

7 HVM Evaluation

In Section 5.3.1 we have demonstrated how the HVM can be used to add Java software components into an existing C based execution and development platform. Additionally Section 5.3.2 demonstrated the scalability of the HVM method to large SCJ applications.

In this Section we will show measurements comparing the execution efficiency of the HVM to other similar environments. Eventhough the HVM can be used to program Java for embedded systems it is also very important to engineers that the efficiency by which Java can run is close to the efficiency they are accustomed to for their current C environments.

For high-end embedded platforms we have already results regarding execution speeds of Java programs as opposed to the same program written in C: in their paper [39] the authors show that their Java-to-C AOT compiler achieves a throughput to within 40% of C code on a high-end embedded platform. This claim is thoroughly substantiated with very detailed and elaborate measurements using the CDj and CDc benchmarks[26].

Since the memory requirements of the CDj and CDc benchmarks (see Section 5.3.2) prevents us from running them on low-end embedded systems we made a small range of additional benchmarks. We have been inspired by the idea from CDj/CDc of comparing a program written in Java with the same program written in C.

7.1 Method

We have written 4 benchmark programs in both C and Java. The characteristics of the programs are,

- *Small.* The benchmarks are small. They don't require much ROM nor RAM memory to run. The reason why this principle has been followed is that it increase the propability that they will run on a particular low-end embedded platform
- *Self-contained.* The benchmarks are self-contained, in that they do not require external Java nor C libraries to run. They don't even require the `java.util.*` packages. The reason is that most embedded JVMs offer their own JDKs of varying completeness, and not relying on any particular Java API will increase the chance of the benchmark running out-of-the-box on any given execution environment
- *Non-configurable.* The benchmarks are finished and ready to run as is. There is no need to configure the benchmarks or prepare them for execution on a particular platform. They are ready to run as is. This will make it easier to accurately compare the outcome from running the benchmarks on other platforms, and allow other JVM vendors to compare their results with ours

- *Simple.* The behaviour of each benchmark is simply understood by a quick scrutinization of the source code. This makes it easier to understand the outcome of running the benchmark and assess why the result is as it is

The benchmark suite of only 4 benchmarks is not complete and the quality and relevance of the suite will grow as new benchmarks are added. We consider the guiding principles of the benchmarks very important, especially the principle of being self-contained, since this in our experience is a principle most important for being successful at running a benchmark on a new embedded platform.

The current benchmarks are:

1. *Quicksort.* The **TestQuicksort** benchmark creates an array of 20 elements initialized with numbers from 0 to 20 in reverse order. Then a simple implementation of the quicksort method sorts the numbers in place. This benchmark applies recursion and frequent access to arrays
2. *TestTrie.* The **TestTrie** benchmark implements a tree like structure of characters - similar to a hash table - and inserts a small number of words into the structure. This benchmark is focusing on traversing tree like structures by following references
3. *TestDeterminant.* The **TestDeterminant** benchmark models the concept of vectors and matrices using the Java concepts of classes and arrays. Then the Cramer formula for calculating the determinant of a given 3x3 matrix is applied
4. *TestWordReader.* The **TestWordReader** benchmark randomly generates 17 words and inserts them into a sorted list of words, checking the list before each insert to see if it is not there already. Only non duplicates are inserted

The nature of these benchmarks are not exhausting all aspects of the Java language, but we believe that they still reveal interesting information about the efficiency of any given JVM for embedded systems. The purpose of the benchmarks are to reveal how efficiently Java can be executed in terms of clock cycles as compared to C and how much code space and RAM memory is required. The benchmarks are not intended to test garbage collection, and none of the benchmarks require a functioning GC to run. Nor do they give any information about the real-time behaviour of the system under test. To test GC efficiency and/or real-time behaviour of a given JVM the CDj/CDc benchmarks are available.

In Section 7.2 we will compare the results from running these benchmarks on GCC, FijiVM, KESO, HVM, GCJ, JamVM, CACAO and HotSpot. This will give us valuable information about the efficiency with which these environments can execute Java code as compared to each other and as compared to C based execution environments.

7.1.1 Benchmark execution - High-end Platforms

Since only three of the tested execution environments (GCC, KESO and HVM) are capable of running these benchmarks on low-end embedded systems, we have first run the benchmarks on a 32 bit Linux PC. On this platform we could execute the benchmarks using all JVMs under test. We measured the number of instructions required to run the benchmarks using the **P**erformance **A**pplication **P**rogramming **I**nterface (PAPI) [7, 36]. The reason for measuring the instruction count and not the number of clock cycles is that the instruction count is a deterministic value for the benchmarks, but the clock cycle count is not on advanced processors. This first run of all the benchmarks on a 32 bit Linux PC will not by itself give us the desired results for low-end embedded platforms, but it will allow us to compare the JVMs under test against each other and against C on high-end platforms. To achieve the desired results for low-end embedded platforms we will run the benchmarks on a particular low-end embedded environment as well using GCC, HVM and KESO. This will give us the desired results for these two JVMs but compared with the results for high-end environments we will make statements about what could have been expected had it been possible to run all JVMs on a low-end embedded environment.

For all execution environments we measured the native instruction count by calling the PAPI API before and after each test run. We ran the tests several times until the measured value stabilized - this was important for the JIT compilers especially, but also for the other environments. E.g. calling `malloc` for the first time takes more time than calling `malloc` on previous runs. All in all the measurements reported are for *hot* runs of the benchmarks.

7.1.2 Benchmark execution - Low-end Platforms

To obtain a result for low-end embedded platforms we ran the benchmarks using GCC, HVM and KESO on a ATmega2560 AVR microcontroller. This is an 8 bit microcontroller with 8KB of RAM and 256KB ROM. On this simple platform there is a linear, deterministic correspondence between number of instructions executed and clock cycles. We used the AVR Studio 4 simulator to run the benchmarks and accurately measured the clock cycles required to execute each benchmark. Figure 24 shows an example of running the `TestQuicksort` benchmark using GCC. To produce the executable we used the `avr-gcc` cross compiler configured to optimize for size.

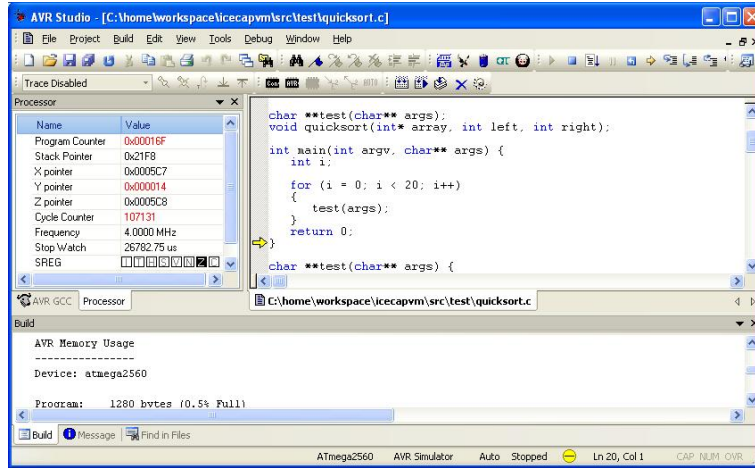


Figure 24: AVR Studio 4

In this test run we noted the clock cycles spent to get to the for-loop (in this case 125 clock cycles), since we eventually want to subtract this time spent in initialization from the time taken to perform the benchmark. Then the test was run 20 times, in this case yielding a clock cycle count of 107131. We conclude that GCC takes $(107131 - 125) / 20 = 5350$ clock cycles to perform the benchmark.

To obtain similar results for KESO we compiled the C source produced by the KESO Java-to-C compiler using the avr-gcc cross compiler and created an AVR Studio 4 project allowing us to measure clock cycles as above. Again we measured the start up time and ran each benchmark a number of times to arrive at an average time taken for KESO to execute the benchmark. Similarly for HVM. All projects configured to optimize for size.

These measurements are directly relevant for low-end embedded platforms and allow us to validate how the HVM compares to GCC and KESO. Since these three environments also appear in our high-end platform measurements, where they can be related to results from the other environments, we get a chance in Section 7.3 to predict how these other high-end environments would have performed had they been able to run on the ATmega2560.

7.2 Results

The measurements performed using the PAPI API on a 32 bit Linux PC platform is listed in Figure 25 and 26.

The instruction count taken for the C version to execute is defined as 100. The instruction count taken for the other environments is listed relatively to C above. E.g. the HVM uses 36% more instructions to execute the Trie benchmark than native C.

	C	KESO	FijiVM	HVM	GCJ
Quicksort	100	101	136	111	172
Trie	100	93	54	136	245
Determinant	100	59	37	96	171
WordReader	100	251	218	177	328
Total	100	126	111	130	229

Figure 25: Instruction count comparison - part 1

	C	JamVM	HVMi	CACAO	HotSpot
Quicksort	100	697	4761	147	156
Trie	100	772	1982	294	234
Determinant	100	544	1664	294	48
WordReader	100	975	4979	263	142
Total	100	747	3346	250	145

Figure 26: Instruction count comparison - part 2

The results from comparing HVM and KESO on the ATMega2560 are listed in Figure 27.

	C	KESO	HVM
Quicksort	100	108	130
Trie	100	223	486
Determinant	100	190	408
WordReader	100	331	362
Total	100	213	347

Figure 27: Cycle count comparison

This is an accurate cycle count comparison for KESO and HVM.

7.3 Discussion

The most interesting results are contained in Figure 27. From this we can conclude that for the benchmarks tested, *KESO is approximately 2 times slower than C and the HVM is approximately 3 times slower than C.*

There are several observations that should be taken into account when considering the above experiment:

- KESO supports GC, the HVM does not but relies on SCJ memory management. Eventhough GC is not in effect above, the KESO VM propably pays a price in terms of execution efficiency for the presence of GC

- The HVM supports Java exceptions, KESO does not. Based on very rudimentary experiments not shown here we get a hint that the cost of exception support is an approx 25% decrease in performance for the HVM
- Scrutinizing the C code produced by KESO we see that the Java type `short` is used in places where this is not correct. E.g. we had to manually fix the code produced for the WordReader benchmark to reintroduce the correct data type `int` in various places. Using `short` where `int` is required might be reasonable in several cases, and this will have a significant impact on performance, especially on 8 bit platforms as the ATMega2560.

Still we believe that we can make the following substantiated observations for low-end embedded platforms,

- Java-to-C compilers are a little slower than native C, but not by an order of magnitude. It is likely that they can achieve a throughput of approx. 100% that of native C.
- KESO is faster than HVM. The HVM achieves a throughput of approx 50% that of KESO