Parallelization of Garbage Collection in a CCP System, Penny

DRAFT

Galal Atlam
galal@sics.se

Johan Montelius
jm@sics.se

Abstract

An important property of logic, object-oriented, functional, and other high-level programming languages is their automatic management of dynamically allocated storage. These languages often take the burden of memory management away from the programmer. The language support system provides the programmer with a virtually unlimited amount of storage by running a garbage collector to reclaim storage no longer needed.

The goal of this article is to present implementation and evaluation of a parallel garbage collection scheme used in a parallel concurrent constraint programming system called, *Penny*, running on a shared memory multiprocessors machine. Our parallel garbage collector employs multiple collectors working in parallel to make efficient use of the underlying architecture. Four strategies have been investigated, these strategies range from a sequential strategy to a full parallel one.

1 Introduction

Advanced programming environments such as functional, object-oriented, logic and concurrent constraint programming environments are based on dynamic object creation and automatic reclamation of storage during computation. A common problem in realizing such advanced programming environments, is how to manage efficiently the heap storage with automatic garbage collection. A garbage collector retains a program's data objects that are in use (active) and reclaims the space occupied by data objects that the running program can never access again (garbage). In addition to reclaiming storage, a garbage collector can also restore *locality* to fragmented data

by dynamically compacting it, thereby improving the performance of caches and virtual memory systems and reducing memory requirements [6, 17].

Now processors are continually getting cheaper multiprocessor machines, are widely in use, and a lot of researchers try to utilize this opportunity to speed up the execution of programs.

Advanced programming environments have been implemented on different parallel architectures and the shared-memory multiprocessors are the ones most used as a test makes the beds. It is important to develop a memory management system that efficiently manage the storage allocated by the parallel program and best use the underlying architecture.

One of the advanced programming enviroments is the Agents Kernel Language (AKL), being developed at SICS. It is a general purpose concurrent constraint language [10]. It combines the programming paradigms of search-oriented languages such as Prolog and Process-oriented languages such as GHC. AKL also allows nondeterministic execution. The parallel AKL system, *Penny* is implemented on a SparcCenter2000 (a shared-memory multiprocessor with 8 processors) machine.

The subject of this article is to investigate different ways for parallelizing the memory management part and garbage collection of the AKL system. Four strategies have been investigated, these strategies range from a sequential strategy to a full parallel one.

The article is organized as follows. Section 2 presents different garbage collection techniques and briefly review a number of parallel garbage collectors. Section 3 presents AKL as a CCP system and its parallel implementation, *Penny*. Section 4 describes the sequential and parallel garbage collectors. The implementation of parallel garbage collection will be presented in section 5. Section 6.1 presents an investigation of different garbage collection strategies. The performance and evaluation of the parallel implementation along with how to calculate garbage collection speedup and the speedup results will be presented in Section 6. Section 7 presents a new memory management scheme. The conclusions will be presented in Section 8.

2 Garbage Collection Survey

There exist several criteria for classify garbage collection techniques, one of them is the method used to identify garbage, directly or indirectly. Garbage collection techniques that identify garbage indirectly are: the *"mark-and-sweep"*, and the *"copying"* techniques. The other class which identify garbage

directly is the *"reference counting"* technique. These techniques will be discussed in the following section. In section 2.2 we will briefly review a number of parallel garbage collection schemes for shared memory multiprocessors.

2.1 Garbage Collection Techniques

In this section the three basic garbage collection techniques classified according to the garbage identification criterion will be briefly discussed.

- **The mark-and-sweep** technique [12] identfy the garbage objects by traverses and marks all objects reachable from root pointers. Then all the objects in memory are examined in order of their addresses. Unmarked objects are either put onto a free list, to be reused when the running program allocates new objects, or rearranged in such a way that all cells still in use appear in one compact mass at one end of the area. The remaining space at the other end can then be recycled by the program.

- **The reference counting** technique [7] identfy the garbage objects during program execution. Each object keeps with a counter specifying the exact number of references to it. Each time a reference to the object is created, the counter is incremented. Whenever an existing reference to the object is eliminated, the counter is decremented. When the counter reaches zero, the object is no longer referenced by the running program. And the space occupied by the object may be reclaimed. The spaces occupied by garbage objects are linked in free-list, to be reused.

- **The copying** technique [6] requires two memory spaces of equal size (old space and new space), only one space is used at a time. The collector picking out the worthwhile objects (*live*) amid the garbage from the old space and copies these objects into a new space. A flip operation then reverses the roles of the two spaces.

Copying garbage collector is very attractive for systems that have a high rate of garbage generation as in functional programming and the family concurrent logic programming. This because in the copying technique garbage collection time is proportional to the amount of data in use by the system ($O(|live|)$) [16]. This is in contrast to the mark-and-sweep technique, which has garbage collection time proportional the entire area (*live* and *dead* objects). Reference counting technique avoids this problem by incremental

collection, but it has problems in dealing with cyclic garbage structures and the reference counter themselves take up space, in addition to memory fragmentation problem. Also copying garbage collector is able to reclaim circular structures and reduces page thrashing as an effect of compacting the living data.

2.2 Parallel Garbage Collection Schemes

A number of parallel garbage collection schemes based on the copying technique is briefly reviewed in this section.

In Concert Multilisp, Halstead [9] parallelize Baker's incremental garbage collection scheme [4] for shared-memory machines. He statically dividing the heap area equally between workers, and each worker copies active objects onto its private new heap area during program execution which improves locality of references. There is no dynamic garbage collection load distribution among workers is given by Halstead [9] scheme.

Like Halstead [9], Ali [2] presents a parallelization of Baker's incremental garbage collection scheme for shared-memory multiprocessors. The scheme suitable for contiguous and non-contiguous memory heap blocks which allocated dynamically to the processors on demand. Ali's scheme supports dynamic load balancing. Processors without computational work perform garbage collection work, other processors continue with their computation. He used a local and global queues for maintaining the garbage collection works. These queues can implemented without space overhead by four words as a queue header and the queue body moved into the heap cells.

Imai and Tick [3] parallelized the stop-and-copy garbage collection scheme of Cheney [6] for shared-memory multiprocessors. They implemented the algorithm within a concurrent logic programming system called, VPIM. Objects with equal size are allocated in the same memory block. The scheme supports dynamic load balancing mechanism, that is any worker can garbage collect objects in any memory. This implemented by a global stack for maintaining garbage collection work that can be taken by any worker. Their calculated garbage collection speedups are expected.

Ellis, li, and Appel [8] proposed also the design and the implementation of a concurrent copying garbage collector on a shared-memory multiprocessors. One processor reclaims all the garbage, while others proceed with their normal computation. Heap area organized in pages. They using a memory protections facilities provided by the hardware for operating system synchronization between the collector and the mutators.

R. Sharma and M. L. Soffa [19] presents a parallel generational garbage collection scheme. They used a shared-memory multiprocessors simulator for implementation of their scheme. Heap area is represented by several generation and the garbage collection of each generation can take place in parallel. The preformance of their scheme evaluated by comparison with Ellis, li, and Appel [8] scheme.

We are implemented the scheme proposed by Ali [1] which is a parallelization of a sequential stop-and-copy garbage collection scheme based on traversing active data in a depth-first manner. The heap area is non-contiguous memory blocks with no extra space overhead. The non-contiguous memory blocks organization of the storage heap is much more flexible than the one based on one large contiguous heap area. This is because it is much easier to get a number of smaller memory blocks than to get one big block from any operating system. The garbage collection load balancing depend upon the implemented strategy as we will see section 6.1. The speedups calculated in our implementation are real speedups.

3 A CCP system, AKL

To parallelize the garbage collection of the parallel AKL system, needs to give a brief description of the parallel implementation of AKL.

AKL is a concurrent constraint logic programming language. The program in AKL is a set of predicate *definitions*, where no two definitions define the same predicate. The predicate of every program atom occurring in a clause in the program has a definition in the program. Each predicate is defined by a set of guarded clauses consisting of a head, a guard, a guard operator and a body. The head is a program atom, the guard and body contain program atoms and constraint atoms. There are three types of guard operators: *wait, conditional* and *commit*. All clauses of the definition has the same type of guard operator.

3.1 Parallel Implementation

???? need more, a good introduction to AKL ???????

The parallel AKL system, *Penny*, [14] represents the execution state information needed by a worker. The execution state of each worker represented by, the configuration tree, a-registers and context stacks.

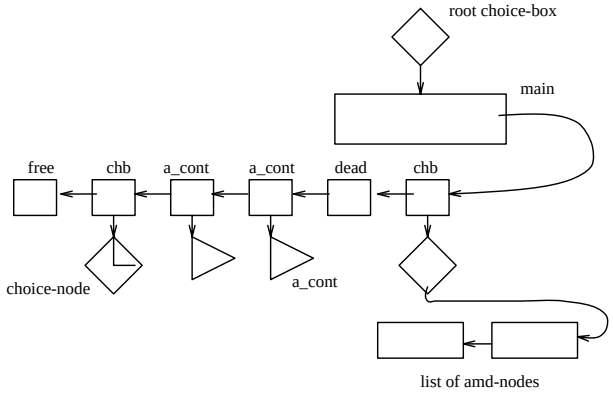

Figure 1: The configuration tree.

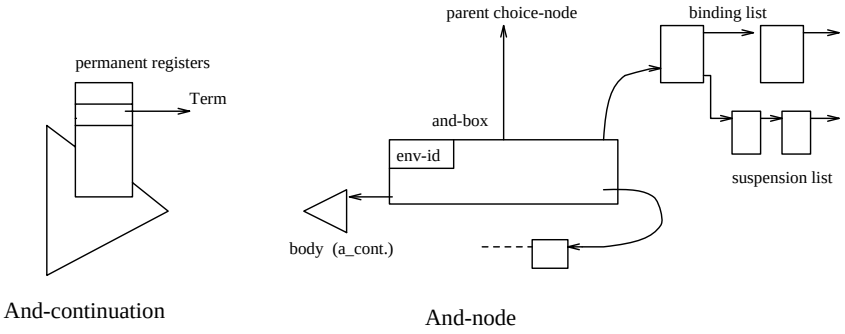

Figure 2: The and-node and the and-continuation.

Configuration tree

The configuration tree, as shown in Figure 1, consists of:

- *and-node*
- *choice-node*

The root of the tree is a dummy choice-node that does not represent any goal but serves only as a sentinel. It holds an empty choice-continuation and a single and-node (the main and-node).

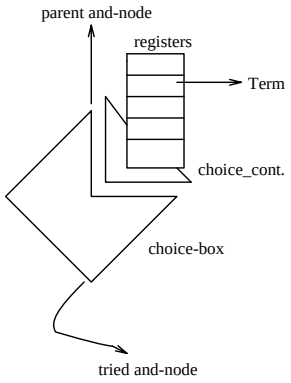

Figure 3: The choice-node: choice-box and choice-continuation.

And-node

It is the representation of a guarded goal. It contains a sequence of goals (*insertion points*), *environment identifier*, a list of *bindings*, a *body* and some status information. The environment identifier is a reference to a cell used by variables to identify the *home* and-box.

The sequence of goals in the and-node are represented by the list of insertion points. Each insertion point is either dead referring to a *choice-box* or an *and-continuation*. An and-continuation represents a sequence of program atoms. It contains the a tuple of permanent registers, a code pointer and the insertion point of the continuation. The *And-node* and The *and-continuation* are shown in Figure 2.

Choice-node

A choice-node represents a choice-box. It contains a sequence of and-nods and *choice-continuation*. A choice-continuation represents a sequence of guarded goals a copy of argument registers. The *Choice-box* and *choice-continuation* are shown figure 3.

Binding lists

It represents constraints on external (conditional) variables Unconditional bindings can never be removed and can therefore be recorded in place. Conditional bindings must only visible in or below the and-box in which the

binding occurs and are therefore recorded in the binding list of the and-node.

A binding list consists of an entry for each constrained external variable. An entry contains, apart from a reference to the variable, the binding of the variable or a list of suspensions. An entry that a list of suspensions is called a *suspension entry*. The suspension entry does not constrain the variable, its purpose is only to keep track of suspensions. An entry that is not a suspension entry is a *proper entry*.

Term representation

Terms are represented as tagged pointers, one high bit reserved for marking during garbage collection. Variables use three primary tags: One for the variable reference (REF), which points to a variable cell, and two more (UVA, CVA), which may be used within a variable cell to mark it as an unbound unconstrained or constrained variable, respectively. An unconstrained variable holds, apart from its tag, a reference to the home and-box, this reference is called environment identifier (envid). A constrained variable holds a reference to a structure holding the environment identifier and a list of suspensions. Atoms hold only an identifier, typically a short integer or an index to an atom table. Structure holds, apart from the functor and its arguments, an environment identifier. Special objects (Generic objects) represented by a reference to a structure holding the reference to its method table, environment identifier and the object. Figure 4 shows different type of terms.

A variable is *local* to an and-box, referred to as the *home* of the variable. Variables that have their home in the path from the root to the and-box are *external* to the and-box.

Notice that the representation of variables is different from the representation of variables in WAM [20] where unbound variables are represented by a self reference.

3.2 Memory Management

All run time terms are allocated on the *heap*. The *heap* is also used for *choice-boxes*, *choice-continuations*, *and-boxes*, *and-continuations*, *constraints* and *suspensions* allocation. The heap is automatically expanded if needed and is subject to garbage collection.

The *code area* holds the instructions of definitions. It has a static size,

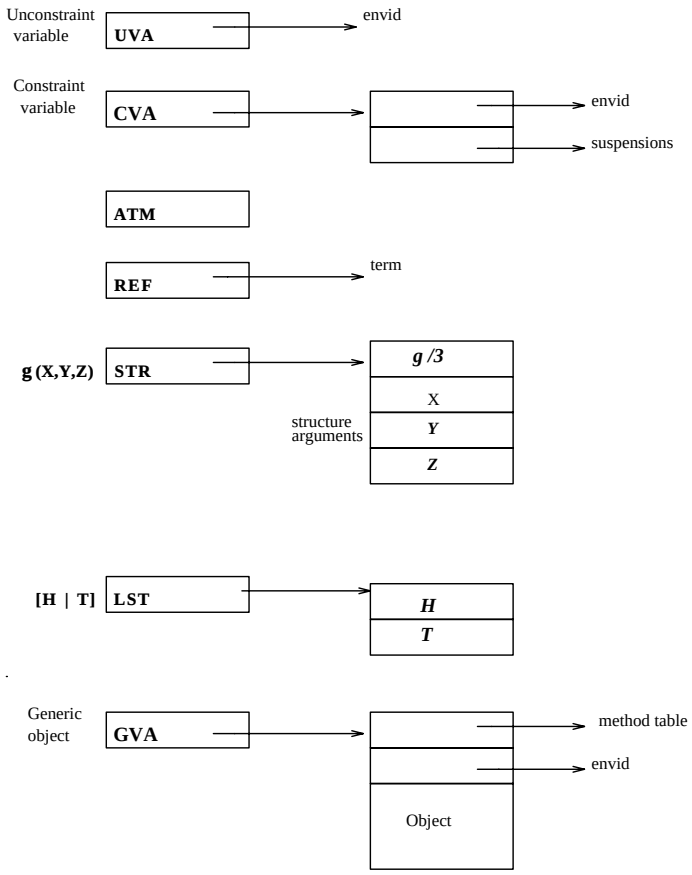

Figure 4: Different types of terms.

and is not subjected to garbage collection so once we run out of space there is no other option but to restart the system.

The *constant area* is used to allocate structures that will not become garbage during the execution (*atoms, functor, definition names* and load time *integers*). The constant area is not subject to garbage collection and does not increase in size.

Storage heap organization

In the parallel implementation, the storage heap is represented by one global pool of non-contiguous free memory blocks and a number of used memory blocks associated with each worker. During computation workers get free blocks from the global pool until certain threshold, whereupon the parallel garbage collector will be invoked. Each worker has a separate heap and when it creates new terms during execution it always places them on its own heap. The garbage collection uses a copying strategy where the remaining free blocks are used.

The segmented heap allows an efficient implementation of incremental heap space. If a larger heap is needed new blocks are requested from the operating system.

Notice that at any time more memory blocks could be requested from the operating system and added to the global free list. Memory blocks are small and non-contiguous, so the chance of getting them from the system is much higher than when requesting one big memory block for the entire storage heap. The global free list could be accessed simultaneously by more than one worker, using global lock prevents any possible conflicts. Every worker can read or write to existing cells on any other heap, hence a worker can create pointers to cells on the other worker's heap.

Stacks

Each worker has its own *context stacks* to maintain its execution state. These stacks are *wake*, *recall*, *a-cont* and *task* stacks. These stacks are divided into segments, each segment referenced from *context* stack entry. Apart of context stacks each worker has two stacks for maintaining garbage collection work, and a stack for maintaining references to unbound variables. Memory spaces of these three garbage collection stacks are returned to the operating system in the end garbage collection cycle.

4 Garbage Collection in AKL

The garbage collection of terms (variables , structures, lists, atoms, ..etc) is, in the sequential system, implemented using a stop-and-copy garbage collection scheme based on traversing active data depth-first. That is, if the current cell to be scanned in a copied structure A points to an uncopied structure B, the next cell to be scanned will be the last cell in a copy of B. The next cell in A will be scanned only when all cells of B and the new copied structures accessible from B have been completely scanned. That is, cells are scanned in last-copied-first-scanned manner, [5]. Section 4.1 presents the idea of the depth-first scanning scheme. Section 4.2 shows how it can be parallelized. Section 4.3 presents how the garabge collctor in the Penny system can use this scheme in its parallel implementation.

4.1 The Idea of Depth-First Scanning Scheme

The chains of cells to be scanned are maintained by two pointers; one points to the current location (LOC) and the other points to the previous location (PREV-LOC), and a register (TAG) contains the tag of the current object. Moreover, one bit of each cell (high-end bit) is used by the garbage collector to mark the copied cells. There are two uses for this mark bit; one for marking already copied data and the other for marking the last cell to be scanned in the copy of an structure. That is, when a structure is copied, the first cell of the old copy is marked and a pointer to the new location (*forward pointer*) is overwritten in the same cell. Cells to be scanned are divided into two chains; one for cells in the current structure and the other for all remaining cells to be scanned. LOC is used to point to the beginning of the first chain and PREV-LOC is used to point to the beginning of the second chain. The first chain consist of consecutive memory cells and is terminates by a marked cell whereas the other chain consist of linked structures that are terminated by a dummy (DUMMY) cell. A *link-reversal* (reversal pointer) technique, similar to the one described in [18, 11], is used for managing the latter chain. Each cell in an object O used to link objects contains also the tag of O. The register TAG contains the tag type (STR or LST) of the current object[1].

It is better to explain the scheme by a simple example. Assum a term structure foo(g(a,X),b), this structure is represented by a tagged pointer (STR,address) pointing to an area of (2+1) contiguous cells (see structure

[1] Let us assume that we have two object types: STR (structure) and LST (list).

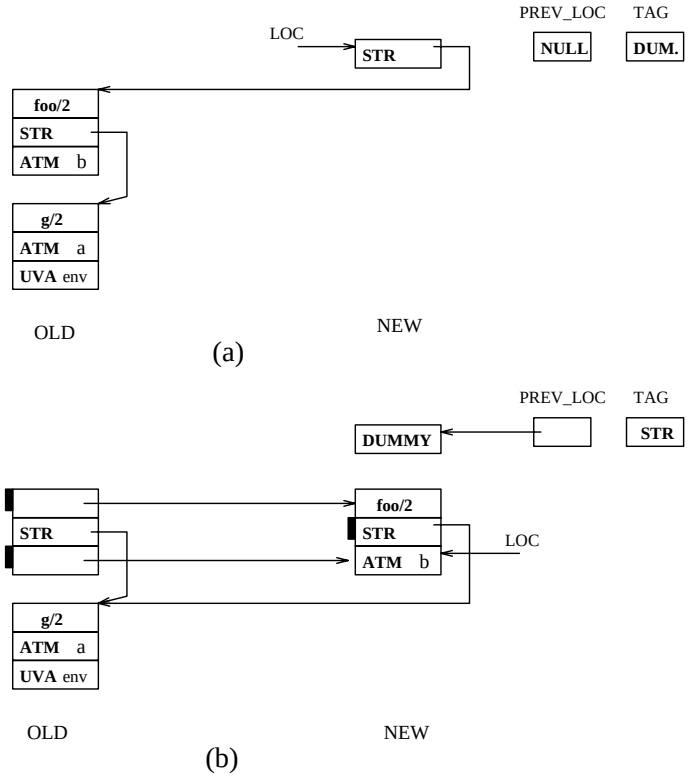

Figure 5: (a) Step(1), (b) Step(2).

representation in section 3.1). As shown in Figure 5(a) the address part of the tagged cell (STR,address) points to the OLD space. The scheme explained in the following steps:

Initially, the tagged pointer *(STR,address)* is the cell to be scanned and points to a three contiguous cells in the OLD space. LOC points to this cell, PREV-LOC is NULL and TAG is DUM, Figure 5(a).

Step(1): the strusture (functor and two arguments) pointed to by address is copied into the NEW space, the first cell in the old strusture is marked and a forward pointer to the new location of the strusture overwritten in it. Mark the last cell to be scanned in the new copy (the first argument). Push the copied strusture (object) in the scavenger stack, that is, PREV-LOC and TAG are saved into the cell pointed to by

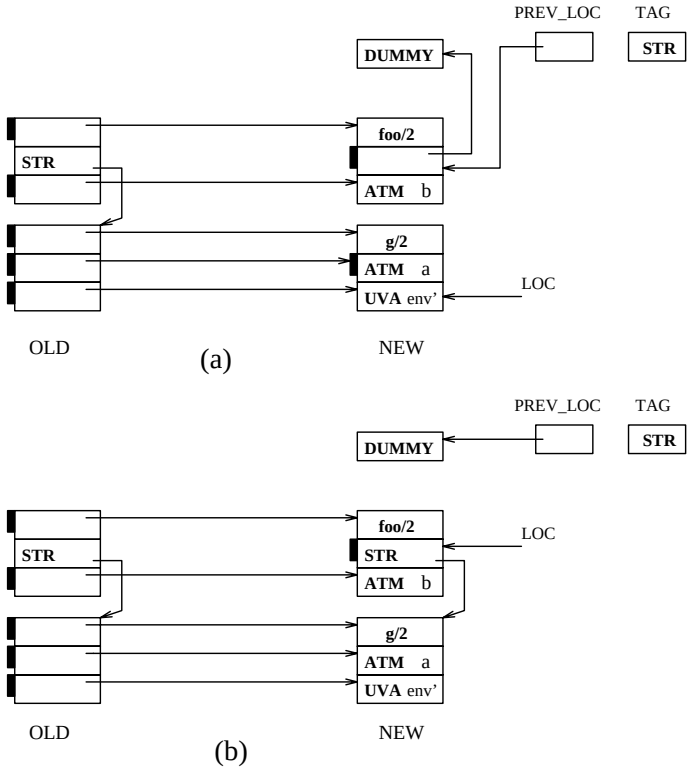

Figure 6: (a) Step(3), (b)Step(5).

LOC, and set PREV-LOC to LOC. Set LOC to point to the first cell be scanned and set TAG to type STR, Figure 5(b).

Step(2): the cell be scanned is an ATM cell "b", then LOC is decremented to point to the next cell to be scanned.

Step(3): the current cell be scanned is a tagged pointer to the structure (g/2). Then repeat setp(1) for the structure (g/2), Figure 6(a).

Step(4): the cell be scanned is an unbound variable[2] "X" and it is not marked as a last cell to be scanned in the current structure. Then LOC is decremented to point to the next cell to be scanned.

Step(5): the cell be scanned is an ATM cell with "a" value and this cell was marked as a last cell to be scanned in the current structure g/2.

[2]During copying this variable, its "envid" is dereferenced and stored in the new copy.

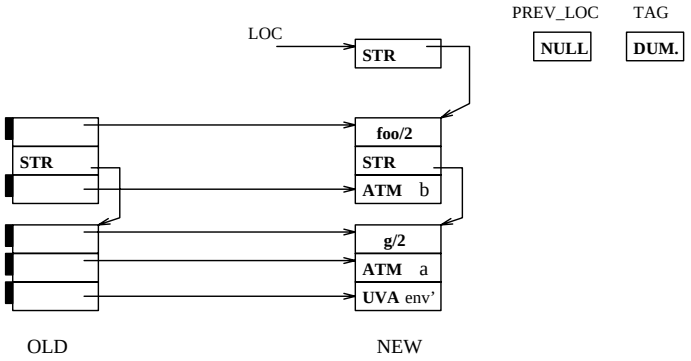

Figure 7: Structure foo(g(a,X),b) after copying to the NEW space.

Then, pop the next object to be scanned (foo/2 structure) from the scavenger stack. that is, set LOC to PREV-LOC, PREV-LOC to its contents and TAG to the tag stored in the contents of PREV-LOC. The cell pointed to by LOC will be contained a tagged pointer the previous values of LOC and TAG, Figure 6(a).

TAG is not DUM and PREV-LOC is not NULL, this means that the scavenger stack contains unscanned objects. from it.

Step(6): LOC points to the marked cell, the last cell in the current object (foo/2), and the scavenger stack is not empty. Then , pop the next object to be scanned as in step(5). TAG is a DUM and PREV-LOC is NULL, this means that scavenger stack is empty and there is no more objects to be scanned.

Figure 7 shows the situation after the scanning is completed by copying all structures (objects) accessible from the *(STR,address)* cell.

4.2 Parallelizing the Depth-First Scanning Scheme

In the above sequential scheme, cells to be scanned are maintained in two chains, as shown in Figure 8(a). One chain for a contiguous sequence of cells in the current object, pointed to by LOC. The second chain for all remaining cells to be scanned, pointed to by PREV-LOC. Ali, in his paper [1] proposes the idea for parallelizing the scanning of cells, by dividing the latter chain into a number of smaller sub-chains and to make them available to the other workers. Each chain is a piece of work that can be processed by any worker.

Figure 8(b) illustrates the idea of deviding the chain pointed to by PREV-LOC into two chains. The first chain is pointed to by PREV-LOC and the the second by Pi. Each chain ends with a dummy (DUMMY) cell. Li contains the pointer to the last object in the chain pointed to by PREV-LOC and the tag of this object. Pi and Li contain enough information for any worker to scan this piece of work and also to link this chain with its previous chain. Each worker maintains its chains into its own *chain-work* stack. Idle workers during garbage collection can get work from other workers with excess load by just getting a small amount of information (only two words) describing the chain of cells to be scanned. If any worker gets a piece of work from its own stack or from another worker. It sets its LOC to Pi, and PREV-LOC and TAG by the information found in cell pointed to by Pi. The worker scans each piece of work independently as in sequential scheme.

Workers distribute work between themselves until all workers become idle. Figure 8(b), shows how worker (W0) divids the chain of cells to be scanned and maintains the sub-chains information in its chain-work stack. For more details about: how to divide a chain of cells and how to distribute work among workers with dynamic load balancing, see [1].

4.3 Parallelizing the Sequential System

Is it really the parallel garbage collection an alternative to a sequential one?. The affirmative answer of this question will be given in section 6. To parallelize the sequential system there are two different schemes:

- parallelize the scanning of cells.

  The idea proposed by Ali [1] is attractive for parallelizing the cells to be scanned in our parallel garbage collector. Figure 8(a) shows the chain of cells to be scanned in the sequential garbage collector and how the pointers LOC and PREV-LOC and the tag register TAG are used to maintain this chain. Figure 8(b) shows how worker W0 divids this chain to sub-chains and maintains two references to describe this sub-chain in its own stack.

- parallelize the configuration tree.

  The parallelization achieved in the level of the configuration tree, dividing the tree and distribute it among the workers. One worker, for instance W0, traverses the tree and pushes work into *tree-work* stack of

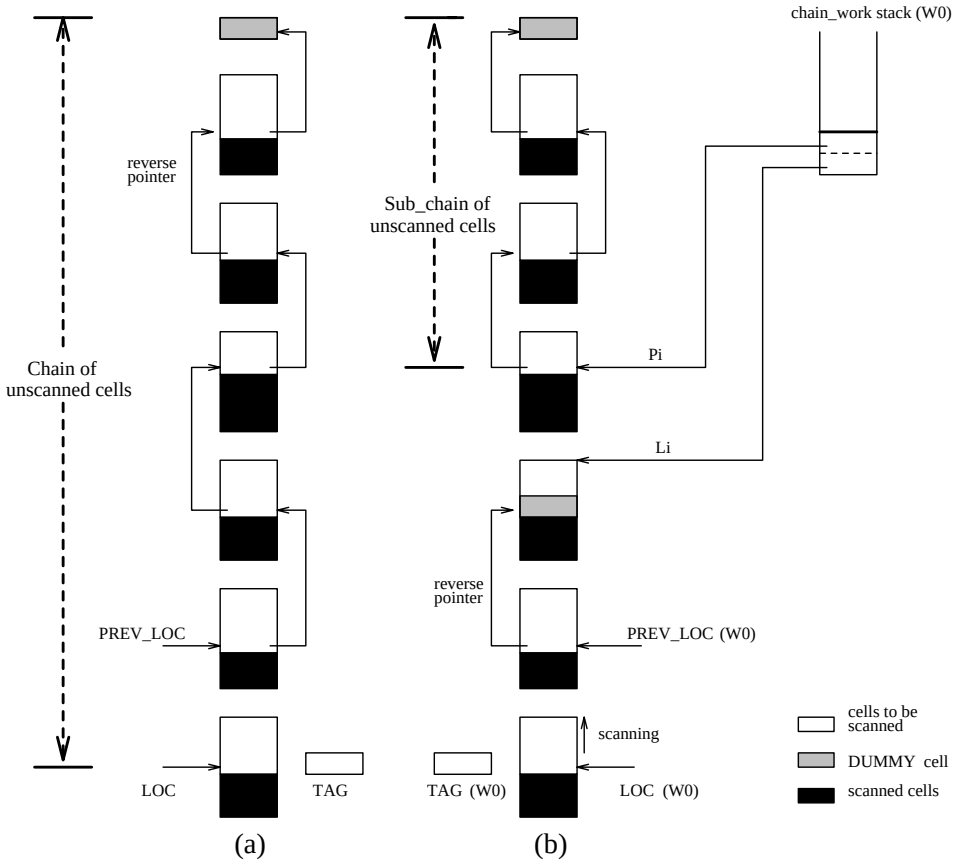

Figure 8: (a) Chain of cells to be scanned, (b) Dividing the chain of cells to be scanned.

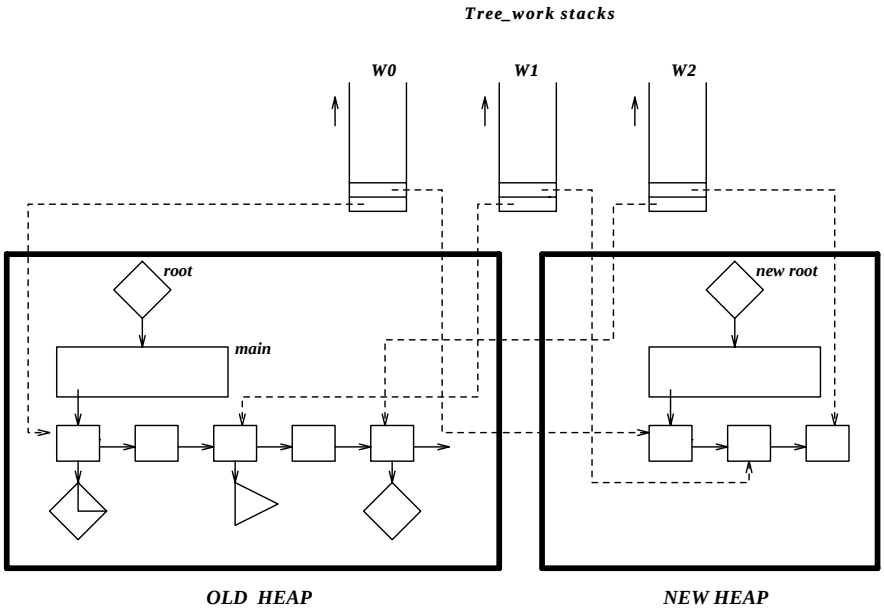

Figure 9: W0 traverses the tree and distributes the work.

each worker. Each piece of work is maintained by two pointers. The work either: and-continuation work or choice-node work. The and-continuation work is copying and-continuation and garbage collect its registers by copying all accessible data structures from it. The choice-node work, is copying choice box and choice continuation (if any) and garbage collect its registers.

When worker W0 completes traversing the tree it signals all other workers to garbage collect the work pushed in their stacks. Figure 9 shows how the W0 worker distribute works in the configuration tree.

5 Implementation

Parallelization of the sequential garbage collection process is not straightforward. There is a lot of problems need to be solved. For instance, managing the workers during garbage collection, how to distribute work among the workers, how to reduce the synchronization barriers as possible and avoid simultaneous update of a cell in the old heap by more than worker.

During normal program execution when a worker has shortage of heap,

try to get a new heap memory block from the global pool. If the used heap blocks reaches certain threshold, whereupon this worker signals all other workers for garbage collection. All workers stop computation and synchronize to enter garbage collection. Each worker, starting from its roots, copies accessible data into free blocks taken from the global pool. One worker, *master*, manages the garbage collection process by signal all other workers, *slaves*.

There are two types of roots in the parallel implementation of AKL system, *Penny*: one of them is a shared root, and the other is a local set of roots. The shared root is the root of the configuration tree, *root choice-box*. The argument registers, *a-registers*, the context stacks and the reference stack are considered the set of local roots. Copying the configuration root and all live nodes accessible from this root and the associated data structures depend upon the used strategy, as we will see in section 6. The data structures (*terms*) copying process, is traverses each copied data structure depth-first. Special objects, for instance *port*, have their own method table. When this object encountered the garbage collection method of this object will be called.

After coping the configuration tree, each worker scan its *a-registers* and copy the trems referenced from these registers using depth-first scheme.
In copying terms not all cells in the old copy are marked. The reason for this is that the garbage collection process includes a dereferencing optimization that changes REF cells to (LST or STR) pointers or ATM value.
The unbound (unconstraint, constraint) variables are handled like in sequential system [5]. When an unbound variable cell in the OLD referenced by reference pointer (REF, address), this cell has two states:

- Marked, this means that it is a part of data structure and this data structure was copied. Then, address of the reference pointer updated by the forward pointer in the marked cell.

- Unmarked, this means that it is not copied and may be it is part of uncopied data structure and it may be copied latter. Then we need to maintain a reference to the reference cell (REF, address) to be investigated after the scanning and copying process finished.

This scheme prevents multiple copies of the unbound variables cells. So, each worker has a stack called *reference stack* similar to the one used in the sequential garbage collector. This stack used to maintain a references to REF cells pointing to uncopied unbound variables during scanning cells in

the NEW space.
After copying the configuration tree and each worker scan its a-registers. Then, each worker traverse its reference stack and copy all uncopied variable referenced from it, and compact the context stacks. That is, remove the entries which point to uncopied nodes. Variables and data structures hold a reference to the home and-box. When updating this reference we ensured that all and-boxes were copied. Also for compacting and update the context stacks we ensured that configuration tree was copied.

Then we can summarized the garbage collector task in the parallel AKL system, *Penny*, as a three steps performed in order:

- garbage collect the configuration tree and the associated terms,
- garbage collect the argument registers, and then
- garbage collect reference stack and compact the context stacks.

Apart from all workers must be synchronize for enter garbage collection and for restart normal execution , there are two barriers, where all workers must synchronize before proceeding:

- Garbage collect the configuration tree must complete before start to garbage collect their a-registers,
- All workers must complete garbage collect their a-registers before start to garbage collect their reference and context stacks, and

In the end of garbage collection process the master worker return the old heap memory blocks to the free-list and signals other workers to restart execution.
We avoid the first barrier by update the term's environment while compact the context stacks, Figure 10 shows the effect of eliminating this barrier on the garbage collection time.

All data structures in the configuration are allocated in a shared memory. So, to avoid simultaneous update of a cell in the old heap by more than one worker, a mechanism for supporting exclusive access to cell is needed.
Each worker has apart from its context stacks, two stacks for maintaining the garbage collection work, *tree-work* and *chain-work* stacks. And has it's own LOC and PREV-LOC pointers and the tag register TAG, for maintaining the chain of cells to be scanned (see section 4.2). The memory space of these stacks along with reference stack are returned to the operating system after garbage collection.

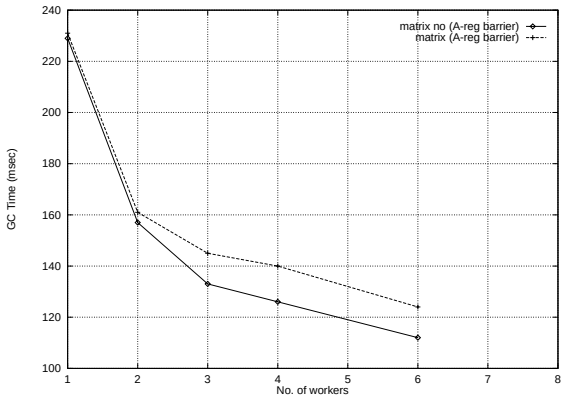

Figure 10: Effect of removing a-registers barrier.

6 Performance Evaluation

In the parallel AKL system, *Penny*, four different versions of garbage collection strategies have been investigated. These strategies range from a sequential strategy to a full parallel strategy.

6.1 Garbage Collection Strategies

In the following descriptions we will talk about *master* and *slave* workers. Any worker can potentially be the master since they all have the information required. In the implementation one worker is the dedicated master.

Strategy 1: Sequential Garbage Collection The master first garbage collect the configuration tree and the associated terms. The workers then, one at a time, garbage collect its own argument registers. In the final phase each worker traverses the reference stack and compact the context stacks.

Strategy 2: Parallelize the Scanning of Cells In this strategy the scheme presented by Ali [1] is implemented. The master will start to garbage collect the configuration tree while slaves, in parallel, garbage collect their argument registers. The workers will, when they garbage collect data structures, divide the cells to be scanned into chains. When a worker's amount of work reaches certain threshold, the worker set it's *load flag* to allow other workers to disturb the worker. A worker that

runs out of work can steel pointer to chains from other workers that have their load flag set.

When all workers have run out of work they will traverse their reference stacks and compact their context stacks.

Strategy 3: Parallelize the Configuration Tree In this strategy the load distribution is achieved by dividing the configuration tree. The *master* worker, traverses the tree and copies the live insertion points. The master will distribute the nodes in the configuration on the tree-work stacks of all workers. The nodes are thus evenly distributed among the workers. While the master traverser the tree the slaves are free to garbage collect their argument registers.

When the master has completed the traversal it will signal to other workers that they are free to start the garbage collection of nodes pushed on their tree-stack. When a worker runs out of work it becomes idle, we have not implemented any dynamic load balancing although this is quite possible. In the last phase the reference stacks and context stacks are handled.

Strategy 4: Full Parallel Garbage Collector This scheme is a combination of strategy 2 and strategy 3. Strategy 3 is used to divide the configuration among workers but once workers are idle they are free to steel work from the other workers chain-stacks. Strategy 2 is thus responsible for the dynamic load balancing.

6.2 What does it cost

In a system garbage collection can be performed either sequentially or in parallel. Performing garbage collection sequentially does not achieve the property of using the available architecture efficiently since workers will be idle during the garbage collection. A parallel implementation does, however, have a price. Barriers and locks adds to the price but there is normally a price to pay even for distributing the work among processors.

When the efficiency of parallel system is evaluated it is important to get an estimation of how big the price is before the benefits are evaluated. In the this section the overhead for maintaining locks and chains are evaluated. We will also discuss how speedup is measured and give a evaluation of the presented garbage collection schemes.

The overhead of locking operations is in one way easy to measure; execute the parallel system on one worker with and without the instructions that implements the locks. This is, however, not a very interesting figure on a modern cache based architecture. The locking operation is done in one atomic swap instruction and since the memory cell that represents the lock will most certainly be in the cache the extra instruction induces a very small overhead. In the parallel implementation only 2% of the garbage collection time can be traceable to the lock operations.

A price is also paid when the cells are divided into chains. This is however something that we can controll. We experimented with different lengths on he chains to get an estimate off the cost and benefit of dividing the cells into chains.

Figure 11 shows the effect in garbage collection time for two programs: "matrix", which have many large data structures, and "life", which have a few larg data structures. Strategy 4 was used and all executions were made with a fixed heap size.

In the matrix program we see that the cost of dividing the cells into chains increase as the length of the chains decreases. When running the program without dividing the cells at all the garbage collection time was 143 ms. for one worker. Setting the length to 30 induces only a small overhead while a length of 5 costs almost 10%. A smaller length increases the parallelism but there is also a penalty since idle workers have to interrupt busy worker more often to steal work.

The life program is, as can be clearly seen, quite indifferent to the length of the chains. Most structures are small and are never divided, there is only one large list in the configuration, the decrease in garbage collection time is almost entirely due to the effects of dividing the nodes of the configuration among workers.

There is of course no best length, since the best value depends on the number of workers and the program being execute, but a length of 20 cells have been chosen as a default value.

6.3 The Method of Calculating Garbage Collection Speedups

The task of garbage collector in the parallel systems can be performed either sequentially or in parallel. The execution state (data structures subjected to garbage collection) can increase as the number of workers increase. The sequential garbage collection time, (i.e., the time taken by one worker to perform the entire garbage collection task) can increase with increasing number

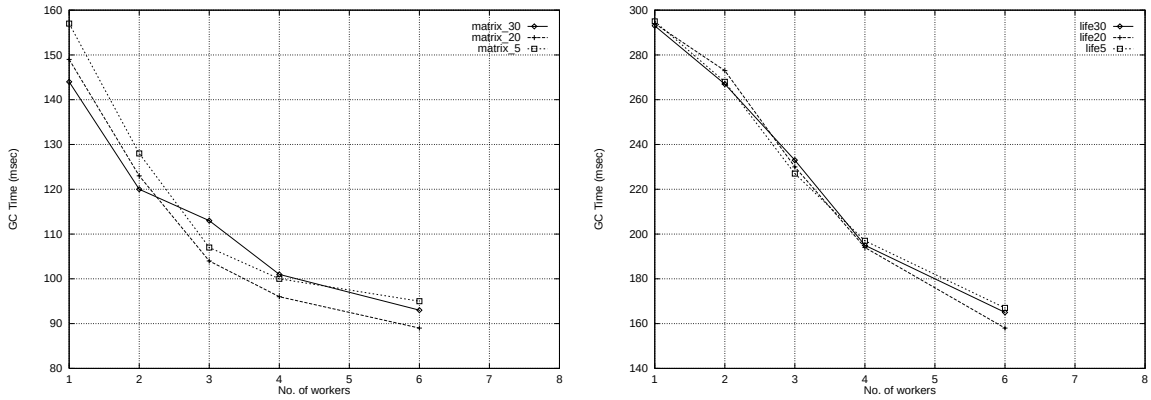

Figure 11: Effect of changing chain length.

of workers.

Imai and Tick [3] calculate GC speedup as following:

$$workload(W_i) = number\ of\ cells\ copied\ + number\ of\ cells\ scanned$$

$$speedup = \frac{\Sigma_{0 \leq i < n} workload(W_i)}{max_{0 \leq i < n}(workload(W_i))}$$

The method of Imai and Tick [3] assume an ideal scheduler for distributing and processing work (without any overheads). However, in a parallel system it is difficult to achieve an ideal scheduler. The speedup figures in [3] are the expected figures and not the measured figures from a parallel system.

Our method for calculating speedups is as follows:

$$speedup = \frac{sgctime}{pgctime}$$

where *sgctime* is the garbage collection time taken by the sequential collector and *pgctime* is the garbage collection time taken by the parallel collector.

6.4 Evaluation of the Different Strategies

Three simple benchmarks where chosen for the evaluation: "matrix", naive reverse" and "life". The benchmarks are quite extreme in their structure.

Matrix is a simple matrix and vector multiplication. In the benchmark the number of nodes in the configuration is proportional to the number of

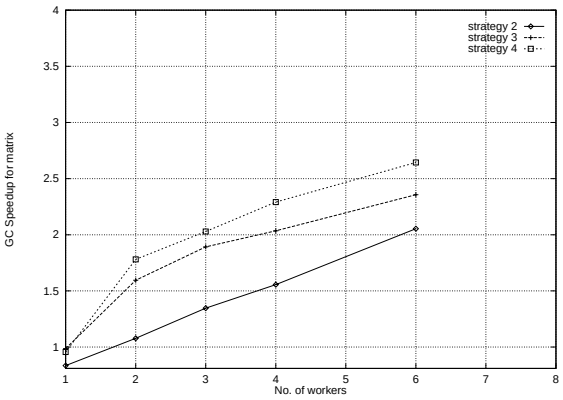

Figure 12: GC speedups for matrix.

workers. It has large data structures (the matrixes) that are ideal for breaking up in to smaller chains. Each garbage collection only takes about 10 milliseconds so speed up is very hard to achieve.

Naive reverse needs no presentation. In the benchmark the number of nodes quickly increases when more workers are added. The benchmark has some large some data structures but not as many as the matrix benchmark. Each garbage collection takes about 40 milliseconds.

Life is the game of life were each cell is represented by a goal. In the benchmark the number of nodes is fairly constant (but large) regardless of how many workers that are used. The data structures are small and mainly consists of lists of one or two elements, only one large list exists. Each garbage collection takes about 300 milliseconds.

To make a reasonable comparison between multiprocessor and single processor performance each program was executed with the same total heap space. This is not an obvious choice since a system with more processors will probably also have more memory available, but to evaluate the different strategies a fixed heap size is used.

Figures 12, 13 and 14 show the speedup obtained with strategies 2, 3, and 4.

Strategy 2: Parallelize the scanning of cells The matrix program shows a reasonable speedup using this strategy. The naive reverse program

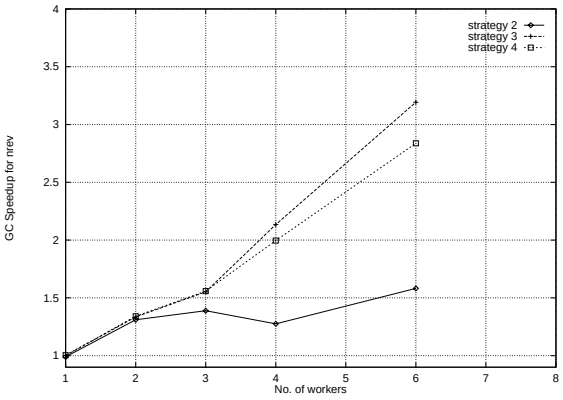

Figure 13: GC speedups for naive reverse.

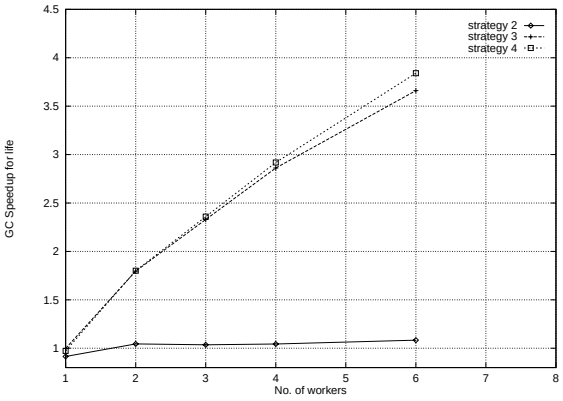

Figure 14: GC speedup for life

does show a speedup but not as good as for the matrix program, but the life program shows almost no speed up at all. This was expected since only one data structure in the life program is divided up into chains. Moreover, this data structure is a flat list and the dominating time is spent by the worker that traverses the list and divides it in to chans.

Strategy 3: Parallelize the configuration tree This strategy shows better speedup than strategy 2. The initial load balancing works well even for naive reverse which has a very un-regular configuration.

Strategy 4: Full parallel garbage collection The combined strategies give an increased speedup for all the programs. Even for the life program where strategy 2 almost gave no speedup is the combined strategy an improvement compared to strategy 3.

The evaluation shows that the combined strategy works well and better than either of the two strategies alone. Of strategy 2 and 3 the latter seams to be the most important but it is of course easy to construct a program where the reversed is true.

If only one was to be implemented strategy 3 would be the correct choice. Most configurations grow when the number of workers increase and thus guarantees that enough parallelism can be found. Strategy 3 is also easier to implement and does not induce an overhead at all since the configuration is easy to divide. Strategy 3 could of course be extended to allow dynamic balancing of nodes. This could increase the speedup even more.

Strategy 2 has a deficiency in its handling of flat lists. Even though the list is divided into chains the time it takes to traverse and copy the skeleton of the list totally dominates the total garbage collection time of the list. This is quite noticeable in programs where a single list constitutes a major part of the living data structures. This is not uncommon and a problem that must be addressed in future implementations.

One might be disappointed on the overall speedup obtained but the figures are acceptable. In the matrix program the total garbage collection time running strategy 4 with six workers was 115 ms. There were 28 garbage collections which gives an average of 4 ms. per collection. The garbage collection time is thus very sensitive for the slightest unbalance in the distribution of work. The time it takes the master to traverse the tree and for workers to synchronize also becomes more dominating.

It is worth noticing that the speedup of the garbage collector need only be as good as the speedup of the runtime excluding garbage collection. A better figure does of course not make any harm but it does not really contribute to the overall speedup. If, on the other hand, the speed up of the garbage collector is worse then the speedup of the remaining computation we will run into trouble. The garbage collection time for larger programs is roughly 5% to 10% of the total execution time. If we don't manage to reduce the garbage collection time but reduce the remaining computation with a factor four the total speedup, given a 10% garabage colection time, is only a factor three. The garbage collection time in the naive revere and the life programs show a speedup that is quite comparable with the overall speedup.

The main cause of reduced speedup arises as a result of many things in a real system. For instance, unbalanced distribution of work, locking overhead, synchronization but most important the decreased cache performance. The next section introduces a new memory management scheme that improves the cache performance of the system and also makes the update of context stacks unnecessary.

7 New Memory Management in AKL

One deficiency in the presented implementation is that nodes of the configuration are allocated on the heap. This means that nodes must be copied in each garbage collection. This is can be avoided if we are able to explicitly reclaim nodes and allocate them in a free-list separated from the heap.

Another draw-back of the implementation is that the garbage collector will "destroy" the data in the caches of the processors. A worker will during execution fill its cache with data that is relevant for the tasks of the worker, after a garbage collection the data will be more or less randomly distributed between workers. The workers must then invalidate the contents of the caches of other workers before it can write to structure copied by other workers. This can not completely be avoided but must be reduced as far as possible.

7.1 Reclaiming Nodes

When a node is reclaimed we must make sure that it is not referenced from any part of the configuration. In order to do this we must protect nodes from direct references from living data structures.

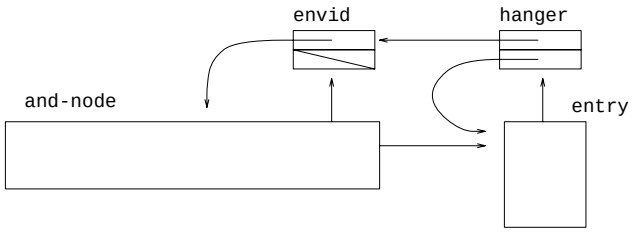

Figure 15: The new representation of an and-node

And-continuations and choice-nodes do not constitute a problem since these nodes are never referred to from any place in the configuration or any context stack once they are removed. And-nodes do however introduce a problem, these nodes can still be referred to from variables even when they are removed. An and-node is referenced (apart from the list of nodes) from variables that are local to the and-node and entries of the and-node. The entries of the and-node will not survive if the and-node is removed but variables will survive if the and-node is promoted. The variables of a promoted and-node is of course not interested in the and-node it self it only needs the forward pointer to determine its new home. The entries of an and-node are referenced from suspensions hanging both on variables and from other entries. If an entry is reclaimed it must also be protected from pointers.

To solve this problem we extract the forward pointer from the and-node in a structure by its own. Each entry is also given a "hanger" that will protect it from multiple references. The representation of an and-node and a unifier entry is shown in Figure 15, the exact representation and handling of and-nodes is given in [15].

The new memory management scheme reduced the job of the garbage collector to copy living data structures (not the whole configuration) and update the configuration so it refers to the the new copies. The garbage collection time is thus reduced but the total runtime is reduced even more. This is a consequence of better cache performance during execution time.

7.2 Cache Performance

To improve cache performance we make use of the property that a worker will probably use and-continuations that are pointed to by its continuation tasks. The garbage collection algorithm is changed so that and-continuations are

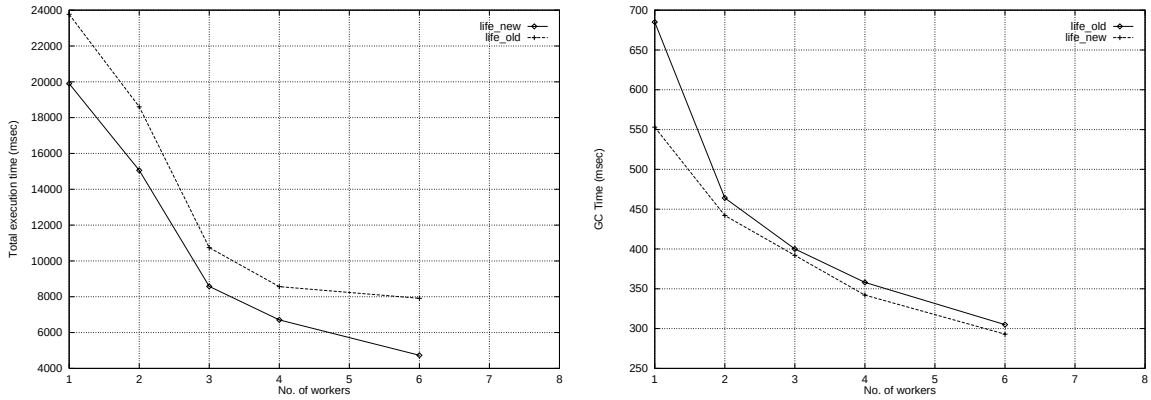

Figure 16: Total execution time and garbage collection time of the two systems.

garbage collected by the worker that has a pointer to it on its stack. Notice that all and-continuations are accessible from the stack of one of the workers, there will not be any un-collected and-continuations in the configuration after a garbage collection.

7.3 Preliminary Results

The new memory management scheme improve the total execution time and garbage collection time for the matrix, naive reverse and life programs. The total execution time (including garbage collection) and the garbage collection time for a life program is shown in Figure 16. The total execution time in the new system compared with the old one is improved. The improved execution time is not only due to the reduced garbage collection time but also due to the improved cache performance during execution.

8 Conclusions

We have presented memory management and garbage collection in the parallel AKL implementation. Two memory management systems have been designed and implementated. Different strategies for parallelizing the garbage collection process have been investegated and their performance results have been discussed. We have obtained reasonable speedups for a number of programs by exploiting two sources of parallelizem in the garbage collection

process.

References

[1] K. M. Ali. A parallel copying garbage collection scheme for shared memory multiprocessors. *New Generation Computing*, 14(3), August 1995.

[2] K. M. Ali. A parallel Real-time garbage collection scheme for shared memory multiprocessors.¡ Submitted for publication in *IEEE Transactions on Parallel and Distributed System*, Febraury 1995.

[3] Akira Imai, and Even Tick. Evaluation of parallel Copying Garbage Collection on a Shared-Memory Multiprocessor. *IEEE Transactions on Parallel and Distributed Computing*, 4(9): 1030–1040, September 1993.

[4] Henry. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4): 280-294, 1978.

[5] Björn Danielsson, Sverker Janson, Johan Montelius, and Seif Haridi. Design of a Sequential Prototype Implementation of the Andorra Kernel Language, DRAFT, May 1994.

[6] C. J. Cheney. A nonrecursive list compacting algorithm. *communications of the ACM*, 13(11): 677–678, November 1970.

[7] George E. Collins. A method for overlapping and erasure of lists. *communications of the ACM*, 2(12): 655–657, December 1960

[8] J. R. Ellis, K. Li, and A. W. Appel. Real-time Concurrent Collection on Stock Multiprocessors. *SIGPLAN' 88 Conference on Programming Language Design and Implementation*, pages 11-20, June 1988. Also Digital SRC Research Report number 25.

[9] R. H. Halstead Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions Programming Languages and Systems*, 7(4): 501-538, October 1985.

[10] S. Janson. AKL a Multiparadigm Programming Language. *Uppsala Thesis in Computing Science* 19, *SICS Dissertaion Series* 14. Uppsala University, SICS, 1994.

[11] D. E. Knuth. The art of computer programming. vol. I: Fundamental algorithms, Addison-Wesley. Reading, Mass 1973.

[12] John McCarty. Recursive functions of symbolic expressions and computation by machine. *communications of the ACM*, 3(4): 184–195, April 1960.

[13] J. Montelius, and K. M. Ali. An and/or-parallel abstract machine for AKL (extended abstract). *In Technical Report* CIS-TR-94-04, *University of Oregon*, 1994.

[14] J. Montelius, G. Atlam, and H. Ueda. The Penny System. *ACCLAIM* deliverable 4.1/4 June, 1995. URL http: //www.sics.se/ps/research/penny.html.

[15] J. Montelius, and H. Ueda. The Penny Abstract Machine. *ACCLAIM* deliverable 4.1/3 June, 1995.

[16] S. C. North, and J. H. Reppy. Concurent garbage Collection on Stock Hardware. *Proceeding of the Third Conference on Functional Programming Languages and Computer Architecture*, PP. 113-133, Portland, OR, September 1987.

[17] R. R. Fenichel, and J. C. Yochelson. A Lisp garbage-collector for virtual memory computer systems. *communications of the ACM,* 12(11): 611–612, November 1969.

[18] H. Schorr, and W. M. Waite. An efficient machine-independent proceduer for garbage collection in various list structures. *Communications of the ACM*, 10(8): 501-506, August 1967.

[19] R. Sharma, and M. L. Soffa. parallel Generational garbage Collection. *OOPSLA'91*, pp. 16-32.

[20] D. H. D. Warren. An abstract prolog instruction set. *Technical Report* 309, SRI International, 1983.