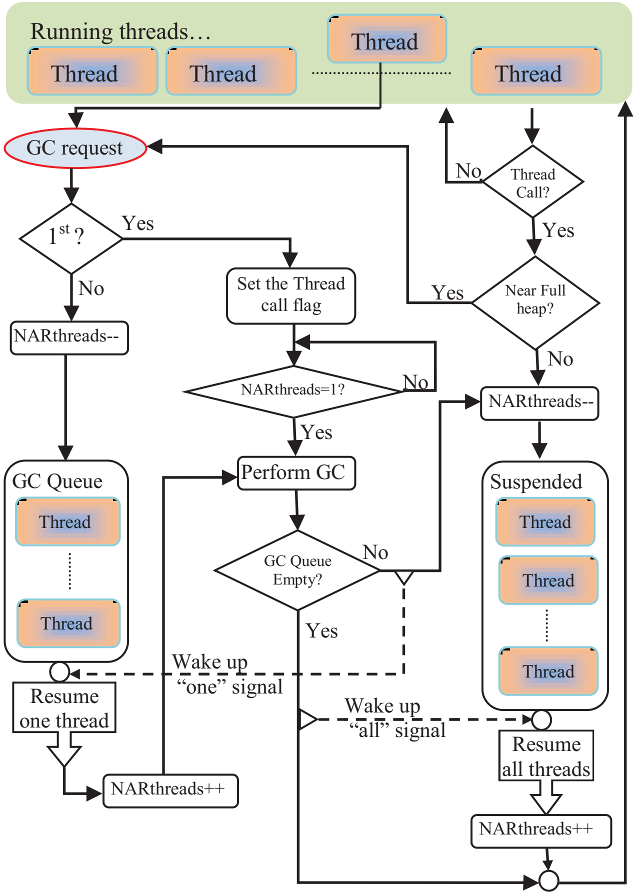


Figure 5. A high level view of the dynamics of suspending/resuming threads for Garbage Collection

Two methods proved to be very effective in fixing such problems. The first method was the use of wrapper functions and macros around the thread and synchronization functions, a quick and easy way to control their behavior and to add debugging print information. The second method was the use of the powerful tools for analyzing and debugging multi-threaded applications that are part of the Sun Studio suite, available on Solaris, and for many Linux distributions. Section 6.1 discusses the use of the above two techniques to debug Unicon's source code. Section 6.2 presents some preliminary evaluation of concurrency in Unicon.

6.1 Parallel Programming and Debugging Techniques

The first method included writing a macro that encapsulates calls to pthread library functions. These macros are exceptionally useful in catching deadlock problems. If a test program exhibits a deadlock behavior, the macros used to lock/unlock mutexes were modified to print information about the mutexes that they operate on. After getting to the deadlock state, the information printed by macros would reveal any mutex lock that was not paired with a matching unlock. In many situations, this output pointed directly at the problem or enabled a trace to its source. A fix was then applied and the test repeated until the deadlock problem was eliminated. Figure 6 shows an example macro applied to encapsulate the function pthread_mutex_lock().

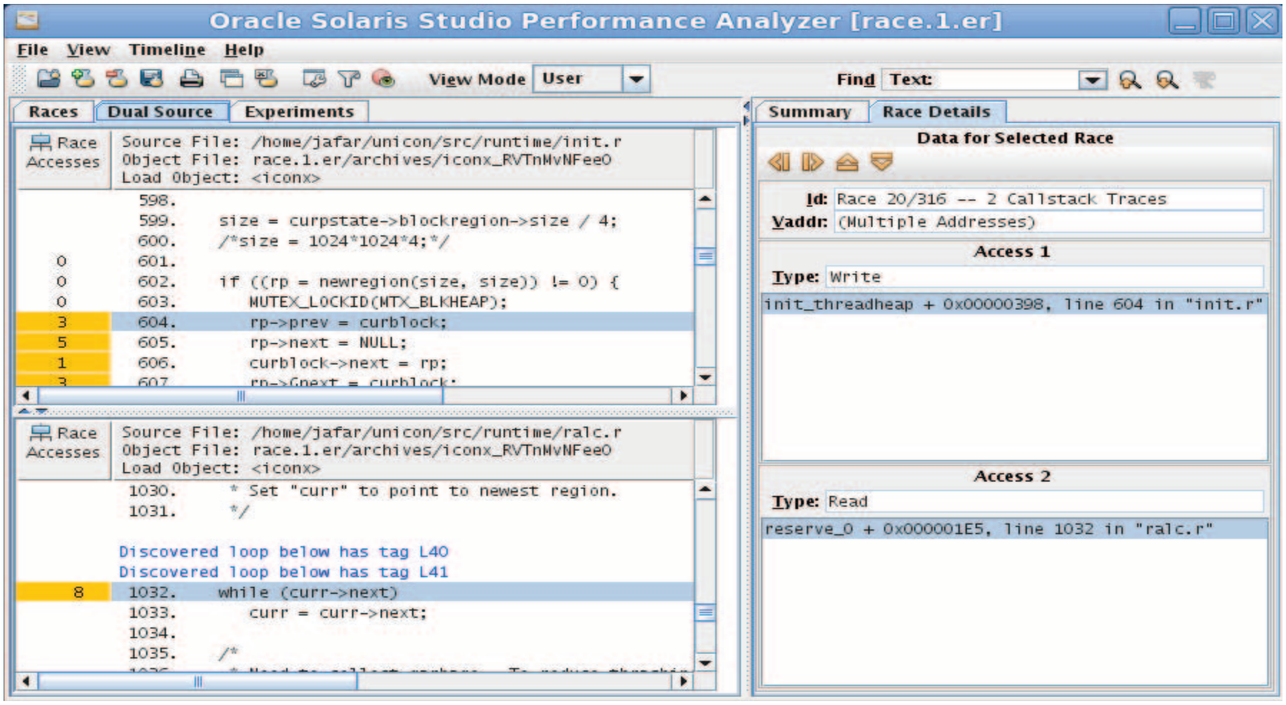
The macro/wrapper function method can only be used to solve a deadlock problem if one actually occurs under testing. It does not help reveal hidden but possible deadlock scenarios that are waiting for the right conditions to happen. It also has nothing to do with diagnosing or discovering race conditions. The Solaris *performance analyzer, collect* tool, and the *thread analyzer* were used [20] [21].

```
#define MUTEX_LOCK( mtx, msg) { int retval;\
printf(“locking mutex:%s\n”, msg);\
if (retval=pthread_mutex_lock(&(mtx))) != 0) \
        handle_thread_error(retval);}
```

Figure 6. A debugging printf in a mutex macro

One of the trickiest race conditions that *collect* and *thread analyzer* revealed were the self-modifying instructions discussed in section 4.2. These race conditions were not noticed during weeks of manual code analysis and debugging, but after a few test runs under *collect*, the *thread analyzer* caught them all.

The *thread analyzer* detects *potential,* rather than actual problems. Some race conditions that the tool may discover are “false positives” or “benign” data races. A false positive might happen when a memory address gets recycled, for example after a garbage collection, and then gets used by a different thread. It might happen also when the code is set up in a way where different accesses to a given address would never overlap. An example where this happens in Unicon's VM and the thread analyzer reports a race condition is in Figure 7. No thread can get to the case Op_Areal before one of the threads changes the offset to an absolute address in the case Op_real. The code in Op_real is protected by a mutex but the code in Op_Areal is not. Both cases work on the same memory address. After switching to the absolute address in the first case all of the subsequent accesses by the different threads will be diverted to the read-only access in the second case which is thread-safe. The *thread analyzer* cannot deduce this and reports a race condition.



```
Source File: /home/jafar/unicon/src/runtime/init.r
Object File: race.1.er/archives/iconx_RVTnMvNFeeO
Load Object: <iconx>
     598.
     599.     size = curpstate->blockregion->size / 4;
     600.     /*size = 1024*1024*4;*/
0    601.
0    602.     if ((rp = newregion(size, size)) != 0) {
0    603.        MUTEX_LOCKID(MTX_BLKHEAP);
3    604.        rp->prev = curblock;
5    605.        rp->next = NULL;
1    606.        curblock->next = rp;
3    607.        rp->Gnext = curblock;
```

```
Source File: /home/jafar/unicon/src/runtime/ralc.r
Object File: race.1.er/archives/iconx_RVTnMvNFeeO
Load Object: <iconx>
     1030.      * Set "curr" to point to newest region.
     1031.      */

     Discovered loop below has tag L40
     Discovered loop below has tag L41
8    1032.    while (curr->next)
     1033.       curr = curr->next;
     1034.
     1035.    /*
```

Data for Selected Race

Id: Race 20/316 -- 2 Callstack Traces
Vaddr: (Multiple Addresses)

Access 1

Type: Write
init_threadheap + 0x00000398, line 604 in "init.r"

Access 2

Type: Read
reserve_0 + 0x000001E5, line 1032 in "ralc.r"

Figure 7. The thread analyzer, showing a possible data race in Unicon's runtime system

A benign data race on the other hand is an intentional data race that enhances the performance of the program, while not affecting its correctness. For example, if a few threads call a pollevent() function every few hundred iterations, and they all share a global counter to keep track of how many iterations they have done already, but the exact number of iterations is not really important, it would be a waste of time to have them lock/unlock to increment the counter. Doing an extra 20 or 50 iterations might be acceptable by the program logic, so it is better to leave the loop running at full speed as opposed to locking and unlocking to protect the counter.

6.2 Thread Performance Evaluation

Even after a successful implementation of concurrency, improving application performance by using threads is not trivial. The performance gain depends on the application coding style (whether the application was a concurrent design starting from a clean slate, or a conversion of an originally sequential program) and the dependencies in the code and data. Synchronization poses a challenge in some applications. Synchronization and thread-local storage are also a source of slowdown for the language infrastructure. Garbage collection remains the biggest factor in preventing some applications from taking full advantage of multiple cores available in a given machine, especially for memory- intensive and long-running programs.

Several experiments were conducted to measure the performance of threads in Unicon under different conditions, including the number of threads, heap size, garbage collection frequency, and many other factors. Results were obtained on a Sun SPARC M9000 with 128 cores running Solaris 10 (64-bit). Experiments used three simple multi-threaded Unicon programs. The first, *sum*, includes a simple loop that counts to a fairly large integer. The program can split this job among several threads. The loop does not do any memory allocation and so is not affected by garbage collection at all. On the other hand, the program *heavy* is a garbage collection-intensive test that keeps allocating memory and trashing it, filling the heaps very quickly and forcing very frequent garbage collections. The last program, *sl*, finds the sum of a long list of integers. All times are in seconds and were measured using the shell command time(1), which means the time measured include the creation, setup and initialization of the program and the threads, and also the time to setup data in the case of *sl* test. Figure 8 shows the performance of these programs with an increasing number of threads. The three programs accomplish completely different tasks that require different times to finish. Plotting them on the same figure is not to compare the absolute numbers from one program against another, but rather to compare the behaviors of the programs under changing conditions, and show how each program is affected by changing some of the parameters of the experiment.

sum runs twice as fast when the number of threads is doubled, as expected, since the threads are independent and the program does not do garbage collection. *heavy* follows the same pattern with 2 and 4 threads. However, because it is memory-allocation demanding with a very high rate of garbage collection, adding more threads has a negative impact: more threads require more synchronization and more time to suspend and resume for each garbage collection. *sl* gets high speedup with more threads but a little less than *sum*. That is mainly because *sl* first initializes the big list of integers to be used, and this is done by the main thread before the other threads start. This initialization takes a constant time, independent of the number of threads.

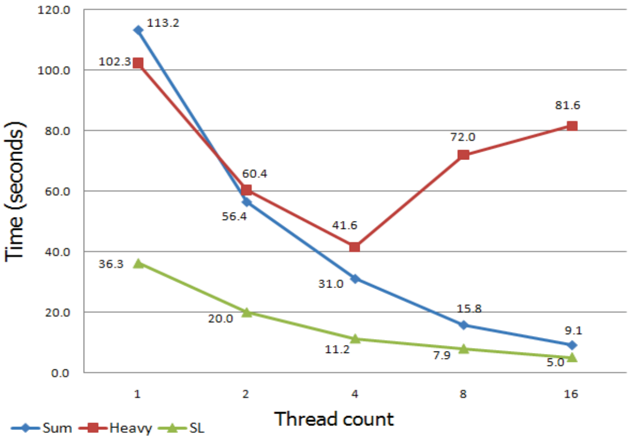


Figure 8. The effect of adding more threads in several programs. *heavy* is garbage collection very intensive

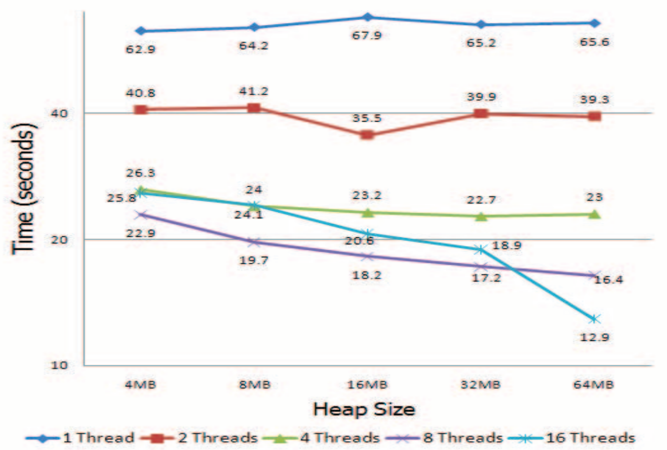


Figure 9. Heap size effect on the performance of garbage collection intensive program

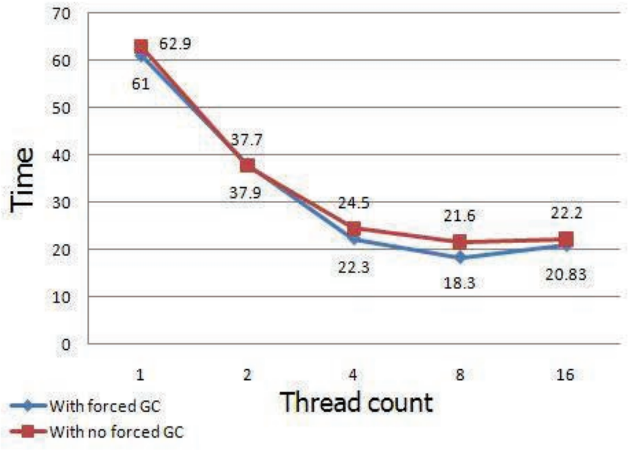


Figure 10. Forcing threads with semi-full heaps to GC

Figure 9 demonstrates the effect of the heap size allocated for each thread on the performance of a program. The heap size affects how frequent the garbage collection is. For these results we used the garbage collection-intensive *heavy* program. The figure shows that in an extreme case of garbage collection, a program does not benefit, or gets only a modest increase in speed from larger heaps with few threads running. The speedup increases with the number of threads if combined with an increase in heap size. Figure 10 illustrates the value of forcing some threads that have almost full heaps to garbage collect if another thread triggered a garbage collection.

6.3 Concurrency Extension Efforts in Similar Languages

Section 2 gave an overview of related work, but it is useful to compare the Unicon effort to similar projects in which concurrency is added to formerly sequential very high level languages. This excludes the many worthy languages designed specifically to support concurrency, but the focus helps identify languages that might benefit from mutual study and cross fertilization of ideas. An extended discussion of many languages' approaches to concurrency is available online [16].

It is difficult to identify peer languages, since there may be dozens or hundreds. Unicon is frequently characterized as being similar to Python or Ruby in language level; Object Rexx or JavaScript and its cousins might also be peers. Only those implementations supporting true concurrent execution on multiple processors/cores are comparable, and of those, the most comparable would be efforts to extend pseudo concurrency to true concurrency. For such comparable languages, it would be useful to compare the designs as well as the implementations. An exhaustive comparison of many languages' designs and implementations is beyond the scope of this paper. Space allows only for highlights to be presented.

In terms of language design, Karaorman and Bruno point out that concurrency (when not attained via invention of a new, intrinsically concurrent language) has often been introduced with no language extension at all, by adding a concurrency class library [11]. While this answers the question of what is the minimal language design possible, it does not provide insight on how to balance the needs of programmers for simplicity versus expressive power.

Python and Ruby had threads introduced to their design early on, but as mentioned in Section 2, a global interpreter lock prevents true execution in their primary implementations. Replacing a global interpreter lock with many dozens of mutexes (Unicon's runtime system uses 34, plus however many the programmer requests) does not guarantee higher parallel performance, it just allows for the probability of attaining it. Since alternative Python and Ruby implementations with true concurrency have not unseated their primary C implementations, concurrency integration in a language must be balanced against other practical aspects of the language implementation.

JavaScript has cooperative concurrency analogous to Unicon's co-expressions prior to the start of this project. Open Object Rexx has object-based concurrency with a complex feature set [2]. A study of the evolution of its concurrency support might require understanding of its relationship to its proprietary commercial predecessor.

The overhead cost of true concurrency is non-negligible, especially when it affects non-concurrent execution. When CPython was parallelized in 1999 by Greg Stein, removal of its global interpreter lock reduced sequential performance by 50%, and true concurrency in CPython has been prevented for over a decade due to this negative result [19].

For Unicon, similar initial negative performance was ameliorated by aggressively reducing the necessity for mutexes and redundant lookups in thread-local storage as described in Section 4.1. The overhead associated with concurrency in Unicon is presently around 12-27%, and should be reduced to less than 8% by passing a reference to each thread's local storage on the stack as a parameter to frequently executing functions, eliminating their need to call the pthreads API. For non-concurrent execution it should be further reduced, to less than 4%, by providing concurrent and non-concurrent versions of selected virtual machine functions, and switching over to concurrency-supporting versions of the functions only when threads are activated.

Compared with pure functional or logic programming languages, implicit parallelism is a challenge in a pragmatic language such as Unicon. Implicit parallelism is a major opportunity due to the very high level of the language. Both co-expressions and generators represent natural units for analysis; in some instances the code may be parallelized implicitly. Generators are especially exciting prospects for implicit parallelism, as they naturally describe a fine-grained, demand-driven parallel computation.

7. FUTURE WORK

While they have already proven useful in an important real-world application, the Unicon concurrency facilities will become more useful with refinement. Future work includes completion of the implementation, further performance tuning, and extension to a broader range of concurrency modes appropriate to the Unicon's domain.

Completion and maturing of the implementation includes a study of implicit locking for the language's built-in list, table, set, record, and object types. At present, explicit mutexes are available to protect heap-allocated variables, but a language at this semantic level would benefit from implicit (automatic) protection for shared structures.

Garbage collection is not yet aware of threads blocking on IO. There is already a technique used to handle blocking in mutexes and the `sleep()` function that can be generalized for this purpose.

Preliminary work has been done toward the implementation of SIMD data-parallel operators that will complement the more general concurrency of threads. Another major area for future work is improvement of the garbage collector. Analysis can determine that some threads may garbage collect without stopping all the other threads; for example, producers and consumers that have no shared memory references and communicate purely through message passing. When synchronous collection is required and threads do have to be stopped, the garbage collector uses a mark and sweep algorithm; the marking phase could be readily parallelized.

8. CONCLUSIONS

This paper describes the design and implementation of concurrency in Unicon, a very high level object-oriented, goal-directed language. Concurrency was added to the virtual machine and runtime system by extending an existing non-concurrent thread type. A primary design goal was to add explicit concurrency with a minimal impact on the syntax. This goal was achieved initially, but the goal has changed over time into a new goal to find an appropriate balance between simplicity and power for explicit concurrent programming.

The primary contribution of the work consists of the invention of thread-safety mechanisms that in many cases avoid the use of mutex-based synchronization. This effort was especially important in the memory allocator and garbage collector. For the majority of the runtime system, the thread-safety mechanism of choice was to migrate data into thread local storage and then focus on reducing the cost of accessing that storage.

Unicon's concurrency features have been deployed successfully in a large piece of research software running on the Solaris platform. The act of porting Unicon's concurrency to Linux, OS X, and Windows resulted in lessons learned and forced code improvements (such as avoiding __thread) that increased performance, benefitting all platforms.

9. ACKNOWLEDGEMENTS

This work was made possible by the generous support of AT&T Labs Research, as well as a contract with the Specialized Information Services division of the National Library of Medicine.

10. REFERENCES

[1] Andrews, G., and Olsson, R. 1993. *The SR Programming Language: Concurrency in Practice*.: Benjamin/Cummings.

[2] Ashley, D. W., et al. 2009 . *Open Object Rexx: Programming Guide*. http://www.oorexx.org/docs/rexxpg/book1.htm (accessed 01/30/2012)

[3] Bal, H., Steiner, J., and Tanenbaum, A. 1989. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*. 21, 3, 261-322.

[4] Beazley, D. 2009. Inside the Python GIL. Python Concurrency Workshop. http://www.dabeaz.com/python/GIL.pdf (accessed 01/30/2012)

[5] Capman, B., Jost, G., and van der Pas, R. 2007. *Using OpenMP*. MIT Press.

[6] Griswold, R., and Griswold, M. 1999. *The Icon Programming Language, 3rd ed*. Peer-to-Peer Communications., San Jose, CA.

[7] Gustafson, P. Sun Developer Network. http://developers.sun.com/solaris/articles/cpp_race.html (accessed 01/30/2012)

[8] Hasselbring, W. 1997. Approaches to High-Level Programming and Prototyping of Concurrent Applications. *Software-Technik Memo 91*. ftp://ls10-www.cs.uni-dortmund.de/pub/Technische-Berichte/Hasselbring_SWT-Memo-91.ps.gz (accessed 01/30/2012)

[9] Hudak, P. 1988. Exploring parafunctional programming: Separating the What from the How. *IEEE Software*, January, 1988, 54-61.

[10] Jeffery, C., Mohamed, S., Parlett, R., and Pereda, R. 2003. *Programming with Unicon*. http://unicon.org/book/ub.pdf (accessed 01/30/2012)

[11] Karaorman, M., and Bruno, J. 1993. Introducing concurrency to a sequential language. *Communications of the ACM*, 36, 9, 103-115.

[12] Kolosov, A., Ryzhkov, E., and Karpov, A. 2008. Intel Software Network. http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers (accessed 01/30/2012)

[13] Mitsolides, T. 1992. The Design and Implementation of ALLOY, a Higher Level Parallel Pro-gramming Language.

[14] Mitsolides, T., and Harrison, M. 1990. Generators and the replicator control structure in the parallel environment of alloy. In *Proceedings of PLDI '90 - ACM SIGPLAN conference on Programming language design and implementation*, 189-196.

[15] Oikonomou, K. N. 2010. Network Performability Evaluation. *Guide to Reliable Internet Services and Applications*, C. R. Kalmanek, S. Misra, and C. R. Yang, Eds. Springer, 113-135.

[16] Oshineye, A. Discussion of "The Free Lunch is Over: A Fundamental Turn Towards Concurrency in Software". http://lambda-the-ultimate.org/node/458 (accessed 01/30/2012)

[17] Paalvast, E. M., Sips, H. J., and Breebaart, L. C. 1991. "Booster: a High-Level Language for Portable Parallel Algorithms," *Applied Numerical Mathematics*. 8, 2, 177-192.

[18] Peyton-Jones, S. L. 1987. *The Implementation of Functional Programming Languages*. Prentice Hall International.

[19] Python FAQ. http://docs.python.org/faq/library#can-t-we-get-rid-of-the-global-interpreter-lock (accessed 01/30/2012)

[20] Sun Studio Express - Using The Thread Analyzer - Tutorial. http://www.oracle.com/technetwork/testcontent/tha-using-141353.html (accessed 01/30/2012)

[21] Sun Studio Performance Analyzer Quick Start Guide. http://www.oracle.com/technetwork/server-storage/solaris/analyzer-qs-136436.html (accessed 01/30/2012)

[22] Thread-safety and POSIX.1. http://www.unix.org/whitepapers/reentrant.html (accessed 01/30/2012)