

Energy-Aware Modelling of Garbage Collectors for New Dynamic Embedded Systems

Jose M. Velasco^{*}, David Atienza^{*}, Luis Pinuel^{*}, Francky Catthoor[‡],
Francisco Tirado^{?*}, Katzalin Olcoz^{*}, Jose M. Mendias^{*}

^{*}DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain. Email: mvelascc@fis.ucm.es,
{datienza, lpinuel, ptirado, katzalin, mendias}@dacya.ucm.es

[‡]IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium.

Email: {Francky.Catthoor}@imec.be

Abstract

Modern embedded devices (e.g. PDAs, mobile phones) are now incorporating Java as a very popular implementation language in their designs. These new embedded systems include multiple complex applications (e.g. 3D rendering applications) that are dynamically launched by the user, which can produce very energy-hungry systems if they are not properly designed. Therefore, it is crucial for new embedded systems a better understanding of the interactions between the applications and the garbage collectors to reduce their energy consumption and to extend their battery life. In this paper we present a complete study from an energy viewpoint of the different state-of-the-art garbage collectors mechanisms (e.g. mark-and-sweep, generational garbage collectors) for embedded systems. Our results show that typical optimizations aiming at performance improvement for Java-based systems do not necessarily produce low-power final solutions.

1 Introduction

Currently Java is becoming one of the most popular choices for embedded/portable environments due to its high portability. Nevertheless, the great abstraction level provided by Java creates an additional major problem, which is the performance degradation of the system due to the inclusion of an additional component, i.e. the Java Virtual Machine or JVM, to interpret the native Java code and to execute it onto the present architecture.

In recent years, a very important research effort has been done for Java-based systems in order to improve performance up to the level required in new multimedia embed-

ded devices. This research has been mainly performed in the JVM. More specifically, it has focused on optimizing the execution time spent in the automatic object reclamation or Garbage Collector (GC) subsystem, which is one of the main sources of overall performance degradation of the system. However, the increasing need for power efficient systems limits very significantly the use of Java for new embedded devices since GCs are usually efficient enough in performance, but very costly in energy and power. Thus, efficient (from the energy viewpoint) automatic Dynamic Memory (DM) reclamation mechanisms and methodologies to define them have to be proposed for a complete integration of Java in the forthcoming low-power embedded systems.

In this paper we present a detailed study of the energy consumed in current state-of-the-art GCs, which is the first step to design custom energy-aware GCs for actual dynamic applications (e.g. multimedia) of embedded devices. The remainder of this paper is organized in the following way. In Section 2 we summarize some related work. In Section 3 we describe in detail the experimental setup used to investigate the energy consumption features of state-of-the-art GCs and the representative GCs used in our study. In Section 4, we briefly introduce our case studies and present the experimental results attained. Finally, in Section 5 we draw our conclusions.

2 Related Work

Nowadays a very wide variety of well-known techniques for uniprocessor GCs are available in a general-purpose context [18]. Recent research on GC policies has mainly focused on performance. The performance of the different GC strategies in a Java context is exhaustively studied in Blackburn et al. [15]. Our work extends their research to the contexts of energy consumption and pause time, where very little work has been done up today.

^{*}This work is partially supported by the Spanish Government Research Grant TIC2002/0750.

Eeckout et al. [11] investigate the microarchitectural implications of several virtual machines including Jikes. In this work, each virtual machine has a different garbage collector, so their results are not consistent related to memory management. Similar to this work is the Sweeney et al. [13] study. They conclude that the garbage collection increases cache misses for both instruction and data. However, they do not analyze the impact of different strategies in the total energy consumed in the system as we do. Farkas et al. [10] evaluate the energy impact of manual versus automatic memory management strategies in the context of C/ C++ programs. The garbage collector is the Boehm-Demers-Weiser conservative mark-and-sweep collector. Our study differs from this in that it is focused on the energy impact of the different automatic memory management strategies on a Java context.

In addition, Chen et al. [4] describe a technique that triggers the GC frequently without waiting for filling the heap. This way they can turn off memory banks thus reducing energy consumption. Nevertheless, they do not study the influence of their strategy in the performance or the pause time and their work is limited to the mark-and-sweep algorithm. Also, in [5], Chen et al. propose to compress the heap to reduce the memory footprint and therefore the memory restrictions. They use a mark-and-sweep collector, but their approach is GC strategy independent, so it can be used in addition to our proposal.

Finally, a large body of research on memory optimizations and techniques exists for static data in embedded systems (see e.g. [3, 12] for good tutorial overviews). All these techniques are complementary to our work and are applicable in the part of the Java code that accesses static data in the dynamic applications under study. Furthermore, they are useful as back-end for our approach, once the amount of dynamic memory used by the system is allocated into memory banks that can be statically declared and optimized.

3 Experimental Setup

In this section we first describe the whole simulation environment used to obtain detailed memory access profiling of the JVM (more specifically of the GCs). It is based on cycle-accurate simulations of the original Java code of the applications under study. In the second part of this section we briefly summarize the representative set of GCs used in our experiments.

3.1 Simulation Environment

Our simulation environment is depicted in Figure 1 and consists of three different parts. First, the detailed simulations of our case studies have been obtained after modifying

significantly the code of Jikes RVM (Research Virtual Machine) from the Watson Research Center of IBM [6]. Jikes RVM is a Java virtual machine designed for research. It is written in Java and the components of the virtual machine are Java objects [7], which are designed as a modular system that enables the possibility of modifying extensively the source code to implement different GC strategies and custom GCs. We have used version 2.3.2 along with the recently developed memory manager JMTk (Java Memory management Toolkit) [6].

The main modifications performed in Jikes have been done to integrate in it the Dynamic SimpleScalar framework (DSS) [18], which is an upgrade of the well known SimpleScalar simulator [2]. DSS allows a complete Java virtual machine simulation by supporting dynamic compilation, threads scheduling and garbage collection. It is based on the PowerPC ISA and has a fully functional and accurate cache simulator. We have included a cross-compiler [9] to be able to port our whole Jikes-DSS system and run it in the Pentium-based platform available for our experiments instead of the PowerPC traditionally used for DSS. We have used the option provided by DSS of implementing virtual devices to allow the simulator and simulated program to communicate with each other, using the Java Native Interface (JNI) and the Linux function M-ADVISE.

Finally, after the simulation in our Jikes-DSS environment, energy figures are calculated with an updated version of the CACTI model [1], which is a complete energy/delay/area model for embedded SRAMs that depends on memory footprint factors (e.g. size, internal structure or leaks) and factors originated by memory accesses (e.g. number of accesses or technology node used). One of its main advantage is that it is scalable to different technology nodes. For all our results shown in Section 4, we use the .13 μ m technology node. In our energy results for the SDRAM main memory used we also include static power (e.g. precharging of a bank, page misses, etc.) values that have been derived from a power estimation tool of Micron 32Mb/64Mb mobile SDRAM.

3.2 Studied State-of-the-art Garbage Collectors

In this section we briefly describe the studied GCs to show how they can cover the whole state-of-the-art spectrum of choices in current GCs. We refer to [8] and [16] for a complete overview of garbage collection techniques and for further details of the specific implementation used in our experiments with Jikes.

In our study all the collectors are from the categories of GCs known as *Tracing* and *stop-the-world* [18]. We study the following representative GCs for embedded devices:

- The classic tracing collectors Mark-and-sweep (or MS) and SemiSpace copying collector (SemiSpace or SS).

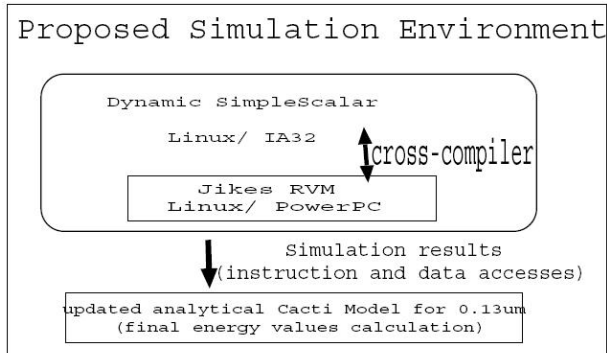


Figure 1. Graphical overview of our whole simulation environment

- Generational Collectors: in this kind of GCs, the heap is divided in areas according to the antiquity of the data. The collector can manage the distinct generations with the same policy or assign to each one different strategies. We consider in this case two typical examples:

- GenCopy: A generational collector with semispace copying policy in both nursery and mature generation.
- GenMS: A hybrid generational collector with semispace copying policy in the nursery and mark-and-sweep strategy in the mature generation. It does not need to reserve space for objects in the mature space where the generational hypothesis expects to find a high surviving rate.

- Copying collector with Mark-and-Sweep (or CopyMS in our experiments): It is the non-generational version of the previous one. Objects that survive a collection are managed with a mark-and-sweep strategy and therefore they are not moved any more.

4 Case Studies and Experimental Results

We have applied the proposed experimental setup to the GCs presented in the previous subsection running into the most representative benchmarks in the suite SPECjvm98 [14] for new embedded devices. These benchmarks could be launched as dynamic services and extensively use dynamic data allocation. They are the following: `_201_compress`, `_202_jess`, `_205_raytrace`, `_213_javac` and `_222_mpegaudio`.

The results shown in this section were obtained on a Pentium III processor at 866 MHz with 1024 MBytes SDRAM and running GNU/Linux 2.4.

As we said earlier, the collectors in this study are *stop-the-world*. This implies that the running application (more frequently known as mutator in the GCs context) is paused during the garbage collection to avoid inconsistencies in the references to dynamic memory in the system. This pause is

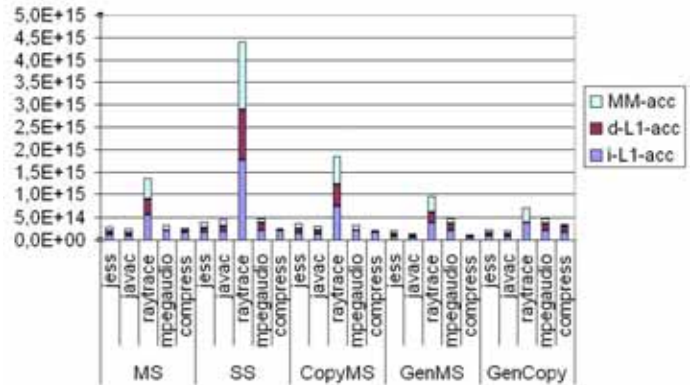


Figure 2. Energy figures for traditional GCs for embedded devices.

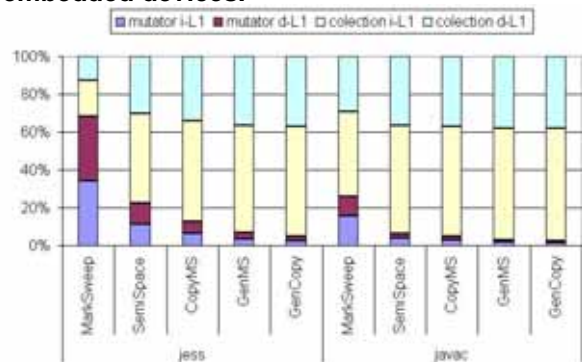


Figure 3. Energy breakdown for all GCs in jess and javac with a heap size of 16MB

the main obstacle for applications using GCs to be executed with real-time requirements.

We have performed two different kind of experiments: In our first set of experiments we have measured the global energy consumption for all GCs Figure 2. In this case, the L1

D-cache and I-cache have sizes of 8 KB each and the main memory is 16 MB. As these results show, all GCs based on Generational collectors (i.e. *GenMS* and *GenCopy*) achieve the best energy results compared to more typical GCs implemented in the JVM of real-life Java-based embedded devices (e.g KVM [4]).

Figure 3 shows the results for the most representative case studies of this set of experiments (i.e. jess and Javac) since they are the ones that most intensively access memory. Here we can see that the percentage of energy spend by the garbage collector increases with the strategies based in the copying policy, being the biggest percentage in GenCopy. However, our second set of experiments measure the effective processing time of an application (mutator) for a time interval spectrum. The minimum mutator utilization (MMU) gives us the percentages the JVM is spending in the collector and the program. This is the usual way of characterizing the response time of a collector, that is to say, its capacity for running applications

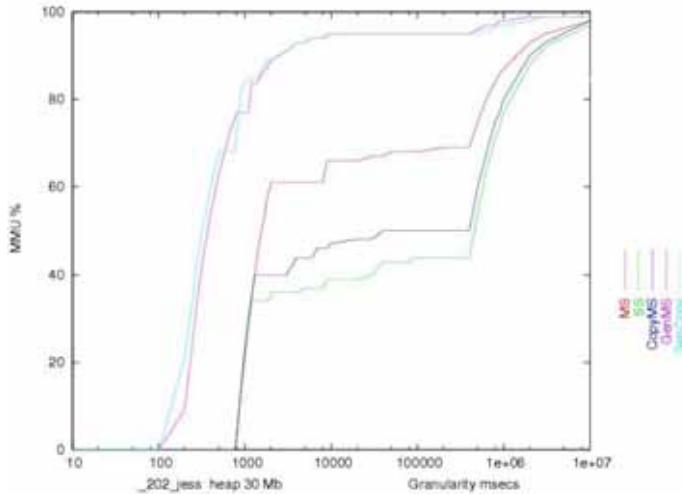


Figure 4. Effective processing execution time percentage for jess and for different GCs.

GCs do not need to make a full heap collection, their MMU is one order of magnitude better than non-generational collectors. So, we find here a trade-off between energy consumption and pause time.

5 Conclusions

New embedded devices can presently execute complex dynamic applications (e.g. multimedia). These new complex applications are now including Java as one of the most popular implementation languages in their designs due to its high portability. Hence, new Java Virtual Machines (JVM) should be designed trying to minimize their energy consumption while respecting the soft real-time requirements of these embedded systems. In this paper we have presented a complete study from an energy viewpoint of the different state-of-the-art garbage collectors mechanisms used in current JVM (e.g. mark-and-sweep, generational garbage collectors) for embedded systems. We have shown how typical optimizations aiming at performance for Java-based systems have to be modified to match the low-power requirements of new embedded systems. In the future, we would explore more in detail the different sizes of L1 caches used in Java-based systems because our energy results show that they heavily depend on the application under study.

This shows that it would be convenient to use generational GCs in embedded devices, contrarily to what is more usually done nowadays because of the most traditional belief of the inherent complexity of this kind of GCs. The main reason for these results is because since generational GCs group objects with similar temporal behaviour, temporal locality in the D-caches is improved. Also, the heap does not get so frequently fragmented (i.e. Dead objects mixed with alive ones in memory) and less garbage collections are needed.

References

- [1] V. Agarwal, S. Keckler, and D. Burger. The effect of technology scaling on microarchitectural structures. Technical report, Technical Report TR2000-02, University of Texas at Austin, USA, 2002.
- [2] T. Austin. Simple scalar llc, 2004. <http://simplescalar.com/>.
- [3] L. Benini and G. De Micheli. System level power optimization techniques and tools. In *ACM Transactions on Design Automation for Embedded Systems (TODAES)*, April 2000.
- [4] G. Chen, R. Shetty. Tuning garbage collection for reducing memory system energy in an embedded java environment. *ACM Transactions on Embedded Computing Systems (TECS)*, 1(1), november 2002.
- [5] G. Chen, M. Kandemir and M. Wolczko. Heap compression for memory-constrained java environments. 2003.
- [6] IBM. The jikes' research virtual machine user's guide 2.2.0., 2003. <http://oss.software.ibm.com/developerworks/oss/jikesrvm/>.
- [7] The source for java technology, 2003. <http://java.sun.com>.
- [8] R. Jones and R. D. Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons.
- [9] D. Kegel. Building and testing gcc/glibc cross toolchains, 2004.
- [10] G. Keith I. Farkas, Jason Flinn and J. M. Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *SIGMETRICS 2000*, Santa Clara, California, June 2000.
- [11] G. Lieven Eeckhout and K. D. Bosschere. How java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, California, USA, 2003.
- [12] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, and C. Kulkarni. Data and memory optimizations for embedded systems. *ACM Transactions on Design Automation for Embedded Systems (TODAES)*, 6(2):142–206, April 2001.
- [13] Peter F. Sweeney, Matthias Hauswirth and M. Hind. Using hardware performance monitors to understand the behavior of java application. In *USENIX 3rd Virtual Machine Research and Technology Symposium (VM'04)*, 2004.
- [14] SPEC. Specjvm98 documentation, March 1999. <http://www.specbench.org/osg/jvm98/>.
- [15] P. C. Steve Blackburn and K. McKinley. Myths and reality: The performance impact of garbage collection. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, June 2004.
- [16] P. C. Steve Blackburn and K. McKinley. Oil and water: High performance garbage collection in java with mmtk. In *Proceedings of 26th International Conference on Software Engineering*, SIGMETRICS, May 2004.
- [17] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.
- [18] Xianglong Huang, J. Eliot B. Moss and D. Burger. Dynamic simple scalar: Simulating java virtual machines. Technical report, University of Texas at Austin Department of Computer Sciences, february 2003.