Realtime Java: A Real Alternative to C and C++ for Programming Embedded Systems

While cloud and desktop computing has moved on, most embedded systems still are programmed in C and C++. There is some use of Java, Java Script, Scala, Python, and other managed languages, but their penetration is small. Managed languages have contributed to a drastic increase in productivity for cloud systems. So why are managed languages not more prevalent in embedded systems?

Better Safety and Security with Managed Languages

Managed languages have been successful for cloud systems because they make the programmer's job easier. Managing the lifespan of objects is time consuming and failure prone. Memory can be lost (leaked) by not releasing it before it becomes unreachable. Worse, memory that is still in use can be deallocated and then allocated for another use, which can cause errors in other, unrelated parts of the system. This nonlocal effect makes debugging difficult and time consuming, since the cause is unclear at the point of failure. Such errors are often not found during normal testing and show up randomly in the field.

Managed languages do not allow the programmer to explicitly deallocate objects; instead, an automatic process (garbage collection) is employed to collect all unreachable memory (garbage) and return it to the free list. This means that live memory is never collected and unreachable memory does not remain unreachable. The developer does not have to debug deallocation errors because they cannot happen.

There is an additional benefit to garbage collection in managed languages that most people do not realize. A garbage collector must know precisely which memory elements hold pointers and which do not, and where the boundaries of each structure are. This means that a programmer must neither be allowed to create a pointer, nor to write over the end of a structure. Bounds checks and safe casting must be implemented in the language. These guarantees make programming in managed languages more robust.

But wait, aren't there garbage collectors for C++? Well there are, but adding such a garbage collector in C++ does not make it a managed language. We need to distinguish between an exact garbage collector, which has full knowledge of where pointers are in the system, and a conservative garbage collector. A managed language must have an exact garbage collector and must give all the aforementioned guarantees. A conservative collector must guess what a pointer is. It must assume that every memory cell is a pointer, even if it might be data, and handle it as such. This means that it can keep memory alive that is no longer reachable. Determining the size of objects is not trivial either and also error prone. For this reason, they are not used much in practice.

There are safety standards for C and C++ that disallow the use of manual heap management (malloc and free). All allocation must be made at the beginning of the program and then that memory may not be freed. This limits the complexity of what one can do. To get around these limitations, such programs often use object pools to manage memory. Unfortunately, this does not improve the case much. Table 1, taken from DO-332, illustrates this point.

DO-332 is a supplement to the software certification standards DO-178C and DO-278A for object-oriented technology and related techniques and shows which safety objective must be met for each application or can be done once for the memory management infrastructure. The advantage of garbage collection is that almost all objectives are certified for the garbage collector and can be reused from project to project.

Comparison of Memory Management Techniques

TECHNIQUE	OBJECTIVES						
	Unambiguous Reference	Fragment Avoidance	Timely Deallocation	Reference Consistency	Deterministic Deallocation	Atomic Move	Sufficient Memory
Object Pooling	AC	AC	AC	AC	MMI	N/A	AC
Stack Allocation	AC	MMI	MMI	AC	MMI	N/A	AC
Scope Allocation	MMI	MMI	MMI	AC	MMI	N/A	AC
Manual Heap Management	AC	??	AC	AC	MMI	MMI	AC
Garbage Collection	MMI	MMI	MMI	MMI	MMI	MMI	AC

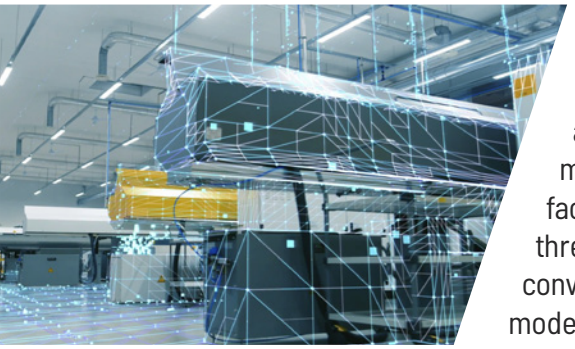
AC = application code, MMI = memory management infrastructure,
N/A = not applicable, and ?? = difficult to ensure by either AC or MMI.

Managed languages provide security advantages over C and C++. Buffer overruns are still a primary security hole in software systems. A managed language must always prevent such overruns to be able to use exact garbage collection in the first place. Both Microsoft and Google have reported that 70% of all security bugs are memory safety issues[1][2]; the very issues that managed languages prevent.

1 https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/
2 https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/

The Problem with Managed Languages

If it was all so simple, managed languages would replace unmanaged languages everywhere; however, embedded systems, i.e., operations technology (OT) systems, have some additional requirements over cloud, i.e., information technology (IT) systems. Most OT systems have some cyber-physical aspects to them. In other words, they must interact with the outside world. This means that they must react within well-defined time bounds, as well as work with more limited resources. Very few managed languages were designed to address these requirements.



Unmanaged languages tend to be simpler and only provide a minimum of functionality; whereas managed languages provide more isolation from the underlying OS to provide more consistent behavior across various platforms. For instance, C does not have a threading model. Instead, the programmer must rely on operating system interfaces to provide threads. On the other hand, the Java VM provides a full thread and memory model for programs running on it. The problem with conventional Java and most other managed languages is that its threading model was designed for throughput, not realtime behavior.

The other problem with most managed languages is the garbage collector. Typically, this runs in a separate thread and must routinely block other threads to do critical parts of its work. This results in unpredictable pauses and delays, thereby severely limiting the deadline guarantees such system can give. It even affects systems that do not strictly require realtime, such as human machine interfaces, where pauses cause annoyance and distraction for the task at hand.

Systems relying on Just-In-Time (JIT) compilation for performance, such as conventional Java, have additional problems. Firstly, each program must carry a JIT compiler with it and provide the working memory to compile code on the fly. Secondly, the program must run a while before it has compiled enough to compile most of its critical paths. Finally, a critical path may be executed infrequently, so deadline guarantees are difficult to give, even after a warmup period.

Fortunately, these difficulties are surmountable. More appropriate threading models can be provided. There are alternatives to JIT compilation. There are even alternative algorithms for garbage collection.

Realtime Java: The Best of C and Conventional Java

Realtime Java is an extension to Java that addresses the concerns of OT systems. This is accomplished by providing additional features and semantic refinements to conventional Java for embedded and realtime systems. In some sense, Realtime Java is a new language that is fully backward-compatible with conventional Java. This means that all conventional Java code runs correctly on a Realtime Java VM, but the semantics of the system are more precisely defined.

As with C, Realtime Java uses a static (Ahead of Time) compiler instead of a JIT compiler. Since not all code is time critical and Java byte code is about a factor of five more compact than machine code, Realtime Java virtual machines can also interpret code. Profiling is used to determine which parts of the application and system code should be compiled and which should not. Most of the performance gain can be achieved with compiling as little as 20% of application and system code. In fact, standard library code that is not used can be held outside the resulting executable.

As with conventional Java, a garbage collector is used to ensure heap memory consistency and hence memory safety. For realtime and embedded systems, it is essential that the garbage collector does not inhibit response time. Realtime Java implementations address this by providing a deterministic garbage collector.

Where realtime Java goes beyond both conventional Java and C, as well as most other languages with the notable exception of Ada, is by providing APIs for realtime programming. In fact, realtime Java inherits quite a bit from Ada. This includes realtime threads with first-in-first-out, run-to-completion semantics, and realtime priorities, as well as event handlers for handling timed or other asynchronous signals.

For example, doing something every 10 milliseconds is quite simple: it can be done in just a few lines of code where the system takes care of the timing. Realtime Java also provides APIs to access device registers directly from Java with type safety, so it is possible to get down to the metal without using JNI code or losing type safety.

Realtime Java brings together the advantages of C, Ada, and conventional Java without many of the problems these other languages have. If the job needs to get done quickly and robustly, realtime Java can provide the edge. It is no longer necessary to chase segmentation faults and there is a plethora of tools available to speed up system implementation.