Region-based Memory for CLI

Alexandru Stefan Florin Craciun Wei-Ngan Chin
Department of Computer Science, National University of Singapore
{alexandr, craciunm, chinwn}@comp.nus.edu.sg

Abstract

Region-based memory management can offer increased time performance, providing support for real-time constraints in program execution. We have implemented region-based memory support into the SSCLI 2.0 platform and also devised a region inference system for CIL programs, with the aid of newly introduced instructions. Results seem promising, as the programs running with regions have considerably smaller interrupting delays compared to those running with garbage collector.

1. Introduction

The Common Language Infrastructure (CLI) [5] is an ECMA standard that describes the core of the .NET Framework. The Microsoft *Shared Source CLI* [4] (SSCLI) is one of the implementations of the ECMA CLI standard, made publicly available for research purposes (while the Microsoft .NET Framework is a commercial implementation of CLI).

We modified Microsoft's SSCLI memory system such that its default garbage collector can co-exist with our *region-based memory system*. Our system is targeted for using a stack of lexically scoped regions in which the last region created is the first deleted. A region is used to give bounds to the lifetime of objects allocated within it. Deallocating a region deletes all its contents in one operation, resulting in better memory utilization at the cost of some predictable risks: the dangling references. A reference is said to be dangling if it points to (an object within) a region that has been deallocated. Dangling references can lead to unsafe memory access because, by having the referenced memory freed too early, new unexpected allocations could be interposed. Our solution to this problem is a no-dangling approach, preventing programs from creating dangling references at all.

We also formalised and implemented a *region inference system* for CIL (the language used in CLI). The region inference is based on our previous work [2], using mechanisms that guarantee that CIL programs never create dangling references while running on our modified SSCLI platform.

2. SSCLI and GC

The default memory system in SSCLI is a generational Garbage Collector, managing two generations of objects. This GC is moreover a copying/mark-and-sweep collector depending on the generation being inspected. There is a difference made between small objects (stored in the two generations) and large objects, considered as 'large' when having a size bigger than 85KB. The large objects are treated differently, being allocated in a *large object heap*, a special heap which is logically part of the older generation.

Any allocated object occupies memory even after becoming dead, until garbage collection takes place. Garbage collection is triggered by memory scarcity and other runtime points. The simplified GC algorithm:

- For generation 0 copy all live objects to generation 1
- For generation 1 and large heap - mark and sweep (without compaction)
- Additional cleaning-up for both generations

The GC keeps track if an object is alive by 'tracing the roots' (marking of all live objects), consisting of first finding live objects from direct references and then further searching for live objects scanning the newly found. *Copying* live objects, done only for generation 0, also includes updating references to new correct values pointing to the moved objects. *Marking* refers to tagging objects as being alive. *Sweeping* represents freeing 'dead' objects - the objects which haven't been previously marked. The mark-and-sweep is only done for generation 1 and large object heap; there is no compaction made, so objects from the old generation and large heap are never moved.

3. Our Approach

We briefly present the modifications made to the SSCLI 2.0 source code for integrating a region subsystem, then we describe the compile-time region inference able to automate program adaptation for using this subsystem.

3.1 Design Decisions

We are using the large object heap (from the existing GC) for allocating regions. This decision is accounted by the usually broad sizes of regions and the fact that regions' memory address need not change (no copying is made for the large object heap). We make use of fixed sizes for regions and region extensions - whenever filled, regions will extend with fixed size increments.

A *Region* class is now used in the code to internally represent all vital information that makes up a region (a set of pointers, similar to how Hallenberg et al. [3] represented regions):

- `p_region_start`, the region's starting address
- `p_region_end`, region ending address
- `p_new_alloc`, new allocation pointer
- `p_next`, next region extension

The first two pointers are used to simply delimit the region's contiguous memory space. Their values are obtained at the point of explicit allocation of a region inside the large heap. The third pointer is used to designate where a new object can be allocated Its value is initialized with the region's starting address added with the usual region header size. So that the regions can be expandable, the forth pointer `p_next` was introduced to link a region with its extension, which in turn can have an extension and so on.

Regions can be identified by an assigned index (or id) and thus, are conveniently placed in an array of Region objects. Whenever some operation is necessary on a particular region, the pointers found at the correct index in the array will reflect the region's state prior to that operation. To support even larger numbers of

runtime regions, the storage of Region instances is extended past the region array's size, through a linked list of region handles, being practically unbounded. Note that the Region objects, array and linked list described here are environment entities, and invisible to the executing program.

3.2 New CIL Instructions

Languages which target the CLI standard (C#, Visual Basic, managed C++ and other .NET languages) compile to CIL (Common Intermediate Language), which is assembled into bytecode. CIL resembles an object oriented assembly language, and is entirely stack-based. With the region features being encompassed into the SSCLI platform, the following new CIL instructions (opcodes) become necessary:

`letreg` - Allocates a region. Permits specifying a region index and even the initial space size (which will usually be a fixed one).

`freereg` - Deallocates a region. Requires an index, denoting which region to deallocate.

`newrobj` - Create an object inside a region.

`newrarr` - Create an array object (arrays are objects in .NET) inside a region.

For ensuring harmony of these features with the GC, we use some of GC's locks for memory operations. Also, we are enforcing:

- regions are never garbage collected

- objects in regions are part of reference tracing but never part of sweeping

The programmer can either manually intervene at CIL level, inserting region control in what areas of his program he deems necessary, or as a second option, the region instructions can be inserted automatically by a compile-time region inference.

3.3 Region Inference

Region inference translates a standard CIL program into a CIL program with regions. The main algorithm was adapted from [2]. However, CIL is a stack-based language, therefore we had to modify the original flow-insensitive region inference to trace the type of the stack operands in a flow-sensitive manner. As first step of the inference, every object (variable) in the program code is parameterized with region variables. Then region lifetime constraints are gathered for each class and method. These constraints express the order in which regions should outlive each other - the aim is simply to forbid dangling references from being created. After all the constraints are known for each of the program's logical blocks, the region variables can either escape a block or be localized in it. Former situation implies that the objects going to a certain region (denoted through their parameterization) are still being referenced after the analyzed block ends, so that region needs to live longer than the referencers (the region escapes). All regions found in a block that are non-escaping are considered localized regions, meaning they will be discarded as soon as the block ends. The final steps of the inference are inserting additional parameters in the method signatures to capture region handles and also insert region instructions for region allocation/discarding (for the localized regions) and region-contained object allocations.

The inference is also handling features like method recursion, method overriding, static fields and others. Finally, we identified some optimizations regarding the management of regions:

- region resetting - prevents repetitive allocation/deallocation of regions; useful for loops: at the end of each cycle, we are resetting the loop-localized region
- region relegation - instead of creating a new region at a block level to allocate new objects into it, using an existing region whenever convenient (as to avoid excessive fragmentation of the memory into regions)
- forced localization - forcing a new local region for a particular sequence of instructions in order to prevent the storage of too many dead objects

4. Experimental Evaluation

Our prototype of the region inference was implemented in F# [6]. We tried mostly small-sized benchmarks (adapted from RegJava and Olden suites also used in [2]) and a tree-constructing GC test [1] for testing our CIL inference. The execution speed improvement can be observed in the tables below (timings are in milliseconds):

	Input size	GC	Regions
Eratosthenes	n=5000	703	672
	n=10000	1422	969
Ackermann	n=3*10+8	1454	1328
	n=3*10+9	3765	2422
Merge Sort	n=20000	1610	1391
	n=25000	2359	1484
Mandelbrot	n=800*400	1016	1016
	n=1000*500	1281	1281
GC Trees	fixed	7297	6313

Four out of ten programs from the Olden suite had notable speed advantage when run with regions, with this being a more general benchmark (not specifically comprising memory-intensive applications). Speed gain in these tests mostly comes from the absence of live object tracing for regions.

Olden suite	Input size	GC	R	Improv.
BH	300	8859	8688	1.9 %
	400	12141	11938	1.7 %
MST	500	5859	5469	6.7 %
	700	10641	9781	8 %
Perimeter	300	2875	2781	3.3 %
	400	8484	8031	5.3 %
TreeAdd	20	2401	1575	34.4 %
	21	5161	2136	58.6 %

The other Olden programs that have not shown significant speed advantage use simpler memory structures and have low memory demand. Nonetheless, we observed that the region programs are usually at least as fast in execution as the ones running with GC.

5. Conclusion

We built region-based memory support into the SSCLI 2.0 environment and added new instructions in the CIL opcode set for supporting region operations. We implemented a region inference system that automates the translation of initially garbage collected CLI (.NET) programs into the region-aware versions. Then we measured performance aspects of executing programs obtained with the inference. Execution of region programs exhibits a bigger speed improvement for programs using complex data structures. Because regions have bounded delays throughout execution (unlike GC's pauses), they also possess a better real-time performance.

References

[1] H. Boehm. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[2] W.N. Chin, F. Craciun, S.C. Qin, and M. Rinard. Region Inference for an Object-Oriented Language. In *ACM PLDI*, Washington, DC, 2004.

[3] N. Hallenberg, M. Elsman, and M. Tofte. Combining Region Inference and Garbage Collection. In *ACM PLDI*, Berlin, Germany, 2002.

[4] D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials*. O'Reilly, 2003.

[5] *ECMA-335 Standard: Common Language Infrastructure (CLI)*, 4th edition, 2006.

[6] F# Language. http://research.microsoft.com/fsharp/fsharp.aspx.