Journal of Systems Architecture 58 (2012) 61–72

Contents lists available at SciVerse ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Memory power optimization of Java-based embedded systems exploiting garbage collection information ☆

Jose Manuel Velasco [a,*], David Atienza [b,a], Katzalin Olcoz [a]

[a] DACYA-Complutense University of Madrid (UCM), Avda Complutense s/n, 28040 Madrid, Spain
[b] Embedded Systems Laboratory-EPFL, EPFL-STI-IEL-ESL, 1015 Lausanne, Switzerland

ARTICLE INFO

Article history:
Received 25 August 2009
Received in revised form 22 August 2011
Accepted 18 November 2011
Available online 3 December 2011

Keywords:
Garbage collection
Java
Memory exploration
Embedded systems

ABSTRACT

Nowadays, Java is used in all types of embedded devices. For these memory-constrained systems, the automatic dynamic memory manager (Garbage Collector or GC) has been always a key factor in terms of the Java Virtual Machine (JVM) performance. Moreover, in current embedded platforms, power consumption is becoming as important as performance. Thus, in this paper we present an 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. This is a starting point for a better understanding of the interactions between the Java applications, the memory hierarchy and the GC.

Hence, we subsequently present two techniques to reduce energy consumption on Java-based embedded systems, based on exploiting GC information. The first technique uses GC execution behavior to reduce leakage energy consumption taking advantage of the low-power mode of actual multi-banked SDRAM memories and it is intended for generational collectors. This technique can achieve a reduction up to 50% of SDRAM memory leakage.

The second technique involves the inclusion of a software-controlled (scratch-pad) memory that stores GC instructions under the JVM control to reduce the active energy consumption and also improve the performance of the target embedded system and it is aimed at all kind of garbage collectors. For this last technique we have experimented with two different approaches for selecting the GC code to be stored in the scratchpad memory: one static and one dynamic. Our experimental results show that the proposed dynamic scratchpad management approach for GCs enables up to 63% energy consumption reduction and 25% performance improvement during the collector phase, which means, in terms of JVM execution, a global reduction of 29% and 17% for energy and cycles, respectively.

Overall, this work outlines that the key for an efficient low-power implementation of Java Virtual Machines 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 GC behavior.

1. Introduction

According to recent studies of the programming community [32], Java is probably, one of the most popular programming languages in the world. We are used to the presence of Java in all kind of servers, personal desktops and, more recently, embedded systems. In fact, nowadays, the Java Standard Platform is being used in all types of embedded devices, ranging from routers, smart phones, 3G telecommunication devices and gas pumps [33,28]. 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 run in almost all 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 a major issue, 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 performed in Java-based systems to improve performance up to the level required in new embedded devices. This research has been mainly performed in the JVM. Much work has been focused on optimizing the execution time spent in the automatic object reclamation

☆ This work is partially supported by the Spanish Government Research Grants TIN2008-00508, CSD00C-07-20811 and the Swiss NSF Grant No. 200021-127282.

* Corresponding author.
E-mail addresses: mvelascc@fis.ucm.es (J.M. Velasco), david.atienza@epfl.ch (D. Atienza), katzalin@dacya.ucm.es (K. Olcoz).

1383-7621/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2011.11.002

or *Garbage Collector (GC)* subsystem, which is one of the main sources of overall performance degradation of the system [3,20,8,34]. 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 [6]. Thus, optimized (from the energy viewpoint) automatic dynamic memory reclamation mechanisms and methodologies to tune them have to be proposed for a complete integration of Java in the design of latest high-performance embedded systems, which include tight low-power constraints for portability purposes.

In this paper we present a detailed exploration of energy versus performance memory hierarchy trade-offs for embedded systems in presence of a broad range of different state-of-the-art GCs, which is the first step to define suitable memory management optimizations for energy-aware Java-based embedded systems. Next, we propose a static and a run-time energy management approach, which rely on two techniques to reduce the energy consumption of the whole memory system, namely: (1) using specific GC information for leakage energy consumption reduction and (2) using an instruction scratchpad memory directed by the virtual machine to store the most used methods of the GC, thus reducing the active energy consumed by the JVM. We have tried two approaches for this last technique: the first approach uses a scratchpad memory assignment at compile time to store critical code of the GC. The second approach performs an additional run-time code reallocation to improve the static assignment of Java methods to scratch-pad memories by the JVM, adjusting at run-time the assigned methods based on a continuous history monitoring of recent GC behavior. Our results show up to 45% leakage reduction with negligible performance impact, as well as 50% overall energy and 30% performance improvements for the GC with respect to classical cache-based JVM memory architectures [10,3,6].

The rest of the paper is organized as follows. In Section 2 we summarize related work in the area of JVM design and GC optimizations. In Section 3 we describe the experimental setup used 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 memory hierarchy exploration and in Section 5 and 6 the main proposals for energy optimization based on the analysis of the memory behavior of the garbage collectors. Finally, in Section 7 we summarize the main conclusions of this work.

2. Related work

Nowadays a 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 [15]. A substantial amount of research on GC policies and architectural exploration has mainly focused on performance [3,20,10]. Our work extends this research for an overall memory hierarchy exploration of high-performance embedded systems.

Eeckout et al. [8] investigate the micro-architectural implications of several virtual machines including Jikes. In their work, each virtual machine has a different GC, so their results are not consistent with respect to memory management. Similarly, Sweeney et al. [27] 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 explore in this work. In a recent study, Hu and John [10] explore 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. Thus, the static leakage is decreased by turning off the unnecessary banks. So, this approach is suited for applications with a small memory footprint relative to embedded system memory size. Then, a hybrid Mark&Sweep and reference counting collector is developed by Griffin et al. [9]. This hybrid collector reduces the number of collections compared to a classical Mark&Sweep strategy and therefore the impact of garbage collection, in terms of energy consumption, is also reduced. Nevertheless, the benefit of a reference counting collector is only limited to applications with no cyclic data structures.

In addition, several works are based on dynamic profiling of the behavior of data [17], instructions [13] or both [35] to choose the scratchpad memory contents. These techniques are complementary to our strategy, which relies on the fact that garbage collection is a highly predictable virtual machine phase because its usage of methods is largely application independent. Then, closer to this work, in [20] it is proposed a scratchpad allocation scheme implemented inside the JVM without compiler support, but the authors do not analyze the effects in power consumption and do not explore different GC algorithms, as we perform in this work. In this regard, a more complete study regarding energy consumption is performed in [19], which proposes two implementation strategies for allocating objects that can significantly reduce the memory system energy consumption of Java applications. The first strategy uses part of on-chip memory resources as a local memory to achieve better performance than a cache-only architecture. Thus, Java application objects are allocated using an annotation-based approach to improve the memory system energy consumption. Then, the second strategy proposes the use of 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 proposed object co-location approach and the use of local memories to reduce energy consumption is complementary to our memory hierarchy exploration and GC allocation methods. Therefore, these mechanisms can be additionally used in combination to our proposed memory hierarchy customization (cf. Section 6) for high-performance embedded systems.

In addition, Badea et al. [2] focused on reducing the impact of the just-in-time virtual machine compiler. This is achieved primarily through the use of *superoperators*, namely, new bytecodes that combine a sequence of standard bytecodes and optimize their combined execution. In fact, this approach is also complementary to ours as the use of superoperators could potentially enable more virtual machine functionally to be stored on our proposed scratchpad memory.

Finally, in [34] it has been recently performed an initial study of GC behavior with respect to the memory hierarchies of embedded systems and potential energy savings of software-controlled memories with respect to cache-based systems. In this work we extend this preliminary exploration of memory hierarchies and GCs performance for high-performance embedded systems, as well as quantify the benefits of using static and run-time GC code allocation for energy optimization, controlled by the underlying JVM. Thus, we achieve 15–20% additional energy consumption savings by suitably selecting the most appropriate GC and JVM methods for each phase of the high-performance embedded system execution.

3. Experimental framework

In this section we first detail the simulation environment used to obtain detailed memory energy consumption of the JVM and performance estimation (for both the application and the collector phase), which are 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. Java-based simulation environment for embedded systems

Our simulation environment consists of three different parts. First, the detailed simulations of our case studies have been obtained after modifying and instrumenting the code of Jikes *Research Virtual Machine (RVM)* from the IBM Watson Research Center [11]. 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 [12]. Jikes is 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) [11].

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 [1]. 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 cache simulator. Therefore, as proposed in [18], we have included a cross-compiler to be able to run our whole Jikes-DSS system onto the 32-bit x86-based platform available for the experiments instead of the PowerPC-based system traditionally used for DSS.

Finally, after the simulation in our Jikes-DSS environment, energy figures are calculated with an updated version (v5.1) of the CACTI model [31], 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 in this paper, we use the 90 nm technology node. Regarding 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 for a 16 MB mobile SDRAM [29,30].

3.2. Studied state-of-the-art GCs

In this section 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 [15] for a complete overview of GC techniques used in our experiments with Jikes [11].

First, we need to distinguish between the garbage *collector* and the *mutator*, as described in Dijkstra's terminology [15]. 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 (class loader, code interpreter, code compiler, scheduler, I/O, etc.). 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* [15]. 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 from garbage, the tracing strategy relies on determining which objects are not pointed to by any living object. To this end, it needs to traverse a whole relationship graph through the memory recursively [15]. 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* [15]. This means that the sweep is delayed up to the allocation phase. This is a classical collector implemented in several typical embedded JVMs, such as, Kaffe [16], JamVM [24] or Kissme [25], and even in the virtual machines of other languages as Lisp, Scheme or Ruby. It is also used as a complement to traditional reference counting collectors [15], 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. Thus, when there are tight memory constraints, it is by far the worst GC. In our experiments the number of cycles and global energy of SS are much bigger ($3\times$ more) than those of the remaining algorithms and so we did not include this GC in the figures.
- *Generational Collectors (or GenMS and GenCopy in our experiments):* 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 JVM 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 overhead and makes the difference relative to the non-generational ones during the mutator phase (see Section 4 for more details). However, collecting generations instead of the full heap produces a larger amount of collections of less cost each.

We have experimented with a flexible nursery size generational collector, which is usually known as Appel collector [15]. 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 proven to be the best regarding performance for generational GCs [34,15].

The generational collector can manage the distinct generations with the same policy or assign to each one different strategies. The nursery always uses copying collectors because objects that survive a collection are always copied to the mature generation. For the mature generation 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, which employs 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 [15] 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, as our results show (cf. Section 4).

In Jikes, these five collectors (MS, SS, CopyMS, GenMS and GenCopy) manage objects bigger than a certain threshold (by default 16 K) 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 [15], 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 demonstrate in this work that generational GCs achieve significantly better global results than more classical GCs used in the JVMs proposed for embedded systems [19,6,15].

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 benchmarks in the suite SPECjvm98 [26] and some additional applications, 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:

- Sixlegs Java PNG Decoder (javapng) [14]: it decodes PNG images. It supports all valid bit depths (grayscale/color), interlacing, paletted images, alpha channel/transparency, gamma correction, etc. The set of images belongs to PngSuite [23] and BrokenSuite [5]. It dynamically allocates to up to 36 MB, none of them in the LOS.
- Java Blowfish [7]: it encrypts a text file (first 5 chapters from Don Quixote [22]) using symmetric block cypher with a variable-length key. It dynamically allocates objects in the LOS (up to 4 MB) and uses up to 76 MB of small objects.
- Java Crypto from Bouncy Castle [4] package (JDK 1.2). It encrypts and decrypts a text file (first 5 chapters from Don Quixote), and allocates up to 90 MB in small objects and 2 MB in the LOS.
- 222 mpegaudio: it is an MPEG audio decoder. The workload consists of about 4 MB of audio data. The dynamic data is about 28 MB. This application does not allocate data 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 4 MB of small objects.
- 202 Jess: it is the Java version of an expert shell system using NASA CLIPS. It is essentially a combination of if–then structures. 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 large amount 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.
- 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, due to the complexity of the simulation framework. The simulations of the different benchmarks correspond to multiple executions of each benchmark to reach a total execution time of 10 min, which corresponds to reaching a stationary situation for each benchmark 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 forcing 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 (variations of less than 4%).

4. Analysis of embedded memory hierarchy influence in GC efficiency

This section shows the application of the previously explained experimental setup (see Section 3 for more details) to perform a detailed 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 90 nm technology node (Section 3). We have run our experiments with four different L1 sizes: 8, 16, 32 and 64 K, using a block size of 32 bytes and testing associativity between 1-way and 4-ways, which are typical for high-performance embedded systems with low-power constraints [20,6,10]. The experiments have been repeated using different blocks replace-

ment policies, namely, *Least Recently Used (LRU)*, *First-In First-Out (FIFO)* and random, but only the best results (LRU policy) are shown in the paper due to space limitations (identical trends were observed for the other policies). 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. Dependence of GC algorithms on the cache memory organization

Fig. 1 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, namely 30 for Mark&Sweep, 57 for CopyMS, 121 for GenMS and 141 for GenCopy (but only three and eight are global collections for GenMS and GenCopy, respectively). 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 ways. 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 Fig. 2 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, namely, 50% less execution time for GenMS with respect to CopyMS. Besides, the number of cycles decreases significantly in each type of GC when the associativity increases.

Similarly, Fig. 3 shows that the energy consumption (including leakage and dynamic parts) for the different levels of the memory hierarchy (L1, L2 and main memory), for both the mutator (mut- in Fig. 3) and the GC (col-), is significantly smaller in generational collectors than in more classic non-generational GCs. Furthermore, as Fig. 3 also 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 overall energy consumption increases more with associativity than execution time decreases.

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 Fig. 4, we present the different energy and performance trade-offs for the best collection algorithm, i.e., GenMS. In this figure, the total execution time (in seconds) is depicted against the global energy consumption of the memory (in Joules) for the twelve more relevant L1 configurations explored, namely, L1 data and instruction caches using associativity values of 1, 2 and 4, with different sizes of 8, 16, 32 and 64 KB.

The Pareto-optimal curve is composed of the points: 32 K size and direct associativity (light diamond), 16 K and 2 ways (big dark asterisk), 32 K and 2 ways (light square), 16 K and 4 ways (big dark cross) and 32 K and 4 ways (light triangle) in Fig. 4. 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 (light diamond the figure), followed by a 2-way cache of 16 KB (big dark asterisk) and a 2-way 32 KB (light square). Conversely, the fastest solutions are always the ones corresponding to both sizes with 4-way associativity (big dark cross and light triangle). Furthermore, the 4-way 64 KB L1 cache (small light cross) 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.

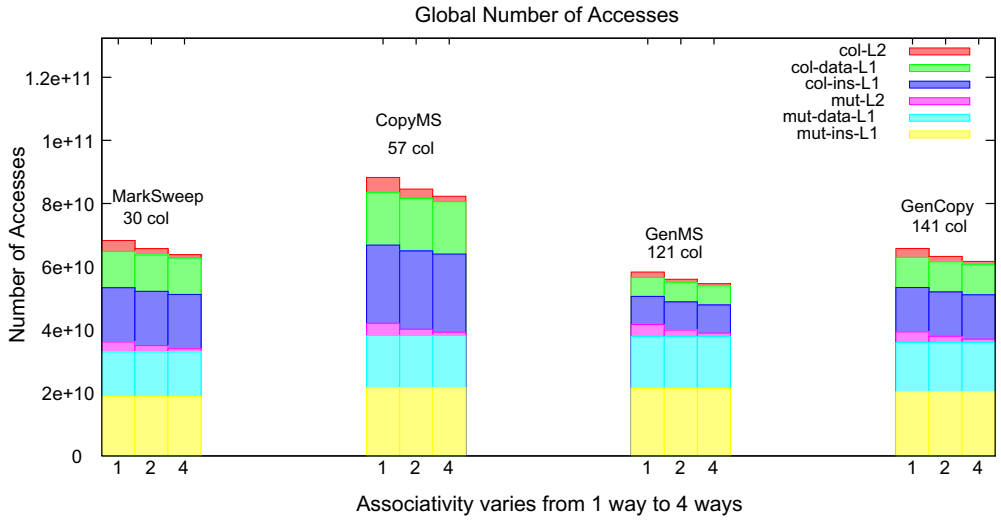

Fig. 1. Cache memory hierarchy accesses for all studied GCs.

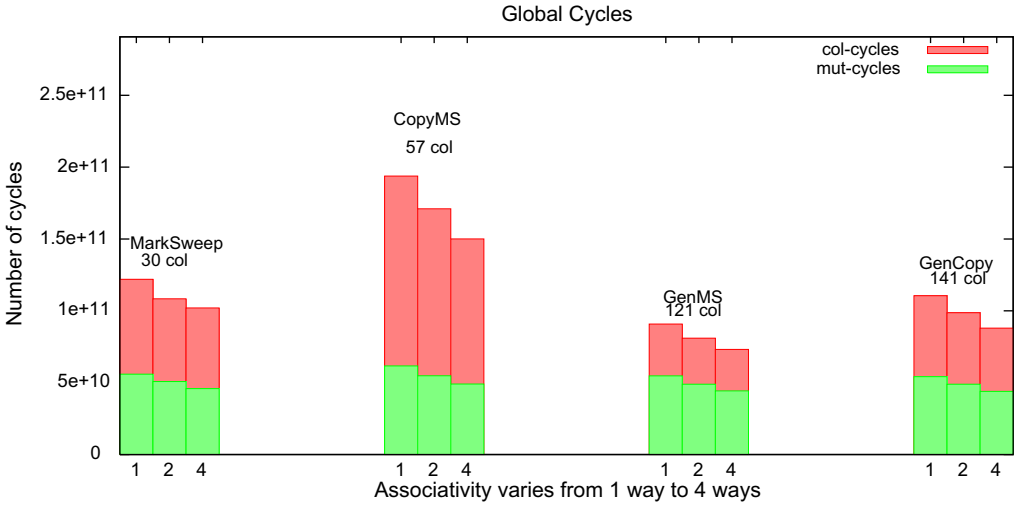

Fig. 2. Global execution cycles for different GCs.

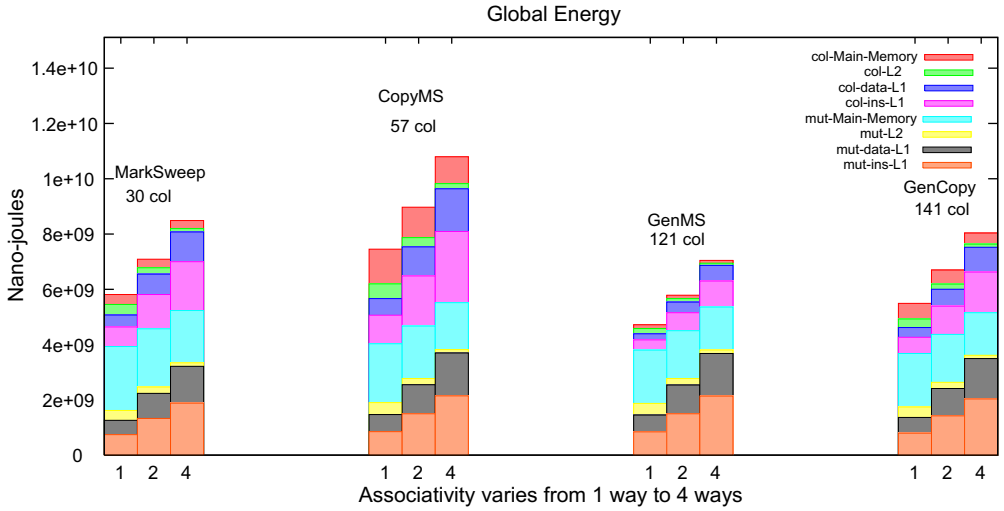

Fig. 3. Energy breakdown for a 32 KB L1-cache.

5. Leakage reduction opportunities in generational GC

Once we have analyzed the design space we propose to reduce the energy consumption figures of GCs due to leakage by taking advantage of the knowledge of the allocation and GC behavior of the JVM. Moreover, this energy reduction is accomplished with negligible performance degradation (less than 2% in overall JVM performance).

5.1. Analysis of static energy consumption

We have measured the percentage of the global consumption of main memory due to leakage for all L1 cache sizes (8, 16, 32 and 64 KB) and associativities (1, 2 and 4). We only present the results for the generational GCs, which have shown to be the best ones regarding overall energy consumption in high-performance embedded systems (cf. Section 4). For GenMS the percentage of global energy of main memory due to leakage varies from 20% to 22% for the mutator and from 70% to 75% for the collector, and the bigger values correspond to the bigger caches with higher associativities. For GenCopy the figures are from 20% to 21% for the mutator and from 46% to 51% for the collector. Table 1 depicts the leakage figures for the 32 KB direct mapped configuration.

From these results, we can draw two conclusions. On the one hand, the amount of static energy consumption during the collector phase is relevant, and much bigger than for the mutator phase. So, it is worth trying to reduce it. On the other hand, the static energy depends strongly on the collector algorithm, being larger for the GCs based on the mark and sweep strategy [15] than for the copying collectors. The reason is twofold. First, a higher number of global collections of GenCopy generates more memory accesses and energy consumption. Second, copying collectors not only scan the mature space to find live objects but also copy all of them. Thus, in the following we study the behavior of both types of generational GCs to find opportunities for static energy reduction in each case.

5.2. Leakage reduction analysis per type of generational GC

As explained in Section 3.2, generational GCs divide the heap in different areas. The areas are the following ones: immortal, *Large Object Space (LOS)* or objects bigger than a certain threshold, ma-

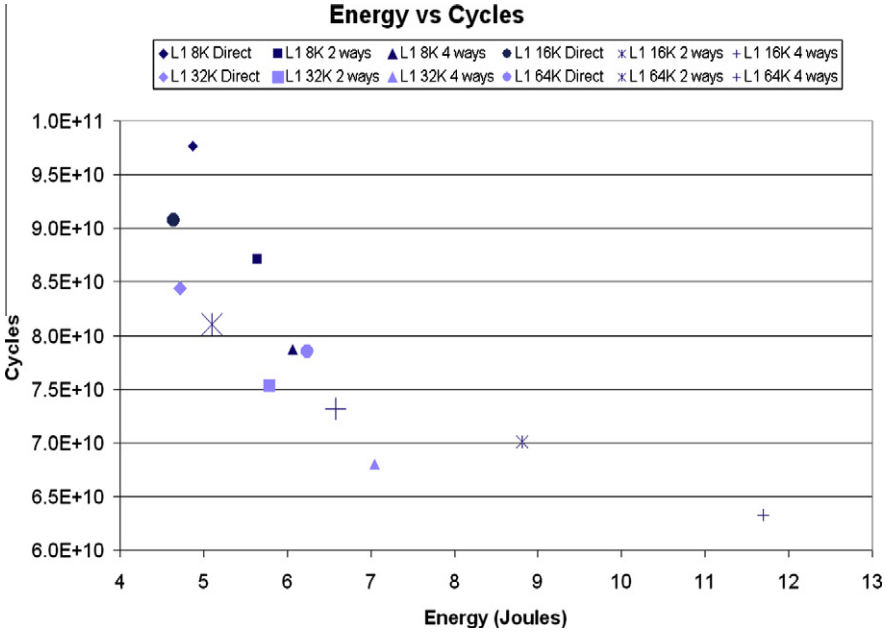

Fig. 4. Design space, including the Pareto curve points, of memory energy consumption (in Joules) versus global execution time (in seconds) for the GenMS collector.

Table 1
Summary of leakage and system energy consumption reduction for generational GCs with a main memory size of 16 MB and 8 banks.

	Leakage% (of energy)		Leakage reduction% (of total leakage)		Final reduction%	
	Mutator	Collector	Mutator	Collector	Leakage	Global energy
GenMS	20.7	70.1	8	22.5	10.8	2.6
GenCopy	20.4	49.5	23	69.2	42.1	11.3

ture and nursery. There is an additional zone, called nursery reserve, equal to the size of the nursery, which is kept free for allocating objects that survive a garbage collection.

During minor collections only the nursery is scanned to find live objects and copy them in the nursery reserve, which becomes part of the mature space. Therefore, the immortal, LOS and mature areas are not accessed during minor collections, thanks to generational write barriers (cf. Section 3.2), but they still consume static energy. As a result, if the allocation of mature objects is contiguous, as for GenCopy, we can put the banks holding this data in low-power mode during minor collections, thus avoiding leakage power consumption. On the contrary, for GenMS the mature area is not contiguous, so only the banks holding immortal and LOS objects can be put in low power mode. In any case, we must turn on these memory banks before the GC processes the remembered set, so that the references in mature generations can be updated without performance penalties. Overall, this predictive wake-up can be estimated correctly based on the executed garbage collection methods, so this phase only entails in the end between 6% and 9% of the total collection time (less than 2% in overall execution time). As shown in Table 1, this technique reduces leakage energy consumption during collection by 22% for GenMS and 69% for GenCopy.

Furthermore, a similar leakage power optimization approach, controlled by the JVM, can be applied on copy-based generational GCs, during the mutator phase, because they allocate space for the nursery reserve contiguously. Hence, in particular in the case of GenCopy, while the mutator is running, no access occurs 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 depicts, 23% of leakage can be saved during the mutator execution phase. In the case of GenMS, this technique can be applied during the period of execution time before the first global collection. After this first collection, the nursery reserve is no longer contiguous because it is interleaved with mature objects. Thus, the total benefit for GenMS during the mutator phase is 8%.

In Table 1 we can see a summary of energy reduction of main memory due to leakage reduction for generational collectors using this technique and with a 16 MB main memory size. In the first column we show the name of the generational collectors. In the second and third column, we show the percentage of total main memory consumption due to leakage during mutator and collector phase. That is to say: for the GenMS collector and during the collector phase, 70.1% of the energy consumed is due to leakage loses and only 29.9% is due to dynamic accesses. During the mutator phase the situation is quite different: 20.7% is due to leakage and 79.3% is due to dynamic accesses. In the fourth and fifth columns, we show the percentage of leakage reduction that can be achieved with this technique, distinguishing between the mutator and collector phase. That is to say: for the GenMS collector and during the collector phase, this technique reduces the leakage up to a 22.5% (69.2% for GenCopy). During the mutator phase, the leakage is reduced up to a 8% for GenMS and up to 23% for GenCopy. In the sixth column, we can observe the final leakage reduction percentage, which is 10% of the total leakage for GenMS but reaches up to 42% for GenCopy. Finally, in the last column it is shown the reduction percentage in the total main memory energy consumption. So, this technique itself can achieve a reduction up to 11.3% of the total energy consumed in the main memory for GenCopy and up to 2.6%

for GenMS. In Table 2 we show the same values in the case the main memory size is 32 MB. As we can see, the increase of main memory size means a bigger reserved space size and a bigger mature generation size and, therefore, bigger leakage figures. This way, it is likely that the relevance of this technique scales with future main memory sizes.

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 [29] available for high-performance embedded systems. Furthermore, these results indicate that these leakage energy savings would be even larger with bigger main memories, as expected to be included in forthcoming high-performance embedded systems. In addition, they show that Mark-and-Compact GC strategies [15], which provide contiguous allocation of mature objects, can be beneficial in the future to fully exploit leakage reduction in embedded systems with larger main memories.

6. Exploiting scratchpad memories for active energy consumption reduction in GCs

The exploration of the memory hierarchy of high-performance embedded systems (cf. Section 4.2) has indicated that energy increases with memory size and associativity more than execution time and L1 cache misses decrease. Indeed none of the L1 configurations with 64 KB was part of the Pareto curve of Fig. 4, and the 4-way configurations with 16 and 32 KB were the most energy consuming points in the Pareto curve. Therefore, in this section we explore power-efficient ways of increasing performance, i.e., without increasing cache size and associativity. One way for doing so is the use of a software controlled *scratchpad (SP)* memory for JVM instructions. We can select the JVM instructions to be included in the SP using two different approaches, namely:

- Statically, choosing the instructions among all the JVM code, which can lead to under-utilization of the SP memory depending on the dynamic Java application behavior.
- Dynamically, with the inherent profiling overhead, but with a more efficient use of the SP memory.

Thus, in this paper, we explore the different trade-offs in order to exploit the fact that garbage collection is predictable and mostly application independent [10,8]. In particular, based on our exploration of the JVM, we have observed that:

- From all the different JVM tasks: class loading, dynamic linking, code compiling, threads scheduling, type verification, etc., and in the context of limited memory devices, the GC has one of the biggest percentage in both execution time and energy consumption.
- Once the memory manager runs out of memory, and before the garbage collection starts, the JVM must execute several management tasks. Thus, the JVM is able to anticipate the start of a garbage collection operation several hundreds of thousands of instructions in advance. The same situation occurs from the moment the garbage collection finishes until the moment the application resumes its execution. As a result, there is enough time to turn on/off a scratchpad memory (as well replacing its content) before any new mutator or collector phase actually start.
- The GC behavior does not depend significantly on application profile.

Therefore, the GC code is one of the best candidates to store in a software- controlled or SP memory. Since SP memories are smaller, less energy consuming and faster than cache memories, this approach is feasible if the size of GC methods is small enough to fit in a SP memory of reasonable size and we find locality phases (in execution time or memory address regions) in the used GC methods at run-time. To this end, we have measured the code size of the different GCs used in our experiments and found out that even though the size of all collector methods is about 350 KB, only a small number of them is responsible for the majority of the instruction executed. Even in the worst case, which corresponds to GenMS, just using 8 KB we can store the methods of 35.8% of the accessed instructions, with 16 KB we can cover 53.8% of instructions, with 32 KB we can reach 73.6% of instructions and with 64 KB we can cover more than 93% of the GC instructions. Thus, by adding up to 64 KB of scratchpad memory, the final memory hierarchy can be very power-efficient with respect to increasing the cache memory size. Note that a configuration with 32 KB of L1 for data and another 32 KB of L1 for instructions and 64 KB for scratchpad is slightly smaller in area than a configuration with 64 KB of L1 cache [31,20], which did not result in a good energy-performance trade-off.

The selection of methods to store in the scratchpad is defined at compile time. Thus, when compiling the JVM, the selected methods are mapped contiguously in the range of memory addresses that belong to the scratchpad memory. In addition, we have also considered the option of using a SP memory to allocate the most accessed methods of the mutator phase in the JVM as well as the GC methods, or to allocate data structures of the JVM. In this regard, our experiments have indicated that the methods of the JVM, outside the GC ones, that could exploit the SP are those related to the compilation and dynamic class loading tasks, which consume more cycles and memory. Nonetheless, the reduced size of the GCs methods and their higher locality and predictability with respect to the dynamic compilation classes related to the mutator imply that an optimal energy reduction can be achieved by only considering allocating GC methods in scratchpad memories. Thus, we focus on the GC methods from now on.

6.1. Static SP assignment of GC methods

Our first approach is to use a SP for the most frequently used GC methods and to store the rest of them in the L1 instruction cache. Then, during the mutator phase the SP is switched to low-power mode by our modified JVM. Since the content of the scratchpad is always the same, this assignment can be statically decided, i.e., at design time. To this end, the most frequently used methods (on average) are selected based on profiling of the executed benchmarks (cf. Section 3.3).

Table 2
Summary of leakage and system energy consumption reduction for generational GCs with a main memory size of 32 MB and 16 banks.

	Leakage% (of energy)		Leakage reduction% (of total leakage)		Final reduction%	
	Mutator	Collector	Mutator	Collector	Leakage	Global energy
GenMS	33.8	76.1	10.5	28.2	12.2	4.3
GenCopy	31.3	60.5	38.4	80.9	52.3	19

This simple static approach reduces both GC time and energy for all L1 sizes and associativities, particularly when 64 KB of scratchpad are added to the memory system. We only show the results of adding 64 KB of SP to the points in the Pareto-optimal curve for GenMS of Fig. 4, as this point captures almost all the critical GC methods. For the other GC algorithms, the gains are even larger because these GCs consume more power and execution time as we will show.

The new figures of execution cycles and energy consumption after adding the SP are depicted in Fig. 5, represented in a light shade and labeled SP-64 K. Only time and energy during collection are represented in this figure because the mutator values remain the same. As this figure shows, the configurations with 16 KB of L1 cache are no longer part of the Pareto curve. In fact, the configurations with 32 KB of L1 have the best energy-performance tradeoffs, because the GC energy is reduced between 30% and 40% and the GC cycles are reduced between 20% and 25%. For example, the addition of a 64 KB scratchpad to a system with 32 KB of direct mapped L1 for instruction and data produces a 25% reduction in the number of cycles of the collector and a 40% reduction in GC energy consumption. Since the energy consumed by GenMS with the cache-only memory configuration is minimal for a 32 KB 1-way L1, this 40% GC energy reduction produces almost 8% global energy reduction. However, for the same configuration with GenCopy (the second best GC), the 54% collector energy reduction obtained adding a scratchpad reaches almost a 20% global energy reduction.

6.2. Dynamic SP allocation for GC methods

Although garbage collection is mostly application independent, there are some differences in the most used methods depending on the number of objects that survive a collection and depending on whether the collection is minor or not. Thus, in this section we dynamically adapt the allocated GC methods into the SP memory according to the current run-time behavior captured by the JVM information. Based on a compile-time analysis of the profiling of GC methods, we have divided the GC methods into three different categories:

- Mature: the most used methods during global collections.
- Copy: the most used methods for minor collections with many survivors.
- Mark: the most used methods for minor collections with few survivors.

Then, at run-time we dynamically select the actual SP content based on specific application behavior, as shown in the pseudocode of Fig. 6. As this figure indicates, the algorithm starts by preparing the garbage collection (globalPrepare part), thus the JVM needs to load a selected set of GC methods in the SP memory. To this end, if a global collection occurs, the JVM loads the set of methods intended for the mature space (imageMature). Conversely, if the collection is limited to the nursery space (else branch), the JVM can choose what to load in the SP (scratchpadImage) between two sets of methods, which have been monitored and selected in the previous garbage collection. Finally, this first part of the execution of the algorithm indicates the JVM to record the amount of memory used by the GC in the nursery and mature space before the collection starts.

Once the garbage collection process has finished (globalRelease part), in order to select the contents of the two possible sets of methods to be chosen from in the next collection, the algorithm calculates the amount of copied memory as the difference between the memory used by the mature space after and before the garbage collection. Then, we calculate the ratio of memory currently being used in the mature space by dividing the amount of copied memory by the initial amount of assigned memory in the nursery before collection, and then we increase our own internal counter to keep track of when was the last time that we updated the list of methods to be loaded (profileCounter). Every N collections (set to five in our experiments after multiple tests), indicated by our internal counter, the JVM obtains an average ratio. This ratio indicates which methods to load in the SP memory during nursery collections. In case the ratio is bigger than a certain threshold (thresholdRatio, fixed to 20% of the copied memory with respect to the assigned memory), the JVM loads the most used methods for minor garbage collections with many survivors (imageCopy set). Otherwise, the JVM loads the most used methods for minor collections with few survivors (imageMark set). At the beginning, before the first ratio can be calculated, the algorithm indicates the JVM to load the imageMark set of methods, as there are no elements and, hence, it falls indirectly in the category with few survivors.

In order to assess the efficiency of our proposed algorithm and run-time SP memory loading approach we have compared it with

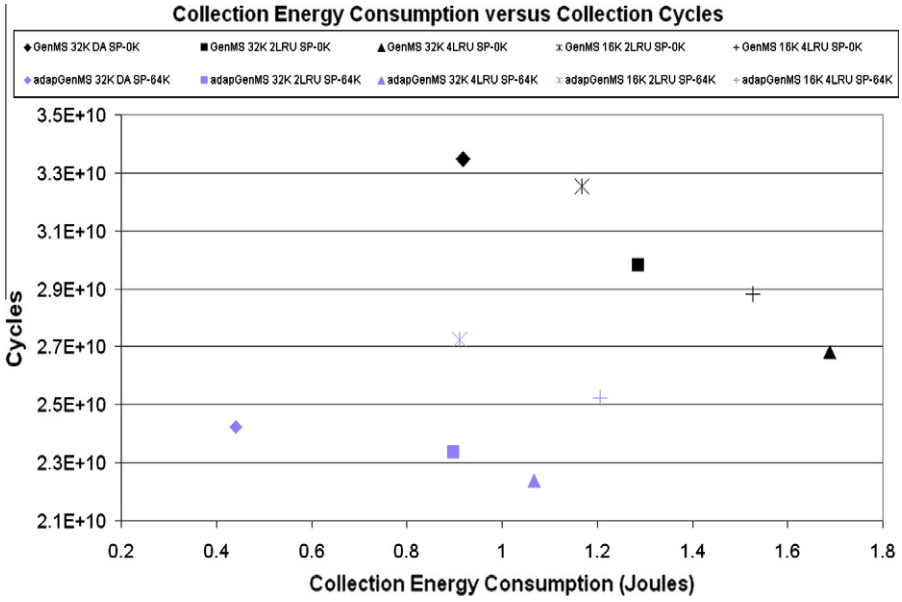

Fig. 5. New design space, including the new Pareto curve points, after including a scratchpad memory.

```
globalPrepare
-----------------------------------------------------------------------
if (globalCollection==true){
        load(imageMature)
}
else{
load(scratchpadImage)
recordAllocationNurserySizeBeforeCollection
recordMatureSizeBeforeCollection
}
-----------------------------------------------------------------------

globalRelease
-----------------------------------------------------------------------
recordMatureSizeAfterCollection
copiedMemory= MatureSizeAfterCollection - MatureSizeBeforeCollection
lastRatio = copiedMemory / allocationNurserySizeBeforeCollection
sumRatio += lastRatio
profileCounter ++

if  (profileCounter== N) {
                newAverageRatio = sumRatio/ N
                sumRatio =0
                profileCounter = 0

        if (newAverageRatio<=thresholdRatio){
                scratchpadImage=imageMark
        }
        else{
                scratchpadImage=imageCopy
        }
}
-----------------------------------------------------------------------
```

Fig. 6. Pseudocode algorithm for runtime selection of GC methods.

the possible optimum selection of methods to store in the SP during GC, assuming a full knowledge of future application and JVM behavior, i.e., as an oracle. To this end, for every benchmark we have obtained the optimum set of GC methods to store in the SP by exhaustive off-line profiling. The results obtained with this approach are labeled as Off-line in Fig. 7. We then compare them with the static approach presented in Section 6.1, labeled as Average in this figure, and with the run- time dynamic selection approach we have just introduced in this section, labeled as Adaptive. The results in Fig. 7 correspond to the lowest energy solution (L1 of 32 KB, direct mapped) of the Pareto curve of memory configuration. The reduction in cycles is not shown because it follows a similar trend. The results for each GC algorithm are normalized to the only-cache configuration for the same GC algorithm.

As Fig. 7 depicts, for the GenCopy GC, the proposed run-time adaptive approach is very close to the optimum off-line oracle of future GC behavior for every size of SP (only 1% worse). In the case of the GenMS algorithm, the adaptive approach reaches also near-optimal results for the 32 and 64 KB SP memory sizes (only 3% worse than the oracle), being worse (up to 40%) than the optimum off-line approach for smaller SP memory sizes (8 and 16 KB). The reason is that the garbage collection algorithm of GenMS is more complex and includes a very variable set of methods to be selected at each moment in time of its execution, which is much more difficult to capture with just the average history. Nonetheless, note that a very similar behavior degradation would occur if caches of only 8 or 16 KB are used instead of 32 KB, as considered in the baseline embedded systems. Overall, the main conclusion is that by just adding 64 KB of SP memory, it is possible to reduce by 50% the consumed energy and by 30% the execution time for both generational GCs, which results in a much more power-efficient way of increasing memory than increasing cache size and associativity.

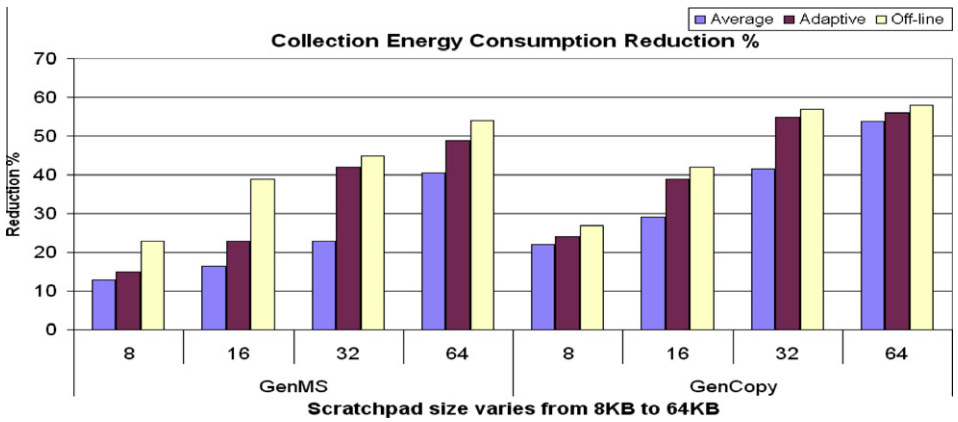

Fig. 7. Comparisons of method selection algorithms for different scratchpad sizes, and with a 32 KB direct-mapped L1 cache.

Table 3
Summary of final reduction achieved during collection and global execution, for both energy and cycles, using a 64 KB scratchpad and a 32 KB L1 cache.

Collector	Collection %			Reduction%			
				Collection		Global	
	Access	Energy	Cycles	Energy	Cycles	Energy	Cycles
M&S	47	33	54	52	34	17	18
CopyMS	52	46	68	63	25	29	17
GenMS	29	20	40	44	27	9	11
GenCopy	40	33	51	57	32	19	16

Table 3 presents summary of how the reductions during collector phase (for both energy consumption and execution cycles) translate into global reductions for the four collectors with the adaptive strategy. The L1 size is 32 KB, the best energy point in the Pareto curve, and the scratchpad size is 64 KB. First, we show the collection percentage relative to global values for access, energy consumption and cycles. Second, we show the reductions achieved using a scratchpad memory during the collector phase for energy and cycles. And finally, how these values translates into global reductions. As we can see, in the case of generational collectors (best suited collectors), this technique reduces global energy consumption and global cycles about a 10% for GenMS and more than a 15% for GenCopy.

7. 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) require large processing requirements and imply very energy-hungry features for 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 detailed 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 a scratchpad memory in the memory hierarchy of high-performance embedded systems, controlled by the JVM, to store critical code of the GCs, which optimize the energy and performance figures of the GCs up to 50% and 40%, respectively, by using a run-time adaptive approach in comparison to classical cache-based memory architectures. Furthermore, our experimental results have shown that more than 45% energy consumption 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, this results outline that 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 directly by the JVM.

References

[1] T. Austin, Simple scalar llc, 2004, <http://simplescalar.com/>.
[2] C. Badea, A. Nicolau, A. Veidenbaum, Impact of jvm superoperators on energy consumption in resource-constrained embedded systems, SIGPLAN Not. 43 (7) (2008) 23–30.
[3] S.M. Blackburn, P. Cheng, K.S. Mckinley, Myths and realities: The performance impact of garbage collection. In: In Proceedings of the ACM Conference on Measurement and Modeling Computer Systems, ACM Press, 2004, pp. 25–36.
[4] Bouncycastle, The bouncy castle crypto apis for java, <http://www.bouncycastle.org/java.html>.
[5] BrokenSuite, Suite of broken png images, <http://code.google.com/p/javapng/wiki/BrokenSuite>.
[6] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, M. Wolczko, Tuning garbage collection for reducing memory system energy in an embedded java environment, ACM Trans. Embedded Comput. Syst. 1 (2002) 27–55.
[7] Chilkat, Java encryption examples, <http://www.example-code.com/java/encryption.asp>.
[8] L. Eeckhout, A. Georges, K.D. Bosschere, How java programs interact with virtual machines at the microarchitectural level, in: Proceedings of the ACM Con- ference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, California, USA, ACM Press, New York, USA, 2003.
[9] P. Griffin, W. Srisa-an, J.M. Chang, An energy efficient garbage collector for java embedded devices, in: LCTES, ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems, 2005, pp. 230–238.
[10] S. Hu, L. John, Impact of virtual execution environments on processor energy consumption and hw adaptation. in: Proc. VEE, 2006, Ottawa, Ontario, Canada, ACM Press, 2006.
[11] IBM. Thejikesrvm, <http://oss.software.ibm.com/developerworks/oss/jikesrvm/>.
[12] S.M. Inc, The source for java technology, 2003, <http://java.sun.com>.
[13] A. Janapsatya, A. Ignjatovic, S. Parameswaran, A novel instruction scratchpad memory optimization method based on concomitance metric, in: ASP-DAC, 2006, pp. 612–617.
[14] JavaPNG, Sixlegs java png decoder, <http://code.google.com/p/javapng/>.
[15] R. Jones, Garbage Collection Algorithms for Automatic Dynamic Memory Management, fourth ed., John Wiley and Sons, 2000.
[16] Kaffe, Kaffe is a clean room implementation of the java virtual machine, 2005, <http://www.kaffe.org/>.
[17] M. Kandemir, I. Kadayif, U. Sezer, Exploiting scratch-pad memory using presburger formulas, in: ISSS '01: Proceedings of the 14th international symposium on Systems synthesis, New York, NY, USA, ACM, 2001, pp. 7–12.
[18] D. Kegel, Building and testing gcc/glibc cross toolchains, 2004, <http://www.kegel.com/crosstool/>.
[19] S. Kim, S. Tomar, N. Vijakrishnan, M. Kandemir, M. Irwin, Energy-efficient java execution using local memory and object co-location, In: IEE Proceedings – Computers and Digital Techniques on-line, 2004.
[20] N. Nguyen, A. Dominguez, R. Barua, Scratch-pad memory allocation without compiler support for java applications, in: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems, 2007.
[21] D. T. U. of Massachusetts Amherst and the University of Texas, Dynamic simple scalar, 2004, <http://www-ali.cs.umass.edu/DSS/index.html>.
[22] Project-Gutenberg, Don quixote by miguel de cervantes saavedra, <http://www.gutenberg.org/etext/996>.
[23] W. Schaik, The "official" test-suite for png. <http://www.schaik.com/pngsuite/>.
[24] Sourceforge, Jamvm – a compact java virtual machine, 2004, <http://jamvm.sourceforge.net/>.
[25] Sourceforge, kissme java virtual machine, 2005, <http://kissme .sourceforge.net>.
[26] SPEC, Specjvm98 documentation, March 1999, <http://www.specbench.org/osg/jvm98/>.
[27] P.F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, 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.
[28] D. Takahashi, Java chips make a comeback, 2001, <http://www .redh erring .com/>.
[29] M. technologies Inc., <http://www.micron.com/>.
[30] M. technologies Inc., System-power calculator, <http://www.micron.com/>.
[31] S. Thoziyoor, J.H. Ahn, M. Monchiero, J.B. Brockman, N.P. Jouppi, A comp rehensive memory modeling tool and its application to the design and analysis of future memory hierarchies, in: ISCA, 2008, pp. 51–62.
[32] Tiobe, <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.
[33] B. Vandette, Deploying java platform. standard edition (java se) in today's embedded devices, <http://developers.sun.com/learning/javaoneonline//pdf/TS-2602.pdf?>.
[34] J.M. Velasco, D. Atienza, K. Olcoz. Exploration of memory hierarchy configurations for efficient garbage collection on high-performance embe dded systems. in: GLSVLSI '09: Proceedings of the 19th ACM Great Lakes symposium on VLSI, New York, NY, USA, ACM, 2009, pp. 3–8.
[35] M. Verma, L. Wehmeyer, P. Marwedel, Dynamic overlay of scratchpad memory for energy minimization. in: CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, New York, NY, USA, ACM, 2004, pp. 104–109.

Jose M. Velasco received a bachelor degree in Education in 1994, a MSc in Physics in 1999, both from Complutense University of Madrid (UCM), a Master degree in Telecommunication Engineering in 2001 from the Technical University of Madrid (UPM) and a PhD degree in Computer Science (UCM) under the supervision of Francisco Tirado, Katzalin Olcoz and David Atienza. Currently, he holds a position as Assistant Professor at the Department of Computer Architecture and System Engineering of Complutense University of Madrid. His research interests include automatic memory management algorithms, hardware and software optimizations for virtual machines, and high-performance distributed garbage collection. He has published in several reviewed conferences from a wide range of topics as the Great Lakes Symposium on VLSI (GLSVLSI), European Conference on Object-Oriented Programming (ECOOP) or the International Conference on Parallel Computing (ParCo).

Prof. David Atienza received his MSc and PhD degrees in Computer Science from Complutense University of Madrid (UCM), Spain, and Inter-University Micro-Electronics Center (IMEC), Belgium, in 2001 and 2005, respectively. Currently, he is Professor and Director of the Embedded Systems Laboratory (ESL) at EPFL, Switzerland, and Adjunct Professor at the Computer Architecture and Automation Department of UCM. Additionally, he is Scientific Counselor of long-time research of IMEC Nederland (IMEC-NL), Holst Centre, Eindhoven, The Netherlands. His research interests focus on design methodologies for high-performance embedded systems and Systems-on-Chip (SoC), including new thermal management techniques for 2D/3D Multi-Processor SoCs, dynamic memory management and memory hierarchy optimizations for embedded systems, novel architectures for logic and memories in forthcoming nano-scale electronics and 3D integrated circuits, as well as Networks-on-Chip design. In these fields, he is co-author of more than 100 publications in prestigious journals and international conferences. He has received a Best Paper Award at the IEEE/IFIP VLSI-SoC 2009 Conference and three Best Paper Award Nominations at the HPCS 2010, ICCAD 2006 and DAC 2004 conferences. He is an Associate Editor of IEEE Transactions on CAD (in the area of System-Level Design), IEEE Letters on Embedded Systems and Elsevier Integration: The VLSI Journal. He is also an elected member of the Executive Committee of the IEEE Council of Electronic Design Automation (CEDA) since 2008 and an elected member of the Board of Governors of the IEEE Circuits and Systems Society (CASS) since 2010.

Prof. Katzalin Olcoz received the MS degree in Physics from the Complutense University of Madrid (UCM) in 1991. After that, she joined the ArTeCS group, where she has held several research and teaching positions. She received her Ph.D degree in the summer of 1997. Her advisor was Dr. Francisco Tirado. Her dissertation, Testable hardware allocation in High Level Synthesis, deals with the automatic design of RT-level data paths that include the hardware needed for their self-test. Since 2000, she is an associate professor in the Department of Computer Architecture and System Engineering of the UCM. As a Ph.D. student she visited the German National Center for Research in Computer Science (Bonn), in 1993, where she worked under the supervision of Prof. Raul Camposano. She also collaborated with Prof. Jean Francois Santucci, from University of Corsica during the years 1996-1998. Her research interests include processor design, virtual machines and hardware and software optimizations of dynamic memory management on multimedia applications, with special emphasis on low-power embedded systems.