Vladimir A. Livshits
livshits@cs.cornell.edu

10/7/99

MCC for Java

The purpose of this document is to describe the steps I took in porting MCC from ML to Java and outline some ways to make to collector faster up to the level of being competitive with the other two collectors that come with Marmot. It's assumed that the reader is familiar with the two papers on MCC and the overview paper on Marmot.

Changes to the collector

File organization.

MCC is integrated with the directory organization of Marmot. The runtime is located under `marmot\scaffold` and this is also where all MCC files are located. The core of the collector consists of the following C files: `mccgc.c, collect.c, conservative.c, debug.c, deque.c, external.c, heap.c, implicitQueue.c, inline.c, machine.c, object.c, page.c`. All these files were present in the original version of MCC except for `mccgc.c`, which is modeled after `copygc.cpp`. This files implements some native functions the system expects, but most of the time it just calls some of the functions in the collector such as `collect()`. The following header files are present: `dataStructures.h, pageHeader.h, prototypes.h, machine.h, object.h, header.h, debug.h, marmot.h, mcc.h`. The last two files are new to the Marmot version of MCC: `marmot.h` declares some Marmot types needed in the collector. `mcc.h` includes the most common MCC headers.

To compile a Java program with MCC, one can use the `makefile` in `scaffold` like this:

```
nmake gc=mc HiMom.exe
```

That builds the runtime system (look for `JRT_OBJ` in `makefile`) and links it with MCC files (`MCC_OBJ` in `makefile`). Note: when you are switching between collectors, you need to recompile the Java source code, otherwise, you'll get a runtime exception when trying to run the executable the system produces. It's important that you use the same set of compile flags when compiling MCC and the rest of the system, otherwise, strange things may occur at runtime (Marmot uses the `__fastcall` calling convention). There were several preprocessor flags I added in addition to the many flags MCC already had:

Flag	Value	What it does
`CONSERVATIVE_HEAP`	`Off`	Controls whether the collector supports conservative objects in the heap, i.e. ones that don't have *vtable* pointers. This is tuned off for Marmot, since all Java objects have *vtable* pointers as their first field. Turning it off also greatly reduces the amount of source code that has to be lined with the runtime system.
`MCC_COMMIT`	`Off`	This controls how the allocation works. If this flag is on, when each a block is allocated, it's immediately committed. If not, it's only reserved and whenever a page is requested by the system using the functions in `page.c`.
`MCC_TRIGGER`	`Off`	If this flag is on, the collector uses the MCC triggering condition, that is start collecting when 2/3 of pages is used. When it's on, MCC uses the same condition used by the other Marmot collectors, it starts collecting when the amount of data allocated since the

		last collection exceeds a certain boundary.
MCC_ALLOC	On	If this flag is on, MCC uses its own allocation algorithm, otherwise malloc is used. In conjunction with the next flag, this is useful for debugging to figure out where the problem lies.
MCC_COLLECT	On	Perform garbage collection, turn it off for debugging.
INITIAL_ALLOC	On	Allocate several blocks in the beginning of the program.
SIZE_PROCEDURES	On	Use procedures in mccgc.cpp for computing the size of objects instead of macros in macros.h. Useful for debugging.
WASTED_SPACE	On	Fill wasted space. Useful for debugging.
PAGE_HINT	On	Supply a hint to reservePage.

Some other MCC flags:
HARD_LIMIT, FAST_CLOCK LOGGING, INLINE_INNER_LOOP, LOOK_FOR_GOOD_STACK, LOGGING, FAST_CLOCK – these were in MCC from the beginning. grep the code to find out what they are. Also Marmot's GC_DEBUG_PRINTS and YADDA_YADDA are useful.

Object headers
Every object in Marmot has the following layout at runtime:

vtable
monitor
fields
…

The first filed points to a vtable located in the static area. The vtable contains f_baseLength and pointerTrackingCrap among other things. The first is the total size of the object for non-array objects. There are macros for retrieving objects sizes in object.h. The second field contains information as to which fields are pointer fields. object.c parses this information. Most of the parsing code is taken from copygc.cpp.

MCC supported manipulation of both typed objects and conservative objects (that could be created using routines in header.h). These routines are not used in the Java version and are guarded by #ifdef/#endif CONSERVATIVE_HEAP. Most macros in object.h also had to be changed to use the layout used by Marmot.

Stack Layout
Of the two collectors that come with Marmot, copygc uses stack and static area annotation to figure out what on the stack is a pointer. Since we wanted to experiment with conservative GC, we didn't use them. (It's possible to turn generation of these tables, which are not very big, off in marmot/utility/StageControl.java). To see how these tables can be used, refer to copygc.cpp. Therefore, we only need to find out the boundaries of the static area and the thread stack for each thread to perform conservative scanning. We can do this because pointers in Marmot are word-aligned. The scanning code is in conservative.c. The registers are not pushed on the stack before the collection unlike the original version of MCC, because Marmot doesn't store live pointers in the registers between calls. The current version doesn't support multiple threads, but this should be relatively easy to add. When Marmot compiles a program, it creates marmotsez.h, which defines SINGLE_THREADED. This is how the runtime system knows whether to include thread support or not.

What has been tested.

The collector has been heavily tested with micro- and macro-benchmark. One criterion for correctness is running the TreeHeapWalk benchmark, which stresses GC a lot.

What hasn't been tested.
As mentioned above, multi-threaded programs haven't been tested. I tried to run MCC with some of the benchmarks in JVM98 SPEC suite, but I couldn't compile those programs with Marmot. Generations and page blacklisting hasn't been tested, either.

Performance results
These are some preliminary results on a couple of programs.

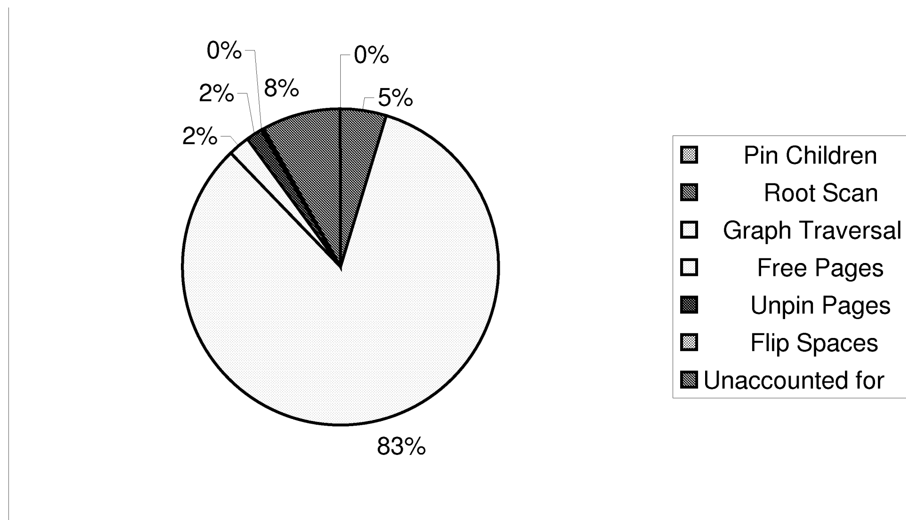
	HiMom.exe		**JLex100.exe**	
	RT, sec	GC, sec	RT, sec	GC, sec
M&S	0.55	0.07	41.3	2.7
MCC	0.66	0.44	39.76	3.5
copygc	0.2	0.47	43.4	1.7

Optimizing the program using MSVC profiler made the collector significantly faster. There's a batch file to run the profiles in scaffold called prof.bat. Just run is with the name of the program and its parameters, it creates an output file by appending the last parameter to the name of the program, i.e.,

```
prof himom 15 DFS 5 4
```

will create `himom.4.txt` with the profiling result. Most of the time seems to be spent in processObject, processChild, and iqEnqueue. Some of these can probably recoded in assembly to make them faster. Another source of optimizations is the fast path of the allocation routine, which is pretty slow now.

The feel is that with some amount of work it's possible to make MCC competitive with BDW, if the latter is used with Marmot. It's hard to expect MCC to be consistently better than a much smaller and probably hand-optimized `copygc`, for example.



Distribution of times among different stages of collection for a sample collection cycle.