Exploration of Memory Hierarchy Configurations for Efficient Garbage Collection on High-Performance Embedded Systems

Jose Manuel Velasco
DACYA-Complutense
Univ. of Madrid (UCM)
Avenida Complutense s/n,
28040 - Madrid, Spain.
mvelascc@fis.ucm.es

David Atienza
Embedded Systems
Laboratory-EPFL
EPFL-STI-IEL-ESL
1015 Lausanne, Switzerland
david.atienza@epfl.ch

Katzalin Olcoz
DACYA-Complutense
Univ. of Madrid (UCM)
Avenida Complutense s/n,
28040 - Madrid, Spain.
katzalin@dacya.ucm.es

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 applications that are dynamically launched by the user, which can produce very energy-hungry systems if the interactions between the applications and the garbage collectors (GCs) are not properly understood. In this paper we present a complete exploration, from an energy viewpoint, of the different possibilities of memory hierarchies for high-performance embedded systems when used by state-of-the-art GCs. Moreover, we explore the potential peformance improvement and energy reductions of using a scratchpad memory directed by the virtual machine to store critical code and data structures of the GCs; thus, enabling up to 40% performance improvements and 41% leakage reduction with respect to classical cache-based memory architectures. Our experimental results show that the key for an efficient low-power implementation of *Java Virtual Machines (JVM)* for high-performance embedded systems is the synergy between the GC choice, the memory architecture tuning, and the inclusion of power management schemes controlled by the JVM, exploiting knowledge of the used GC.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Garbage collection, Storage hierarchies*

General Terms

Design, Measurement, Performance

Keywords

Garbage collection, Java, Memory exploration, Embedded systems

1. INTRODUCTION

Java is becoming one of the most popular choices for embedded portable environments. In fact, currently there are more than 5 millions of Java code developers and it is expected that the number of Java-based systems, such as mobile phones, PDAs, etc. will increase [19]. One of the main reasons for this large growth is that the use of Java in high-performance embedded systems allows developers to design new portable services, which can effectively run in almost all the available platforms without the use of special cross-compilers to port them, 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 (JVM)*, to interpret the native Java code and execute it onto the underlying 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. As a result, state-of-the-art GCs (e.g., generational GCs, incremental mark-and-sweep algorithms) have reduced their latency of response and the amount of time that the system needs to be stopped to compact the whole list of unused objects in Java-based designs. However, the increasing need for low-power 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, optimized (from the energy viewpoint) automatic dynamic memory reclamation mechanisms and methodologies to define them have to be proposed for a complete integration of Java in the design of forthcoming high-performance embedded systems, which include tight low-power constraints for portability purposes.

In this paper we present a detailed exploration of energy vs performance memory hierarchy trade-offs for embedded systems in presence of a complete range of different state-of-the-art GCs (e.g., generational GCs, mark-and-sweep, etc.), which is the first step to define suitable memory management optimizations for energy-aware Java-based embedded systems. Then, we present two techniques for reducing the energy consumption of the whole memory system: using an instruction scratchpad memory directed by the virtual machine and using GC information for leakage energy consumption reduction. Our results show up to 40% performance improvements and 41% leakage reduction with respect to classical cache-based JVM memory architectures.

The rest of the paper is organized as follows. In Section 2 we summarize related work in the area of JVM design and GCs optimizations. In Section 3 we describe the experimental setup used

GLSVLSI'09, May 10–12, 2009, Boston, Massachusetts, USA.
Copyright 2009 ACM 978-1-60558-522-2/09/05 ...$5.00.

to investigate the energy consumption features of the various memory hierarchies possibilities, the representative state-of-the-art GCs used and the considered applications. In Section 4, we present the experimental results. 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 GCs) are available in a general-purpose context within the software community [11]. A lot of research on GC policies and architectural exploration has mainly focused on performance [3]. Our work extends this research for an overall memory hierarchy exploration of high-performance embedded systems.

Eeckout et al. [7] investigate the micro-architectural 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. In a recent study, [5] explores the performance and power consumption of the overall Java virtual machine to propose microarchitectural changes, but for much larger memories in desktop and server systems. Thus, their results are complementary to our analysis for embedded systems.

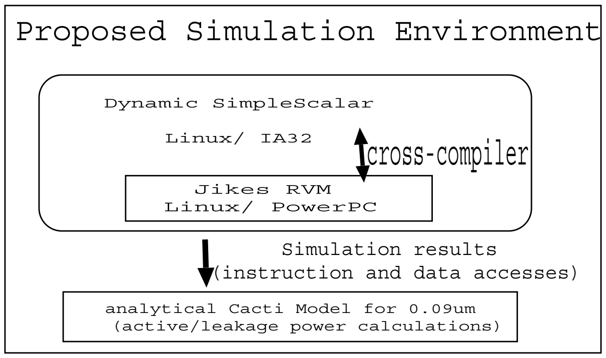
Chen et al. [6] focus on reducing the static energy consumption in a multi-banked main memory by tuning the collection frequency of a Mark&Sweep-based collector that shuts off memory banks that do not hold live data. The static leakage is decreased by turning off the unnecessary banks. In [8], it is proposed a scratchpad allocation scheme that is completely implemented inside the JVM, which does not use compiler support, but this paper does not explore the effects in power consumption and does not explore different garbage collection algorithms. A more complete study regarding energy is performed in [4], which proposes two implementation strategies for allocating objects that can significantly reduce the memory system energy consumption of Java applications. The first strategy uses a part of the on-chip memory resources as a local memory to achieve better performance than a cache-only architecture, where the object allocation strategy is implemented using an annotation-based approach and shown to be effective in improving performance and reducing the memory system energy consumption. The second strategy is object co-location, which exploits the temporal locality already present in heap references to achieve better spatial locality and less cache misses, with a subsequent energy consumption reduction. The object co-location approach and the use of local memories to reduce energy consumption is parallel to our memory hierarchy exploration; thus, the previous mechanisms can be additionally used in combination to our memory hierarchy customization for high-performance embedded systems.

3. EXPERIMENTAL FRAMEWORK

In this section we first describe the overall simulation environment used to obtain detailed memory access profiling of the JVM (for both the application and the collector phase), which is based on cycle-accurate simulations of the original Java code of the applications under study. Then, we summarize the representative set of considered GCs in our experiments and the set of applications used 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 *Research Virtual Machine (RVM)* from the IBM Watson Research Center [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 *Java Virtual memory Management Toolkit (JVMTk) [9].*


Proposed Simulation Environment
Dynamic SimpleScalar
Linux/ IA32
cross-compiler
Jikes RVM
Linux/ PowerPC
Simulation results
(instruction and data accesses)
analytical Cacti Model for 0.09um
(active/leakage power calculations)
Figure 1: Overview of the overall simulation environment

The main modifications in Jikes have been performed to integrate in it the *Dynamic SimpleScalar framework (DSS)* [20], which is an upgrade of the well known SimpleScalar simulator [2]. 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 [13] 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.

Finally, after the simulation in our Jikes-DSS environment, energy figures are calculated with an updated version (v4.1) of the CACTI model [1], which is a complete energy/delay/area model, scalable to different technology nodes, for embedded SRAMs and that includes leakage as well as active power in the different components of the memory cells. For our results of Section 4, we use the $.09\mu$m technology node. In the energy results for the SDRAM main memory, we also include static power values (e.g., bank precharging, page misses.) derived from a power estimation tool of Micron 16 MB mobile SDRAM [14].

3.2 Studied State-of-the-art GCs

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 used in our experiments with Jikes [9].

First, we need to distinguish between the garbage *collector* and the *mutator*, as described in Dijkstra's terminology [11]. During the collector phase, the JVM is executing the garbage collection algorithm (distinguishing and reclaiming garbage), while the mutator phase refers to the JVM executing the user application along with its remaining tasks. We report the performance and energy results for these two phases for all the considered GCs. 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 (or mutator) 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:

- *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 unreachable 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* [11]. 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 [12], JamVM [15] or Kissme [16], and in the Virtual Machines of other languages as Lisp, Scheme or Ruby. It is also used as a complement to traditional reference counting collectors [11], like in the Perl VM or in the Python VM.

- *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. Finally, the references between the blocks and from the root set are updated to the new semispace. Allocation can be performed easily incrementing a pointer across the unused semispace. Since both the new objects and the copied ones are allocated into contiguous blocks, the memory shows little fragmentation. By counterpart, this strategy entails other disadvantages that arise in the reclaiming phase. The immortal data, during the time of an execution, are scanned and copied repeatedly with the consequent unproductive overhead. The available memory is reduced to half and most of the time this space is wasted.

- 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. In order to operate correctly the virtual machine has a write-barrier for instructions that can modify a pointer to an object. This way the collector is able to follow the references of objects in the mature generations on objects in the youngest generations without collecting the mature spaces. These references are saved in the *remembered set*. This task seems to entail an important overload and makes the difference relative to the non-generational ones during the mutator phase (see Section 4 for more details). Thus, collecting generations instead of the full heap produces a bigger amount of collections of much lesser cost.

We have experimented with a flexible nursery size generational collector, which is usually known as Appel collector [11]. The generational Appel collector divides the heap into two generations: nursery and mature. When an object is created, it is assigned to the youngest generation, the nursery space, in which all free space is contained. When the nursery is full, the collector copies all surviving objects to the mature space, and then reduces the nursery size by the same volume. It repeats this process until the nursery size falls below a certain threshold, at which point it performs a full heap collection. The collector returns the freed space to the nursery. In Jikes RVM, this threshold is fixed by default to 0.5 Mb. This strategy has been proved to be the best performing one for generational GCs [11].

The generational collector can manage the distinct generations with the same policy or assign to each one different strategies. We consider here two options. First, the *GenCopy*, which is a generational collector with semispace copying policy in both nursery and mature generation. The SUN Java 2 Standard Edition (J2SE) JVM by default uses a GC very similar to this one, with a Mark&Compact strategy in the mature generation. Second, the *GenMS*, which is a hybrid generational collector with semispace copying policy in the nursery and mark-and-sweep strategy in the mature generation. The Chives Virtual Machine [11] uses a hybrid generational collector but with three generations instead of only two.

- *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. Since it is not generational it avoids the overhead instructions of the write barriers in the mutator phase. It is the best performing considering the number of collections, but all collections are full heap; Thus, consuming more energy per collection.

In Jikes, these five collectors manage objects bigger than a certain threshold (by default 16K) in a special area. The JMTK reserves for larger objects a region of the heap, the Large Object Space (LOS). The new large objects are allocated in a Baker's tread-mill style [11], namely using a doubled linked list of fixed-size blocks. 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, although 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, they theoretically put less pressure in the processing power of the final embedded system and achieving good overall results (e.g., performance of memory hierarchy, L1 cache behavior, etc.). We will demonstrate that generational GCs achieve much better global results.

3.3 Case Studies

We have implemented the GCs presented in the previous subsection in the proposed experimental setup, and tested them while running the most representative benchmarks in the suite SPECjvm98 [17], modeling a complex memory hierarchy, representative of latest high-performance embedded devices. These benchmarks are launched as dynamic services and extensively use dynamic data allocation. The applications considered in our experiments are:

_222_mpegaudio: it is an MPEG audio decoder. It dynamically allocates to up to 8 MB and 2MB 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 Śif-thenŠ. It allocates 48 MB (plus 4 MB in the LOS) and most objects are short-lived.

_209_DB: builds an in-memory data base and operates on it. The data base is a 1 MB file, which is resident in memory. It allocates up to 224 MB of data.

_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-immortal objects (35 MB + 3 MB in the LOS).

_228_jack: it is a Java parser generator with lexical analysis. It allocates up to 480 MB of short lived data.

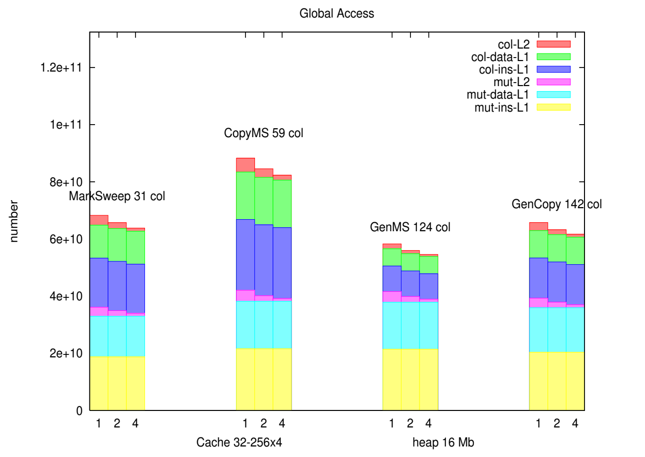


Figure 2: Cache memory hierarchy accesses for all studied GCs

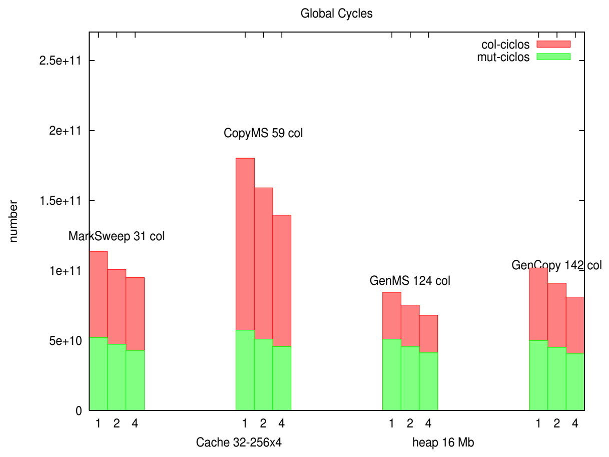


Figure 3: Global execution cycles for different GCs

_227_mtrt: it is a dual-threaded version of raytrace. It can allocate up to 355 MB of data.

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 size s100 is not representative of the different possible input sets and may give too much influence to the garbage collection phase with respect to realistic working conditions. The simulations of the different benchmarks correspond to multiple executions of the different benchmarks to reach an average execution time of 10 minutes, in order to reach a stationary situation regarding processing and memory utilization. Furthermore, to better explore the influence of garbage collection, the considered execution mode allowed us to run a predefined number of times the different benchmarks without detonating a memory flush between them. Finally, our results report average figures from 10 iterations of our experimental setup in each case, where all the results were very similar.

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 garbage collection mechanisms for high-performance embedded systems according to their key metrics (i.e., energy, power and performance of the memory subsystem). 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 and an off-chip SDRAM main memory, both distinguishing leakage and dynamic power in a .09μm technology node (Section 3). We have run our experiments with four different L1 sizes: 8K, 16K, 32K and 64K, using a block size of 32 bytes and testing associativity between 1-way and 4-ways, typical for high-performance embedded systems with low-power constraints. The experiments have been repeated using different blocks replacement policies, namely, *Least Recently Used (LRU)*, *First-In First-Out (FIFO)* and random, but only the LRU results are shown in the paper due to space limitations. Then, the L2 size is always fixed to 256 KB, with a basic block size of 128 bytes, using a 4-way associativity, and an LRU-based replacement policy. Finally, the main memory size is 16 MB.

4.1 Pressure of GCs in the cache memory organization

Figure 2 indicates the number of accesses of the mutator and collector to the different caches, differentiating both instructions and data accesses, and reporting the number of collections for each GC (31 for MarkSweep, 59 for CopyMS, 124 for GenMS and 142 for GenCopy). The configuration shown uses 32 KB for the L1 data and instruction caches, but the results are very similar for other cache sizes. These results sweep the associativity range from 1 to 4. As this figure shows, the mutator accesses are very similar for all the GCs, but the number of accesses to L1 caches (for both instructions and data of the mutator and collector) are always smaller in the generational collectors, i.e., approximately 33% less for GenMS than CopyMS. The reason is that, although the generational GCs have a much larger amount of collection phases, they are mainly local (i.e., covering in the end only a small percentage of the heap), while the non-generational collectors perform complete heap collections. Then, the number of accesses to the L2 caches decreases linearly when the L1 size increases, but there is a more important reduction effect in the number of accesses when the associativity of the L1 cache increases for a certain size. Indeed, for 32 KB, comparing a direct cache with a 4-way one, the number of L1 misses can vary up to 65% for the different configurations, while the GC algorithm does not seem to have a large influence. This conclusion is valid for all the tested L1 cache sizes, as a 4-way configuration achieves a 45% decrease with respect to a direct cache for 8 KB, 50% for 16 KB and 75% for 64 KB. Furthermore, L2 misses are smaller than 2% in all the cases, and L1 is the main source of power consumption reductions in the different memory configurations.

Then, as Figure 3 shows for 32 KB L1-caches (other L1 cache sizes show similar trends), the number of total cycles for the execution of the application and the virtual machine is less for generational collectors and decreases significantly in each type of GC when the associativity increases, namely, 50% less execution time for GenMS with respect to CopyMS. Similarly, Figure 4 shows that the energy consumption (distinguishing leakage and dynamic parts) for the different levels of the memory hierarchy (L1, L2 and main memory), for both the mutator and the GC, is significantly smaller in generational collectors than in more classic non-generational GCs.

However, regarding energy consumption, as Figure 4 indicates, the decrease in cache misses as the associativity increases is not large enough to compensate for the larger energy per access to caches with higher associativity. Therefore, the energy consumption increases more with associativity than execution time decreases.

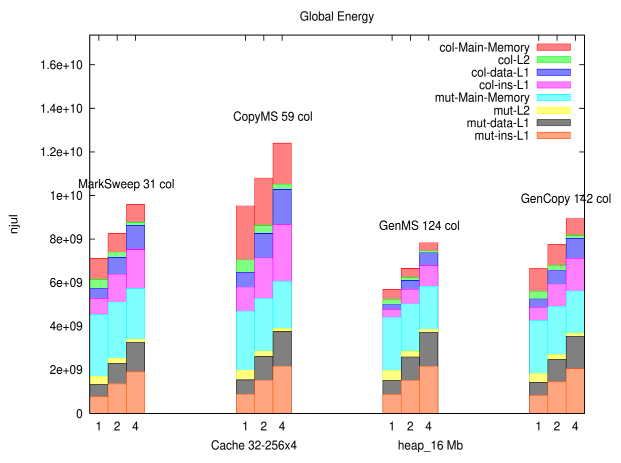


Figure 4: Energy breakdown for a 32KB L1-cache

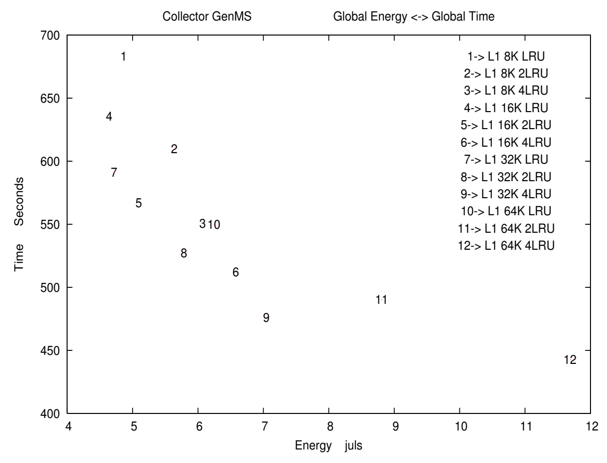


Figure 5: Memory energy consumption (in J.) versus Global Time (in sec.) for the GenMS collector

4.2 Exploration of energy-performance trade-offs for cache memory configuration in GCs

The previously observed conflicting trends between energy and performance for different memory configurations create a very interesting design space to be explored for each type of GC. In Figure 5, we present the different energy and performance trade-offs for the best collection algorithm, GenMS. In this figure, the total execution time (in seconds) is depicted against the global energy consumption of the memory (in J) for the twelve more relevant L1 configurations explored, namely, L1 data and instruction caches using associativity values of 1, 2 and 4, with different total sizes 8K, 16K, 32K and 64K. The Pareto-optimal curved is composed of the points 7, 5, 8, 6 and 9 in Figure 5. Similar Pareto-optimal curves have been obtained for the other GC algorithms explored. These results indicate that the lowest L1 cache energy solution is obtained using a direct cache of 32 KB (point 7 in the figure), followed by a 2-way cache of 16 KB (point 5) and a 2-way 32 KB (point 8). Conversely, the fastest solutions are always the ones corresponding to both sizes with 4-way associativity (points 6 and 9). Furthermore, the 4-way 64KB L1 cache (point 12) is only slightly faster than the previous one of 32 KB (less than 5%), but it consumes 40% more energy than the other solutions. Thus, although theoretically it can be considered part of the Pareto curve, it cannot be considered a good solution for embedded systems.

4.3 Leakage reduction opportunities in GCs

In the previous set of results, the generational GCs have shown to be the best ones regarding overall energy consumption. Indeed, this effect is the result of performing a large number of local garbage collections instead of global ones. Furthermore, this type of GCs use write barriers to remember the references to the explored parts of the mature region, the stored large objects (LOS region) and the immortal objects stored [11]. Therefore, the main memory devoted to store these elements is not accessed during the non-global collections, except the phase when the pointers to these regions are updated, which occurs when the part of the stack that stores the write barriers is processed. Hence, apart from the few main memory banks that store the nursery generation, nursery reserve and the part of the stack with the memory barrier references, the rest of the main memory banks can be put in low power mode, showing important leakage power reductions in this type of GCs.

First, as shown in the first column of Table 1 for GenMS, 70% of the main memory energy consumption of the GC is static. The second columns shows the percentage of saved leakage power. For GenMS, more than 60% of the leakage consumption of the GC can be saved. Thus, we reach main memory energy savings of 24%, as we use the low-power mode of the memory banks devoted to the mature generation, immortals and LOS during the local collection phases.

Furthermore, a similar (but even more aggressive) leakage power optimization approach, controlled by the JVM, can be applied on copy-based generational GCs. In this type of GCs, the available space is divided in two halves [11]. In the first half, a continuous assignment of objects is done, and the second half is used to copy the objects that survive the collection. Hence, in particular in the case of GenCopy, while the mutator is running, no access can occur to the main memory space reserved to copy the nursery generation and the mature generation, which enables shutting down these memory banks. Therefore, as Table 1 shows, more than 40% gains in overall main memory power reductions can be achieved for GenCopy, since this GC enables to shut-down the memory banks both in the case of the mutator and collector: using the low-power mode of the memory banks devoted to reserves during the mutator, as well as the memory banks devoted to the mature generation, immortals and LOS during the collector phase.

These results show very clear opportunities for leakage power reductions in JVM by using the knowledge of the specific GC mechanism used, in combination with the low-power technology features added to the latest SDRAM memories [14] available for high-performance embedded systems. Thus, further research in the future needs to be performed in this area.

4.4 Exploiting scratchpad memories in GCs

According to the misses penalties observed by the different cache memories (Section 4.1), in this section we assess the potential benefits (in performance increase and energy consumption reduction) of including a scratchpad memory that is controlled by the JVM. In particular, after a design time analysis, we have included in the JVM a control of the segregation of the GC virtual machine instructions stored in the different memories, such that we can use the scratchpad memory to store the most accessed methods of the collector while its execution occurs, and we can disable the scratchpad during the mutator phase. Thus, few conflicts exist between the mutator and the collector during the execution. Hence, Figure 6 and Figure 7 show the energy consumption and execution cycles of a 32-KB direct-mapped configuration (corresponding to the lowest energy solution in the design space) with respect to the same

	Leakage % (of energy)		Leakage Reduction % (of total leakage)		Final Overall Reduction %
	mutator	collector	mutator	collector	final
GenMS	20.7	70.1	0	62.5	24.6
GenCopy	20.4	49.5	23	59.2	41.2

Table 1: Summary of energy reduction of main memory due to leakage reduction for different GCs

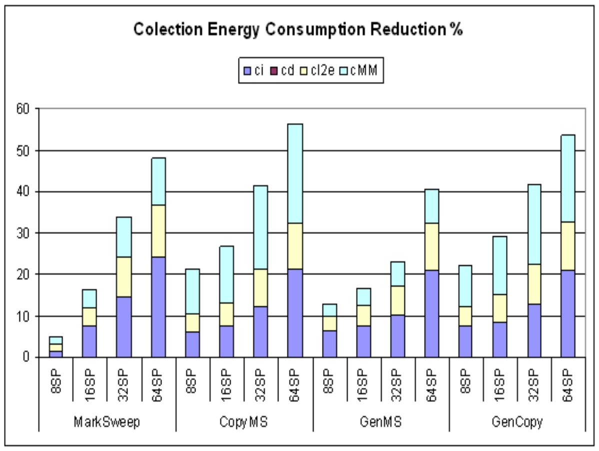


Figure 6: Memory energy consumption of the collector for different scratchpad sizes, with a 32 KB direct-mapped L1 cache

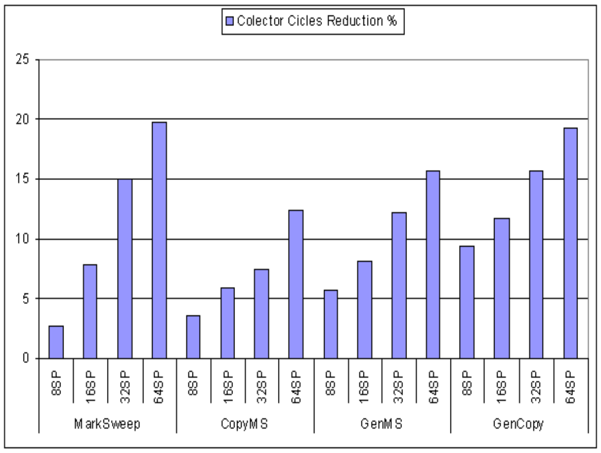


Figure 7: Cycles of the collector for different scratchpad sizes, with a 32 KB direct-mapped L1 cache

configuration after adding an scratchpad of 8 KB, 16 KB, 32 KB and 64 KB. The results for each GC algorithm are normalized to the only-cache configuration for the same GC algorithm. This approach achieves a significant reduction in the GC energy consumption (50%) and execution time (15%), which saturates with the 64-KB scratchpad configuration, since most of the GCs accesses are already captured in this scratchpad size, and further increases in the size of the scratchpad only produce a larger overhead per access, but not a significant reduction in the amount of L1 cache misses.

5. CONCLUSIONS

Due to the portable nature of Java applications, new high-performance embedded devices are now including Java in their designs as one of the most popular implementation languages. However, new complex dynamic embedded applications (e.g., multimedia) demand large processing power and possess specially energy-hungry features for these latest embedded devices. Therefore, JVMs should be designed trying to minimize energy consumption while preserving a minimum level of processing power. In this paper we have shown that the GC is a critical element in the overall amount of energy consumed by the JVM. Also, we have evaluated the importance for energy consumption of the interactions between the GC choice and the underlying memory hierarchy configuration. In particular, we have presented a complete energy-performance trade-off exploration of the different possibilities of memory hierarchies for high-performance embedded systems when used by state-of-the-art GCs. In addition, we consider the potential benefits of including an scratchpad memory in the memory hierarchy of high-performance embedded systems, controlled by the JVM to store critical code and data structures of the GCs, which optimize the energy and performance figures of the GCs, by 50% and 15%, respectively, in comparison to classical cache-based memory architectures. Furthermore, our experimental results have shown that up to 40% performance improvements and 41% energy reduction can be achieved in the main memory by efficiently exploiting the low-power mode in the banks of the latest memories. All in all, efficient low-power implementation of JVM can be achieved for high-performance embedded systems by exploiting the synergy between the specific GC algorithm used and the inclusion of power management schemes, exploiting the hardware features of latest embedded memories, controlled by the JVM.

6. REFERENCES

[1] V. Agarwal, et al. The effect of technology scaling on microarchitectural structures. Tech. Report TR2000-02, University of Texas at Austin, USA, 2002.
[2] T. Austin. Simple scalar llc, 2004. `http://simplescalar.com/`.
[3] S. Blackburn, et al. Myths and reality: The performance impact of garbage collection. In *Proc.ICMMCS*, 2004.
[4] S. Kim, et al. Energy-efficient Java execution using local memory and object co-location In *IEE Proc-CDT*, 2004.
[5] S Hu, et al. Impact of virtual execution environments on processor energy consumption and HW adaptation. In *Proc. VEE*, 2006.
[6] G. Chen, et al. Tuning garbage collection for reducing memory system energy in an embedded java environment. *ACM TECS*, November 2002.
[7] L. Eeckhout, et al. How java programs interact with virtual machines at the microarchitectural level. In *Proc. OOPSLA*, 2003.
[8] N. Nguyen, et al. Scratch-pad memory allocation without compiler support for java applications. In *Proc. CASES*, 2007.
[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*. J.Wiley and Sons, 2000.
[12] Kaffe. A Java virtual machine, 2005. `http://www.kaffe.org/`.
[13] D. Kegel. Building and testing gcc/glibc cross toolchains, 2004. `http://www.kegel.com/crosstool/`.
[14] ZBT@ sram and sdram products, 2006. `http://www.micron.com/`.
[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, et al. Using HW performance monitors to understand the behavior of java applications. In *Proc. VM*, 2004.
[19] D. Takahashi. *Java chips make a comeback*. Red Herring, 2001.
[20] The Univ. of Massachusetts Amherst and the Univ. of Texas. Dynamic simple scalar, 2004. `http://www-ali.cs.umass.edu/DSS/index.html`.