On Enabling the WAM with Region Support

Henning Makholm[1] and Konstantinos Sagonas[2]

[1] DIKU, University of Copenhagen, Denmark
henning@makholm.net
[2] Computing Science Department, Uppsala University, Sweden
kostis@csd.uu.se

Abstract. Region-based memory management is an attractive alternative to garbage collection. It relies on a compile-time analysis to annotate the program with explicit allocation and deallocation instructions, where lifetimes of memory objects are grouped together in *regions*. This paper investigates how to adapt the runtime part of region-based memory management to the WAM setting. We present additions to the memory architecture and instruction set of the WAM that are necessary to implement regions. We extend an optimized WAM-based Prolog implementation with a region-based memory manager which supports backtracking with instant reclamation, and cuts. The performance of region-based execution is compared with that of the baseline garbage-collected implementation on several benchmark programs. A region-enabled WAM performs competitively and often results in time and/or space improvements.

1 Introduction

High-level languages like Prolog relieve the programmer from worrying about mundane programming details like managing the memory which is needed for a program's execution. Memory allocation happens implicitly by simply creating data structures, and deallocation is the responsibility of the runtime system. The traditional means of doing automatic memory management is *garbage collection* where decisions about what to deallocate are made at run time. Though very sophisticated schemes for collecting garbage efficiently now exist, the process is still potentially time-consuming and hard to predict. It would be desirable to move some of the workload to the compiler. Several proposals for doing *compile-time garbage collection* have been made; see e.g. [8, 7] and the references therein.

Region-based memory management [10] takes this principle to the limit. Here all deallocation points in the program are determined by a compile-time analysis, and the runtime system needs only to carry out the preselected actions. Though not all programs are well-suited to having their memory usage reasoned about statically in this way, many are. Moreover, each of the preselected actions operates on a single region and so have a bounded worst-case running time. This makes it easier to guarantee running times of real-time programs.

Region-based memory management was originally proposed for strict functional languages. Much work has been done trying to enlarge the scope of the technique to mainstream imperative languages, but so far the only work on adapting it to Prolog has been a preliminary study by the first author [6]. In this paper we take that work a big step further by adding region support to a state-of-the-art WAM-based Prolog implementation and comparing its performance with the same implementation when using a garbage collector. We find that the region-based implementation performs competitively with garbage-collected ones, and in some cases offers significantly better time and/or space behavior.

The next section briefly discusses memory management in the WAM and introduces region-based memory management in general. Section 3 introduces the flavor of our region annotations and region-enabled WAM assembler using a simple example. Section 4 introduces properties of our region model, but we do not describe in detail how to do region inference. Section 5 contains the main contribution of this paper: An abstract machine design for adding region support to the WAM. Section 6 briefly presents the current status of our implementation, and in Sect. 7 we evaluate its performance. Finally, Sect. 8 concludes.

2 Preliminaries and Related Work

2.1 The WAM: Architecture and Heap Memory Management

Due to space limitations, we assume familiarity with the WAM [11]. We depart, albeit only slightly, from the WAM instruction names and adopt the naming convention actually used in our Prolog system: Depending on their classification, variables are denoted as `t` (temporary), `p` (permanent), or `u` (unsafe). Also, instruction names are truncated. So for example, a `putpval` instruction involves a permanent variable and corresponds to WAM's `put_value` instruction.

Besides registers, the WAM memory areas consist of a stack (or stacks) where environments and choice points are maintained, the *global stack* or *heap* where lists, compound terms, and variables that outlive their activation record are stored, and the *trail* that maintains information on variables that need to be reset upon backtracking. Upon backtracking to the topmost choice point, the heap and trail segments allocated after the choice point creation can be *instantly reclaimed*. Perhaps due to this cheap reclamation of memory upon backtracking, the WAM has a reputation of being space-efficient.

However, the instant reclamation provided by the WAM is not a panacea. In reality, Prolog programs are often mostly deterministic and Prolog systems do require additional support for automatic memory management. In most implementations this support comes in the form of heap garbage collection. A lot of work has already been done in this area. An excellent account of issues in Prolog heap garbage collection can be found in [1]; a more recent one appears in [3, Section 3]. As a result, several Prolog systems do have a heap garbage collector—in fact, some of them even have more than one—and it might appear that the issue of heap memory management in the WAM has been solved in a satisfactory way. This impression is often strengthened by the effectiveness of Prolog garbage collectors; garbage collections that recover 90–99% of the heap space are not unusual. Notice however, that there is another way of interpreting this figure, namely that heap memory *allocation* in the WAM is suboptimal. Regardless of the view that one prefers, the garbage collection process penalizes a program's execution as it happens during run time rather than statically.

2.2 Region-based Memory Management

Region-based memory management was proposed by Tofte and Talpin [10] as an alternative to garbage collection for functional languages. The basic premise of this technique is that a compile-time analysis called *region inference* annotates the program with explicit instructions for allocation and deallocation of memory. These instructions utilize the *region paradigm*: Memory blocks are grouped together in *regions*. A new block can be allocated in a region at any time, but deallocation can only happen for a region in its entirety. The number of regions varies during the execution of the program and is in principle unbounded. However, the grouping-together of allocations allows a static analysis to keep the number of distinct regions it needs to reason about down to a manageable level.

Several benefits are associated with this scheme:

- During run time, no work is spent on garbage collection (not only collecting the garbage but tracing pointers to find it).
- Because the region inferencer can analyze the possible futures of the computation (whereas a garbage collector typically views the mutator as a black box), it can sometimes deallocate data that GC would consider live.
- The basic region operations can be implemented to all run in constant time—including the deallocation of a region whose size is not statically known. Because there are no GC pauses either, it is possible to reason accurately about the execution time of a region-annotated program in real-time environments.
- Region-based memory management may lead to better cache behavior than garbage collection, because it naturally reuses memory for short-lived objects in a LIFO fashion, whereas garbage collectors usually imply a round-robin usage pattern for the nursery.

There are also certain drawbacks, however. Most prominent is the fact that certain programs are not at all well-suited to static determination of object lifetimes. One such example is an interpreter, whose source code gives no information about the lifetime of the data that represent the interpreted program's data.

Another drawback of the early Tofte–Talpin proposal is that its region inference is not strong enough to handle most real programs with satisfactory results. It is based on the principle that the lifetime of each region must coincide with the evaluation of one source-level expression. In particular, any region that existed at the time of a call must be live though the execution of the entire function-call expression, so the arguments in a tail-recursive call can be deallocated only after the recursion. Several schemes for relaxing this principle have been proposed; the latest one by Henglein, Makholm, and Niss [4], henceforth referred to as the HMN model, is the basis of the region system we employ.

Regions for Prolog. In [6] (and in more detail in [5]), the first attempt to extend region-based memory management to support backtracking and cuts was made. The challenge is that Prolog's control flow makes it difficult to find meaningful places to insert explicit deallocation operations. In a program such as

```
main :- ⟨COMPUTE T SOMEHOW⟩,                                  | foo(1).
        foo(V),   % succeeds twice                            | foo(2).
        ⟨DO SOMETHING WITH T⟩, ❶ ⟨DO SOMETHING WITHOUT T⟩,     |
        bar(V).   % fails the first time we get here          | bar(2).
```

the last use of `T` is when program point ❶ is reached for the *second* time. Ideally, one would like to deallocate (the region of) `T` at that point, but in general the code executing then does not know whether it is running for the first or the second time. It will not do to postpone the deallocation until after the possible failure either, because in less contrived examples than this one it will not be apparent where in the source code the last relevant failure is.

The solution to this is that backtracking should be transparent to regions. Whenever backtracking occurs at run time, it becomes the region management library's job to restore all regions to the state they had when the choice point was created. This includes undoing allocations and region creations made after the choice point (instant reclamation for regions) and recovering regions that the program thought it had deallocated. Algorithms and data structures to do this efficiently in the presence of cuts were described in [6].

The main problem with [6], which this work remedies, is that it is not oriented towards contemporary state-of-the-art implementation models for Prolog. The preliminary performance measurements used an ad hoc Prolog compiler. How to integrate the region operations into a WAM-based Prolog implementation was not addressed. In particular, the handling of conditional bindings inside structures was incompatible with the WAM's data model. In short, although results of [6] show promise, region-based memory management in Prolog *a la* [6] requires a fundamental shift from the abstract machine for Prolog execution: an action which raises concerns (after all, memory management is just a part of a language's implementation) and therefore is a path that most Prolog implementors are probably not willing to take. We address this problem and offer an alternative to [6] which is WAM-based and imposes minimal changes to 'plain' WAM.

3 Compiling with Regions: A Step-by-Step Example

The purpose of this section is three-fold: 1) discuss how issues of region-based memory management translate to the context of WAM-based Prolog, 2) explain our implementation, and 3) introduce our design decisions which are presented in a more detailed manner in Sect. 5.

Consider the familiar naive reverse program shown below:

```
main :- nrev([1,2,3],X), write(X).
nrev([],[]).
nrev([H|T],L) :- nrev(T,V), append(V,[H],L).
append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
```

Analyzing this program to infer in which regions data should be allocated is a process that requires support from type inference and benefits from information about modes. However, note that the above program contains *no programmer-supplied annotations* *about modes and types*; it is up to the region inferencer to infer this information.[3] Such a whole program region analyzer could produce the following region- and mode-annotated Prolog program:

```
:- mode main.
main :-
  °new R2, nrev(R2·[1|R2·[2|R2·[3]]],X)°i(R2)°o(R0), write(X), °release R0.
:- mode nrev(i,o).
nrev([],[])°i(R6)°o(R0) :- °release R6, °new R0.
nrev([H|T],L)°i(R6)°o(R1) :-
  nrev(T,V)°i(R6)°o(R4), °new R1, append(V,R1·[H],L)°c(R1,R4), °release R4.
:- mode append(i,i,o).
append([],L,L)°c(R0,R4).
append([H|L1],L2,R0·[H|R0·L3])°c(R0,R4) :- append(L1,L2,L3)°c(R0,R4).
```

In fact, this is exactly the intermediate program produced by our analyzer. The region inferencer has for example inferred that the list `[1,2,3]` will live in a *new* region named R2 which is created before the call to `nrev/2` and is passed to it as an *input* (i) parameter. The result of `nrev/2` will be placed in an *output* (o) region R0 and can be *released* after the call to `write/1`. Finally, there are some regions which are *constant* (c). These are regions that the callee must not release; the caller expects them to be around even after the predicate call returns.

We now perform the following Prolog program transformation: Rather than passing the region parameters as annotations to heads and calls, we pass them as extra arguments. Also, the °new and °release annotations can be considered new Prolog builtins which are treated specially by the compiler. Finally, we also introduce a new compiler builtin called °return that gets produced whenever a °new region annotation would need to create a region that is annotated as output. For example, since the region variable R0 is annotated as output in the first clause of `nrev/2`, rather than creating a call °new(R0), we introduce a new region variable Å0 and translate the call as °new(Å0), °return(Å0,R0). The °return builtin stores the region reference from Å0 into the region variable R0 which has been already unified with the region variable in the caller. We elaborate on need for this in Sect. 5.2. Performing this program transformation results in the following Prolog program:

```
:- pragma main.
main :-
  °new(R2),'nrev/2'(R2,R0,R2·[1|R2·[2|R2·[3]]],X), write(X),°release(R0).
:- pragma 'nrev/2'(i,o,w,w).
'nrev/2'(R6,R0,[],[]) :- °release(R6), °new(Å0), °return(Å0,R0).
'nrev/2'(R6,R1,[H|T],L) :- 'nrev/2'(R6,R4,T,V), °new(Å0), °return(Å0,R1),
                           'append/3'(Å0,R4,V,Å0·[H],L), °release(R4).
:- pragma 'append/3'(c,c,w,w,w).
'append/3'(R0,R4,[],L,L).
'append/3'(R0,R4,[H|L1],L2,R0·[H|L3]) :- 'append/3'(R0,R4,L1,L2,L3).
```

Note that the above program now contains very few region annotations. Most of the information on which argument positions correspond to region variables is kept in the

[3] The only assumption that our analyzer currently makes is that the program contains a zero-arity "top-level" predicate such as `main/0` in our example.

code for main/0	*code for 'nrev/2'/4*		*code for 'append/3'/5*
allocate	switchonlist r3 L_1 L_2	L_2: *% new clause*	switchonlist r3 L_3 L_4
new_trgn r1	try 4 L_1	allocate	try 5 L_3
putlist_trgn r2 r1	trust 4 L_2	getpvar v2 r2	trust 5 L_4
bldnumcon 3		getlist r3	
bldnil	L_1: *% new clause*	unipvar v3	L_3: *% new clause*
putlist_trgn r5 r1	getnil r3	unitvar r3	getnil r3
bldnumcon 2	getnil r4	getpvar v4 r4	gettval r4 r5
bldtval r2	release_trgn r1	putpvar v5 r2	proceed
putpvar v2 r2	new_trgn r1	putpvar v6 r4	
putlist_trgn r3 r1	return_ttrgn r1 r2	call 7 'nrev/2'/4	L_4: *% new clause*
bldnumcon 1	proceed	new_trgn r1	getlist_tvar_tvar r3 r6 r3
bldtval r5		return_tprgn r1 v2	getlist_trgn r5 r1
putpvar v3 r4		putpval v5 r2	uni_tval_tvar r6 r5
call 4 'nrev/2'/4		putpval v6 r3	execute 'append/3'/5
putpval v3 r1		putlist_trgn r4 r1	
call 4 write/1		bldpval v3	
release_prgn v2		bldnil	
dealloc_proceed		putpval v4 r5	
		call 7 'append/3'/5	
		release_prgn v5	
		dealloc_proceed	

Fig. 1. Generated WAM code for the region-annotated naive reverse program.

form of automatically generated compiler pragmas (w denotes a non-region argument position; other letters denote region variables and the pragma information describes their use). This program can be seen as the intermediate code representation that a region-enabled WAM compiler uses. Compiling this program results in the WAM code shown in Fig. 1. Instructions added to the WAM are shown underlined in the figure. Notice the correspondence between region builtins and the new WAM instructions that implement their functionality. Also, note that e.g. the annotation in the second clause of `'append/3'/5`'s last argument has resulted in a getlist_trgn instruction rather than in a getlist.

4 Our Region Model

The region system for Prolog we employ is based on the HMN region model [4] which, in its original formulation, works for first-order functional programs. In the HMN model, the lifetimes of regions are asynchronous with respect to the call/return discipline of the program. Region handles (the pointers to region control blocks which are used to allocate in, or deallocate, the region) can be passed as parameters in function calls but they cannot be stored in compound terms.

In general, callers pass one or more *input regions* as extra arguments to callees; these are where the ordinary arguments have been allocated, and the callee is responsible for deallocating the region after reading the input. Conversely, the callee returns one or more *output regions* to the caller; the ordinary return value is allocated in the output regions, and the caller deallocates them when the return value has been read. Instead of deallocating its input regions, a function (or predicate in the Prolog setting) may return them as output regions, such as if the output value contains parts of the arguments.

In addition to the input and output regions, the HMN model also has a third kind of region parameters, called *constant regions*. These are passed from the caller to the callee, but the callee does *not* deallocate them; rather they can be used for allocating memory for return values. Constant regions are used when the caller needs to be able to specify a preexisting region for allocating a return value (or parts of it). Operationally, a constant region is equivalent to an input region that is always reused by the callee as an output region, but because this reuse *always* happens, the passing-back-as-output can be optimized away in practice. In the region inference process (and in the specialized type system that guarantees the safety of the region-annotated program) constant regions play a special role, but due to space limitations we do not describe those in this paper; the reader is referred to [4] instead.

A final feature of the HMN model that ought to be mentioned here is that regions are *reference counted*. A count of the number of references to each region—not the number of allocations in the region or pointers into it, but the number of pointers to the region control block which can be used to allocate more memory in it or deallocate the region—is kept at run time, and when the last reference goes away, the region is deallocated. There is no explicit deallocation primitive, but rather primitives to increase (*alias*) and decrease (*release*) the reference count.

5 WAM with Region Support

We now describe how to extend a WAM-based Prolog system to support regions.

5.1 Memory Architecture

Our basic premise is that a region-enabled WAM offers regions as an *enhancement* to the WAM heap, not as its *replacement*. This means that the memory architecture contains both a heap and a region area. As shown in Fig. 2, all other memory areas of the WAM are of course still present. Note that, like the heap and the trail, regions are also segmented according to choice points. Shaded areas denote areas which will be reclaimed on backtracking. The figure does not show the data structures used to implement backtracking of regions.

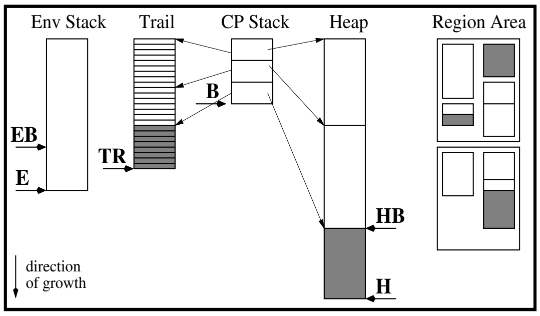

Fig. 2. Memory areas of a region-enabled WAM.

In principle, we allow data in the regions to be freely intermixed with data on the heap. A structure on the heap can reference subterms in a region, and vice versa. This design means that existing non-region-annotated code for the libraries, the compiler, and the top-level interactive loop can coexist with region-annotated code in our implementation. In a wider perspective, it also means that, given a sufficiently smart region inferencer, a program could be annotated to allocate short-lived data in regions but still be able to revert to using the garbage-collected heap where regions cannot give acceptably tight lifetimes.

Consequences of less structured memory layout. On Fig. 2 the region area is shown as being divided into two subareas. That is to indicate that the region area is not necessarily contiguous in memory: New "batches" of memory can be added to it as the need arises, without relocating the existing regions. In fact, each region does not even need to be contiguous; regions are implemented as linked lists of *cards* which might well be from different "batches".

One consequence of this is that the abstract machine cannot enforce a strict spatial relation between the different memory areas, at least not if the implementation is written in (relatively) portable C and does not do its own low-level memory allocation. It is a common optimization trick for WAM implementations to make sure that, say, the local stack is always allocated at higher addresses than the heap, so that a single test can determine whether a pointer points into the stack or the heap. With regions around, there is a third alternative, namely cells pointing into a region area. These pointers should usually be treated as those pointing into the heap, but a single comparison to determine whether they are region or heap pointers does not suffice since region areas can be located on either side of the local stack. Thus, all tests for pointing-into-the-heap in the implementation need to be updated and in a region-enabled WAM become more expensive. Such tests appear in the `uni*val` instructions and the unification subroutine. They are used to enforce the WAM invariant that there should not be any pointers from the heap to the stack. This invariant extends nicely to regions: there should not be any pointers from the region area to the local stack either.

Instant reclamation and conditional deallocation for regions. We adopt the principle of [6] that backtracking should be transparent to the region area. This means that recent allocations in the region area (including recent region creations) must be undone upon backtracking, as indicated by shaded parts of the regions on Fig. 2. Conversely, if a failing computation path releases a region, it must be kept alive and reinstated at backtracking. Techniques for how to do this were developed in [6]; they transfer essentially without change to the WAM-based environment. Unfortunately, space limitations prevent us from presenting the details, and we refer the reader to [6, 5]. We only note that the implementation of all *choice* instructions in the WAM must be extended to support backtracking of the region area. It is not sufficient to use specially enhanced choice instructions when compiling a region-annotated program.

Conditional bindings in regions. The handling of conditional variable bindings that need to be reset upon backtracking is where we deviate most from [6]. In the WAM, conditional bindings are recorded on a separate *trail* stack, which in its basic form is simply an array of addresses of cells that must be reset. Choice points contain pointers into the trail that determine which part of it is relevant in a given backtracking operation. [6] asserted (wrongly) that it would be very complicated to make a single global trail work well with regions. Instead it proposed a private trail for each region, organized as a linked list of bound variables. This required that two words be allocated for each bindable variable, which conflicts with the WAM's use of interior variables in compound terms.

In our implementation we stick to the global trail; we even intermix trail entries for variables in regions with entries for variables on the WAM stacks. The only care this requires is that we only trail bindings that really are conditional.

A binding is conditional if the variable being bound was allocated *before* the most recent choice point. The WAM mandates that this should be checked for each variable binding, which can be done with two pointer comparisons. Nevertheless, some Prolog systems prefer not to do this checking but to instead record every binding in the trail. In fact, even the trail overflow can be omitted; see [2]. This strategy is acceptable because no variable appears twice on the trail, thus the trail never grows larger than the size of the WAM heap (and stack).

When regions are present, one cannot avoid checking for conditionality anymore. Because region memory can be reused without backtracking taking place, the trail can keep growing without bounds when deterministic code runs, if it also records unconditional bindings.

Checking accurately whether a binding in a region is really conditional is complicated, but an approximation suffices: Instead of checking whether the *variable* is older than the most recent choice point, we check whether the *region card* containing the variable was added to the region before the most recent choice point. This test may give rise to a number of "false positives", but not enough to risk the trail growing unboundedly. The reason for this is that a choice point created after a card was added to the region prevents all of that card from being used more than once until backtracking occurs (or the choice point is cut away, in which case the not-conditional-anymore bindings should be purged from the trail anyway).

To make this check possible, we set aside one word in each card to hold a timestamp, using a "clock" that ticks each time a choice point is created. Given that cards are properly aligned in memory, the timestamp's address can be recovered from the address of the variable by a simple bit-masking operation.

5.2 Instruction Set

Instructions for managing regions. The new_trgn instruction creates a new region (with a reference count of 1) and puts a pointer to its region control block in a specified X register. Similarly, new_prgn creates a region and puts the handle in a specified Y register. All the region-specific instruction come in this kind of pairs; henceforth we will just present instructions as ending with _rgn and thereby understand a pair of _trgn and _prgn instructions.

The alias_rgn instruction increases a region's reference count by one, and the release_rgn decreases the reference count by one, and, if it becomes zero, "deallocates" the region. We put "deallocate" in quotes here because it may be necessary to take special measures to ensure that the region can reappear at backtracking.

The alias_rgn and release_rgn are the only ways for the reference counter to change value, except that when backtracking occurs, the region manager resets all reference counters to their previous values. Thus, the notion that the counter really counts references is just a convention, but serves as a guideline for when to use the instructions.

In practice, the alias_rgn instruction is seldom needed, but the ability to emit it is important as a fall-back strategy of the HMN region inference mechanism. The most common reason for needing it is code such as

```
main :- ⟨COMPUTE T⟩, a(T), a(T).
a(T) :- ⟨USE T⟩, ❷ ⟨DO SOMETHING THAT DOES NOT INVOLVE T⟩.
```

where a/1 can release the region for T after its last use (e.g., at program point ❷). Then main/0 can use an explicit alias to keep T alive during the entire first call to a/1, and pass the (then only) reference to the region to the last call to a/1, such that the data will be deallocated when the last call reaches point ❷.

Instructions for allocating data in regions. Next, we add instructions to allocate data in regions. The principle is that every way to allocate something on the heap should have a corresponding way to instead allocate it in a region. For example, the WAM's puttvar and putuval instructions allocate variables on the heap. Their region-allocating counterparts are puttvar_rgn and putuval_rgn.

Allocation of compound term is not so simple. In the WAM, a typical instruction sequence consists of a getstr or putstr instruction followed by a number of uni* or bld* instructions. Each of the instructions in the sequence allocates a single part of the compound term by increasing the **H** register.

In a region, this principle does not work, because subsequent single-cell allocations in a region will not necessarily be contiguous. Instead, getstr_rgn and putstr_rgn must allocate space for the entire compound term in one operation. Then the address of the argument cells must be stored in a register for the subsequent uni*/put* instructions to know where to place the arguments. Which register? The most natural choice would be the WAM's **S** register which is already used for a similar purpose in a READ-mode getstr sequence. But in our baseline implementation, the **S** register is already used in WRITE mode—namely, by being set to zero it signals that we are in WRITE mode. So we have to add a new register for the purpose of filling in fresh compound terms. We call it the **W** register.

For simplicity, and to avoid instruction set bloat, we also change the *non*-region-aware getstr and putstr to allocate the entire functor on the heap at once, and put the argument cell address in the **W** register. Then, say, getstr and getstr_rgn sequences can use the same set of uni* instructions, which we change to use the **W** register instead of **H**.

As a further optimization, we use the **S** register instead of **W** in the putstr(_rgn) and bld* instructions, where it is implicit that we are in WRITE mode. This is because on register-poor architectures, such as the x86, it may be possible to register-allocate **S** in the emulator loop, but probably not both **S** and **W**.

When (get,put)str(_rgn) now need to allocate the entire compound term, they must know how big it is. The arity of the functor can be found by looking it up in the symbol table, but we can save that memory reference by giving the number directly in the WAM instruction. Finally, the optimized list instructions (get,put)list are changed and enhanced the same way as the (get,put)str ones.

Instructions for avoiding interior variables. In the WAM, the argument fields of a structure can be free variables. This is a space-saving device for the WAM, but is not always a good thing when regions are around. The reason is sharing. To see the problem, consider this program fragment (which is somewhat idealized, but one can imagine point ❸ in the following code to also contain other predicate calls):

```
main :- blah(P,Q), ❸ ⟨COMPUTE WITH Q ONLY⟩.
blah(foo(V),bar(V)).      % ... perhaps other blah/2 clauses ...
```

The `blah/2` clause creates two structures with the `bar` structure containing a pointer to an interior variable in the `foo` structure. This means that even though `P` is a singleton variable in `main/1`—and is the only reference to the `foo` structure itself—it is *not* safe to deallocate the region containing the `foo` structure before the computation with `Q` is finished (e.g., in point ❸).

This effect means that the region inferencer must take care to keep the `foo` alive as long as the subterm of `bar` may be referenced. In practice, that need interferes with other approximations made by the region inferencer, so it often leads to large losses of space efficiency. Therefore, our design includes another possibility: The above program can be automatically annotated by our region analyzer to allocate `V` in a region different from the one containing the `foo` structure. The `blah/2` clause above can be annotated and compiled as shown below:

```
:- pragma 'blah/2'(c,c,c,w,w).
'blah/2'(R1,R2,R3,R1·foo(R2·V),R3·bar(V)).
```

```
getstr_trgn r4 r1 2 foo/1
unitvar_trgn r1 r2
getstr_trgn r5 r3 2 bar/1
unitval r1
proceed
```

Here the unitvar_trgn instruction allocates a variable in region R2 and fills in a pointer to it in the `foo` structure in R1 that is being added (aside for storing a pointer to the new variable in X register r1, as the unitvar instruction does).

Instructions for output regions. As mentioned in Sect. 4, the HMN region model requires that functions can return output regions to their callers alongside their normal return values. A decision needs to be made about how to return output regions in the WAM. At first, this may seem trivial. Prolog natively supports "returning" multiple values from a predicate by simply passing unbound variables in and instantiating them in the called predicate. Why not use this mechanism for output regions, too? One might. But doing it naïvely would be wasteful, because output regions do not need the full generality of Prolog variables. The second clause of the following intermediate code would compile to:

```
:- pragma 'foo/1'(o,w).
'foo/1'(R,a) :-
   °new(R).
'foo/1'(R,T) :-
   'foo/1'(R,V),
   T = R·f(V).
```

```
allocate 2 5
getpvar v2 r1
getpvar v3 r2
putpval v2 r1
putpvar v4 r2
call 5 'foo/1'/2
putpval v3 r1
getstr_prgn r1 v2 2 f/1
unipval v4
dealloc_proceed
```

Here the caller of `'foo/1'/2` passes a *pointer* to a free variable (presumably somewhere on the local stack) in r1. That pointer is stored at location v2 in its own stack frame before calling itself recursively. When the recursive call returns, v2 still contains a pointer to the region reference instead of the region reference itself; therefore getstr_prgn and all other region-annotated allocation instructions must be prepared to *dereference* their region parameters. WAM instructions always dereference their value arguments, but region inputs and outputs are supposed to be strongly moded: We know exactly when the region variable is bound at compile time, so why waste cycles on testing it again at run time?

For performance reasons we choose the following slightly more complicated solution. The caller still passes in an address to a free local stack word for each output region. But in the callee, this address is never used in normal region operations. Instead we introduce two new instructions `return_rgn` and `unreturn_rgn` for moving data to and from the pointed-to call, respectively. All region variables other than formal output parameters *always* contain a direct region reference (when they contain anything meaningful at all).

We can now rewrite the above example to use auxiliary variables where the original used a formal output region for a region operation. The intermediate code and the region-enabled WAM code it compiles to are shown below.

```
:- pragma 'foo/1'(o,w).
'foo/1'(R,a) :-
   °new(Å0),
   °return(Å0,R).
'foo/1'(R,T) :-
   'foo/1'(R,V),
   °unreturn(R,Å0),
   T = Å0·f(V).
```

```
    try 2 L5
    trust 2 L6

L5: % new clause
    getcon r2 a
    new_trgn r2
    return_ttrgn r2 r1
    proceed
```

```
L6: % new clause
    allocate 2 5
    getpvar v2 r1
    getpvar v3 r2
    putpval v2 r1
    putpvar v4 r2
    call 5 'foo/1'/2
    unreturn_ptrgn v2 r1
    putdval v3 r2
    getstr_trgn r2 r1 2 f/1
    unipval v4
    dealloc_proceed
```

6 Our Implementation

Our implementation is based on XXX, a WAM-based Prolog system which is a lightweight derivative of the XSB system. XXX is a full Prolog implementation, features a jump-table based bytecode emulator, and comes with both a mark-&-slide and a mark-&-copy heap garbage collector; see [3]. However, it supports tabling exclusively based on CHAT, follows some of the advice on implementing Prolog emulators given in [9] and [2], and its compiler performs instruction merging more aggressively than XSB's. As a result, XXX is a reasonably fast system: On the x86 and on the set of standard Prolog benchmarks, XXX (in the setting *without* region support) is comparable in speed to SICStus 3.8 #4.

A design decision was to do region inference as a source-to-source transformation. Our region inferencer works on a whole program at a time; modular region inference is not currently supported. The original Prolog program is transformed into a region-annotated program in the syntax of the example in Sect. 3. We use region inference algorithms from the original HMN prototype [4][4], adapted to work with Prolog input instead of an SML subset.

The second transformation, which converts region parameters to Prolog-level parameters and introduces auxiliary Å variables for output-region manipulation, is also a separate source-level operation. After that, the annotated program is compiled into bytecodes by an enhanced version of the XXX compiler which recognizes the region

[4] About half of the 12,000 lines of code in our region inferencer are from the HMN prototype.

annotations. Because of the source-level preprocesses, no drastic changes to the structure of the compiler were necessary. Our runtime system has been extended as described in Sect. 5. In addition to the 45 new instructions for regions, several changes to existing code were necessary:

- A new **W** register has been added.
- Certain pointer comparisons become more complex because the region area may be discontiguous; see Sect. 5.1.
- *Choice* and *cut* instructions must test whether it is necessary to invoke the backtracking/cut code in the region manager.
- Choice points take up four more cells than in XXX, to store administrative data for the region manager.

These changes in general make region-less programs run about 5% to 10% slower than on the original XXX emulator (although one benchmark surprisingly runs 5% *faster*).

The region manager code, based on [6], required no major changes apart from integrating it into the XXX runtime system and adapting it to WAM-like trailing. It uses a region card size of 32 words (128 bytes) of which 30 are available for allocation, and allocates cards in batches of 100 cards using `malloc()`.

7 Performance Evaluation

We conducted our experiments on a dual processor Pentium III (Coppermine) 933 MHz machine with 1 GB of RAM and 256 KB of cache running Linux.

To measure the performance of the region-enabled XXX system vs. its plain WAM variant, we used standard Prolog benchmark programs or programs previously used to measure performance of Prolog garbage collectors. In two of them, **browse** and **dnamatch**, we also slightly modified the code to be more region-friendly; the modified programs are referred to **rbrowse** and **rdnamatch** respectively.

Tables 1 and 2 contain results of our experimental evaluation. For the region-enabled system, we used two configurations: one where the region inferencer insists on using the interior variable convention of the WAM (identified with –W), and one where it does not. As mentioned, XXX features two garbage collectors: a non-segment order preserving mark-&-copy (default), and a mark-&-slide collector which traditionally is the one used in the WAM-based Prolog world. In our time comparisons, we use both settings. Times spent in GC are also shown in Table 1.

The **queens** program naturally reclaims heap space by backtracking and needs no GC. Still, the region-enabled system executes it in about the same time as the WAM, while actually requiring less memory space (data shown in Table 2).

A very bad case for regions occurs in the original **dnamatch** program. In this program, the size of the live data is quite small (climbs to ≈65,000 words and then decreases); GC manages to make this program run without the heap needing expansion. Our current region inferencer ends up placing everything in one region (cf. Table 2). As a result, it needs an enormous amount of space, and gets penalized in execution time as well. The region-friendly version (**rdnamatch**) exhibits a much better space behavior which however comes at a time cost.

Table 1. Time performance comparison (in ms). The rows labeled WAM correspond to the *unmodified* XXX system (i.e., without region support), and include GC times.

	queens	dnamatch	rdnamatch	browse	rbrowse	serial	gsort	nreverse
Regions –W	630	8260	8380	6010	5360	1540	1660	1350
Regions	630	9230	9920	5940	5410	1570	1950	1350
WAM –sliding	600	7550	7560	5330	5310	1730	2170	2130
WAM –copying	620	7290	7360	5160	5010	1630	2060	1880
gctime –sliding	0	690	730	690	1290	600	680	750
gctime –copying	0	480	540	490	890	490	550	400

Table 2. Space performance comparison (in words resp. thousands of words).

	queens	dnamatch	rdnamatch	browse	rbrowse	serial	qsort	nreverse
Regions –W	196	15,751,504	85,259	1,434,696	2,548,822	2,085 k	6,179 k	20 k
Regions	196	10,471,636	76,111	1,439,576	285,546	500 k	500 k	20 k
WAM	379	194,530	194,533	391,115	391,147	1,571 k	3,144 k	195 k
Allocations	556,324	15,751,504	15,756,504	7,744,666	8,858,792	3,028 k	6,179 k	25,015 k

The situation is somehow reversed in the **[r]browse** set: **rbrowse** runs faster than **browse** (the region-improved algorithm is inherently faster than the original), and region-based execution is competitive in time with that based on WAM. Moreover, without support for WAM's internal variables, the region-based system results in the program having significantly lower space needs.

Region-based execution is a winner for both **serial** and **qsort**: By avoiding the cost of garbage collection, the total execution times are better by 10% on average (and up to 30% if one compares the –W region system with WAM using a sliding collector). Data in Table 2 clearly show that the space-efficiency of the WAM is a myth. With the most space-efficient region annotation, a region-based system can run in three (**serial**) to six times (**qsort**) less space than the WAM. Interestingly enough, space economy does not seem to pay much in today's machines. The performance of the memory subsystem is apparently quite good in our machine: **qsort** under –W runs faster than without even though it allocates 12 times more space without ever bothering about its deallocation. However, in all machines there is clearly a limit to such space-recklessness.

Finally, we examine the familiar **nreverse** program reversing a list of 5,000 integers. This is a case where region-based execution has the advantage that the size of the used memory is never more than twice the size of the live data (i.e., is linear to the size of the input list). On the other hand, WAM-based execution requires garbage collection and this penalizes execution times. Moreover, note that region-based execution is even faster than WAM-based execution even *excluding* time for GC. This is because the region area fits within the processor's cache (whereas WAM+GC would waste large amounts of time to achieve this).

8 Concluding Remarks

This paper investigated an alternative to WAM's heap allocation- and garbage colection-based memory management: region-based memory management. In particular, we presented a complete scheme for adding region support to the WAM and reported on the performance of the resulting system. In short, our conclusion is that region-based execution is competitive with garbage collected ones, and often offers significantly better time and/or space behavior.

This does not necessarily imply that Prolog systems must abandon the WAM framework. It simply means that alternatives for memory management of logic programming languages do exist, they can be nicely and tightly integrated in a WAM (or WAM-like) environment, and their performance characteristics can be extremely attractive. We hold that because of this, these memory management schemes, and regions in particular, should be investigated further.

This paper focussed on the abstract machine extensions that regions require: memory architecture and instruction set additions. An orthogonal issue is that of the static analyses that will guide the compiler in generating these new instructions. Although our proposal is not tied to some concrete region inference method, it is clear that the effectiveness of such an analysis also depends on the characteristics of the language that is being analyzed. Adapting our region inferencer so that it is more tailored to Prolog is one direction for future work. Another, perhaps easier to pursue, is to explore region-based memory management in the context of logic programming languages where the concept of static analyses that infer modes and types is not so foreign. Languages such as Ciao Prolog, Mercury, or HAL seem particularly suited for this endeavor.

References

1. Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic memory management for sequential logic programming languages. In Y. Bekkers and J. Cohen, eds, *Proceedings of IWMM'92*, number 637 in LNCS, pages 82–102. Springer-Verlag, Sept. 1992.
2. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd *et al*, eds, *Proceedings of CL-2000*, number 1861 in LNAI, pages 1240–1254. Springer, July 2000.
3. B. Demoen and K. Sagonas. Heap memory management in Prolog with tabling: Practice and experience. *J. of Functional and Logic Programm.*, 2001(9):1–56, Oct. 2001.
4. F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of PPDP 2001*, pages 175–186. ACM Press, Sept. 2001.
5. H. Makholm. Region-based memory management in Prolog. Master's thesis, University of Copenhagen, 2000.
6. H. Makholm. A region-based memory manager for Prolog. In *Proceedings of ISMM 2000*, pages 25–34. ACM Press, 2000.
7. N. Mazur, P. Ross, G. Janssens, and M. Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury. In Codognet, ed., *Proceedings of the 17th ICLP*, number 2237 in LNCS, pages 105–119. Springer, 2001.
8. A. Mulkers, W. Winsborough, and M. Bruynooghe. Live-structure dataflow analysis for Prolog. *ACM Trans. Prog. Lang. Syst.*, 16(2):205–258, Mar. 1994.
9. V. Santos Costa. Optimising bytecode emulation for Prolog. In G. Nadathur, ed., *Proceedings of PPDP'99*, number 1702 in LNCS, pages 261–267. Springer, Sept./Oct. 1999.
10. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb. 1997.
11. D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.