A Shared-Memory Parallel Extension of KLIC and Its Garbage Collection

Nobuyuki Ichiyoshi   Masao Morita
Mitsubishi Research Institute *
ichiyoshi@mri.co.jp   morita@boss.mri.co.jp

Takashi Chikayama
Institute for New Generation Computer Technology (ICOT) †
chikayama@icot.or.jp

Abstract

An extension of KLIC (a portable sequential implementation of KL1) for shared-memory multiprocessors has been developed. The design goal was to add shared-memory parallel processing features without modifying the sequential core implementation. It was realized by not adding synchronization code to variable handling routines but introducing shared variables (variables which potentially need synchronization) and making data sharing explicit. Apart from the obvious merit from the software engineering standpoint, the separation minimizes the overhead imposed on the sequential core by a parallel extension. It is also justified by the growing discrepancy between local memory access cost and shared memory access cost in shared-memory multiprocessors using high-performance processors. In order to avoid the bus bottleneck problem in the stop-and-copy type garbage collection, an asynchronous garbage collection in which each processor can independently garbage-collect the shared memory area while others are working, has been introduced, (In the final paper, we will include more performance data.)

1 Introduction

KLIC [3, 2] is a portable implementation of parallel logic language KL1 [12, 1]. It is written in C, and compiles a KL1 program into a C program. KLIC is intended as a vehicle for porting the software that was developed in the Japanese Fifth Generation Project which would otherwise runs efficiently only on the specially-built parallel inference machines (PIMs) [11]. More importantly, it is to be a platform for application programmers to write KL1 programs for various hardware, from personal computers to single- and and multi-processor workstations to clusters of workstations to massively parallel computers. KLIC has a framework called "generic objects" for introducing new features, such as linking with foreign language subroutines, and with numerical or graphics libraries. To introduce a new class of generic objects, the user is to provide standard methods, such as deref(), unify(), gc() and print(). (These method are called implicitly from within the kernel code or from builtin predicates.) The user can also provide methods specific to the class. (A special syntax is used to call them: `generic:`*Method*(*Args*).)

A sequential implementation runs on various workstations and proves to be reasonably efficient (approx. 2 MLIPS peak performance on SPARCstation 10 with 36 MHz

*2-3-6 Otemachi, Chiyoda-ku, Tokyo 100, JAPAN
†1-4-28 Mita, Minato-ku, Tokyo 108, JAPAN

SuperSPARC processor). Two parallel extensions of KLIC have been developed: one for distributed-memory parallel computers and one for shared-memory parallel computers [7]. This paper describes the outline of the shared-memory parallel extension of KLIC and its garbage collection scheme (Distributed-memory parallel version of KLIC is described in [10].)

Shared-memory implementations of concurrent logic languages developed so far have taken either a UMA model (all memory is shared) or a NORMA model (no memory is shared). Examples of the UMA scheme include the shared-memory execution scheme in the PIM implementations [9] of KL1 and the JAM [4] implementation of Parlog [8]. Except for the local goal stack and various control data per processor, all data in the heap area are shared. Since a variable may be simultaneously accessed by multiple processors, its instantiation requires locking: lock the variable, check if it is still uninstantiated, write on it, unlock it. The problem is that a variable is always be locked before instantiating, even if it may not be really shared. For instance, the performance of a sequential program suffers when it is run on one processor in such a parallel implementation.

The implementation of Strand [6] for shared-memory multiprocessor is an example of the NORMA scheme [5]. Each processor has a private memory area, and inter-processor communication is realized by writing on and reading from communication buffers (one for each processor). In effect, it is a shared-memory porting of a distributed-memory implementation. A merit of this scheme is that sequential execution speed does not suffer from the parallel extension. (Locking is necessary only when writing to a message buffer.) However, the software overhead in interprocessor communication is considerably larger than the base hardware overhead (shared-memory read/write). Also, since communication is one-to-one, single-writer-multiple-reader communication requires $O(n)$ software overhead where n is the number of reader processors.

The design rationale of the shared-memory implementation of KLIC is to avoid the defects in both of the two "extermes", by mixing the two (a Mixed scheme). It tries to keep interprocessor communication overheads reasonably low by directly reading and writing shared area. However, it also tries to minimize the parallelizing overheads by separing the local and shared areas and localizing shared data operations. Under this Mixed scheme, the garbage collection of the local heaps can be done independently. The garbage collection of the shared heap does not adopt a synchronous parallel garbage collection (all-stop-and-all-copy), because it would make the shared bus a bottleneck. Instead, asynchronous garbage collection is introduced, in which processors perform shared heap garbage collection asynchronously: Each processor can enter a garbage collection independently (it copies those data which are accessible by itself), while other processors are running non-GC code. Preliminary performance figures are given in the last section.

2 Outline of the Shared-Memory Parallel Execution Scheme

2.1 Overview of Sequential KLIC

KLIC consists of a compiler and a runtime library. The compiler translates a KL1 source file into a C source file. These C source files are compiled and linked with the KLIC runtime library. User supplied object files and system libraries can also linked. Foreign routines (such C functions) can be invoked from the KL1 part via "generic objects".

A data word of KLIC is of the size of a pointer. Data words in memory are aligned at word boundaries (4 byte boundaries, at least) and the lower two bits of a pointer are used as tag bits. All four patterns are used: `REF`, `ATOM`, `CONS` and `FUNCTOR`. A REF is a reference to another data word ("a cell"). An uninstantiated variable without hooked

goals[1] is represented as a cell referring to itself, one with hooked goals is represented by a cell with a REF pointer to a two word suspension record. The first word of the suspension record points back to the variable cell and the second word contains a pointer to the suspension chain (list of goals hooked on the variable). The first type (unhooked variable) is recognized as a REF loop with length one, and the second type (hooked variable) is recognize as a REF loop with length two, respectively.

There is no room for adding new tags, but one can introduce a new class of data as *generic objects*. Generic objects come in three varieties: data objects, consumer objects, and generator objects. A data object looks like a FUNCTOR data, and is semantically a concrete value, such as a character string and a vector. A consumer object looks like a goal hooked on a variable, and is semantically a goal awaiting instantiation of a variable. Instantiating a variable on which a consumer object is hooked, triggers a specific object-dependent action. A generator object looks like a hooked variable, and is semantically a variable which becomes some other value on demand (normally it produces a concrete value). Generic objects are comparable to slots with procedural attachment in frame-based AI shells. Apart from the obvious merits from the software engineering standpoints, generic object mechanism prevents the added features (expected to be invoked not very often) from imposing overhead on the sequential core (which mostly determines the performance).

2.2 Design Goals

Because a high priority of the design is to not degrade the sequential performance of KLIC, it is decided that the shared-memory parallel version of KLIC should not have a special kernel which differs largely from the sequential base kernel (which is optimized for sequential execution), but it should be realized by introducing the shared-memory parallel processing feature as add-on features — i.e., by using generic objects. We regard shared-memory operations (non-local operations) as costly, and expect those to be infrequent.

In the parallel extension, there are multiple *workers*. A worker is a sequential process running KLIC program. It can execute Kl1 goals on its own, and it can also communicate with other workers by reading from and writing to shared memory.[2]

The most important difference between data handling in sequential implementation and shared-memory implementation is that there are shared variables in the latter. (By definition, a *shared* variable is one which may be accessible by more than one worker.) A shared variable needs a special treatment: locking is necessary for instantiating it and for hooking a goal on it[3]. In order not to impose locking overhead on non-shared variables, shared variables have been introduced as a separate implementation data type. Specifically, shared variables are implemented as generator objects of the type SHVAR. Non-shared variables remain a kernel data type. and operations on them do not involve costly locking/unlocking. They are referred to as *local variables* in the following.

Only those variables guaranteed to be not shared by multiple processors can be local variables. It follows that a shared data structure must contain local variables. It follows also that a shared variable may not be instantiated by a data structure containing local variables. To differentiate between data structures that contain local variables and ones that do not, they are allocated in different memory areas. There is one local heap for each worker, and one shared heap (global heap) shared by all workers (Fig. 1). A local heap is

[1]A hooked goal is one awaiting instantiation of a variable.

[2]It is intended that distinct workers run on distinct processors, although multiple workers can run on a single processor.

[3]When a goal must wait for instantiation of a variable, the goal is put on the *suspension chain* of that variable. It is called hooking of the goal on the variable.

read and written only by its owner worker. When unifying a shared variable with a local data, a worker may not overwrite the variable cell with a pointer to the local data, but it must copy the data to the shared heap and make the variable point to it.[4]

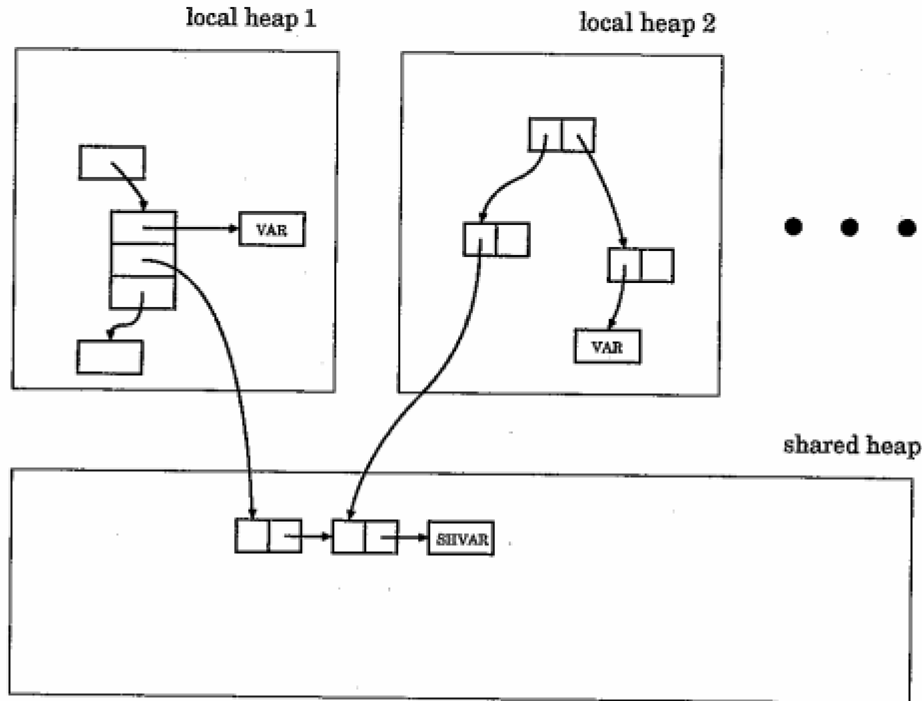

Figure 1: Local and Shared Heaps

2.3 Local and Shared Areas

A local area consists of the maintenance data for local execution and local heap for dynamic allocation of local goals and KL1 data. The maintenance data includes the local heap top pointer, the root of the local goal stack, etc. The shared area consists of the maintenance data for worker interaction and shared heap for dynamic allocation of shared goals and KL1 data. The shared maintenance data includes external goal pools (one for each worker), an interrupt flag for each worker, etc. An external goal pool of a worker holds the goals allocated to the worker by other workers for load distribution. It is the same as the one in the sequential KLIC, but is now allocated in the shared area, so that any worker can set the interrupt flag of another worker.

Each local heap is divided into two spaces for copying garbage collection. The shared heap area is divided into three spaces — *old*, *new* and *unused* — that are used in a circular manner (cf. Section 4.2.1). In general, there may not be pointers from the shared heap to a local heap. However, there are a few cases when such a pointer is necessary. Such a pointer is implemented by a two word record containing the target worker number and an index in the indirection table (called an *export table*) of the worker. The entries of the export table are updated at local garbage collection to reflect the moves of exported data.

Each worker has one "current" shared heap space (either the old or the new) for allocation of shared data. Each space has a worker table listing those workers whose

[4]This corresponds to the sending of a *unify* message to an external reference in the distributed-memory KLIC implementation. The main difference is that, in the shared-memory implementation, (1) a worker does not encode a data into a message, and (2) a data copied to the shared heap becomes accessible to all workers.

current space is that space. To minimize synchronization overhead, each worker keeps a page of shared heap for allocation of shared goals and data. When the page has run out, a worker demands a new page from the current space. (In the current implementation, the size of one page is 4 KB.) A worker "moves" from the old space to the new space when it performs copying garbage collection from the old space to the new. There may not be pointers from the new space to the old. The absence of such pointers guarantees that all data in the old space has become garbage when the last worker has moved to the new space. The old space can now be reclaimed and it becomes an unused space in the next stage.

2.4 Goal Distribution

Load distribution was dynamic and automatic in the shared-memory PIM implementations: An idle worker stole a goal from other workers' (or global) goal stacks. In the shared-memory KLIC implementation, one has to explicitly attach a goal distribution pragma to a body goal (syntactically, *Goal*`@node(`*N*`)`) to move that goal to a worker other than the worker which has reduced its parent goal. This is called *goal throwing*. In a way, it is a regress. One excuse is that as shared-memory access becomes more and more costly than local memory access, data clustering will become more and more important, and the programmer should be provided with some mechanism to control data and process allocation. A global goal pool could be introduced, if it should be desirable. A worker could put a goal to it by a special pragma `@node(global)`, and an idle worker could fetch a goal from the global pool.

To show how goal allocation is actually done, look at the following code.

```
p(X) :- ... | ..., q(X,foo(Y))@node(4), ... .
```

The worker executing the clause first allocates the goal q in the local heap (in the compiled code of the predicate `p/1`). The worker then copies the goal to the shared heap (in the runtime routine called "throw_goal").[5] In copying, (1) a CONS or FUNCTOR record is copied and its substructures are recursively copied, (2) a local variable is moved to the shared heap by allocating a new shared variable and overwrite the local variable cell by a REF pointer to that shared variable, and (3) a pointer into a shared heap need not be further traversed. Some cases must be handled exceptionally. (A) In moving a hooked local variable, the worker copies the suspension record to the shared heap. Each copied hook record refers to the corresponding goal in the local heap via the export table. (B) When the current space of the worker is *new* and it encounters a pointer into the *old* space, the worker has to copy the data to the *new* space. (C) When the current space of the target worker is *new* and the throwing worker encounters a pointer into the *old* space, it has to copy the data to the *new* space (it happens only when the current space of the throwing worker is old). This prevents a worker in the new space from seeing old data.

After the goal has been copied to the shared heap, it is put into the external goal pool of the target worker. The current priority level of the target worker is then checked. When the priority level of the thrown goal is higher than the current execution priority level of the target worker, the interrupt flag of the target worker is set so that the new goal may be scheduled right after the current reduction. Otherwise, the throwing worker does not interrupt the target worker. Each worker checks its external goal pool whenever it moves to a lower priority level, and copy thrown goals if any into the local goal stacks according to the priority level of each one of them.

[5]The data could be allocated directly in the shared heap. It is not currently done, because the compiler has to be modified to accommodate that. Since goal throwing is rare dynamically, the performance penalty should be little.

3 Some Optimizations

3.1 Instantiation of a Shared Variable

As mentioned before, shared variables are implemented as generator objects of type SHVAR. A SHVAR record contains a lock word and a pointer to the suspension chain. A worker has to acquire the lock to change the state of the variable (instantiating and hooking). A worker has to unlock the lock word after hooking itself on a variable. But, it need not unlock after instantiating the variable (it is a small optimization). This is because, in our implementation, a worker that fails to acquire the lock does not simply spin on the lock word, but it spins by watching the value part of the variable (the variable cell itself) and the lock word alternately. If the lock word is unlocked, the worker acquires the lock. If the value part of the variable changes, it indicates that the variable ceases to be a variable: it has been instantiated, or it has been unified with another variable. In that case, the worker will do appropriate things.

3.2 Generator Hooking

Suppose a process (producer) generates a list which is to be read by another process (consumer), and they reside in different workers. At first the producer and the consumer share a variable `S` (tail of the stream). When the producer generates a stream element `X1`, it first creates a list `[X1|L1]` in the local heap (because it is performed by the sequential core), and then unifies the list with `S`, which is implemented as a SHVAR generic object. The unification method of SHVAR copies the local list to the shared heap and places a pointer to the copied list `[X1|S1]` into `S`. Similar operations take place every time the producer generates stream elements. The successive instantiating and creating of shared variables incur a large overhead, especially in throughput-oriented programs (e.g., prime number generator).

To alleviate this problem, the *generator hook* scheme has been introduced. In the generator hook scheme, when a shared variable which is not waited upon is instantiated by a local structured data, the instantiation is not done at the point, but the variable is transformed into a generic object of the type *generator hook*. When some goal tries to read the generator hook object the object will produce the requested data by copying the local data to the shared heap. It is expected that that the whole or most of the structured data has been generated when it is requested. Note that in such a case the producer changes a shared variable into a generator hook only once, and the generator hook copies the whole data once.

In case some goal is already waiting on the value of the shared variable, the variable is not changed into a generator hook, but is instantiated immediately and the waiting goal is awakened.

In summary, when there is someone already waiting for a concrete value, local data is copied eagerly, otherwise, local data is copied lazily (on demand). The former is typically the case with data-driven or throughput-oriented programs, while the latter is typically the case in demand-drien or response-oriented (or, object-oriented) style programs.

4 Garbage Collection

4.1 Garbage Collection of Local Heap

The separation of local and shared heaps allows each worker to perform local garbage collection independently. The sequential KLIC implementation has a copying garbage

collector, and it is modified so that pointers to the shared heap are treated just as pointers into the new local space (i.e., the garbage collector does not copy shared data into its new local space).

4.2 Garbage Collection of Shared Heap

4.2.1 Asynchronous Garbage Collection

In shared-memory execution of PIM implementations, garbage collection of the shared heap was synchronous: all workers stop normal execution and start garbage collection. However, as single processor performance increases much more rapidly than that of shared bus throughput, and since there is little locality in garbage collection, the shared bus is becoming a bottleneck in parallel garbage collection.

To alleviate the above problem, an asynchronous garbage collection in which workers can perform shared heap garbage collection asynchronously is introduced. Instead of forcing all workers to become collectors (bus intensive) simultaneously, some workers can become collectors while others are mutators. In this way, it is expected that the level of bus request becomes more even along the time axis, with the result that the total idle time in waiting for bus response becomes smaller.

Asynchronous garbage collection is similar to on-the-fly garbage collection in that the collector(s) and the mutator(s) run simultaneously, but asynchronous garbage collection is different from on-the-fly garbage collection in that the role of mutator and collector is not fixed, but each worker switches between mutator and collector.

In our asynchronous garbage collection, a mutator working in the old space becomes a collector when it detects the shortage of the old space. A collector copies data in the old space to the new space, and then becomes a mutator working in the new space. If there were only two spaces ("old" and "new"), there is no target space of copying when the new space is exhausted. Therefore, one more space ("unused") is added. The reservation of an unused space guaranteed that, as long as the amount of active data does not exceed the size of a single space, shared heap overflow does not occur.[6]

Fig. 2 illustrates asynchronous garbage collection by snapshots of heap spaces. In this example, Workers 1 and 2 are initially working in the old space. Then, Worker 1 detects shortage of the old space (Fig. 2 (a)), and copies the active data it references to the new space (Fig. 2 (b)), while Worker 2 is still executing in the mutator mode. Worker 1 then resumes normal code. Later, Worker 2 also detects shortage of the old space and copies the active data it references to the new space (Fig. 2 (c)).

Suppose that the new space has run out. If the old space is still used by some workers, they are interrupted to be forced to do garbage collection. They copy the active data in the old space to the unused space. At this point, the spaces are rotated: the old space becomes the unused space, the new space becomes the old space, and the unused space becomes the new space.

In asynchronous garbage collection using three spaces, the following problems must be solved.

- Root set determination

  What are the root set of active data in the shared heap?

- Concurrent reads and writes

[6] The space efficiency of the shared heap is 1/3. Note that, depending on the timings of garbage collections, the amount of active data in the old and new spaces together may exceed the size of one space *without* causing shared heap overflow.

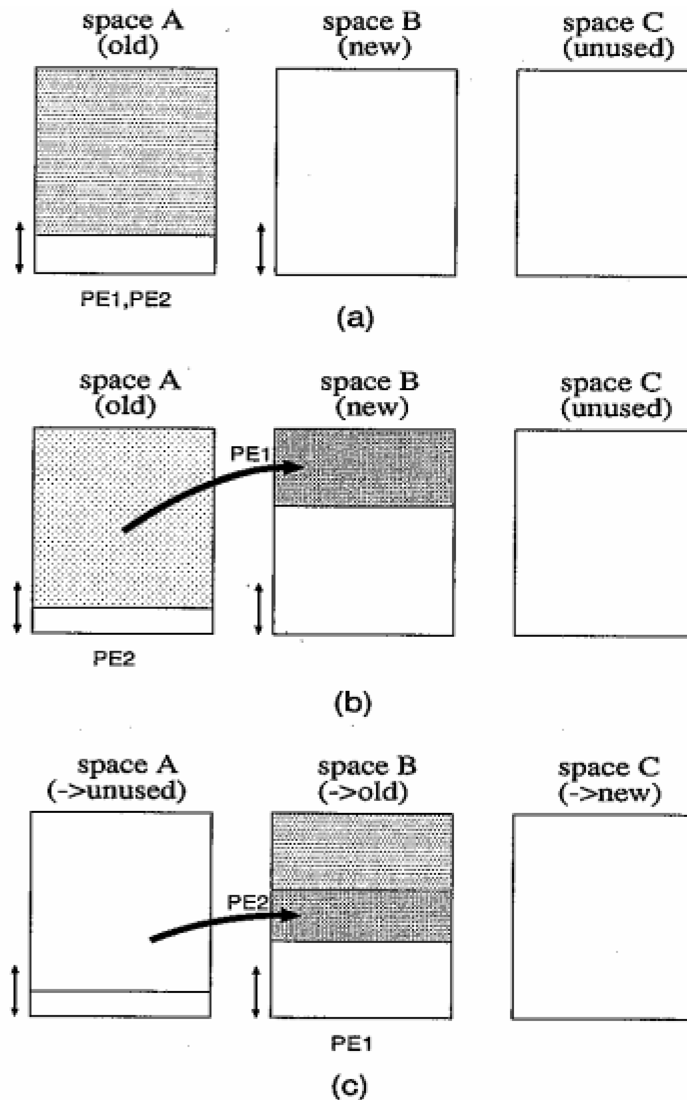

Figure 2: Asynchronous GC of Shared Heap

The mutators and collectors run concurrently and they may write on and read from the same locations.

- Reclamation of the old space

  In order to reclaim the old space (and make it a *new* unused space), there must not be pointers to it from the new space, or at least it must be possible to identify such pointers at reclamation. How is it to be done?

Our solutions to the above problems are described below.

4.2.2 Root Set Determination

Determination of the root set of active data is one of the main issues in on-the-fly garbage collection (or, incremental tracing collection) [13]. However, it is not an issue in asynchronous garbage collection. When a worker changes from a mutator to a collector, the collector simply inherits active data from the mutator. (To do this, pointers from the local heap to shared heap are registered in the root set during local garbage collection.) Thus,

expensive cooridation between mutator(s) and collector(s) in on-the-fly garbage collection (especially, on the part of the mutator(s)) is unnecessary.

4.2.3 Concurrent Reads and Writes

Care must be taken in copying data since mutators may be accessing the same data. In ordinary breadth-first copying algorithms, when a structure record in the old space is copied to the new space, a forwarding pointer to the new record is written into the old record. Since forwarding pointers must be distinguished from normal pointers, they are *illegal* data for the mutator.

The following are a few possible solutions to this. Some of them are KLIC implementation dependent, but the techniques may be of interest to designers of other parallel implementations of logic languages as well.

Read Only Old Space The collector only reads the old space but does not write on it, thus doing no harm to mutators. This mean that the collector does not write forwarding pointers into old records.

Note that shared variables are an exception: the collector must not split a single variable into two variables, by making a copy of the old variable in the new space. Splitting a variable into two separate variables would destroy the logical semantics. This can be handled by allocating a new variable in the shared space and unifying it with the old one. The value cell of the old variable will be overwritten by a REF pointer to the new variable. (The REF pointer effectively serves as a forwarding pointer.)

Because of the the absence of the forwarding pointer, the collector will not know where the copy of a record in the old space is in the new space (it does not even know whether or not the old record has already been copied). Thus, a record in the old space will be copied to the new space as many times as there are distinctive active paths to it, which could be a very large number.

A separate forwarding table could be maintained by the collectors. But the added space and runtime overhead makes it an unviable choice.

Bottom-up Copying A data structure in the old space is copied bottom-up to the new space, and pointers in the old record to substructures in the old space are overwritten by the pointers to the copied substructures in the new space . The copying of a list `[a,b]` (Fig. 3) proceeds as in Figs. 4 (a) and Fig. 4 (b). Since copied substructures in the new space are logically equivalent to the old counterparts, it is safe to replace it for the old substructures at any time (assuming that storing a pointer is an atomic operation, which it usually is). For example, the reference κ from some mutator represents `[a,b]` at any time.

This scheme suffers from the problem of duplicate copying, too. In particular, if a collector encounters a circular data, it will go into an infinite loop. But the degree of duplication is not too much: Unlike the Read Only Old Space scheme, an old record is copied as many times as there are active pointers to it (not as many times as there are active paths to it, the number of which could explode combinatorially).

Fortunately, most KL1 programs do not create circular data structures, and only a low percentage of records are referenced by more than one pointer.

Explicit Forwarding A new pointer tag (FWD) indicating a forwarding pointer could be introduced. The collector places the forwarding pointer into some field in the old

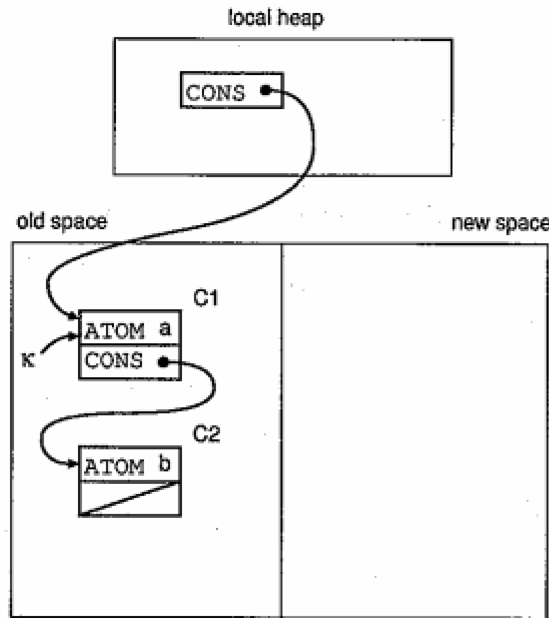

Figure 3: Original Data Layout (Before Copying)

record (predetermined for each record type) and make it point to the corresponding field in the new record. The mutator treats it just as an indirection pointer (REF pointer). For example, if a CONS record [a|b] is copied, the CAR part may be overwritten by a FWD pointer pointing to the CAR part of the new record, which is a.

This somewhat elegant (?) solution is not implementable, because sequential KLIC implementation uses up all tag bit patterns. Representing forwarding pointers by generic objects should be a natural alternative. But, this is ruled out, because a forwarding generic object takes up at least two words (a pointer to the method table and the forwarding pointer itself). Consuming four words (two for the CONS record and two for the forwarding generic object) per copied CONS record would not be tolerated.

Implicit Forwarding In the Implicit Forwarding scheme, forwarding information is represented by the presence of a indirection pointer (REF pointer) in one of the fields, say the first, of a record. The pointer references the corresponding field in the new record. For example, in copying a list [a,b], the first CONS record C1 is copied to the new space, the pointer from the local heap is updated to point to the new CONS record (C1'), and the first field (CAR part) of the CONS record C1 is overwritten by a REF pointer to C1' (Fig. 5 (a)). The REF pointer serves as the forwarding pointer. Then, similarly, the second CONS record C2 is copied, and a forwarding pointer is written (Fig. 5 (b)). For example, although the data structure referenced by pointer κ changes as the old list is copied to the new space, it logically represents [a,b] at all times before, during and after copying the list.

The problem with this scheme (especially, as compared to explicit forwarding) is that there may be look-alike REF pointers other than forwarding pointers, and these two must be distinguished. In our implementation, the REF pointers created by the mutator in the shared heap can be classified as follows:

(1) A REF pointer from a field of a record to a shared variable.

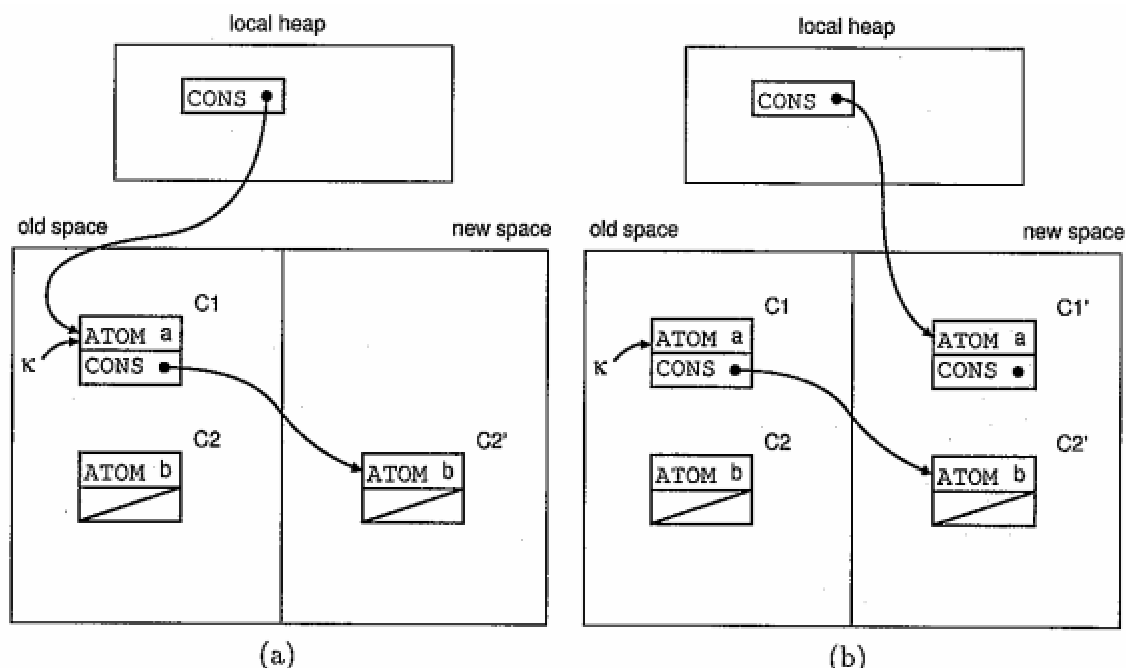

Figure 4: Bottom-Up Copy

(2) A REF pointer resulting from unification of two shared variables.

In case (2), there is no way to tell whether the pointer is created by a mutator or by a collector (in moving a shared variable in the old space to the new space), and there is no need to tell them apart. In case (1), if the pointer is from the old space to a shared variable in the new space, it cannot be distinguished from a forwarding pointer. Such pointers are created when a mutator copies a nested data structure from the local heap to the shared heap. Thus, the runtime copy routine was modified so that it inserts an indirection cell in such a case (a slight overhead in copying).

Pointers from the copied records in the new space to the substructure in the old space seem to violate the restriction of the pointer direction. It is allowed, however, because those mutators whose current space is new will never see such pointers (Otherwise, they could see the original structure which lies in the old space in the first place).

From the above, we implemented both the bottom-up copying scheme and the implicit forwarding scheme (one can choose either one by conditional compilation).

5 Preliminary Performance Data

The current implementation has been developed on a SPARCcenter 2000 (with Super-SPARC 40 MHz) running Solaris 2.3. It has been ported to a DEC 7000 AXP running OSF/1 SMP. The following measurements were taken on the SPARCcenter.

In the following, Seq KLIC stands for the original sequential KlIC, Shm KLIC for the shared-memory KLIC implementation described here ("Mixed scheme"), and Dist-on-Shm KLIC for the distributed KLIC implemented using the shared-memory message passing ("NORMA scheme"). In Dist-on-Shm KLIC, when a worker sends a message, it writes the message on the message buffer of the target worker and alerts the target worker by setting the interrupt flag.

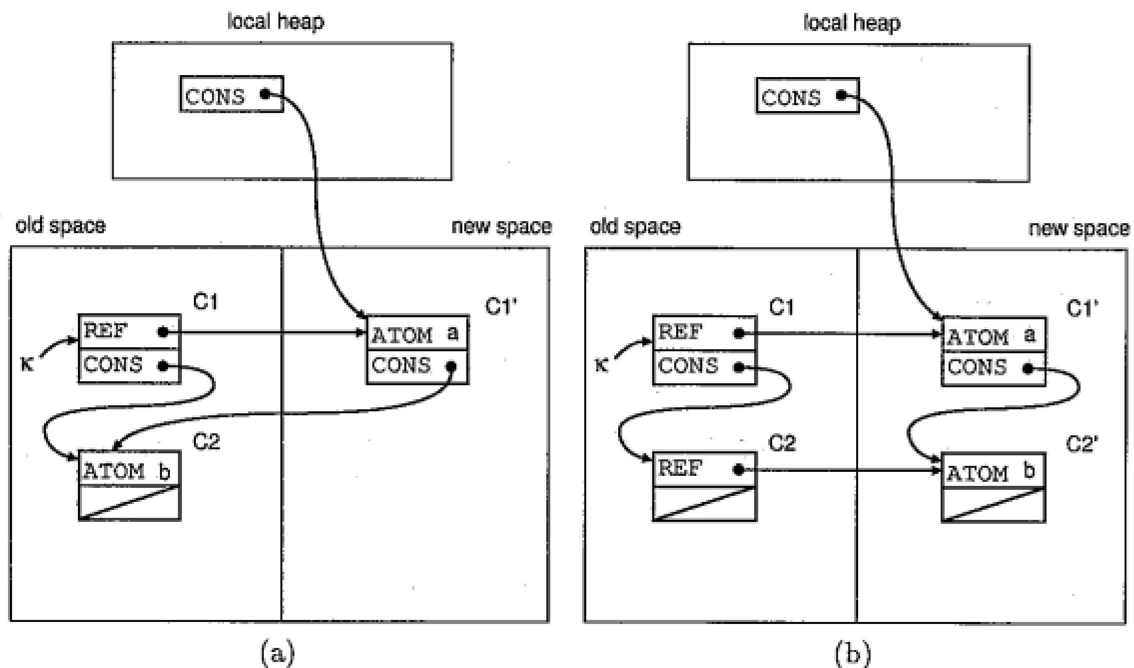

Figure 5: Top-Down Copy

Table 1: Round Trip Time

	time
Shm KLIC	176 μsec
Dist-on-Shm KLIC	493 μsec

First two sets of measurements confirm the two advantages of Shm KLICover Dist-on-Shm KLIC: low inter-worker latency and data sharing among workers.

Round trip times on Shm KLIC and Dist-on-Shm KLIC are shown in Table 1. Two KL1 processes are connected by two streams (represented by CONS lists) in the opposite directions, and each one sends an atomic message as soon as it receives a message from its counterpart. By definition, a round trip time is the average time between two sends by one of the processes. It represents double the message latency between two processes on different workers. The figures show that message latency of Shm KLIC (Mixed scheme) is roughly one third of that of a NORMA implementation.

Table 2 shows the time it takes for a given of readers to read 100,000 atomic messages, where each one of the message writer process and all reader processes resides on a distinct worker. The row indicated by "*" in the "readers" column is 50,000 times that of the round trip time (it should roughly the same as one-reader execution time). The difference between the first row and the second of Shm KLIC represents the effect of generator hooking optimization (when one of the reader requests for the first message, the generator copies all the messages to the shared heap). We have not found a plausible explanation of why the time jumps from 8 to 16 readers (it seems to be a SPARCcenter-specific phenomenon). In Dist-on-Shm KLIC measurements, the times of the first and second rows are close, because Dist-on-Shm KLIC returns only the surface level of the data to a read request. Since data is not physically shared, the writer has to copy the same data as many times as there are readers. Therefore, the reading time grows as the number of readers grows. The sublinear growth is probably due to the fixed cost of registering

Table 2: Multiple Reader

	readers	time
Shm KLIC	*	8,800 msec
	1	260 msec
	4	280 msec
	8	290 msec
	16	430 msec
Dist-on-Shm KLIC	*	24,650 msec
	1	28,570 msec
	4	41,490 msec
	8	62,460 msec
	16	133,640 msec

Table 3: Execution Time of Twelve Queens Program

	processors	time	relative performance
Seq KLIC	1	22.39 sec	1.00
Shm KLIC	1	22.40 sec	1.00
Shm KLIC	12	2.34 sec	9.57

addresses to the export table.

Table 3 shows the time to execute a twelve queens program. on Seq KLIC, on Shm KLIC on one processor, and on Shm KLIC on one processor on twelve processors. The program finds the number of all solutions of the twelve queens problem by divide-and-conquer. Goals are distributed at the second level cyclically. Each distributed goal represents a partial solution with column positions of queens in the first two rows fixed, e.g., [1,3], [1,4], ..., [5,1], ... (110 goals are distributed). The subproblems return the number of full solutions (not the solutions themselves). The figures show that sequential part of the shared-memory extension does not suffer from parallelizing overhead, as we have designed. The parallel speedup is lower than the ideal 12, mostly due to load imbalance.

For shared-heap garbage collection, we implemented six combinations: asynchronous bottom-up sequential copying, asynchronous bottom-up parallel copying, asynchronous top-down sequential copying, asynchronous top-down parallel copying, synchronous top-down sequential copying, and synchronous top-down parallel copying. A synchronous collection is a two semispace stop-and-copy collection. In a sequential collection, only one worker can be in the copying phase of garbage collection at one time. A parallel collection is one in which more than one worker can perform shared-heap garbage collection simultaneously.

It is hard to evaluate the different garbage collection schemes, because with the benchmarks we have ported, shared-heap garbage collection occurs only rarely and the execution time overhead is very low, although the number of local heap garbage collections are fairly large. This is mainly because once goals have been distributed, the execution of the "inner-most loops" are mostly closed in single workers. This very fact tells of a merit of the mixed-scheme over an UMA scheme: In the latter, there is only shared heap garbage collection. Measurements using a couple of artificial programs show that the garbage collection times of asynchronous bottom-up parallel bottom-up collection and of

asynchronous top-down parallel collection are the smallest among the six.

Acknowledgments

We thank the members of the KLIC development group and the KLIC working group for valuable discussions. In a broader sense, we are indebted to Shun'ichi Uchida and other people who worked hard to extend the FGCS project with the two-year follow-on project. It is our hope that younger generations will build on our humble efforts, and the seeds of the follow-on project may grow, new flowers bloom, new fruits be borne.

References

[1] T. Chikayama. Operating system PIMOS and kernel language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 73–88, 1992.

[2] T. Chikayama. Parallel basic software. In *Proceedings of FGCS'94*, 1994.

[3] T. Chikayama, T. Fujise, and D. Sekita. A portable and efficient implementation of KL1. In *Proceedings of the International Symposium on Programming Language Implementation and Logic Programming 1994*, 1994.

[4] J. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors.* PhD thesis, Heriot-Watt University, Edinburgh, May 1988.

[5] I. Foster. *Systems Programming in Parallel Logic Languages*, page 89. Prentice Hall, 1990.

[6] I. Foster and S. Taylor. *Strand — New Concepts in Parallel Programming.* Prentice Hall, 1990.

[7] T. Fujise, T. Chikayama, K. Rokusawa, and A. Nakase. KLIC: A portable implementation of KL1. In *Proceedings of FGCS'94*, 1994.

[8] S. Gregory. *Parallel Logic Programming in PARLOG — The Language and its Implementation.* Addison-Wesley, 1987.

[9] K. Hirata, R. Yamamoto, A. Imai, H. Kawai, K. Hirano, T. Takagi, K. Taki, A. Nakase, and K. Rokusawa. Parallel and distributed implementation of concurrent logic programming language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 436–459, 1992.

[10] K. Rokusawa, A. Nakase, and T. Chikayama. Distributed memory implementation of KLIC. In *ICOT/NSF Workshop on Parallel Logic Programming and Its Programming Environments*, 1994.

[11] K. Taki. Parallel inference machine PIM. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 50–72, 1992.

[12] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, 1990.

[13] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, pages 1–42. Springer LNCS 637, 1992.