PennBench: A Benchmark Suite for Embedded Java

G. Chen, M. Kandemir, N. Vijaykrishnan and M. J. Irwin
Microsystems Design Lab, The Pennsylvania State University, University Park, PA
{gchen, kandemir, vijay, mji}@cse.psu.edu

Abstract

Currently, there are 23 million Java-enabled handsets with more than 50 different models from 17-plus suppliers. With the growing popularity of such devices, there is a need in the embedded industry for a set of applications for accurate indications of the performance of embedded Java solutions. Towards addressing this problem, we gather a set of 12 Java applications running on PDA and cell phone like devices. We present a machine-independent characterization of the applications, specifically focusing on their memory characteristics. Our focus is motivated by the criticality of memory size constraints and heap footprints on both performance and power budgets.

1 Introduction

In the post-PC era, the usage of embedded/mobile devices is expanding at an enormous rate. Java is one of the widely used technologies targeted for developing interoperable applications on different type of devices in the widely diversified embedded world. For example, Java has recently been deployed by several handset manufacturers and mobile carriers for various cell phone-like devices. According to marketing numbers from Sun Microsystems, currently there are 23 million Java-enabled handsets with more than 50 different models from 17-plus suppliers [20].

Java applications targeted for constrained environment need to address a different set of concerns as compared to their high-end counterparts. This is because developing applications for embedded systems brings a whole new set of constrains such as small memory, low cost, unreliable connectivity and time determinism. Java is preferred in such embedded environments due to the ease in which the same application can be targeted at different embedded devices built by a wide variety of vendors. In fact, Java is currently portable to 80 to 95 percent of platforms and lets developers design and implement portable applications without the special tools and libraries that coding in C or C++ normally requires [21]. Further, Java allows application writers to embed animation, sound, and other features within their applications easily. These features are important for some of the emerging applications targeted at cell phones such as games, video and audio players.

The availability of appropriate benchmark suites has spurred innovation in hardware and software systems in the past. For example, benchmark suites such as Dhrystone[27], Linpack[11], SPECJVM[5], SPECjbb[4], SPEC2000[3], Mediabench[15], MiBench[13], PowerStone[16], DSPstone[30], Perfect Club[6] have been used widely for evaluating different architectures and software optimizations. These benchmark suites are generally targeted at non-Java environments or high-end/desktop Java environments. With the proliferation of embedded devices that support Java, it is important to have access to an embedded Java benchmark suite. To our knowledge, there is no benchmark suite available for embedded Java. The importance of such a suite is also shared by industry as indicated in the ongoing effort in EEMBC to develop an embedded Java suite. According to Rod Crawford, who leads the EEMBC's Java technology efforts, "there is a vital need in the embedded industry for fair, honest, certifiable benchmarks that provide customers with accurate indications of the performance of embedded Java solutions" [1]. The design of such applications has gained momentum due to the several embedded Java accelerators announced by various companies over the last year. Access to such benchmarks can help in evaluating the performance of Java virtual machines (JVMs), just-in-time Java compilers, Java accelerators and co-processor, other Java execution mechanisms, and the interaction of all the above with a host operating system in an embedded application.

This paper introduces PennBench, a set of embedded Java applications targeted at cell-phone and PDA like devices. We present a machine-independent characterization of the applications, specifically focusing on their memory characteristics. Our focus is motivated by the criticality of memory size constraints and heap footprints on both performance and power budgets.

The rest of this paper is organized as follows. We provide an overview of the benchmarks and the experimental setup in the next section. Section 3 characterizes the method invocation behavior of our applications. Section 4 presents the details of the memory allocation behavior of the PennBench suite. Finally, we summarize our conclusions in Section 6.

2 Overview of PennBench

The Mobile Information Device Profile (MIDP)[2] is a set of Java APIs which, together with the Connected Limited Device Configuration (CLDC), provides a complete Java 2 Platform, Micro Edition (J2ME) application runtime environment targeted at mobile information devices, such as mobile phones and entry level PDAs. The MIDP specification addresses issues such as user interface, persistence storage, networking, and application life cycle. Table 1 shows the minimal requirements for MIDP devices. Our work in this paper is based on SUN MicroSystem's MIDP reference implementation v1.0.3. To study the behavior of applications running on embedded devices that support MIDP, we selected 12 Java applications. When selecting applications, we considered following criteria. (1) All the applications should run on a Java environment that satisfies MIDP/CLDC specification. (2) The suite should cover both utility and entertainment applications. (3) The suite should cover CPU intensive, memory intensive and GUI intensive applications. (4) The suite should make use all the major components of a MIDP devices, including CPU, volatile memory (heap, stack, etc), non-volatile memory, wireless network, display, keyboard and touch screen. (5) The applications should be useful (for utilities) or interesting (for entertainment) so that we believe that they will be popular in today's or future hand-held devices. (6) The applications should be publicly available.

To observe the behaviors of these 12 applications, we executed them on KVM (a small-footprint Java virtual machine that satisfies the J2ME/CLDC configuration) running on a Solaris platform. Most of the information collected in the study are machine-independent and are obtained by adding instrumentation codes to the KVM. Specifically, to characterize method invocation behavior, we record each method invocation. To characterize memory allocation behavior, we instrument the memory management code to keep track of each memory allocation. To obtain the information about object lifetimes, we invoke garbage collector at (i) each memory allocation, and (ii) each return from a method. In order to make the results of interactive applications repeatable, we also instrumented KVM's GUI event delivery codes to record user activities (e.g., keyboard stroke, mouse click, etc.). These events are played back to KVM when gathering profile data for different configurations.

Before presenting detailed results of our study, we briefly describe each of the selected applications that form PennBench.

calculator This is a calculator application that supports addition, multiplication, subtraction and division operations. The operands for these are entered by using a virtual keyboard. The execution of four different expressions is used in obtaining the profile.

chess This is a game application in which the user plays against the computer. The player can move the pieces by selecting and dragging them using a pen-type device. A sequence of 24 moves taken by the user before winning a game was used in the profile.

emailviewer POP3 client In this application, the MIDP device receives 3 short text-only emails from a POP3 server and each of these text messages are browsed in sequence.

jpegviewer This application is used to display 100x67, 24-bit color JPEG images on the screen. In the profiling, we load an image 7 times and choose different options for color, gray scale and brightness. These operations are performed to mimic changes performed in handhelds to adjust to lighting differences.

manyballs Drawing bouncing balls on the screen. Each ball corresponds to a Java thread. Using keyboard, we increase the number of balls from 1 to 10 and then reduce it from 10 to 1 again. The procedure is repeated 4 times. This application is selected as a representative common ball games.

mdoom This application is a 3D shooting game. Here, the player navigates through a maze carrying a gun to track and shoot a robot. The keyboard or cell phone keys can be used to both control navigation direction and also to decide on when to fire. The profiling was performed until the robot was found and killed.

mpg This application is a MPEG movie player with a display size of 100x67 pixels. We play the movie for around 35 seconds and then stop the application.

mvideo This application plays a streaming movie from a web site. The video file is in the smv format and is obtained using a HTTP connection. We stop the application 40 seconds after the video starts playing.

pushpuzzle This is a conventional puzzle game in which user push boxes to their desired location in the maze. In the profiling, we solve the first two levels of the puzzles.

scheduler This application is a personal monthly scheduler. It displays a calendar on the screen. The user

Display	Screen-size: 96x54 Display depth: 1-bit Pixel shape (aspect ratio): approximately 1:1
Input	One or more of the following user-input mechanisms: - one-handed keyboard, - two-handed keyboard, - touch screen
Memory	Non-volatile: MIDP components: 128KB application-created persistent data: 8KB Volatile: Java runtime (e.g., the Java heap): 32KB
Networking	Two-way, wireless, possibly intermittent, with limited bandwidth

Table 1. Minimal requirements for MIDP devices

can touch the date on the screen and then input a text reflecting the schedule for that day. The schedules are stored in the non-volatile memory. In the profiling, we first input schedules for three days of August 2002. Then, we retrieve these schedules, modify them, and store them back into the non-volatile memory.

sfmap This is a utility that is used by tourists when visiting a new city. It identifies addresses, restaurants and attractions on a digital map of San Francisco. We input three addresses through the keyboard and ask the application to display the portions of San Francisco map containing those addresses.

webviewer This is a web browser for MIDP devices. We browse through three web pages, one is the home page of Penn State University and the other two are students' personal pages.

Table 2 provides the important characteristics of the applications explained above. Specifically, we provide the total size of class files, the number of classes, the number of dynamic bytecodes executed and the number of user threads created during our profiling runs for each application.

3 Method Invocation Behavior

We characterize the number of methods used in each of the applications. The methods are grouped as either user methods or library methods. The user methods are the methods of the classes that belong to a user application. They are loaded when the application is launched and make use of the volatile memory space of the MIDP devices. The library methods are the methods in the class libraries that are preloaded and stored in the non-volatile portion of the memory associated with the MIDP components.

Table 3 shows the numbers of user and library methods in the first two columns and the total number of method invocations in the last column. The breakdown of the invocations into user and library parts is shown in the left side of Figure 1. It can be observed that more than 80% of invocations are for library methods. This is also shown to translate to dominance of the number of bytecodes executed from the library methods for most benchmarks. This can be seen from the right side of Figure 1 that shows the distribution of dynamic bytecodes between user and library methods. It should also be noted that this observation holds for most benchmarks in spite of the exclusion of the some of the I/O or GUI related library methods. These methods are implemented as native methods and hence do not contribute to the dynamic bytecode count.

This distribution also indicates that significant performance gains can be obtained for these applications if just the class libraries are pre-compiled to native form. This would obviate the need for runtime interpretation of bytecodes of the class libraries and provide a good performance without the need for a dynamic compiler that may be costly in terms of space for an embedded device.

benchmark	Number of methods User	Number of methods Library	Number of Invocations
calculator	36	243	37273
chess	78	281	767249
emailviewer	117	631	1333508
jpegviewer	36	243	21177
manyballs	13	217	2720403
mdoom	39	245	351832
mpg	59	254	175287
mvideo	19	466	2694535
pushpuzzle	47	380	44825
scheduler	26	454	39553
sfmap	26	396	193074
webviewer	108	669	154695

Table 3. Number of methods and invocations

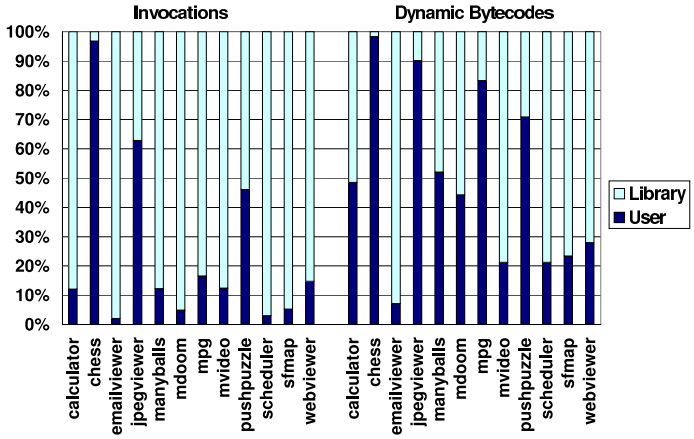

Figure 1. Distribution of user and library method invocations.

Next, we quantify the number of distinct methods that consume most of the execution time. Figure 2 shows the percentage of methods that contribute to the overall number of executions. We observe that, for all our benchmarks, 20% of methods account for over 90% of total invocations. This indicates that dynamically compiling a small portion of the methods to the native code can provide a significant performance benefits. Note that dynamic translation incurs both space and time penalties. These overheads are important factors influencing the design of light-weight dynamic compiler (e.g. [23]). Specifically, the space for storing the translated code in small in embedded devices is a major constraint. Hence, it would be beneficial to identify a small set of critical methods to compile.

One common approach to decide on when to compile a method is to use a threshold for the invocation count. If the invocation count of the method is less than the threshold, the method is interpreted. When the threshold is reached, dynamic compilation is triggered and subsequent executions of the method make use of the compiled native code. Fig-

Benchmark	Total Size of Classes (Bytes)	Number of Classes	Number of Dynamic Bytecodes	Number of User Threads	URL
calculator	15715	3	723523	1	www.spruce.jp/freemidlets/
chess	47625	7	34585042	24	www.spruce.jp/freemidlets/
emailviewer	125285	33	75153058	5	www.reqwireless.com
jpegviewer	23646	7	317171	1	www.jshape.com/midp/index.html
manyballs	11514	5	57484608	43	an example comes with KVM
mdoom	21903	5	8946741	2	www.jshape.com/midp/index.html
mpg	53580	8	2243471	4	www.anfyteam.com/dev/j2me/
mvideo	8383	4	56908069	2	www.jshape.com/midp/index.html
pushpuzzle	23065	6	920306	1	an example comes with KVM
scheduler	12101	3	591768	1	holycow.tripod.co.jp/cooldownboy/
sfmap	15834	5	3275078	3	www.jshape.com/midp/index.html
webviewer	98936	25	1686041	4	www.reqwireless.com

Table 2. Applications in the PennBench suite.

ure 3 shows the cumulative percentage of methods in the x-axis and the number of invocations of a method in the y-axis. These results indicate that, as an example, if we set the threshold on when to compile to 10 invocations, up to 40% of the methods of benchmark chess will be compiled. However, by setting the threshold to 15 invocations, the percentage of compiled methods drops to 30%. This profile data can be used to tune the thresholds according to the size of the memory available for storing translated code. In fact, we can also estimate the size of the translated methods by having a knowledge of the size of the bytecodes in the method. Figure 4 shows the sizes (in bytes) of each method of each code in PennBench. An interesting observation is that all the benchmarks have a peak at 5. This indicates that a big portion of methods are property access methods. A typical property access method has such bytecodes:

```
Method int length()
  0 aload_0
  1 getfield #22 <Field int count>
  4 ireturn
```

4 Memory Allocation Behavior

4.1 Heap Usage

The KVM maintains a heap. All the data structures created during the runtime of Java applications (e.g., Java objects, stack frames, loaded user classes, etc.) are allocated from this heap. Some of these objects become inaccessible as execution of the application progresses and are called as garbage. The objects in the heap that are not garbage are called as live objects. The maximum live heap size indicates the minimum configuration required for the volatile memory to support the Java runtime.

Figure 5 shows the heap size of the live objects during the execution of PennBench applications. The maximum

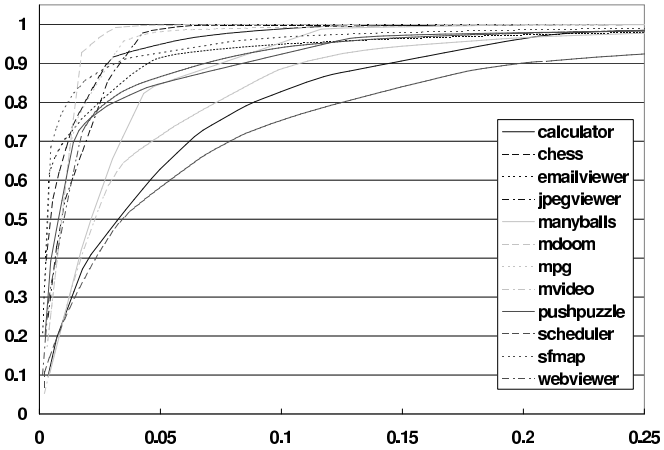

Figure 2. Accumulated percentages of method invocations. x-axis: cumulative percentage of methods. y-axis: accumulated percentages of method invocations.

value of the curve of each application indicates the minimum size of the heap for the application to run without out-of-memory exception. It can be observed that among the 12 applications, only the manyballs and scheduler application can run in devices supporting the minimal MIDP configuration of 32KB non-volatile memory. Further, we observe that five of the applications in PennBench require at least a 256KB heap to execute. These include data intensive video/image applications such as mpg, mvideo, sfmap and webviewer. Thus, the selection of the non-volatile memory size needs to consider the application set that are to be supported. We observe that a 256KB heap is sufficient to support all applications in PennBench. However, seven of the twelve applications, are able to run with a heap smaller than 128KB.

The choice of the heap memory size is also influenced by the energy budget of the device (influencing both dynamic and leakage energy). A larger memory consumes more dynamic energy for each access and leakage energy. Parti-

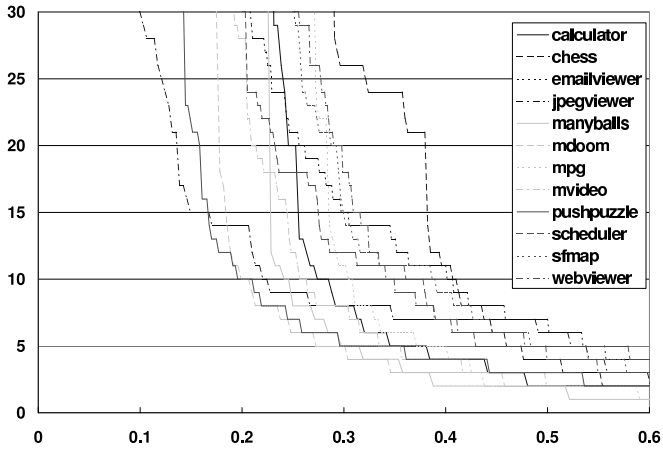

Figure 3. Number of method invocations. x-axis: cumulative percentage of methods. y-axis: the number of invocations of a method.

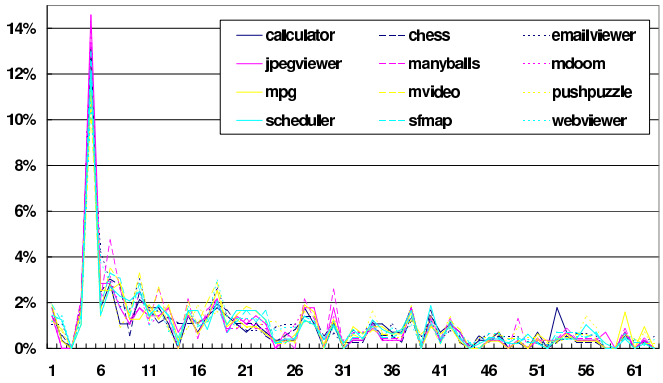

Figure 4. Distribution of method sizes. x-axis: method size in bytes. y-axis: percentage of methods.

tioning the memory banks to smaller parts and activating only those banks that are currently used can help in mitigating the power consumed in the memory. For example, for reducing the leakage energy consumed by the memory, the supply rails can be selectively disconnected to heap portions that do not contain any live data [8]. The size of heap space currently being used is a function of both the allocation rate and the rate at which the memory occupied by the dead objects are reclaimed. Thus, if the space occupied by dead objects is not recollected immediately after they become inaccessible, the size of the occupied space in the heap can be much larger than that shown in Figure 5. Each graph in Figure 6 has two curves. One shows the accumulated sizes of objects that have been allocated by the application since the application was launched. The other curve shows accumulated sizes of garbage that have been generated by the application since the application was launched. The slopes of the two curves in each graph indicates the rate of memory allocation and garbage generation. Note that the difference between the two curves is the minimum heap size required for the application assuming that garbage is collected immediately. While reference counting garbage collectors [14] exhibit such a behavior of collecting most of the garbage immediately, they have the drawback of not being able to identify garbage in circular structures. In more commonly used collectors such as the mark-sweep-compact collector [14] employed in the KVM, garbage is collected only during an invocation of a collector. While invoking the collector is beneficial in reclaiming the unused space, the invocation pauses normal program execution and also consumes energy in scanning and marking the heap for accessible objects during collection. The frequency of such periodic collectors can be tuned based on the rates of the allocation and garbage generation to maximize opportunities of turning off the banks of the memory by striking a balance between the cost of garbage collection and the energy savings by preventing the enabling of additional banks [7].

4.2 Garbage Collection Pause Time

The frequency of the garbage collector also has an implication on another key parameter in real time applications, namely, the application pause times. This is because the execution of the application is paused when the garbage collector reclaims the memory space that are occupied by garbage objects. The existing implementations of the garbage collectors are invoked only when the heap space is used up and there is not enough space for the next allocation. This is referred to as the base case. In Table 4, we compare the number of collections and pause times of the base case with that of a collector that is invoked after every K object allocations. Note that such collectors that are invoked more frequently than the base case collector can be desirable from energy perspective. While the number of pauses due to more frequent collector increases, the pause times during garbage collection are observed to reduce (generally) as the frequency of collection increases. Minimization of pause times is important as it affects the response time in interactive applications. This observed behavior in Table 4 is due to the complexity of mark-sweep-compact GC algorithm used in KVM is proportional to the number of objects (both live and dead) in the heap. When the frequency of GC is reduced, the collector has to scan more objects at each GC invocation, thereby increasing pause times. The reason for minor deviation from this trend is because other factors such as the amount of heap fragmentation, which also influencing pause times.

4.3 Object Life Times

Generational garbage collection [29] has been widely used in many implementations of Java Virtual Machines. Generational garbage collection is based on the observation that most objects live a very short time, while a small percentage of them live much longer. Generational collector

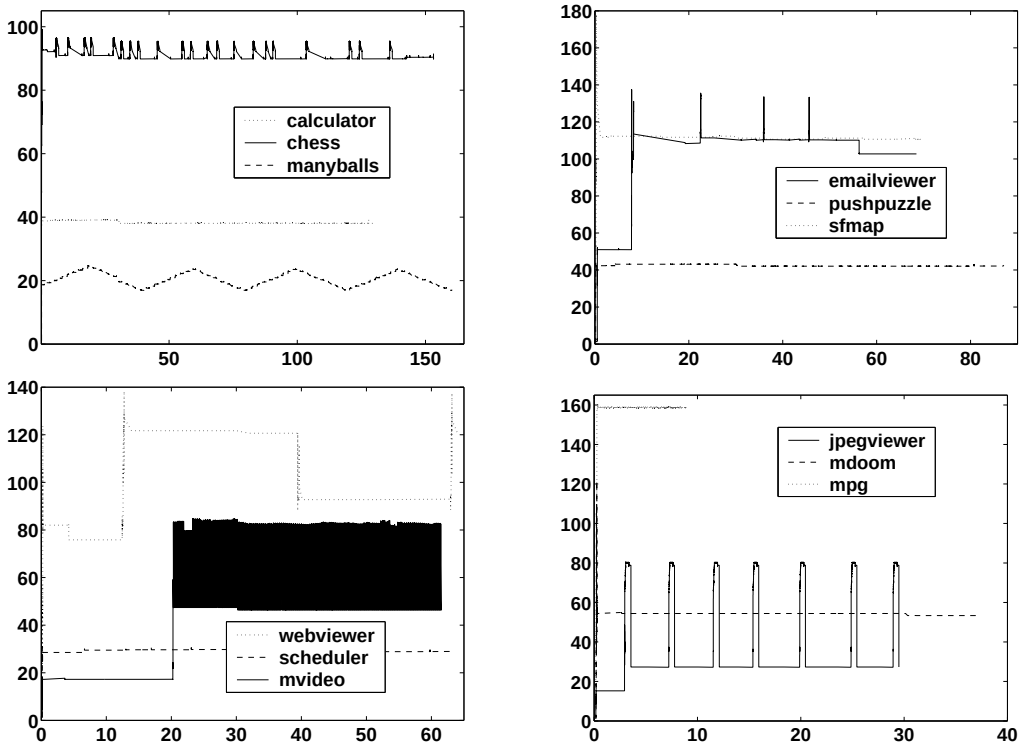

Figure 5. Memory footprints of PennBench codes. x-axis: total size (KB) of live objects in the heap. y-axis: time (in seconds, measured by running KVM on SunBlade 1000, SPARC v9 at 750 MHz).

	k						
Benchmark	**10**	**40**	**75**	**100**	**200**	**500**	**base**
calculator	822	207	113	87	47	24	9
	0.101	0.132	0.151	0.166	0.198	0.244	0.248
chess	898	234	135	106	64	39	27
	0.131	0.160	0.182	0.190	0.253	0.259	0.288
emailviewer	679	203	128	106	74	55	43
	0.210	0.208	0.235	0.251	0.289	0.349	0.354
jpegviewer	3348	843	454	343	176	76	13
	0.226	0.353	0.338	0.362	0.378	0.360	0.509
manyballs	174	47	27	21	13	8	5
	0.137	0.175	0.195	0.196	0.213	0.216	0.193
mdoom	147	46	30	26	19	15	13
	0.081	0.108	0.123	0.134	0.147	0.172	0.182
mpg	720	189	108	86	54	34	22
	0.197	0.212	0.214	0.235	0.239	0.270	0.270
mvideo	450	117	65	50	45	34	28
	0.128	0.160	0.190	0.183	0.187	0.206	0.259
pushpuzzle	202	57	35	29	20	14	11
	0.106	0.147	0.163	0.168	0.193	0.194	0.191
scheduler	258	69	40	31	19	11	7
	0.092	0.121	0.137	0.156	0.190	0.203	0.191
sfmap	508	134	76	60	35	20	11
	0.213	0.238	0.249	0.256	0.273	0.290	0.292
webviewer	774	215	114	107	71	50	36
	0.223	0.233	0.226	0.250	0.272	0.307	0.342

Table 4. Impact of GC frequency on GC pauses. The heap size is 512KB. GC is invoked every k times of allocations. The top and bottom numbers in each cell are the number of GC invocations and average length of GC pauses, respectively. Average pauses are in milliseconds. The numbers are obtained by running the KVM on SunBlade 1000 (SPARC V9 at 750MHz).

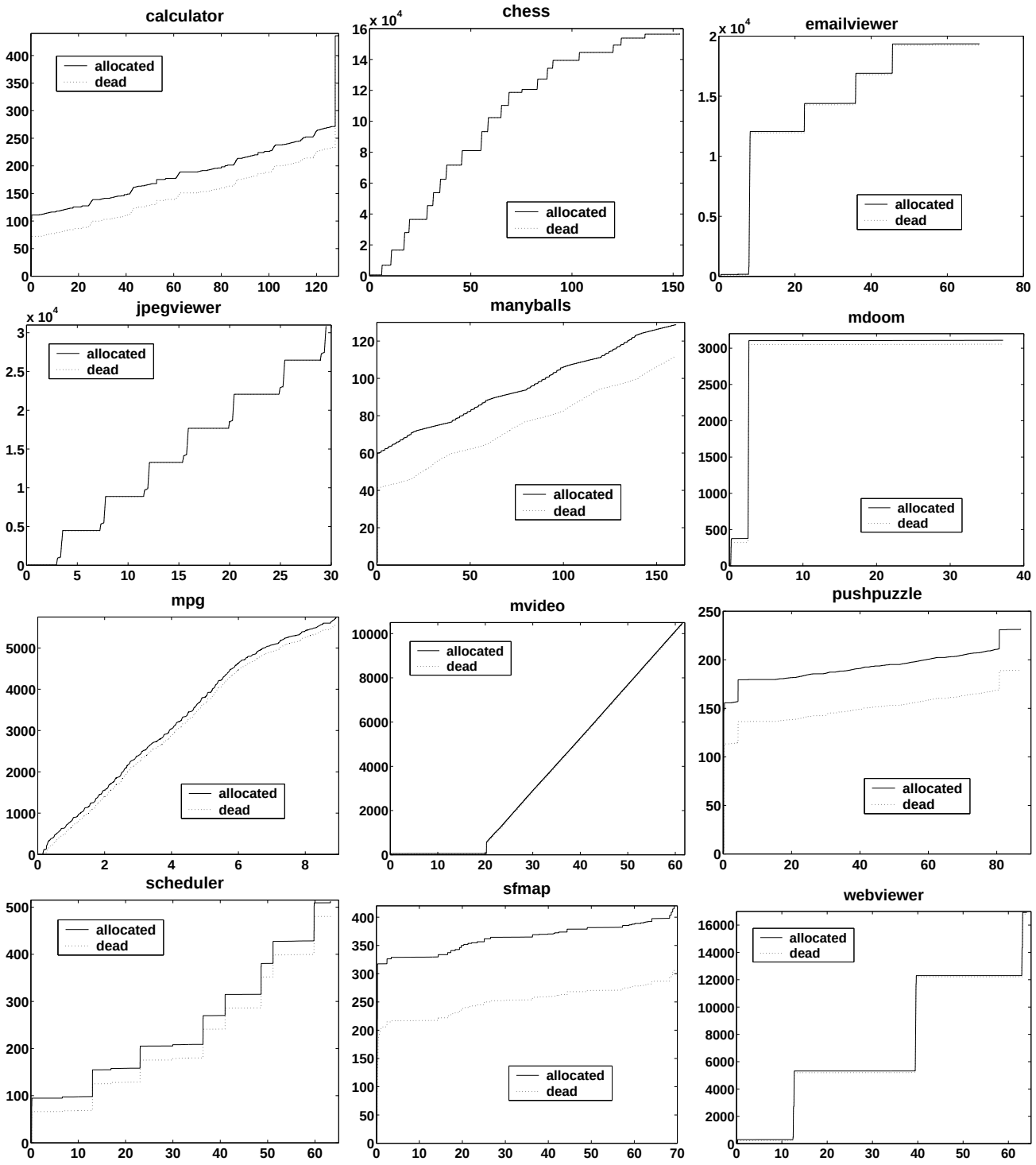

Figure 6. Memory footprints. x-axis: total size (KB) of live objects in the heap. y-axis: time (second, measured by running KVM on SunBlade 1000, SPARC v9 at 750 MHz).

segregates the heap into multiple areas by age. Areas containing older objects are collected less frequently. All the objects are initially allocated in the youngest area. When the area for a younger generation is used up, a collection on that generation is invoked. The objects that survive a certain number of collections are promoted to a less frequently collected area for older objects. It is important to identify the size of heap area for each generation. If the size of the area for a generation is too small, the intervals between collections will be too short and too many objects will survive each collection. Both these increase overall GC cost. Too large an area for a generation, however, makes pause time for each GC very long, which is undesirable for interactive or realtime applications.

The tuning of parameters of such a generational collector strongly depends on the object life times. Figure 7 shows the distribution of objects lifetimes in PennBench. Lifetime is indicated in terms of the number of bytes allocated between the creation of the object and when that object becomes garbage. This metric is useful in determining the size of each generation. We observe from Figure 7 that, in the applications' starting stage (first 0.5 second after the application is launched) and stable stage (the running time after the starting stage), heap objects have different lifetime distributions. For most of the applications in PennBench, during the stable stage, around 90% of objects die before the next 1KB memory have been allocated. During the starting stage, however, this percentage is much lower (less than 30% for most of PennBench benchmarks). Further, most of the long-lived objects are created during this initialization phase. We can also observe that applications such as calculator, manyballs, pushpuzzle and sfmap exhibit a generational behavior with objects of different lifetimes. We can conclude that a heap size between 1-4KB would be a suitable choice for the young generation. Further analysis such as tracking inter-generational references are required to make more precise estimates.

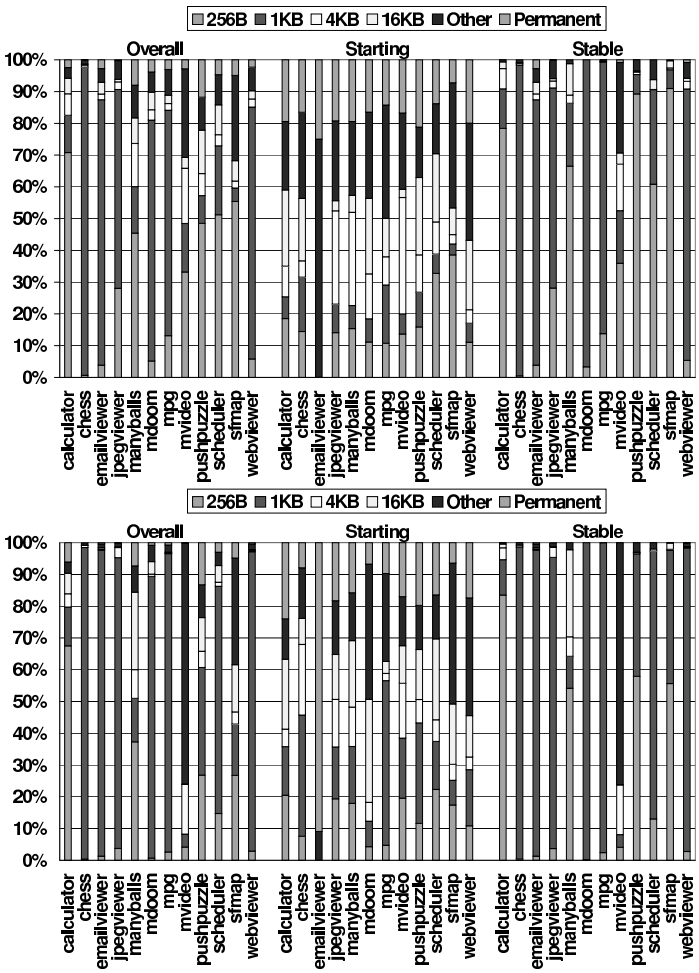

Figure 7. Distribution of object lifetimes in PennBench. y-axes in the top and bottom graphs are the percentage of objects and allocated bytes, respectively. 256B, 1KB, 4KB and 16KB represent the objects that become garbage before allocation of next 256B, 1KB, 4KB and 16KB heap memories, respectively. Permanent objects are the objects that are identified to live throughout the application's running time when they are created, such as constant pools, field tables, virtual method tables and bytecodes.

4.4 Method and Thread Local Objects

Next, we characterize the locality of the objects with reference to their lifetimes and access patterns.

A method local object is an object whose lifetime does not exceed the return of the method in which this object was allocated [9, 28]. In other words, method local objects can be guaranteed to be garbage when the method returns. It is possible to exploit the presence of method local objects for both performance and energy improvements. First, a method local object can be allocated in the stack along with the stack frame of the method. Since stack allocation is easier and faster than heap allocation, it provides performance enhancements. Next, due to the stack allocation of the objects, garbage collection is implicit and objects are removed along with the methods frame on a method return. Hence, no performance or energy overhead is incurred in explicitly collecting the objects. From an energy perspective, since the objects are collected implicitly close to when the become garbage, the leakage energy consumed by unused objects can be reduced. In addition, since the stack is always "first in, last out", the accesses pattern of the stack is more regular and predictable than that of the heap. Therefore, memory banks allocated for stack may be turned on and off more predictably. This also implies opportunities for energy saving if a large number of objects can be stack allocated. Our goal is not to quantify the exact performance gains or energy savings but to identify the number of objects that are method local. It can be observed from Figure 8 that except for one application, 20% of all objects are method local in our suite.

Next, we define thread local objects as objects that can

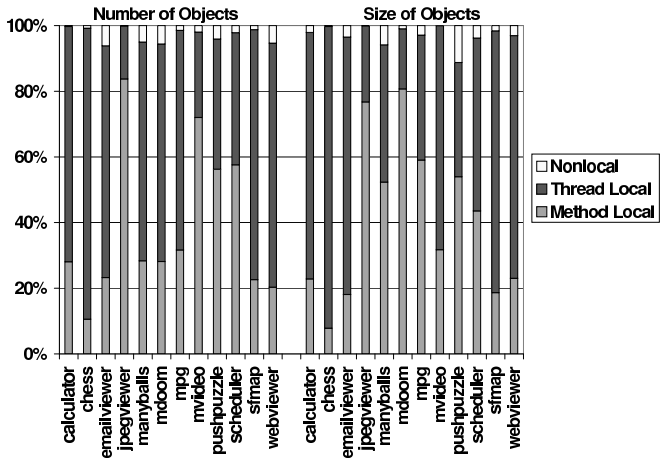

Figure 8. Distribution of method local, thread local and nonlocal objects. Only Java objects, i.e., the objects are created and visible by Java bytecodes, are considered. Data structures internal to the KVM implementation are not counted.

be accessed only by the thread in which the object was created [9, 28]. It should be noted that method local objects are also thread local; thread local, however, are not necessarily method local objects. Since thread local objects are guaranteed to be accessed by only one thread, synchronization operations on thread local objects are not necessary and thus, they can be safely eliminated. The elimination of synchronization operations on these objects improves performance. Further, the thread local objects provide opportunities for energy savings. When a thread is blocked or preempted, all its local objects will not be accessed until it is scheduled again. Thus, we can place memory banks that contain only local objects of the specific thread into a low power mode. Figure 8 shows that more than 90% of all objects are thread local indicating potential for significant performance and energy savings. This is specifically important to multi-threaded applications such as manyballs.

5 Related Work

A number of studies profiled the behaviors of Java applications and Java Virtual machines. Farkas et al. [12] examined the energy consumption of a pocket computer running Java applications. G. Chen et al. [8] evaluate the impact of garbage collection frequency on energy consumption. The set of applications used in [8] while representative of embedded applications is based on the obsolete com.sun.kjava package. Our work looks at the behavior of embedded applications more generally and our applications are MIDP compliant. Vijaykrishnan et al. [25] presented a characterization of the energy consumption by the cache and main memory when executing the SPEC JVM98 benchmarks in the JIT and interpreter modes. Radhakrishnan et al. [22] identified some execution characteristics of SPEC JVM98 benchmark suite at both bytecode and native code levels. Li et al. [19] investigated the influence of architecture and operating system on the execution of Java applications using complete system simulation. Based on the behaviors of Java applications, Vijaykrishnan et al. [26] proposed architectural support for object manipulation, stack processing and method invocation to enhance the execution speed of Java bytecodes based on characterizing some Java applications. In [18] and [17], the authors investigated the predictability of branches in the control flows of Java workloads. In [10] and [24], the memory behavior of Java workloads were characterized. More specifically, the studies in [10] were focused on the heap allocation behaviors and provided useful information for the designing of garbage collectors. In [24], the focus was on the characterization of memory access behaviors in order to expose opportunities for improving cache and TLB performance.

All these studies (except [12, 8]) used SPEC JVM98 benchmark suite, which is designed for high performance JVM with plenty of resources (such as memory, energy supply and computational power of CPU). Some general conclusions drawn from the studies on SPEC JVM98 may also be applicable to the embedded environments, however, when evaluating JVMs running on resource limited devices, SPEC JVM98 are not appropriate. For example, most applications of SPEC JVM98 require a heap no smaller than few mega bytes [10] and they are not able to run on a mobile phone that has only 4MB memory. In addition, all the benchmarks in SPEC JVM98 are non-user-interactive, computation-intensive applications. They are not representive for the typical workloads running on hand-held devices where GUI activities play an important role. When tuning configuration related parameters (such as the size of each generation in a generational garbage collector and optimization strategies in a JIT compiler) in resource constrained and user-interactive-intensive environments, the results that are obtained by using SPEC JVM98 may not be optimal for the applications specially designed for those systems.

6 Conclusion

In this study, we gathered a set of applications that are representative of the embedded Java applications running on cellphone/PDA-like devices. First, we looked at the method invocation behavior of these applications. We observed that off line compilation of library methods and dynamic compilation of a small set of methods would obviate the need of interpretation of most of the methods used by these applications. Next, our study of the heap behavior of PennBench codes revealed different opportunities that exist for optimizing the performance and energy for such codes. Specifically, we looked at object characteristics that influence the performance of garbage collectors and potential for stack allocation of method local objects. We believe that the availability of this suite of applications publicly will help other JVM architects, Java-enabled device manufacturers and embedded Java researchers in evaluating new hard-

ware/software optimizations.

Acknowledgement

This research is supported in part by NSF Awards 0073419, 0103583 & 0139143; NSF CAREER Awards 0093082 & 0093085; GSRC.

References

[1] EEMBC - embedded microprocessor benchmark consortium. http://www.eembc.org/Press/Default.asp.

[2] Mobile information device profile (MIDP) specification. http://java.sun.com/j2me/docs/.

[3] SPEC CPU2000 V1.2. http://www.specbench.org/osg/cpu2000/.

[4] SPEC JBB2000. http://www.spec.org/osg/jbb2000/.

[5] SPEC JVM98 benchmarks. http://www.specbench.org/osg/jvm98/.

[6] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. H. andJ. Schwarzmeier, L. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall,1989.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Adaptive garbage collection for battery-operated environments. In *The 2nd USENIX Java Virtual Machine Research and Technology Symposium (JVM'02)*, Aug. 2002.

[8] G. Chen, R.Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded Java environment. In *The 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*, Cambridge, MA, Feb. 2002.

[9] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, Denver, Colorado, Nov. 1999.

[10] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *The European Conference on Object-Oriented Programming (ECOOP'99)*, 1999.

[11] J. Dongarra. Linpack benchmark. http://performance.netlib.org /performance/ html/ linpack.data.col0.html.

[12] K. Farkas, J. Flinn, G. Back, D. Grunwald, and J. Anderson. Quantifying the energy consumption of a pocket computer and a Java virtual machine. In *SIGMETRICS '00*, Santa Clara, CA, June 2000.

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: a free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, Dec. 2001.

[14] R. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1999.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[16] L. H. Lee, B. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *International Symposium on Low Power Electronics and Design*, Aug. 1999.

[17] T. Li, R. Bhargava, and L. John. Rehashable BTB: an adaptive branch target buffer to improve the target predictability of Java code. In *The International Conference on High Performance Computing (HiPC)*, 2002.

[18] T. Li, L. John, and J. Robert H. Bell. Modeling and evaluation of control flow prediction schemes using complete system simulation and Java workloads. In *The 10th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2002.

[19] T. Li, L. John, N. Vijaykrishnan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *ACM International Conference on Supercomputing*, pages 22–33, 2000.

[20] J. Lyman. Java's surprising comeback. http://www.newsfactor.com/perl/story/18365.html.

[21] L. D. Paulson. Handheld-to-handheld fighting over Java. *IEEE Computer*, page 21, July 2001.

[22] R. Radhakrishnan, J. Rubio, and L. K. John. Characterization of Java applications at bytecode and Ultra-SPARC machine code levels. In *1999 IEEE International Conference on Computer Design*, Austin, Oct. 1999.

[23] N. Shaylor. A just-in-time compiler for memory constrained low-power devices. In *The 2nd USENIX Java Virtual Machine Research and Technology Symposium (JVM'02)*, Aug. 2002.

[24] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing memory behavior of Java workloads: A structured view and opportunities for optimizations. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 2000.

[25] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of Java applications from the memory perspective. In *The 1st USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, Monterey, California, Apr. 2001.

[26] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla. Object-oriented architectural support for a Java processor. In *ECOOP'98, the 12th European Conference on Object-Oriented Programming*, July 1998.

[27] R. Weicker. Dhrystone: A synthetic systems programming benchmark. *Communications of the ACM*, 27:1013–1030, 1984.

[28] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, Denver, Colorado, Nov. 1999.

[29] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, St. Malo, France, Sept. 1992.

[30] V. Zivojnovi'c, J. Mart'inez, C. Schlager, and H. Meyr. DSPstone: a DSP-oriented benchmarking methodology. In *ICSPAT'94*, Dallas, Oct. 1994.