ESPRIT PROJECT 6062

Requirements Specification for Parallel Extensions to TDF to support Dylan - including Run-time Support

Written by: Tony Mann, Eliot Miranda
Harlequin Ltd.

Issued by: Peter Edwards
DRA

Delivered by: Gianluigi Castelli
Project Director

The status of this document is "WORKING" if signed only by its author(s); it is "ISSUED" if signed by the Workpackage Manager; it is "DELIVERED" if signed by the Project Director.

Project deliverable id: TR5.7.1-01

Document code: Harlequin GLUE5.7.1 - 01

Date of first issue: 1994-03-16
Date of last issue: 1994-03-16

Availability: Confidential

omi/glue

CHANGE HISTORY

This is the first version.

1. Purpose

The purpose of this document is to identify the features missing from TDF [TDF92] which are required to support a multi-threaded implementation of Dylan with garbage collection. In order to do this, a study is made of some possible implementation techniques, and an analysis is made of how much support TDF already provides for these techniques.

This work was sponsored by the Commission of the European Communities.

2. Executive Summary

Dylan is an advanced, modern, object oriented programming language [Apple92]. It contains features not found in most other languages — such as automatic memory management, and an exceptions mechanism. The language is still under development, and proposals are starting to appear for standard libraries for the language.

As part of their work with OMI/GLUE, Harlequin have undertaken to study the requirements of a Dylan producer with support for a threads library. This will establish the adequacy of the parallel TDF extensions developed by DRA in task 5.1, and point out any areas where TDF might be lacking.

Although a standard threads library for Dylan does not yet exist, Harlequin intend to develop their own library, which may later be proposed as a standard. It is not expected that the support required from TDF will depend on the details of the final threads standard chosen for Dylan.

This document looks at the features any Dylan threads library must provide, and attempts to map them onto the experimental basic thread model described in [Edwards93]. This basic thread model offers suitable support for synchronization, thread creation and thread deletion. The requirements of Dylan for garbage collection and other advanced features have been discussed for a serial computation model in [Mann93][Mann93a][Currie93]. These areas require extension for parallel computation models, and are not covered by the basic threads model.

Implementation techniques for Dylan's advanced features on some important types of parallel architectures are discussed. As explained in [Edwards93], there is no common model of parallel computation, and TDF does not attempt to provide an architecture neutral solution. This document shows, in principle, how architecture neutrality may be provided for these advanced features, just as for basic threads, using a combination of TDF token definitions and platform specific run-time systems. A variety of very different garbage collector designs may be used with this level of abstraction.

By leaving the details of the support for threads to token definitions and a suitable run-time system, the introduction of threads to the Dylan producer is not expected to introduce any further loss of efficiency, compared to an architecture specific implementation.

3. Threads

A *thread*, or lightweight process, is an independent flow of control in a single address space that may share that address space with other threads. Three basic kinds of thread scheduling can be distinguished. Threads may run truly concurrently on a shared memory multiprocessor. Threads

may be pseudo-parallel if scheduled by the host operating system, as is the case with a native thread implementation on a uni-processor. Finally, threads may be run pseudo-parallel as coroutines, as is typically the case with a thread library. Truly concurrent implementations may provide pseudo-parallel thread scheduling when there are more threads than processors.

Scheduling models may be pre-emptive or cooperative. A pre-emptive model allows a thread to be suspended at an arbitrary point, whereas a cooperative model only deschedules a thread when the thread performs some action, such as calling a routine that will yield to other runnable threads. Scheduling models may provide thread priorities; for example, a model may provide the invariant that no lower priority thread runs while a runnable higher priority thread is waiting. Such a model is partially pre-emptive because higher priority processes can pre-empt those of a lower priority, but processes at the same priority are scheduled cooperatively.[1]

Since threads share a common address space they can communicate via shared variables. Some means of synchronization must be provided to coordinate the use of such (see the following section). Asynchronous communication between threads need not be provided primitively, since it can be synthesised from other thread primitives.[2]

3.1 Synchronization

Parallel programs depend on synchronization mechanisms to ensure their harmonious and correct interaction. Synchronization provides ways for compound operations to be performed indivisibly, by serialising concurrent executions of such critical code sequences. Two basic synchronization mechanisms exist: busy waiting mechanisms using test-and-set or load-store interlocked instructions [Edwards93], which may be very quick but may cause excessive locking of the bus, and scheduling mechanisms which are slow to execute because they may entail suspending and resuming a thread, but do not tie up valuable resources while contending for a lock. Such facilities can be built from spin-locks and thread scheduling primitives, but on a uniprocessor one may simply rely on cooperative scheduling. The choice of mechanism will depend on the grain of the activity being serialized. It is only practical to use busy waiting for short activities that are unlikely to provoke contention.

Parallel systems using a garbage collected shared heap have two main requirements for thread synchronization. First, certain operations on the heap need to be *serialized*, so that compound operations are always executed atomically. For example, in many systems allocating an object via incrementing an allocation pointer, this involves reading the value of the allocation pointer to derive a new object's address, and then incrementing the allocation pointer by the object's size. If two threads attempt to allocate an object at the same time one possible interleaving is for both threads to read the same value of the allocation pointer and subsequently increment it, which results in a doubly allocated object, and a mispositioned allocation pointer. It must therefore be

1. One issue is how a pre-empted process is rescheduled. For example, the Smalltalk scheduler as specified in [GR83] will place a pre-empted process at the back of the queue of runnable processes for its priority. Subsequent Smalltalk implementations [ParcPlace] place a pre-empted process at the head of the queue of runnable processes for its priority, maintaining a cooperative scheduling model within each priority.

2. For example, asynchronous i/o notification can be implemented by one thread forking another thread to perform blocking i/o which then sets a shared variable to signal the possibility of i/o. Truly asynchronous notification also typically requires the use of shared variables, because the only communication a signal handler has with its thread is via change of state, for example by setting flag variables.

possible to ensure that allocations occur serially without interleaving. On systems with a cooperative scheduler on a uniprocessor this situation may never occur, and hence no special machinery is required. On pre-emptive schedulers, or true multiprocessors spin-locking is more appropriate than a scheduling mechanism since allocation is a relatively high-frequency operation.

Secondly, course-grain atomic operations need to be serialized using a scheduling mechanism. One such activity peculiar to many practical concurrent garbage collectors is the need to update references from thread-local registers and stacks to moved objects. This depends on the ability to stop and subsequently restart all non-garbage-collector threads.

3.2 A Dylan Thread Model

A high-level object-oriented language such as Smalltalk, Common Lisp, or Dylan provides an excellent basis for the provision of elegant parallel programming constructs. Since the object models in these systems provide information hiding and encapsulation it is natural to hide synchronization behind the languages' own implementations of these features. High-level services such as Smalltalk's SharedQueue (through which producers can safely and asynchronously pass values to consumers) are constructed from lower level facilities such as semaphores and mutexes. These services are used via message passing or generic function invocation, hiding the synchronization and its implementation, and hence require no explicit synchronization code in clients. Anonymous functions (lambdas) provide a convenient basis for thread abstractions; in Smalltalk and many Common Lisp implementations (for example: *LispWorks*, Harlequin's Common Lisp development environment) one can create a thread by invoking a *fork* operation upon an anonymous function. A thread is typically represented in the system as an object which implements methods for suspension, resumption, and termination, and has state defining its priority and execution context. The thread model can be enriched, for example through subclassing, adding slots to hold additional information such as thread parent-child links.

We propose that a Dylan thread model provide at least:

- a thread class, instances of which represent underlying system threads. It is possible to suspend, resume and terminate a thread, and possible to assign the processing priority of a thread while it is suspended.
- a semaphore class, instances of which represent counting semaphores, and support the operations signal and wait. A thread executing a wait on a semaphore with no outstanding signals is suspended and added to the end of a queue of threads waiting on that semaphore. A signal operation either removes the first suspended thread waiting on the semaphore from that semaphore's queue and resumes the thread, or increments the semaphore's count if no threads are queued.
- a scheduler class, instances of which represent the pool of runnable threads, with operations that give access to the current thread, all threads at a particular priority level etc.

3.3 Implementation of the Dylan Thread Model in TDF

The proposed Dylan thread library provides a simple, portable, object oriented interface to thread based computation. While there is an attempt to make the library integrate well with the features of the Dylan language, there is no attempt to make the library make use of any unusual properties

of threads. This means that any threads mechanism which has been designed to support more traditional languages can also be used to support Dylan threads. In particular, basic thread operations such as creation, deletion, synchronization and scheduling can all be implemented in terms of either the low level TDF token interface called the “Experimental Basic Thread Model” [Edwards93] or any standard threads API.

It is anticipated that there will be a separate implementation of the Dylan threads library for each standard threads API, and for the low level token interface. There is a hope, however, that it may be possible to have a single, architecture neutral implementation of the threads library which uses an interface of Dylan-specific TDF tokens. Token definitions for this library would expand into either API calls or the low level tokens, as appropriate. This interface has not yet been designed, and it is not essential to have this architecture neutral threads library for Dylan.

The details of the Experimental Basic Threads Model are not discussed here — since it is expected they will work as well for Dylan as for any other language. However, there are some features of Dylan which require special implementation in TDF for serial computation, and some of these require additional support for parallel computation; these features are: garbage collection, the dynamic environment and exceptions. Garbage collection is a complex issue, and is dealt with in sections 5 and 6. The other requirements are considered in this section.

3.3.1 Implementation of the dynamic environment with multiple threads

Dylan provides language constructs which have *dynamic scope*. Each such construct establishes an entry in the *dynamic environment* (potentially visible from anywhere in the program), and explicitly disestablishes it on exit. An exit may occur either through normal completion of the construct, or by any transfer of control out of the construct. These features are defined in terms of a stack based model of computation, and must be extended to support multiple dynamic environments on separate stacks when threads are introduced. Semantically, this means the constructs only effect the dynamic environment of the thread in which they are executed. Dylan has a subset of the dynamically scoped features of Common Lisp which are described in [Mann93]. The features it does provide are unwind-protect and handler-bind.

The purpose of unwind-protect is to guarantee that a section of *clean-up* code will always be executed after a section of *protected* code — even if a transfer of control should occur during the execution of the protected code (for example, because an exception handler causes a premature exit from the protected code). Implementation techniques for unwind-protect are described in [Mann93]. Basically, the techniques involve building an *unwind protect frame* on the stack whenever the construct is encountered. The normal execution stack is therefore used to maintain the stack of the currently active clean-up code references. Whenever a transfer of control occurs, all intervening unwind protect frames must be located, and the clean-up code executed. The frames can be found by chaining them together on the stack, so each frame points to the next oldest frame. The most recent frame is found by looking in a global variable (at a defined location or in a register).

handler-bind is a language construct which allows handlers to be initialized for *conditions*, which are objects which represent exceptions. Dylan defines a programmer-extensible hierarchy of condition classes, which represent the different types of exceptions which might occur — such as warnings or errors. If an exception should occur within the dynamic scope of the handler-bind body, then the condition signalling mechanism will call the handler, provided the class of the con-

dition matches the class of the handler, and there was no more recent handler to handle the exception. The handler is called by the exception signaller with a normal function call, and it is executed above the stack frame of the function which raised the exception. The handler may choose what action to take, (including declining to handle the exception, by calling the next most recent handler). Typically, the handler will transfer control to the execution context in which it was defined.

For Common Lisp, handler-bind is implemented in terms of lower level primitives, such as catch and *special binding* [Mann93]. In Dylan, these lower level primitives are not available — but the implementation techniques will be similar. Conceptually, a *handler frame* will be created on the stack, in the same way as for an unwind protect frame. Frames for both constructs will be linked in the same chain of dynamic frames (which represents the dynamic environment), and the same location is used to point to the most recent frame (of either type). Whenever an exception occurs, the signaller must test each handler frame for appropriate handlers to determine the ordered list of applicable handlers.

Since handler-bind and unwind-protect are implemented in terms of the normal execution stack, the mechanism scales for multiple threads (and hence multiple execution stacks) in a simple manner. However, it is a requirement for each thread to maintain an independent location for the pointer to the most recent frame. For pseudo-parallel threads, it is possible to achieve this without any TDF support by arranging for a run-time system to update a global variable with the thread local value of this frame pointer on each context switch. However, for concurrent threads it is necessary to have a truly thread local variable to implement this.

A native code compiler might implement the thread local variable for the most recent frame by reserving a register for this purpose (since registers are necessarily thread local). However, there is no way to specify this in TDF. The only portable mechanism is to pass the pointer to the most recent frame as a parameter on every language function call. Of course, this would have a serious impact on performance if the installer is unable to optimize this parameter in a register.

For concurrent threads, it is therefore a requirement for Dylan that there is some means of describing a thread local variable in TDF. As with other features related to parallelism, there is no direct TDF support for this. Instead the feature will be implemented in terms of token definitions and possible support from the run-time system — as with the basic threads library itself. By integrating these token definitions with the Dylan token library definitions for function calling it should be possible to implement thread local variables by parameter passing as a last resort mechanism.

3.3.2 Implementation of exceptions with multiple threads

The dynamic mechanisms for handling exceptions have been described above. But the mechanisms for generating exceptions must also work with threads. These mechanisms are described in [Currie93]. In particular, the following TDF constructs must have an effect which is local to the thread in which they are used:

set_stack_limit give_stack_limit

Similarly, the following constructs which describe exception detection must work for all threads:

~set_signal_handler

It is assumed it will not be difficult for these constructs to support threads — but it is noted here that this is a requirement for Dylan.

4. Garbage Collection Algorithms for Threads

4.1 Heap-Based Automatic Storage Management

In systems that provide heap-based automatic storage management, the heap may be viewed as a directed graph whose nodes are *objects*, variable length regions of the heap, and whose edges are *pointers*, or *references* from objects (*referees*) to other objects (*referents*). Depending on the system an object may contain only pointers, only non-pointer data or a mixture of both. The program contains references to the graph, which are termed the *roots*. The program may alter the graph by allocating new objects and by changing the edges to refer to other objects by assigning to pointer fields in objects, and hence is termed the *mutator*. In the context of concurrent systems there may be many threads, more than one of which may be a mutator.

As a result of mutator activity parts of the graph may become detached from the transitive closure of references (formed by transitively following edges from the roots until all reachable objects have been enumerated). This unused space is called *garbage*; the job of the *garbage collector* is to identify and reclaim it. Garbage collection (GC) can therefore be viewed as a per-object proof of reachability from the roots. Algorithms which perform this proof by traversing the graph are *tracing* collectors, whereas algorithms which perform this proof by counting the number of edges into nodes are *reference counting* collectors.

Incremental collectors attempt to collect objects interleaved with mutator activity, and concurrent systems allow more than one mutator to share the heap in parallel. In both cases heap manipulation must be coordinated. Abstractly, coordination is realised by the mutator(s) having to cross barriers before performing certain heap operations. A mutator may have to use a *read-barrier* to indirect through a reference, a *write-barrier* to store a reference, or an *allocation-barrier* before allocating an object. The barriers can serve to notify the collector of changes to the graph, and to *serialize* accesses to the graph so that all possible interleavings of (possibly multiple) mutator and collector heap accesses leave the heap in a consistent state. For example, if objects are allocated from a free list, an object must be removed from the head of the list, and the head of the list updated atomically if the object is not to be inadvertently allocated twice.

4.2 Basic Algorithms

There are three basic GC algorithms; *reference counting*, *mark-sweep*, and *copying* garbage collection, all of which have been used as the starting point for implementing concurrent garbage collectors. Important variants of the latter two algorithms are *mark-compact*, *generational* and *conservative* collectors. We will first give a brief overview of these algorithms before discussing their parallel variants. A fuller treatment of current GC technology was given in [Mann93a], and thorough reviews can be found in [AMR92][Wilson92]. Reference counting was not discussed in [Mann93a] because the technique has significantly inferior performance to tracing collectors on uni-processors, but it *does* have its uses in a concurrent context[1], and so will be discussed here.

In reference counting garbage collectors each object has an associated reference count which records the number of references to that object. Counts are maintained using a write barrier; when

1. This is widely used in distributed garbage collection algorithms [AMR92].

an assignment to a slot occurs the previous reference in the slot is overwritten, so the previous referent has its count decremented, whilst the new referent acquires a reference, and its count is incremented. If an object's reference count falls to zero (due to a reference count decrement) then the garbage collection system can infer that no references to the object exist and can collect the object, returning it to the free space pool. At this point references from the collected object to other objects can also be destroyed[1], possibly resulting in the reclamation of further objects, hence the term *recursive freeing*.

Reference counting is incremental, because garbage collection activity is interleaved with mutator activity. However, reference counting is inefficient, since it performs work proportional to the number of objects collected, and requires the management of free storage with free lists, which complicates allocation. *Deferred* reference counting [DB76] improves performance by avoiding counting references from the stack(s) by deferring reclamation. The collector batches together collections by queuing objects whose count goes to zero in the *zero count table*. When necessary, the mutator(s) are stopped, objects accessible from the stack are marked and the queue is processed. Any objects in the queue which still have a zero count (or whose count falls to zero during recursive freeing) that are not marked can then be collected. But the major problem for reference counting is its inability, in general, to reclaim circular structures because isolated cycles of objects have non-zero reference counts from their mutual references. For these reasons reference counting has fallen from grace and is no longer considered a state-of-the-art technique for serial garbage collection systems.

Mark-sweep garbage collectors defer garbage detection to a point at which free space is exhausted. At this point all live objects are identified, by transitively tracing object references from the system's roots (any global variables and the current computation's references, for example registers and/or stack). Once this marking phase has completed, memory is swept, and unmarked objects returned to the free space pool. The sweeping phase can be performed by compacting the retained objects into a contiguous region thereby eliminating fragmentation of free space. This is the *mark-compact* variation of mark-sweep. Mark-sweep also copes with cyclic garbage since unreachable cycles are not marked, and hence reclaimed[2].

Copying garbage collectors combine the mark phase with the compaction phase. The heap is divided into two *semispaces*, *from-space* and *to-space*. Only from-space is used to hold objects, new objects being allocated contiguously at the end of from-space. The garbage collector traces objects in from-space, starting from the system's roots, and copies each object into to-space, leaving behind a *forwarding pointer* so subsequent references to the copied object can be relocated to the to-space copy. After copying, all live data are in to-space and free-space is empty. The two spaces are then *flipped*, from-space becoming the new to-space, and vice versa. Copying collectors allow object allocation to be done simply by incrementing an allocation pointer.

One very important variation of the copying algorithm is Baker's incremental copying collector [Baker78], an incremental, real-time algorithm. The algorithm incrementally copies objects from from-space to to-space, leaving forwarding pointers behind. This algorithm maintains the invari-

1. the destruction of references in a collected object can be deferred to the point at which the object is reallocated. This technique is used in some real-time reference counting systems to reduce the pause times caused by recursive freeing.

2. For these reasons most practical reference counting garbage collectors rely on a mark-sweep garbage collector to reclaim cycles and compact free space [GR83][Rovner85].

ant that the mutator sees only to-space pointers. The mutator uses a read barrier; each read is checked, and if made to a forwarded object the reference is updated and the read is redone on the to-space copy. This requires hardware support to be efficient and so was not widely used until the techniques outlined in section [4.3.2] were developed. The collector does a small amount of tracing on each allocation and hence provides real-time performance.

Real-time GC algorithms typically provide two guarantees: first, no invocation of the garbage collector will last longer than a given time, and second, in any chosen period the garbage collector will not run for more than a given percentage of time. The first guarantee allows the mutator to respond to events within some fixed bound, because it is not blocked awaiting termination of GC for longer than the constraints allow. The second guarantee allows the mutator to progress, because it has available to it at least some percentage of available processing power. One convenient way of implementing such real-time GC algorithms is to structure the garbage collector as a cooperating process, the real-time garbage collector's execution being interleaved with the mutator. Note that Baker's algorithm is effectively structured as two cooperating processes, and contains considerable pseudo-parallelism, the collectors' tracing and copying/forwarding activities being parallel to the mutators' allocation and pointer forwarding activities. Consequently, many algorithms for real-time GC are concurrent, and vice versa, but one should not in general equate real-time and concurrent GC algorithms.

4.3 Concurrent variants

Computation in a shared heap will require synchronization to implement certain algorithms correctly. It is not the job of a concurrent garbage collector (CGC) to provide such facilities, but it must at least maintain the integrity of the heap (create no dangling pointers), and provide temporal coherence (preserve the order of stores into memory). In addition, CGCs have two main goals, first to allow concurrent execution of threads that can share the same garbage collected heap, and second to collect garbage concurrently with the execution of threads.

4.3.1 Concurrent Reference Counting Collectors

Both goals are easily met in a reference counting collector. Multiple threads can share a reference counted heap if reference increment, decrement and allocation from free lists are serialized. Recursive freeing involves further reference counting operations, and adding of objects to the free list, so collection can easily be parallelised if the reference counting and free list update therein is similarly serialized [BM93][1].

The Cedar [Rovner85] and Topaz [DeTreville90] collectors are deferred reference counting system with a separate thread to do collection. Mutators use a lock to serialize reference counting operations. Another lock is used to control insertions into the zero count table. A final lock is used to serialize allocations. The collector thread processes the queue asynchronously, performing recursive freeing collection.

4.3.2 Effective Barrier Methods

The Appel, Ellis & Li collector [AEL88], a version of Baker's algorithm, was the first collector to

1. However, the scheme in [BM93] uses per-object mutexes, which appears hard to implement efficiently in TDF, since no assumptions can be made about the space requirements of a mutex.

use virtual memory (VM) facilities [AL91] to implement efficient read and write barriers. The Appel, Ellis & Li scheme sets the VM protection of unscanned pages in from-space to be “no access”. Mutator reads of unscanned objects therefore provoke a VM trap. The collector fields the trap, scans all objects on the page, copies them to to-space, and forwards pointers as necessary. The mutator is then restarted at the faulting instruction. The pointer fetched after the restart will now refer to to-space, and execution continues. Since reads are much more frequent than writes, attention has now focused on algorithms that depend only on a write-barrier, and the efficient implementation of write barriers. The paper by Moss et al. [HMS92] gives a performance evaluation of write barriers implemented using software (such as the sequential store buffer described in below) or virtual memory protection.

4.3.3 Concurrent Tracing Collectors

Tracing collectors include mark-sweep and copying collectors since both strategies involve tracing the graph of live objects starting from the roots to determine the set of live objects. If the second goal of parallelising the garbage collector is to be met using a write-barrier then the garbage collector must cope with mutator modifications to the graph whilst tracing is in progress, for example when newly allocated objects are assigned to parts of the graph that may already have been scanned. This is sufficiently problematical that if pause times for copying are low enough, as they are in generational schemes, it may be expedient to simply stop all threads, perform scavenging, and then resume [PU88].

The abstraction of *Tri-colour marking* [DLMSS78] is useful in explaining the workings of incremental write-barrier schemes in both mark-sweep and copying algorithms. At the start of each trace all objects are untraced, and to indicate this are coloured white. As tracing proceeds white objects are encountered that must be traced. Such objects are coloured grey. Once an object has been traced it is coloured black[1]. The collector *must* trace all of the used portions of the graph, so the write barrier must check for the storing of white objects into black objects.[2] There are two basic approaches: *Snapshot-at-beginning*, and *Incremental-update* [Wilson92].

Snapshot-at-beginning algorithms maintain the invariant that all objects active at the beginning of tracing will be retained. This is done with a write-barrier; the mutator places overwritten pointers and newly allocated objects in a queue that is examined by the tracer once it has traversed the heap [Yuaza90]. This scheme has the disadvantage of being “conservative”[3] because objects that become unreachable are not reclaimed until the after next trace.

Incremental-update algorithms attempt to be “less conservative” by gathering information about updates as tracing proceeds. They effectively maintain the minimum invariant, required of all such garbage collectors, that objects live at the end of the tracing phase are retained. They solve the problem of storing references to white (untraced) objects into black (already traced) objects by using a write-barrier. When such a write occurs the collector can either turn the white object black, or turn the black object grey and rescan it later. At the end of the initial tracing phase trac-

1. This can be seen as an extension of traditional marking where the only colours are white (untraced) and black (marked).

2. The problem of assigning a white object into a blackening object (one in the process of being scanned) can either be handled by scanning an object atomically, or by turning it black before scanning, and rescanning if an assignment occurs.

3. A confusing term! Here conservative refers to the fact that free storage is not reclaimed immediately, but definitely will be by the next cycle, whereas in *conservative* collectors it refers to the fact that mistaken pointer identifications may cause storage retention for an arbitrary length of time, or preclude its collection altogether.

ing must be completed; any remaining grey objects must be scanned. For mark-sweep collectors, only once this final tracing is complete may memory be swept to reclaim storage. In copying algorithms a flip must be performed. In Boehm's mostly parallel collector [BDS91], a mark-sweep algorithm, the tracing of remaining grey objects, and the sweeping of memory is done by the GC thread after it stops all mutator threads. Measurements show that the additional tracing takes little extra time, and that sweeping takes even less, so pause times introduced by this phase are acceptable.

VM write barriers are not always appropriate because trap handling can be expensive, some platforms provide no such facilities, and because page-size granularity may cause too much work to be done. O'Toole and Nettle's concurrent replicating realtime collector [ON93] uses a simple *sequential store buffer* or *storelist* where writes are recorded. On each write the mutator must place a reference to the written object into the storelist[1]. The collector incrementally builds a copy of from-space in to-space using a conventional copying algorithm. The mutators use only from-space. The collector has completed a collection once the storelist is empty, all objects accessible from the mutators' roots have been replicated in to-space, and all the mutators' roots have been updated to refer to the to-space replicas.

Replication proceeds asynchronously; mutators must serialize access to the storelist and the allocation pointer. The collector can process entries in the storelist asynchronously and update the to-space replicas of the from-space objects in the storelist. Once the storelist is empty it stops all mutators, updates their roots, and flips. An important space optimization is to store forwarding pointers in the headers of replicated from-space objects. This requires that mutators use a read barrier to access header words, which is an infrequent activity. No synchronization is required for this read barrier since the collector writes the header word into the to-space replica before writing the forwarding pointer into the from-space object's header; all that is required is that the memory system perform word writes atomically, and that a sequence of writes issued by a processor is performed in order.

Common to all these systems is the need to serialize allocation. Contention for the allocation lock can be radically reduced with copying algorithms by arranging that the allocator in each thread grabs large chunks from the global allocation pointer which it uses to satisfy thread-local allocation requests by using a thread-local allocation pointer [FMY91], but this can pose problems if there are a large number of threads [AEL88].

5. Implementing Garbage Collection Algorithms in TDF

As has been discussed above, and in [Mann93a], garbage collection for Dylan in TDF will be implemented using a combination of a run-time system and a cooperating compiled program. The TDF installer is responsible for the final phase of creating the compiled program, and hence it must be able to generate any support required by the run-time system.

1. A software write barrier can be efficient compared with VM schemes if sufficient care is taken. The main advantage of the sequential store buffer is that on uni-processors it can be implemented as a short in-lined sequence of jump free instructions, giving good performance, with an acceptably small increase in code size. If the pointer to the store buffer is kept in a register then it may be implemented in as little as two instructions: increment the pointer and indirect through it. However, in a concurrent system, explicit synchronization may be required, for example in a system with a shared sequential store buffer. In cases such as these, a software write barrier requires jump instructions, and a VM write barrier using page-based scanning may be more attractive.

The amount and the details of the cooperation between the program and the run-time system will depend on the choice of the garbage collection algorithm. It is not practical to choose a single algorithm for garbage collection on all platforms as algorithms have very different characteristics, and their relative efficiency may be very dependent on the nature of the program or the platform.

Whatever the chosen algorithm, it is known that cooperation is only required for certain distinct operations, each of which may require some form of synchronization. These are:

- Allocation
- Reading from objects
- Writing to objects
- Location of roots

Although the details of the required cooperation for each of these operations is algorithm dependent, portability may still be achieved by describing the cooperation in terms of an abstract interface using TDF tokens. A particular algorithm will be supported by a combination of a run-time system which implements the garbage collector, and suitable token definitions which implement the cooperative support from the program. This principle is also used for implementing garbage collection for serial programs [Mann93a] and for implementing parallelism itself [Edwards93].

The token library which will describe this cooperation has not yet been fully designed — but the sections which follow show it is possible to express the cooperation in terms of tokens.

5.1 Synchronization for Garbage Collection in TDF

It was mentioned in section 3.1 that garbage collectors may require synchronization between threads in order to serialize allocation and tracing operations. The requirement for allocation is simply lightweight mutual exclusion, for which a TDF mechanism is described in [Edwards93]. In fact, this mutual exclusion will occur inside the run-time system, and will probably not be implemented in terms of TDF.

A possible requirement for synchronization during garbage collector tracing is that all mutator threads are stopped while the garbage collector updates references from thread-local variables to relocated objects. This is a coarse-grain form of synchronization which is not a part of the Preliminary Parallel Extensions for TDF. The need for this exclusive right to computation resources only occurs inside the garbage collector, which is a part of the run-time system.

Depending on the host system, it may be possible to implement the suspension of mutator threads without requiring any support from TDF. However, if the host system does not provide a mechanism for stopping and restarting other threads, and accessing their state, then the mutator threads themselves must arrange to cooperate with the garbage collector. It is possible to achieve this by arranging for each mutator thread to poll the garbage collector as part of the read and/or write barrier code. If the mutator thread determines that it is about to attempt a dangerous operation during the critical stage of the garbage collector, then it must arrange to suspend itself and restart itself appropriately. This may significantly add to the cost of the barrier — although it might be possible to do this with little or no extra cost as the part of the normal synchronization in the barrier. In any case, this cost is not related to the use of TDF.

Some simple coding tricks can reduce the cost of barriers. Many synchronization primitives depend on a read-modify-write cycle which requires locking the bus, and is potentially both slow and prevents other bus accesses. A naive spin-lock might be written to attempt to get a lock using a read-modify-write cycle in a tight loop. One common technique is to first use a conventional read cycle to check that the lock is free, and only then attempt the read-modify-write cycle.

Naive

```
get_lock:
loop:
      test_and_set lock
      if not got lock
            jump loop
got_lock:
```

Pragmatic

```
get_lock:
loop:
      read lock
      if not lock free
            jump loop
      test_and_set lock
      if not got lock
            jump loop
got_lock:
```

In this way one can drastically reduce the number of read-modify-write cycles executed for frequently accessed locks. Furthermore, the costs of coarse-grain thread GC synchronization can be amortised within lock access. If threads need to poll to synchronize with the GC one can arrange that polling only happens when an attempt to access a lock fails:

```
get_lock:
      read lock
      if not lock free
            jump poll
      test_and_set lock
      if not got lock
            jump poll
      jump got_lock
poll:
      if GC_wants_to_synch
            jump sync_with_GC
got_lock:
```

When the GC thread wants to synchronize it grabs the lock. Thus a thread only polls when it can’t make progress because it has to wait for the lock to be released, or because the GC actually wants to synchronize.

5.2 Allocation in TDF

The interface between the program and the run-time system will be implemented in terms of a call to a function in the run-time system. All allocation requests will be serviced by the run-time system, which may require an allocation barrier for a concurrent system. The implementation of this is solely the responsibility of the run-time system, and is therefore outside of the domain of TDF.

5.3 Read and Write Barriers in TDF

It has been shown that either a read-barrier or a write-barrier (or both) may be necessary to maintain a consistent view of live data as seen by the garbage collector and the running program. For hardware implementations of barrier there is no need for the code of the program to contain any explicit cooperation with the run-time system — when using a VM barrier, for example.

For software barriers however, there must be explicit code in the program to implement the barrier. The nature of this explicit code is algorithm dependent. For a reference counting collector, it will be in the form of an update of the reference count, synchronized by a mutual exclusion. For tracing collectors, the barrier may be implemented with an extra indirection, or with a sequential store buffer, amongst other successful techniques.

Whatever the details of the read or write barrier, it is possible to provide an algorithm dependent implementation by writing a suitable token definition for read and write operations, and ensuring that the Dylan producer generates a use of this token whenever the fields within objects are accessed. This mechanism is identical to that described in [Mann93a] for the support of incremental serial garbage collectors.

5.4 Locating Roots for Garbage Collection in TDF

Garbage collection roots include global variables and local variables. Techniques for locating these roots in a serial computation model using TDF were described in [Mann93a]. The set of global roots is static information for a given program, and it is straightforward to communicate this to the run-time system. The introduction of parallelism does not affect the location of global roots since these are shared by each thread.

The introduction of parallelism does affect local root determination, since there are local variables associated with each thread. However, the mechanism for locating local roots with serial computation [Mann93a] actually builds a linked chain on the stack of maps of live local variables for each stack frame. Since this chain is maintained on the stack itself, the mechanism may be used with equal effect in the presence of multiple threads.

The garbage collector must arrange to scan the map of local variables for each thread. The way this is done is algorithm dependent, and it may be done either with a parallel garbage collector thread, or by a function call from a thread into the run-time system. In the former case, the run-time system is responsible for maintaining housekeeping information about where to find the head of the chain for each thread. In the latter case, the function call will be part of one of the barrier activities (either at a read, write or allocation request) and will be synchronized as appropriate in the same way as all barrier operations. The head of the chain will be passed as a parameter to a function in the run-time system, with the parameter in a known location, according to the normal calling convention.[1]

5.5 Expected costs and benefits for TDF support

The addition of multiple threads to the garbage collector requirements for Dylan increases the complexity of the garbage collector algorithms. However, these algorithms may be implemented in a platform dependent manner, and will not suffer in performance compared with native code implementations.

Support for locating local variables will incur the same costs as for serial garbage collection [Mann93a].

1. Note that in the important case of a conservative garbage collector it is not necessary for the garbage collector to make use of the map of live variables. In this case, there will be null definitions of the tokens which are defined to build the chain of maps.

6. Conclusions

It should be possible to implement a threads library for Dylan by using a mixture of architecture specific token definitions and an architecture specific run time system. Core TDF provides no direct support for architecture neutrality — but this is a conscious decision for the design of TDF since there is no common model of parallel computation.

The thread models described in the Preliminary Parallelism Extensions document [Edwards93] are well suited for implementing threads in Dylan. However there are some language features of Dylan which are not covered by this model.

There is a requirement for multi-thread support for the exception handling mechanism described in Proposed Extensions to TDF for Ada and Lisp [Currie93], as well as a requirement for a mechanism to define a thread-local variable to support the dynamic features of Dylan.

Implementation of garbage collection for a multi-threaded language is a complex problem — but existing industry-standard techniques can be used with TDF. Since compiler support for garbage collection is limited to the three areas of read barriers, write barriers and root detection, these areas may be implemented in the most appropriate manner for a given algorithm by using the combination of TDF token definitions and a run time system.

The introduction of multiple threads into a TDF implementation of Dylan is not expected to add any extra inefficiency problems, other than those inevitably encountered when implementing a language that uses a parallel garbage collection system.

7. References and Bibliography

[AMR92] Abdullahi, S.E., Miranda, E.E., Ringwood, G.A., “Distributed Garbage Collection”, 1992 International Workshop on Memory Management, St. Malo, France, September 1992, SpringerVerlag.

[Appel89] Appel, A.W., "Runtime Tags Aren't Necessary", Journal of Lisp and Symbolic Computation, Vol. 2, 1989.

[AEL88] Appel, A.W., Ellis, J.R., Li, K., “Realtime Concurrent Collection on Stock Microprocessors”, Proceedings of SIGPLAN ‘88 Conference on Programming Language Design and Implementation, June 1988, ACM.

[AL91] Appel, A.W., Li, K., “Virtual Memory Primitives for User Programs”, Proceedings of SIGARCH/SIGOPS/SIGPLAN Conference on Architectural Support for Programming Languages and Operating Systems, April 1991, ACM.

[Apple92] Apple Computer, “Dylan; An object oriented dynamic language”, April 1992.

[BM93] Balou, A.T., Miranda, E.E., “The Design of SPIRIT Multiprocessor Smalltalk”, Queen Mary Westfield College, Department of Computer Science tech. report #627, March 1993.

[BW88] Boehm, H-J., Weiser, M., “Garbage Collection in an Uncooperative Environment”, Software Practice and Experience, vol. 18, #9, September 1988.

[BDS91] Boehm, H-J., Demers, A.J., Shenker, S., “Mostly Parallel Garbage Collection”, Proceedings of SIGPLAN ‘91 Conference on Programming Language Design and Implementation, June 1991, ACM

[Currie93] Currie, I., “Proposed Extensions to TDF for Ada and Lisp”, GLUE Deliverable 4.1.1, DRA Malvern, September 1993

[Detlefs91] Detlefs, D., “Concurrent Garbage Collection for C++”, in *Topics in Advanced Language Implementation*, Lee, P. ed, 1991, MIT Press.

[DeTreville90] DeTreville, J., “Experience with Garbage Collection for Modula-2+ in the Topaz Environment”, DEC SRC Research Report #63, 1990.

[DB76] Deutsch, L.P., Bobrow, D., “An Efficient Incremental, Automatic Garbage Collector”, Communications of the ACM, vol. 19, #7, July 1976.

[DLMSS78] Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M., "On-the-Fly Garbage Collection: An Exercise in Cooperation", Communications of the ACM, vol. 21 #11, November 1978.

[Edwards93] Edwards, P., “Preliminary Parallelism Extensions”, GLUE Deliverable 5.1.1, DRA Malvern, October 1993

[FMY91] Furusou, S., Matsuoka, S., Yonezawa, A., “Parallel Conservative Garbage Collection with Fast Object Allocation”, OOPSLA ‘91 Conference on Object-Oriented Systems, Languages and Applications workshop on Garbage Collection, `anon. ftp cs.utexas.edu /pub/garbage/GC91.`

[HMS92] Hosking, A.L., Moss, J.E.B., Stefanovi´c, D., “A Comparative Performance Evaluation of Write Barrier Implementations", Proceedings of OOPSLA ‘92 Conference on Object-Oriented Systems, Languages and Applications, SIGPLAN Notices vol. 27 #10, ACM, October 1992.

[KMY93] Kamada, T., Matsuoka, S., Yonezawa, A., “Efficient Parallel Global Garbage Collection on Massively parallel Computers”, OOPSLA ‘93 Conference on Object-Oriented Systems, Languages and Applications workshop on Garbage Collection, `anon. ftp cs.utexas.edu /pub/garbage/GC93.`

[KHM89] Kranz, D.A., Halstead, R.H., Mohr, E., “Mul-T: A High-Performance Parallel Lisp”, Proceedings of SIGPLAN ‘89 Conference on Programming Language Design and Implementation, June 1989, ACM.

[Mann93] Mann, T., “TDF Support required by Lisp”, GLUE Deliverable 4.2.1, Harlequin Ltd., 1993.

[Mann93a] Mann, T., “Initial evaluation of TDF Support for Garbage Collection”, GLUE Deliverable 4.2.2a, Harlequin Ltd., 1993.

[NO93] Nettles, S., O’Toole, J., “Real-Time Replication Garbage Collection”, Proceedings of SIGPLAN ‘93 Conference on Programming Language Design and Implementation, June 1993, ACM.

[ON93] O’Toole, J., Nettles, S., “Concurrent Replicating Garbage Collection”, OOPSLA ‘93 Conference on Object-Oriented Systems, Languages and Applications workshop on Garbage Collection, `anon. ftp cs.utexas.edu /pub/garbage/GC93.`

[PU88] Pallas, J., Ungar, D., “Multiprocessor Smalltalk: A Case Study of a Multiprocessor-Based programming Environment”, Proceedings of SIGPLAN 88 Conference on Programming Language Design and Implementation, June 1988, ACM.

[Rovner85] Rovner, P., “On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically Checked, Concurrent Language”, tech. report CSL-84-7, Xerox PARC, June 1985.

[SS91] Sharma, R., Soffa, M.L., “Parallel Generational Garbage Collection”, Proceedings of OOPSLA ‘91 Conference on Object-Oriented Systems, Languages and Applications, SIGPLAN Notices vol. 26 #11, ACM, November 1991, ACM.

[Steenkiste91] Steenkiste, P., “The Implementation of Tags and Run-Time Type Checking”, in *Topics in Advanced Language Implementation*, Lee, P. Ed, MIT Press, 1991.

[TDF92] Foster, M., Currie, I., “TDF Specification”, DRA Malvern, September 1992.

[Wilson92] Wilson, P., “Uniprocessor Garbage Collection Techniques”, 1992 International Workshop on Memory Management, St. Malo, France, SpringerVerlag, September 1992.