A Real-Time Garbage Collector for Embedded Applications in CLI

Okehee Goh and Yann-Hang Lee
Computer Science and Engineering Department
Arizona State University
Tempe, AZ 85287

Ziad Kaakani and Elliott Rachlin
Honeywell International Inc.
Phoenix, AZ

Abstract — *We are working on scheduling of garbage collector as a concurrent thread for time-constrained applications in Common Language Infrastructure (CLI). We have implemented an incremental garbage collector with fine-grained write barrier in MONO, an open-source implementation of CLI. Our collector is based on existing conservative garbage collector of Boehm et al. By conducting benchmarking experiment, we will derive parameters to predict the behavior and overhead of garbage collection and apply real-time scheduling algorithms that guarantee the timeliness of applications without memory starvation.*

Index Terms — *Common language infrastructure (CLI), Incremental garbage collection, Real-time scheduling.*

INTRODUCTION

Common Language Infrastructure (CLI), introduced as a core technology of Microsoft .NET and standardized by the international standardization organization ECMA in 2002, provides a virtual execution system (VES) that supports multiple-languages as well as multiple platforms. The machine-independent intermediate code that "write once, and run everywhere," contributes to expedite the evolution of this VM technology by minimizing cost and time-to-market of application development. However, real-time embedded systems that require timeliness response do not get benefits using CLI as available implementations of CLI are not designed for timeliness response.

A real-time embedded CLI environment can enormously expand the applicability of CLI, not only for consumer electronics devices but also the embedded systems for home appliances, telecommunication, and industry automation. Among VES's features such as thread scheduling, exception handling etc., we focus on a garbage collector to make it deterministic. Our ultimate goal is to schedule a garbage collector to ensure applications to meet their deadline and satisfy memory requests in CLI. Scheduling a garbage collector primarily requires two conditions. Firstly, the activity of garbage collection (GC) must be controllable as a schedulable unit and should not result in a long pause time. Secondly, the parameters to predict the behavior of a garbage collector must be derived. These parameters such as the execution time of garbage collection, overhead due to incremental GC, amount of reclaimed memory etc. are a basis of scheduling of GC.

We have implemented an incremental garbage collector in CLI, running concurrently to be applied for time-based or work-based scheduling. This WIP report covers the design of the incremental garbage collector. Currently, we are gathering experiment data and developing scheduling algorithms of garbage collection that guarantee time and space bound.

COMMON LANGUAGE INFRASTRUCTURE (CLI)

CLI is aimed to make it easy to write components and applications with multiple languages and for multiple platforms[1]. It is enabled by defining a rich set of types to allow CLI-aware languages to interoperate, having each component carry self-describing information, translating applications into intermediate language codes, and providing virtual execution system (VES) executing intermediate language codes (CIL).

VES, similar to JVM, is an abstract stack-based machine featuring loader, verifier, JIT compiler, garbage collector, security system, multiple threads, exception handling mechanism etc. The ECMA standard for CLI does not confine a specific garbage collection mechanism on VES so that the implementation of a garbage collector does not have any limitation except awareness of "pinned" type signature. CLI defines unmanaged pointer type that is not traced by a garbage collector. A memory object referenced by unmanaged pointer must be "pinned" to prevent a garbage collector from moving the object. Unmanaged pointer referring to managed heap object can happen when a managed object is passed to managed code that operates with unmanaged code. However supporting this feature does not affect the design of a deterministic garbage collector.

As well as commercial products Microsoft .NET and WinCE .NET, SSCLI by Microsoft, MONO by Ximian/Novell, and DotGNU Portable .NET are available open-source implementations of CLI. C#, C++, VB, JavaScript, and Java are available as CLI-compatible languages. So far, there is no implementation aiming to support time-constrained embedded applications. Considering the benefits of using CLI in embedded applications, we believe that this need will grow soon.

GARBAGE COLLECTOR IN MONO

We have worked on the garbage collector by Boehm et al. (BDW)[2] that is associated in MONO[3]. BDW, designed to support languages such as C, C++ etc., is a conservative mark-sweep GC without cooperation of a compiler or a runtime system. In order to reduce a pause time due to GC, BDW supports a mostly parallel GC (partly incremental), and a parallel but not incremental collector for multiprocessor systems.

In mostly parallel GC, GC is triggered per allocation basis and write barrier tracing mutators' heap pointer updates is performed by using dirty pages through virtual memory protection mechanism of operating system. The write barrier algorithm is that all heap pages are protected as read-only at an initial step of mark phase when incremental garbage collection starts and the pages flagged as dirty due to mutators' write operations are re-scanned by a garbage collector in a termination step of mark phase. The advantages using virtual memory protection is that firstly, it does not require a compiler to emit write barrier and secondly, there is no overhead placed on mutators due to write barrier.

However, this design imposes limitation on schedulable GC. Firstly, triggering garbage collection per allocation makes collection work dependent on allocation patterns of applications. Secondly, using virtual memory protection for write barrier is a system dependent feature that limits portability of CLI. Finally, in the mark phase, an initial step to protect all heap pages and a termination step to rescan both dirty page set and root-set may lead to long pause time.

DESIGN OF INCREMENTAL GC

We extended BDW and JIT compiler in MONO to address the problems described above. Firstly, the collector is concurrent for either time-based or work-based scheduling. Secondly, an incremental GC is implemented by having JIT emit write barrier.

Concurrent Garbage Collector

A traditional incremental GC that performs collection work at each allocation does not guarantee consistent collection due to bursty allocation which is general characteristic of applications [4]. A concurrent garbage collector, invoked either at the fixed time or at the threshold of free memory, allows us to apply real-time scheduling algorithms. The GC invocation interval and the pause time in each collection cycle, as shown in Figure 1, is controllable based on applications and platform requirements.

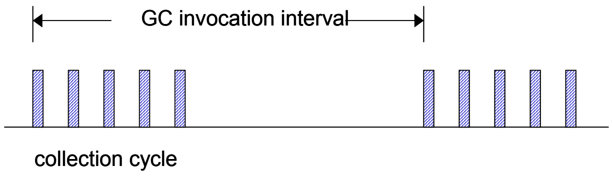

Figure 1. GC invocation and scheduling

Incremental Garbage Collector Using Fine-Grained Write Barrier

We extended BDW GC to perform root-set scan, mark and sweep incrementally. The write barrier used to trap mutators' heap pointer updates is Yuasa's snapshot-at-the-beginning algorithm [5].

- Write Barrier: MONO includes JIT compiler that translates intermediate language codes (CIL) into native codes. Among about 220 CIL instructions, CIL instructions that need write barrier due to their store operation are in Table 1. JIT was modified to generate the native codes for the instructions that include an internal call performing write barrier in MONO runtime system. Applying an indirect function pointer for the internal calls helps avoid comparison of GC phase each time. Write barrier in a root-set scan phase is extended from snapshot-at-the-beginning algorithm.
- New objects allocated in GC cycle: All objects allocated during GC cycle are marked live to ease a termination condition and help avoiding a long-pause in a termination step of mark phase.

CIL	Descriptions
Stind.ref	Store an object reference into the memor
Stfld	Store a value into a field of an object
Stsfld	Store a value into a static field of class
Stelem.ref	Store a value into a vector element

Table 1 CIL codes that require write barrier in GC

STATUS, FUTURE WORK AND CONCLUSION

Currently, we have finished the implementation of the incremental garbage collector and are gathering experiment data. Based on the experiment, we will devise efficient scheduling algorithms which guarantee time and space bounds. At this level, the conservatism of locating heap pointers and deciding liveness of pointers in a root-set, and memory fragmentation are not addressed. However, our incremental garbage collector can establish an experiment environment to schedule real-time applications in CLI.

REFERENCES

[1] Common Language Infrastructure, ECMA TC39/TG3, Oct, 2002

[2] Hans-Juergen Boehm et al. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157-164, 1991.

[3] Ximian/Novell, MONO, http://www.go-mono.com

[4] David F. Bacon, Perry Cheng, V. T. Rajan: A real-time garbage collector with low overhead and consistent utilization. POPL 2003

[5] Richard Jones, Garbage Collection: algorithms for automatic dynamic memory management, John Wiley & Sons, Ltd, 1999.