Energy Characterization of Garbage Collectors for Dynamic Applications on Embedded Systems

Jose M. Velasco[1], David Atienza[1,2,⋆], Katzalin Olcoz[1], Francky Catthoor[2,⋆], Francisco Tirado[1], and J.M. Mendias[1]

[1] DACYA/U.C.M., Avenida Complutense s/n, 28040 Madrid, Spain
[2] IMEC vzw, Kapeldreef 75, 3000 Leuven, Belgium

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 devices 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 traditional solutions of garbage collectors for Java-based systems do not seem to produce the lowest energy consumption solutions.

1 Introduction

Currently Java is becoming one of the most popular choices for embedded/portable environments. In fact, it is suggested that Java-based systems as mobile phones, PDAs, etc. will enlarge their current market from around 150 million devices in 2000 to more than 700 millions at the end of 2005 [20]. One of the main reasons for this large growth is that the use of Java in embedded systems allows developers to design new portable services that can effectively run in almost all the available platforms without the use of special cross-compilers to port them to different platforms, as happens with other languages (e.g. C or C++). Nevertheless, the abstraction 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 to improve performance up to the level required in new embedded 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.

⋆ This work is partially supported by the Spanish Government Research Grant TIC2002/0750 and E.C. Marie Curie Fellowship contract HPMT-CT-2000-00031. + Also professor at ESAT/K.U.Leuven-Belgium

V. Paliouras, J. Vounckx, and D. Verkest (Eds.): PATMOS 2005, LNCS 3728, pp. 69–78, 2005.
© Springer-Verlag Berlin Heidelberg 2005

However, the increasing need for efficient systems (i.e. low-power) 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 DM reclamation mechanisms and methodologies to define them have to be proposed for a complete integration of Java in the design of forthcoming very low-power embedded systems.

In this paper we present a detailed study of the energy consumed in current state-of-the-art GCs (i.e. generational GCs, mark-and-sweep, etc.), 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 the experimental setup used to investigate the energy consumption features of GCs and the representative state-of-the-art GCs used in our study. In Section 4, we 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 (e.g. reference counting, mark-sweep collection, copying garbage collector) are available in a general-purpose context within the software community [11]. Recent research on GC policies has mainly focused on performance [6]. Our work extends their research to the context of energy consumption.

Eeckout et al. [8] investigate the microarchitectural implications of several virtual machines including Jikes. In this work, each virtual machine has a different GC, so their results are not consistent related to memory management. Similarly, Sweeney et al. [18] conclude that GC increases the 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.

Chen et al. [7] focus in reducing the static energy consumption in a multibanked main memory by tuning the collection frequency of a Mark&Sweep-based collector that shuts off memory banks that do not hold live data. The reduction of leakage approach is parallel to ours and can be used complementary.

Finally, a large body of research on memory optimizations and techniques exists for static data in embedded systems (see e.g. [5, 14] 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.

3 Experimental Setup

In this section we first describe the whole simulation environment used to obtain detailed memory access profiling of the JVM (for both the application and the collector phase). It is based on cycle-accurate simulations of the original Java code of the applications under study. Then we summarize the representative set of GCs used in our experiments. Finally we introduce the sets of applications selected as case studies.

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 [9]. 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 [10], which are designed as a modular system to enable 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) [9].

The main modifications in Jikes have been performed to integrate in it the Dynamic SimpleScalar framework (DSS) [21], which is an upgrade of the well known SimpleScalar simulator [4]. DSS enables a complete Java virtual machine simulation by supporting dynamic compilation, threads scheduling and garbage collection. It is based on a PowerPC ISA and has a fully functional and accurate cache simulator. We have included a cross-compiler [12] to be able to run our whole Jikes-DSS system onto the Pentium-based platform available for our experiments instead of the PowerPC traditionally used for DSS. In our experiments, the memory architecture consists of three different levels: an on-chip SRAM L1 memory (with separated D-cache/I-cache), an on-chip unified SRAM L2 memory of 256K and an off-chip SDRAM main memory. The L1 size is 32K and the L1 associativity has been tested between 1-way and 32-ways. The block size is 32 bytes and the cache uses and LRU blocks replacement policy.

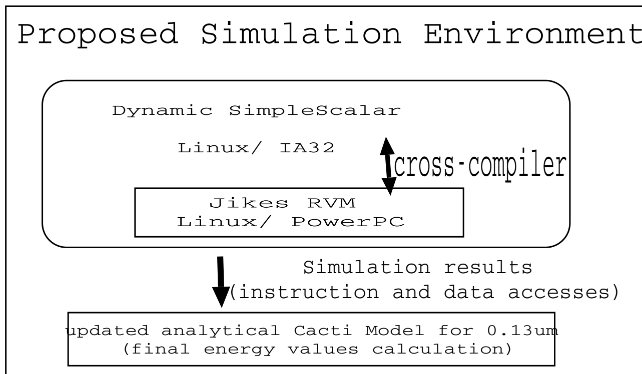


Fig. 1. Graphical overview of our whole simulation environment

Finally, after the simulation in our Jikes-DSS environment, energy figures are calculated with an updated version of the CACTI model [3], which is a complete energy/delay/area model, scalable to different technology nodes, for embedded SRAMs. For all our results shown in Section 4, we use the $.13\mu$m technology node. In our energy results for the SDRAM main memory we also include static power values (e.g. precharging of a bank, page misses, etc.) that have been derived from a power estimation tool of Micron 16Mb mobile SDRAM [13].

3.2 Studied State-of-the-Art Garbage Collectors

Next, we describe the main differences among the studied GCs to show how they can cover the whole state-of-the-art spectrum of choices in current GCs. We refer to [11] for a complete overview of garbage collection techniques and for further details of the specific implementation used in our experiments with Jikes [9].

In our study all the collectors fall into the category of GCs known as tracing *stop-the-world* [11]. This implies that the running application (more frequently known as mutator in the GCs context) is paused during garbage collection to avoid inconsistencies in the references to dynamic memory in the system. To distinguish the live objects among the garbage, the tracing strategy relies on determining which objects are not pointed to by any living object. To this end, it needs to traverse the whole relationship graph through the memory recursively. The way of reclaiming the garbage produces the different tracing collectors of this paper. Inside this class we study the following representative GCs for embedded devices: mbdlma@hotmail.com - Mark-and-sweep (or MS): the allocation policy uses a set of different block-size *free-lists*. This produces both internal and external fragmentation. Once the tracing phase has marked the living data, the collector needs to *sweep* all the available memory to find unreacheable objects and reorganize the free-lists. The sweeping of the whole heap is very costly and to avoid it in the Jikes virtual machine, the sweep-phase is implemented as *lazy*. This means that the sweep is delayed up to the allocation phase. This is a classical collector implemented in several Java virtual machines as Kaffe [2], JamVM [15] or Kissme [16].

- Copying collector (SemiSpace or SS): it divides the available space of memory in two halves, called semispaces. The objects that are found alive are copied in the other semispace in order and compacted.

Generational Collectors: in this kind of GCs, the heap is divided into areas according to the antiquity of the data. When an object is created, it is assigned to the youngest generation, the nursery space. As objects survive different collections they mature, that is to say, they are copied into older generations. The frequency with which a collection takes place is lower in older generations.

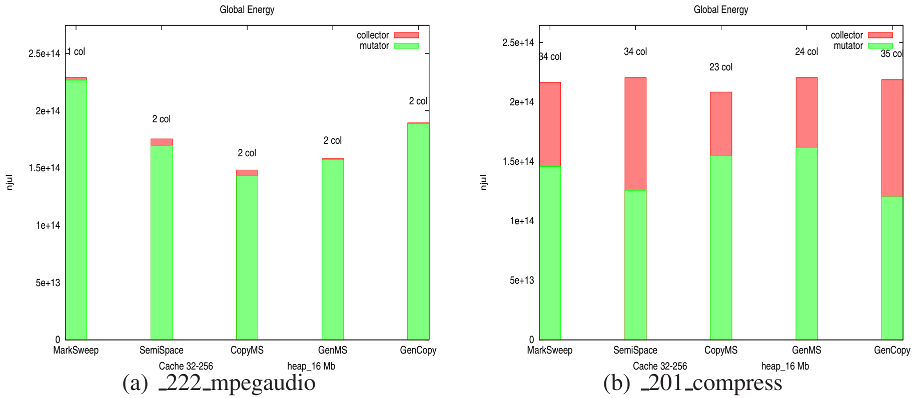
The generational collector can manage the distinct generations with the same policy or assign to each one different strategies. We consider here two options:

– GenCopy: a generational collector with semispace copying policy in both nursery and mature generation. This collector is used in the BEA JRockit virtual Machine [19] and the SUN J2SE(Java 2 Standard Edition) JVM by default uses a very close collector with a Mark&Compact strategy in the mature generation.
– GenMS: a hybrid generational collector with semispace copying policy in the nursery and mark-and-sweep strategy in the mature generation. The Chives Virtual Machine [1] uses a hybrid generational collector with three generations

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.

In Jikes, these five collectors manage objects bigger than a certain threshold (by default 16K) in a special area. Jikes also reserves space for immortal data and meta data (where the references among generations are recorded, usually known as the *remembered set*). These special memory zones are also studied in our experimental results.

Finally, it is important to mention that, even though we study all the previous GCs with the purpose of covering the whole range of options for automatic memory management, real-life Java-based embedded systems typically employ MS or SS since they are initially the GCs that possess less complex algorithms to implement; Thus, theoretically putting less pressure in the processing power of the final embedded system and achieving good overall results (e.g. performance of memory hierarchy, L1 cache behaviour, etc.)



(a) _222_mpegaudio (b) _201_compress

Fig. 2. Energy figures for L1-32K, direct mapped and LRU replacement policy

3.3 Case Studies

We have applied the proposed experimental setup to the GCs presented in the previous subsection running the most representive benchmarks in the suite SPECjvm98 [17] for new embedded devices. These benchmarks could be launched as dynamic services and extensively use dynamic data allocation. The used set of applications is the following:

_222_mpegaudio: it is an MPEG audio decoder. It allocates 8 MB + 2 MB in the LOS.

_201_compress: it compresses and then uncompresses a large file. It mainly allocates objects in the LOS (18 MB) while it uses only 4MB of small objects.

_202_Jess: it is the Java version of an expert shell system using NASA CLIPS. It is compound fundamentally of structures of sentences 'if-then'. It allocates 48 MB (plus 4 MB in the LOS) and most objects are short-lived.

_205_Raytrace: raytraces a scene into a memory buffer. It allocates a lot of small data (155 MB + 1 MB in the LOS) with different lifetimes.

_213_javac: it is the java compiler. It has the highest program complexity and its data is a mixture of short and quasi-inmortal objects (35MB + 3 MB in the LOS).

The suite SPECjvm98 offers three input sets(referred as s1, s10, s100), with different data sizes. In this study we have used the medium input data size, represented as s10, as we think it is more representative of the actual input sizes of multimedia applications in embedded systems.

4 Experimental Results

This section shows the application of the previously explained experimental setup (see Section 3 for more details) to perform a complete study of automatic dynamic memory management mechanisms for embedded systems according to key metrics in embedded systems (i.e. energy, power and performance of the memory subsystem). To this end, in this section we first analyze the dynamic allocation behaviour of the different SPEC benchmarks and categorize them in dynamic behaviour scenarios. Then, we discuss how the different GC strategies respond to each scenario. Finally, we study how several key features of the memory hierarchy (i.e. associativity of the cache and the size of main memory) can affect each GC behavior from an energy consumption point of view.

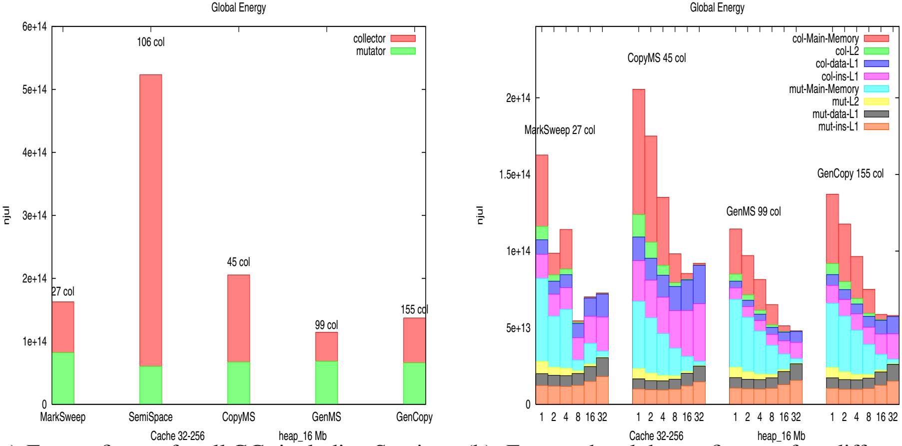
4.1 Analysis of Data Allocation Scenarios for JVM in Embedded Systems

After a careful study of the different data allocation behaviours encountered in the SPEC benchmarks, we have been able to identify three types of scenarios:

1. The first scenario related to benchmarks that employ a very limited memory space within the heap, such as _222_mpegaudio in SPEC. In fact, since it allocates a very reduced amount of dynamic memory, it is usually not considered in GC studies. Nevertheless, as we can see in figure 2(a) the GC choice influences the virtual machine behavior during the mutator phase (due to allocation policy complexity and data locality) and it can achieve, if correctly selected, up to a 40% global energy reduction. Hence, we are including it as representative example of this kind of benchmarks, which in reality are not so infrequent since many traditional embedded applications (e.g. MP3 encoders/decoders, MPEG2 applications, etc.) use static data and only few data structures demand dynamic allocations.

2. The second scenario has been identified in benchmarks that mostly allocate large objects, such as _201_compress. This benchmark is the Java version of 129.compress benchmark from the SPEC CPU95 suite and it is an example of Java programming in a C-like style. The fraction of heap accesses to the TIB (Type Information Block) table is very small relative to the fraction of accesses to array elements (the Lemper-Ziv's dictionaries). This means that the application spends more time accessing static memory via C-like functions rather than generating and accessing dynamic memory using object-oriented methods as in native Java applications. Hence, similarly to the previous type of scenario, it is usually considered an exception and not included in GC studies. Nonetheless, we have included it in our study as we have verified that such kind of non-truly object-oriented Java program with dynamic behaviour is quite common in embedded systems. Moreover, we consider that compressing algorithms are very frequently present in embedded environments and the big amount of large objects allocation required in such systems demands a deep understanding.

3. The third possible scenario has been observed in benchmarks with a medium to high amount of allocated data, and with different life timespan, for instance, in SPEC: _202_Jess, _205_Raytrace and _213_javac. These benchmarks are the ones most frequently considered in performance studies of GCs and in this paper we present complementary results for them taking into consideration energy figures apart from performance values.



(a) Energy figures for all GCs including Semispace collector.

(b) Energy breakdown figures for different GCs. Associativity varies from 1-way to 32-ways.

Fig. 3. Third Scenario energy figures for all GCs with L1-32K

4.2 Comparison of GC Algorithms

Our results indicate that each GC shows a different behaviour regarding which scenario the application under study belongs to. In the first scenario, the percentage of energy wasted in the collection phase is not significative in comparison with the energy spent during the mutator phase. However, as Figure 2(a) depicts, the global JVM energy consumption varies significantly depending on the chosen GC. First, we can observe that it is very important the effect of the allocation policy associated with the GC. In fact, Figure 2(a) indicates that the simplest GC of all tested copying collectors, i.e. SS (see Section 3.2 for more details) attains better energy results than MS, which uses free-lists and lazy deletion. Second, our results indicate that since the number of accesses and L1 cache miss rates (see Table 1) of all copying-based collectors are very similar, the main factor that differentiates their final energy figures is the L2 miss rate. Thus, outlining that hybrid collectors that minimize this factor (i.e. CopyMS and GenMS) produce the best results for energy consumption. In fact, the best choice for this scenario (i.e. CopyMS) achieves an overall energy reduction of 55% compared to the classical Mark&Sweep collector employed in many embedded systems.

In the second scenario, the percentage of energy wasted in the collection phase varies from a 25% to a 50% of the final JVM energy consumption. This indicates that current GCs for embedded systems are not really tuned for this type of embedded application. Then, considering the special characteristics of _201_compress (see Section 4.1), we can observe that, contrarily to the previous scenario, the energy behavior during the mutator phase for the pure copying GCs is better than for the hybrid ones. However, the better space management that hybrids collectors exhibit produce a lesser number of collections; Thus, the energy spent in the collection step is up to 40% less than pure copying GCs. All in all, hybrid collectors attain a final energy reduction of 5% compared to copying GCs, and again CopyMS is the best election.

In the third group of benchmarks, we find that the handicaps associated with the classical SemiSpace copying collector make it not appropriate for memory constrained environments. The fact is that Figure 3(a) (for a heap memory of 16MB) shows that SS consumes less energy than MS during the mutator phase, the one next with less energy consumption in this phase, but it possesses a large penalty in the collection step. As a result, the global energy consumption of this GC is twice the consumption of any other collector studied. Due to this fact, it is discarded and it will not be included in the rest of results presented related to the third scenario for a main memory size of 16MB. Nevertheless, it is important not to discard this GC in other working environments with less constrained main memory sizes. In fact, we have performed an additional set of experiments with larger main memories (i.e. 32MB or 64MB) and, with these larger heap sizes, SS achieves even better results than MS. This is due to the fact that SS performs more memory accesses than MS when there are more objects alive in the heap, while MS and other GCs perform more memory accesses when more objects are dead. Therefore, on the one hand, in smaller heaps (e.g. like with 16 MB), there is not enough time for the objects to die and SS is more expensive than MS in energy consumption. On the other hand, in large heaps where more objects can die between memory collections, SS is favoured compared to MS and other GCs; thus, it consumes less energy.

In addition, we can observe that the best energy consumption results for this scenario are achieved by different variations of the two generational GCs studied (i.e. GenMS and GenCopy in Figure 3(a)). This occurs because for the input data size used (s10) the amount of metadata produced by the write barriers is insignificant. Thus, no performance penalties appear in generational GCs and they attain similar results to the hybrid solutions that were the best options in the other scenarios (e.g. CopyMS) during the mutator phase. Next to this, the minor collections (only in parts of the heap, see Subsection 3.2 for more details) in the generational strategy interfere less in the cache

Table 1. Summary of cache miss rates in all benchmarks with L1-32K and direct mapped

		mut-ins-L1 %	mut-data-L1 %	mut-L2 %	col-ins-L1 %	col-data-L1 %	col-L2 %
1° scenario	MS	9.9	6.6	92.5	13.3	7.3	30.8
	SS	9.9	6.2	67.3	13.3	7.8	41.1
	CopyMS	10.0	6.2	54.8	14.0	5.7	36.4
	GenMS	10.1	6.3	58.9	13.4	6.7	53.6
	GenCopy	10.0	6.3	74.0	14.0	6.8	51.2
2° scenario	MS	8.7	5.1	32.6	13.2	7.3	30.3
	SS	7.7	7.4	22.0	13.3	7.8	42.7
	CopyMS	8.6	7.0	30.5	14.0	5.6	37.1
	GenMS	8.4	7.1	33.6	13.6	5.9	45.1
	GenCopy	7.5	5.0	24.4	14.1	6.5	45.1
3° scenario	MS	13.0	8.1	56.2	13.3	7.4	31.9
	SS	13.4	8.5	52.5	13.4	7.8	44.2
	CopyMS	13.1	8.8	53.9	14.0	5.7	38.4
	GenMS	12.7	8.1	50.3	12.8	6.0	42.5
	GenCopy	12.9	7.9	49.8	13.6	6.5	43.3

behavior than the full heap collections of non-generational ones. Furthermore, GenMS does not need to reserve space for copying surviving objects in the mature generation. This produces a lesser number of both minor and mayor collections and it is the eventual reason why GenMS obtains slightly better results than GenCopy.

Finally, to test if there are important effects of the eventual memory hierarchy in the GCs, we have performed a final set of experiments varying the associativity of the L1 cache from 1-way to 32-ways and with a main memory size of 16 MB. The results accomplished are depicted in Figure 3(b), which indicates that the energy breakdown figures of the different GCs (without SS) for this third scenario, distinguishing the energy consumption during the mutator phase (mut) and collector phase (col). As these results outline, depending on the memory hierarchy (in this case simply modifying the L1 associativity), the influence of the GC algorithm choice can vary significantly. In fact, in this case the energy consumption differences between GCs can vary up to 50% in the collection phase and and its indirect effect in the mutator phase can reach an additional 20% variation in energy consumption. Hence, the global variation in energy consumption can be up to 40%. Also, this study indicates that the L1 miss rates (see Table 1) are very similar for all GCs. Finally, we can observe that Mark&Sweep has the lowest L2 miss rates. Besides, with an associativity of 8-ways, the reduction in the number of misses (Figure 3(a)) is translated in a drastic reduction of the energy spent in both main memory and L2 cache. This reduction produces a final global energy very close to the best results of the generational GCs, figure 3(b). Therefore, the Mark&Sweep handicaps can be diminished with a proper cache paremeters selection. In summary, the GC algorithm choice is a key factor for optimizing the final energy consumption of the JVM, but it should take into account the memory hierarchy to be tuned conveniently.

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 GCs mechanisms used in current JVM for embedded systems. We have shown how the GCs traditionally used in embedded devices (i.e. MS or SS) for Java-based systems do not achieve the best energy results, which are obtained with variations of generational GCs. In addition, the specific memory hierarchy selected can significantly vary the overall results for each GC scheme, thus showing the need of further research in this aspect of Java-based embedded systems.

References

1. Chives virtual machine with a jit compiler, 2005. `http://chives.sunsite.dk/`.
2. Kaffe is a clean room implementation of the java virtual machine, 2005. `http://www.kaffe.org/`.

3. 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.
4. T. Austin. Simple scalar llc, 2004. http://simplescalar.com/.
5. 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.
6. S. Blackburn, P. Cheng, 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.
7. G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection for reducing memory system energy in an embedded java environment. *ACM Transactions on Embedded Computing Systems (TECS)*, 1(1), november 2002.
8. L. Eeckhout, A. Georges, 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. ACM Press New York, NY, USA.
9. IBM. The jikes' research virtual machine user's guide 2.2.0., 2003. http://oss.software.ibm.com/developerworks/oss/jikesrvm/.
10. The source for java technology, 2003. http://java.sun.com.
11. R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 4th edition, July 2000.
12. D. Kegel. Building and testing gcc/glibc cross toolchains, 2004. http://www.kegel.com/crosstool/.
13. Zbt@ sram and sdram products, 2004. http://www.micron.com/.
14. 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.
15. Sourceforge. Jamvm - a compact java virtual machine, 2004. http://jamvm.sourceforge.net/.
16. Sourceforge. kissme java virtual machine, 2005. http://kissme.sourceforge.net/.
17. SPEC. Specjvm98 documentation, March 1999. http://www.specbench.org/osg/jvm98/.
18. P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, 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.
19. B. Systems. Bea weblogic jrockit, 2005. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/jrockit/.
20. D. Takahashi. *Java chips make a comeback*. Red Herring, 2001.
21. The University of Massachusetts Amherst and the University of Texas. Dynamic simple scalar, 2004. http://www-ali.cs.umass.edu/DSS/index.html.