Energy consumption and garbage collection in low-powered computing Technical Report: CU-CS-930-02*

Amer Diwan, Han Lee, and Dirk Grunwald
University of Colorado, Boulder

Keith Farkas
Compaq Western Research Laboratory

Abstract

We have measured the energy efficiency of different memory management strategies on a high performance pocket computer. We conducted our study by measuring the energy consumption of eight C programs with four different memory management strategies each. The memory management strategies are: no deallocation, explicit deallocation, conservative mark-and-sweep garbage collection, and conservative mark-and-sweep incremental garbage collection.

Our measurements show that different memory management strategies have very different energy requirements. In the most extreme case, one program consumed 40 times as much energy with incremental garbage collection than with explicit deallocation. We demonstrate that, although overall energy use is strongly correlated with execution time, the processor and peripheral energies separately do not correlate well with execution time.

1 Introduction

Modern pocket computers [17, 8] provide sufficient processing capability and memory capacity to run traditional desktop-development environments (e.g., Java) and operating systems. The Itsy Pocket Computer [17], an example of this new breed of pocket computer, runs a fully-functional port of Linux, X-windows, Java, and Squeak. In comparison, the Palm V PDA [9] runs a non-general-purpose operating system (PalmOS) and a very limited port of Java (KVM). This new breed of pocket computers offers software developers the opportunity to use a rich programming environment, and to concentrate on adding functionality to applications rather than minimizing their memory footprint. For the user of such computers, their processing and memory capabilities make it feasible to download applications over a wireless connection and run them safely within a flexible and platform-neutral run-time environment such as Java.

A key enabler of rich programming environments and mobile applications is dynamic memory allocation, a technique that permits memory to be allocated by a program as it runs. With one exception [7], previous evaluations [33, 11, 28] of memory managements strategies focused on memory behavior or execution time and not energy. Yet, for pocket computers, which are powered by batteries, energy is one of the most important resources – a pocket computer is useless if its battery has been exhausted. Further, the power consumed by the memory subsystem of a pocket computer is a significant contributor to its overall power consumption [14].

In this paper, we present an evaluation of the energy impact of memory management strategies. To our knowledge, such an evaluation has been done previously only by Chen et al.[7] using an activation-based energy model, which captured the energy consumed by the (simulated) processor core, on-chip caches, and SRAM. In comparison, we measure the energy consumed by an actual pocket computer including the energy consumed by its processor (StrongARM SA1100) and its memory subsystem (64 MB of DRAM).

To evaluate the energy impact of memory management, we have implemented four different memory management strategies, and have measured the performance and energy cost of their use when several benchmark programs were run on an Itsy pocket computer. The strategies we have chosen are representative of those often used in C and C++ programs as well as in run-time environments such as Java. Our results show that the choice of memory management

*This work is supported by NSF ITR grant CCR-0085792 and an NSF Career award. Any opinions, findings, and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

has a profound effect on the energy consumption of programs. The contributions of this paper are:

- We experimentally quantify the energy performance of four memory management strategies using an actually prototype of a modern pocket computer.
- We demonstrate that, although overall energy use is strongly correlated with execution time, the processor and peripheral energies separately do not correlate well with execution time.
- We relate object allocation and deallocation patterns to the energy consumption of specific memory managers.

In the next section, we present some background material on energy and memory managements. Then, in Section 3, we describe the four strategies we examine, our workload, the Itsy Pocket Computer, and our measurement methodology. We then discuss our results in Section 4, and present future work in Section 6.

2 Background

To better understand how the use of a particular management strategy affects the energy consumption of applications that use it, we begin by reviewing energy-consumption concepts. We then follow with a discussion of memory management strategies.

2.1 Energy

The energy E, measured in Joules (J), consumed by a computer over T seconds is equal to the integral of the instantaneous power, measured in Watts (W). The instantaneous power consumed by components implemented in CMOS, such as microprocessors and DRAM, is proportional to $V^2 \times F$, where V is the voltage supplying the component, and F is the frequency of the clock driving the component. Thus, the power consumed by a computer to, say, search an electronic phone book, may be reduced by reducing V, F, or both. However, for such tasks that embody a finite amount of work, reducing the frequency may result in it taking more time to complete the work, and thus, little or no energy will be saved.

In normal usage, pocket computers run on batteries, which provide a finite amount of energy. Furthermore, the amount of energy a battery can deliver (i.e., its capacity) is reduced with increased power consumption [10]. As an illustration of this effect, consider the Itsy pocket computer that was used in this study (described in Section 3.2). When the system is idle, the integrated power manager disables the processor core but keeps active the device drivers. If these drivers are clocked at 206 MHz, a typical pair of alkaline batteries will power the system for about 2 hours, but if they are clocked at 59 MHz, they will last for about 18 hours. Although the battery lifetime increased by a factor of 9, the processor speed was only decreased by a factor of 3.5. The capacity of the battery can also be increased by interspacing periods of high power demand with periods of low power demand [4]. Nonetheless, the extent to which these two non-ideal properties can be exploited is highly dependent on the chemical properties and the construction of a battery as well as the conditions under which the battery is used.

2.2 Memory Management Strategies

A memory management strategy specifies how dynamically-allocated memory is deallocated. Broadly speaking, there are three types of strategies: *no deallocation*, *explicit deallocation*, and *automatic deallocation*. A program using the *no deallocation* strategy never deallocates the memory allocated to its objects, except when it completes. A program using *explicit deallocation*, however, deallocates the memory allocated to objects at the points in the program specified by the programmer. In contrast, a program using automatic-deallocation (i.e., garbage collection) depends on the run-time system to deallocate the memory associated with objects when the objects are not reachable, that is, considered live.

At run time, to support the allocation and deallocation of memory, some bookkeeping information must be maintained. In general, this information must be updated on each allocation and deallocation. In addition, for strategies that rely on garbage collection, the bookkeeping information may also have to be updated on each read and write of memory. Accessing and updating the bookkeeping information requires executing instructions that are not part of the program itself, and that may increase the number of cache misses and page faults the program incurs. Thus, bookkeeping introduces both time and energy overhead to the execution of a program.

No-deallocation has the smallest overhead because there is little overhead in allocating new memory – in essence, the memory available for allocation to a program may be viewed as a linear array, which can be indexed by an increment-only pointer. The simplicity of this scheme makes it appropriate for short-lived programs having small working sets. *Explicit deallo-*

cation incurs a greater overhead due to the need to record when memory is deallocated, to keep track of the blocks of memory that are not presently allocated, and to select from these each time an allocation is to be done.

The overhead associated with automatic-deallocation is highly dependent on the type of garbage collector used. In this paper, we use a conservative mark-and-sweep collector. A *conservative collector* is pessimistic when deciding whether an object is no longer considered live, but as a consequence, does not require support from the compiler or programming language. A *mark-and-sweep collector* does not reallocate frequently-accessed live objects during garbage collection. As a result, over time, the memory space becomes more fragmented, leading to worse memory-system behavior (e.g., higher cache miss rates) than would occur with a compacting collector. Because we use a non-compacting collector, the overhead associated with tracking unallocated memory and allocating memory will be the same as with *explicit deallocation.* But, automatic-deallocation also incurs the cost of searching for those objects that are no longer considered in use, and deallocating the memory associated with them. Each time a garbage collector is invoked, it may either attempt to locate all live objects, or it may seek only a subset of them. In the latter case, an *incremental collector* is said to be used. For more details on dynamic memory allocation, the reader is referred to [32].

The different overheads of the memory management schemes translates into different energy overheads. The study presented in this paper will help software designers for embedded systems make more informed memory management decisions.

3 Methodology

This section first describes the four memory management strategies and the benchmarks we used in our study (Section 3.1), then describes the Itsy pocket computer, which was the platform for our experiments (Section 3.2), and finally describes how we measure energy (Section 3.3).

3.1 Strategies and workload

In this paper we measure the energy consumption resulting from the use of four memory management strategies:

- no deallocation (**no deallocation**),
- explicit deallocation using the memory manager provided in glibc2.1 (**explicit deallocation**),
- automatic deallocation using the Boehm-Demers-Weiser conservative mark-and-sweep garbage collector [2] (**BDW**), and
- automatic deallocation using the Boehm-Demers-Weiser collector in incremental mode (**BDW-inc**)

We modified our benchmark programs to implement each of these strategies, and ran each program on an Itsy pocket computer while measuring its power consumption. In their original form, our benchmarks use the *explicit deallocation* strategy. For *no deallocation*, we deleted calls to the routine for freeing memory (*free*) from the programs. If a program had its own memory recycling mechanism, we also disabled that mechanism. Finally, for both *BDW* and *BDW-inc*, we used the programs modified for *no deallocation* but replaced calls to the memory allocator (system or user-defined) with calls to the garbage collector's allocation routine.

Table 1 briefly describes our benchmark programs and their inputs, and for each one, gives its running time for the explicit-allocation strategy, the number of lines of code, and the amount of memory it allocates dynamically. The running times are measured by using our energy measurement hardware and are accurate to 1/5000 second. All programs are written in C. *Anagram*, *ks*, *ft*, *yacr-2*, and *bc* are from Todd Austin's pointer-intensive benchmark suite [1]; *li* and *ijpeg* are integer benchmarks from the SPEC95 suite; *sed* is a commonly used UNIX tool from GNU.

We chose these programs since we believe they are representative of the tasks a pocket computer might be asked to do: *e.g.,* complex calculations (*bc*), string processing (*sed*), execution of programs using interpretation (*li*), and image processing (*ijpeg*). In addition these benchmarks do a significant amount of dynamic memory allocation, which is an important characteristic of many modern applications. Finally, several of these benchmarks, such as the ones from Austin's suite and the SPEC benchmarks, are commonly used benchmarks in the literature.

3.2 The Itsy Pocket Computer

The Itsy Pocket Computer is a flexible research platform, developed to enable hardware and software research in pocket computing. It is a small, low-power, high-performance handheld device with a highly flexible interface, designed to encourage the development

Name	Running time (sec)	Memory allocated (bytes)	Lines of code	Description and inputs
bc	0.57	27,811,152	7,308	GNU bc calculator Find prime numbers smaller than 1000
yacr-2	0.76	69,916	3,979	Channel router used for integrated circuit layout input1.in distributed with program
sed	0.91	591,512	31,859	GNU stream editor Text processing of 720K text file
ks	0.97	8152	782	Kernighan-Schweikert graph partitioning tool KL-1.in distributed with program
anagram	1.11	259,812	647	Generates anagrams Eight words in four lines
li	2.40	3,835,616	7,597	Lisp interpreter boyer.lsp distributed with benchmark
ijpeg	8.79	9,072,332	31,211	Image compression/decompression Sample image
ft	42.37	1,245,480	2,156	Finds minimum spanning trees. Graph with 8000 vertices, 16000 edges

Table 1: Benchmark programs and their inputs

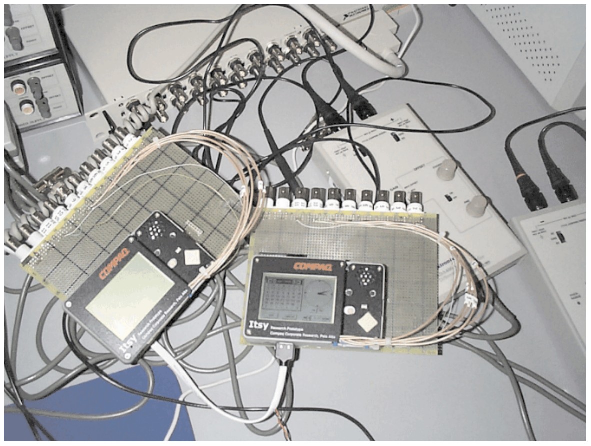
Figure 1: Setup used to measure power.

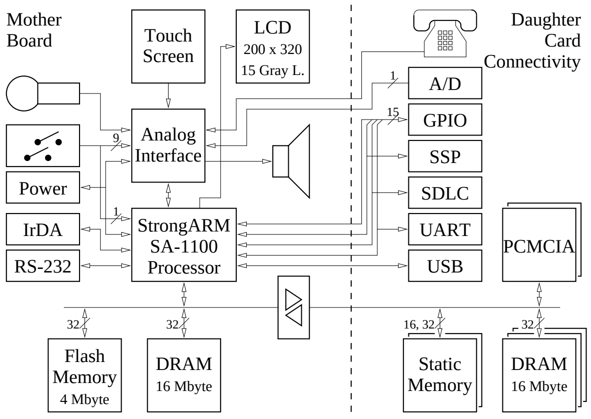

Figure 2: Itsy version 1.5 System Architecture

of innovative research projects, such as novel user interfaces, new applications, power management techniques, and hardware extensions.

There are several versions of the basic Itsy design, with varying amount of RAM, flash memory and I/O devices. We used several version 1.5 units for this study, which Compaq Computer Corporation's Western Research Lab modified to include instrumentation leads for power measurement. Itsy version 1.5 and version 2 differ primarily in their sleep power usage. Figure 1 shows the units along with the measurement equipment we used.

All versions of the Itsy are based on the low-power StrongARM SA-1100 microprocessor. All versions have a small, high-resolution display, which offers 320×200 pixels on a 0.18mm pixel pitch, and 15 levels of greyscale. All versions also include a touchscreen, a microphone, a speaker, and serial and IrDA communication ports. The Itsy architecture can support up to 128 Mbytes both of DRAM and flash memory. The flash memory provides persistent storage for the operating system, the root file system, and other file systems and data. Finally, the Itsy also provides a "daughter card" interface that allows the base hardware to be easily extended.

The version we use in this study (1.5) has 64 Mbytes of DRAM and 32 Mbytes of flash memory. Our units are modified to allow us to run the StrongARM SA-1100 at either 1.5 V or 1.23 V. Although 1.23 V is below the manufacturer's specification, it can be safely used at moderate clock speeds, and yields about a 30% reduction in the power consumed by the processor. The Itsy can be powered either by an external supply or by two size AAA batteries – the batteries provide sufficient energy for the Itsy to run $\frac{1}{2}$ hour in "high power" mode, such as when playing an MPEG video or the popular action game Doom. Figure 2 shows a schematic of the Itsy architecture. All measurements in this paper use the power adapter rather than batteries.

3.3 Measuring Power and Total Energy

To measure the instantaneous power consumed by the Itsy, we use a data acquisition (DAQ) system to record the current[1] drawn by the Itsy as it is connected to an external voltage supply, and the voltage provided by this supply. Figure 1 presents a picture of our setup along with the wires connected to the Itsy to facilitate measuring the supply current and voltage. We configured the DAQ system to read the current and voltage 5000 times per second, and convert these readings to 16-bit binary values. A host computer stores these readings for subsequent analysis. From these measurements, we can compute a time profile of the power used by an application as it runs on the Itsy.

To determine the relevant part of the power-usage profile of a workload, we measure the time required to execute the workload and then select the relevant set of measurements from the data collected by the DAQ system. For each benchmark, we used the `gettimeofday` system call to time its execution; this interface uses the 3.6 MHz clock available on the processor to provide accurate timing information. To synchronize the collection of the voltages with the start of execution of a workload, as the workload begins executing, we toggle one of the SA1100's general-purpose input-output (GPIO) pins. This pin is connected to the external trigger of the DAQ system; toggling the GPIO causes the DAQ system to begin recording measurements. As our measurement technique is very similar to that used in [14], we refer the reader to this reference for a more in depth description.

Once the relevant part of the profile has been determined, we calculate from it the average power and the total energy consumed by the Itsy during the corresponding time interval. To compute the energy, we make the assumption that the power measured at time t represents the average power of the Itsy for the interval t to $t + 0.0002$ seconds, where 0.0002 seconds is the time between each successive power measurement. Thus, the energy E is equal to $\sum_{i=1}^{n} p_i(t) \times 0.0002$, where $p_1(t), \ldots, p_n(t)$ are the n power readings of interest.

In making our power measurements, we used a similar approach as the one used in [14] to reduce a number of sources of possible measurement error. The net effect of these errors is an error of ≈ 0.005 Watts, which in our experiments, yields an error of at most 0.0006 Joules, or at most 0.6% with a mean maximum possible error of 0.4%. These values represent the maximum error; we saw much smaller variation in our measurements.

[1]The supply current was measured by measuring the voltage drop across a high precision small-valued resistor of a known resistance (0.02Ω). The current was then calculated by dividing the voltage by the resistance.

	Time in GC		Number of GC	
Benchmark	BDW	BDW-inc	BDW	BDW-inc
bc	2.51	14.80	158	421
yacr-2	0.02	0.06	3	6
sed	0.06	0.28	9	23
ks	0.01	0.02	1	2
anagram	0.03	0.04	2	3
li	0.53	2.36	35	97
ijpeg	2.28	4.39	2	4
ft	0.48	0.68	7	14

Table 2: Statistics on benchmark programs. All times are in seconds.

4 Results

In this section, we report on the energy consumption of our benchmark programs for each of the four memory management strategies described in Section 3.1. In Table 2, we provide some additional statistics for the two strategies that use a garbage collector. In the table, the columns labeled *Time in GC* give the time spent by the program in the garbage collector when using BDW and BDW-inc. The *Number of GC* columns give the number of times each program invoked the garbage collector when using BDW and BDW-inc. We used the default settings BDW and BDW-inc to trigger garbage collection. We used `getrusage` to measure all times in Table 2.

4.1 Energy consumption

Figure 3 presents the energy consumption for each of our benchmark programs for each memory management strategy. The bars are normalized to the energy consumption of *no deallocation* (Table 3 gives

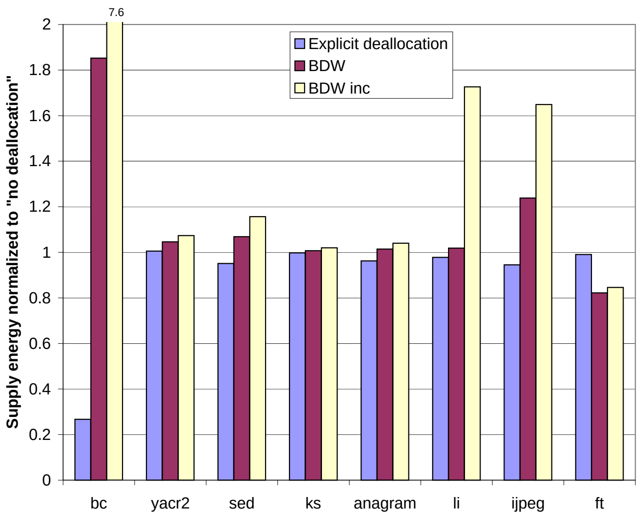

Figure 3: Normalized total energy consumption for benchmark programs

Benchmark	Energy for "no deallocation" (J)
bc	7.40
yacr-2	2.65
sed	3.40
ks	3.17
anagram	3.81
li	8.81
ijpeg	34.62
ft	144.27

Table 3: Energy consumption in Joules of "no deallocation"

the energy consumption of *no deallocation*). For the programs we considered, use of the *explicit deallocation* strategy consumed less energy than all other strategies. Use of the *no deallocation* strategy did not result in less energy demand because the programs allocated a sufficient amount of memory that it was better to deallocate memory as the program ran than to reduce the time spent in the memory manager.

From Figure 3 we note that both *no deallocation* and *explicit deallocation* consume noticeably less energy than garbage collection in three programs (*bc*, *li*, and *ijpeg*) and more energy in one program (*ft*). Given that our garbage collection does not compact objects (and thus significantly change the memory behavior of programs) and that we do not do any energy optimizations in the garbage collector (such as powering down unused DRAM banks [7]), it is not surprising that explicit deallocation typically uses less energy than garbage collection. However, the magnitude of the difference between garbage collection (and especially incremental collection) and explicit deallocation for some programs, such as *bc*, is surprising.

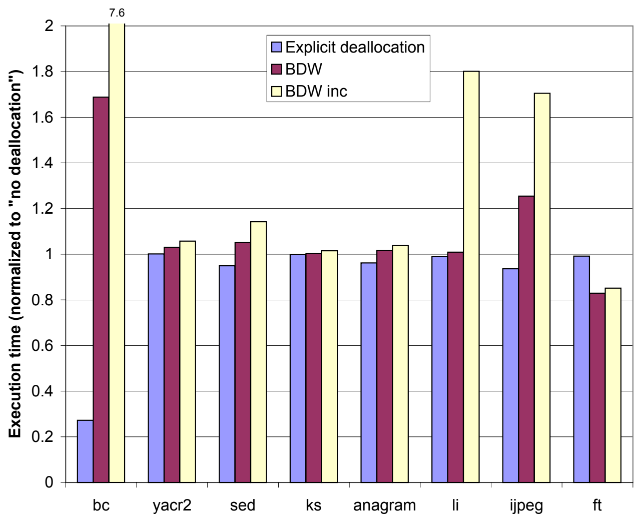

Figure 4: Normalized running time (in seconds) for different memory managers

There are two factors that can account for different energy consumption when a program runs with two different memory managers. First, one memory manager may be more expensive than the other in terms of execution time. Second, one memory manager may have or lead to worse memory system behavior (e.g., more cache misses) than the other. In the following two sections we explore which of these two reasons accounts for the data in Figure 3.

4.2 Correlation of energy consumption to execution time

Figure 4 presents the execution time for our benchmark programs. All bars are normalized to the execution time for *no deallocation*. On comparing these bars to Figure 3 we see clear parallels: in every case, increased execution time implies increased energy consumption.

Figure 5 investigates whether an increase in execution time is fully correlated to increase in energy consumption. It presents the power consumption (i.e., Watts) for each benchmark program. If energy consumption is fully correlated to execution time, then all bars (even across benchmarks) will have exactly the same height.

From Figure 5 we see that there is some variation in the rate of energy consumption of the programs. In other words, while execution time of a benchmark configuration gives a good indication of the total en-

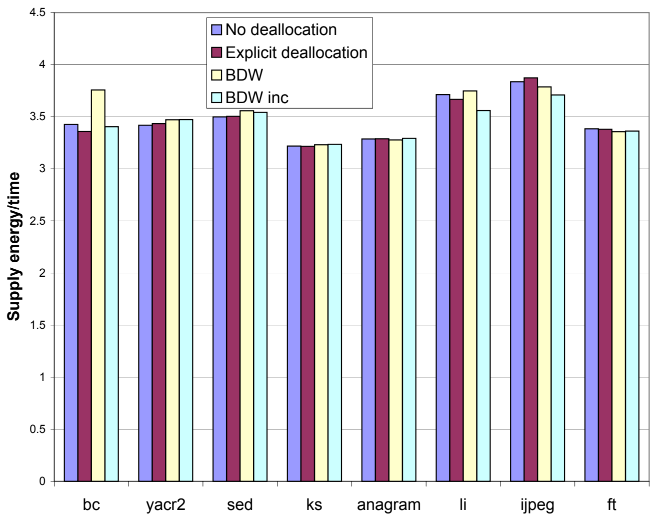

Figure 5: Power consumption (Energy/Time) in Watts.

ergy consumption of the program, it does not tell the whole story: other factors, such as program and memory management behavior also affect energy consumption.

Figure 6 breaks down the power consumption into energy consumed by the processor and the power consumed by the peripherals (e.g., caches and display). Our measurement infrastructure allows us to measure these powers seperately. The lines to the right of the vertical axis give the power consumption of the processor and the lines to the left give the power consumption of the peripherals. There are four set of lines for each benchmark corresponding to the four memory management schemes Note that for a given benchmark and memory management strategy, the difference between the total power reported in Figure 5 and the sum of the power consumed by the processor and the peripherals is due to the power consumed by the voltage regulators used in the Itsy.

From Figures 6 we see that power consumption varies significantly across memory managers and particularly across benchmarks even though we did not see these variations in Figure 5. Moreover, in configurations where the processor consumes energy at a faster rate, we often see that the peripherals consume energy at a slower rate and vice versa. For example, in *bc* explicit deallocation consumes energy at a slower rate than *no deallocation* for peripherals and at a faster rate for the processor. The explanation for this is that if a program incurs many cache misses, then it will spend more of its execution time waiting for memory than a program that incurs few cache misses. While a cache miss is in progress, the

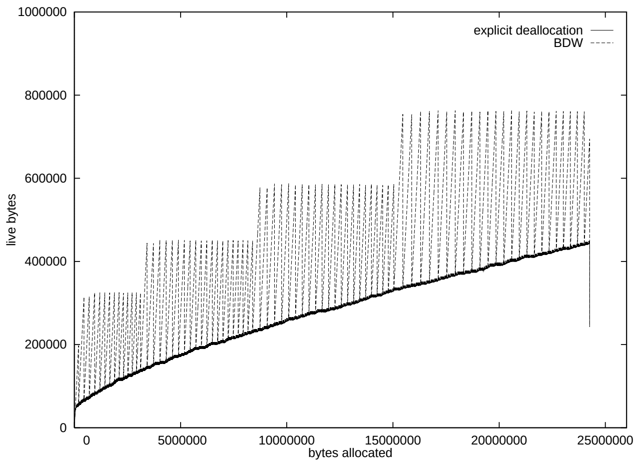

Figure 7: Memory behavior of `bc` running on a Compaq Alpha Workstation

processor used in the Itsy, the StrongARM SA-1100, reduces its clock speed, thereby lowering the power consumption of the processor. Taken together, the data presented in this section suggests that it is not enough to evaluate the energy consumption of a program or strategy by just measuring the energy consumed by the processor: one must measure or simulate the energy consumed by the entire computer to get an accurate picture of energy consumption.

4.3 Correlation of energy consumption to program behavior

The most dramatic variation in rate of energy consumption occurs in benchmark `bc`. To understand this, we need to look at the memory usage pattern of `bc` (Figure 7)[2]. The x-axis of Figure 7 represents time measured in bytes allocated by the program. A point (x,y) in this figure means that at "time" x the live objects occupy y bytes. We obtained these numbers from a "debugging" run with the garbage collector; with the debugging flag set, BDW collector outputs detailed statistics at each garbage collection. This graph has two curves, one for explicit deallocation and the other for the BDW memory manager. Figure 7 shows that `bc` allocates objects rapidly: during the run, it allocates just under 23MB of objects. However, objects also die fairly rapidly and the number of live bytes grows slowly and reaches less than 400K.

[2]We generated the memory usage graphs 7 and 8 by running the same source code and garbage collector as used in our other experiments on a Compaq Alpha Workstation. Thus, while the shape of these graphs are significant, the actual magnitude of the data will be different from the Itsy since data types (such as pointers) have different sizes on the Itsy and the Alpha.

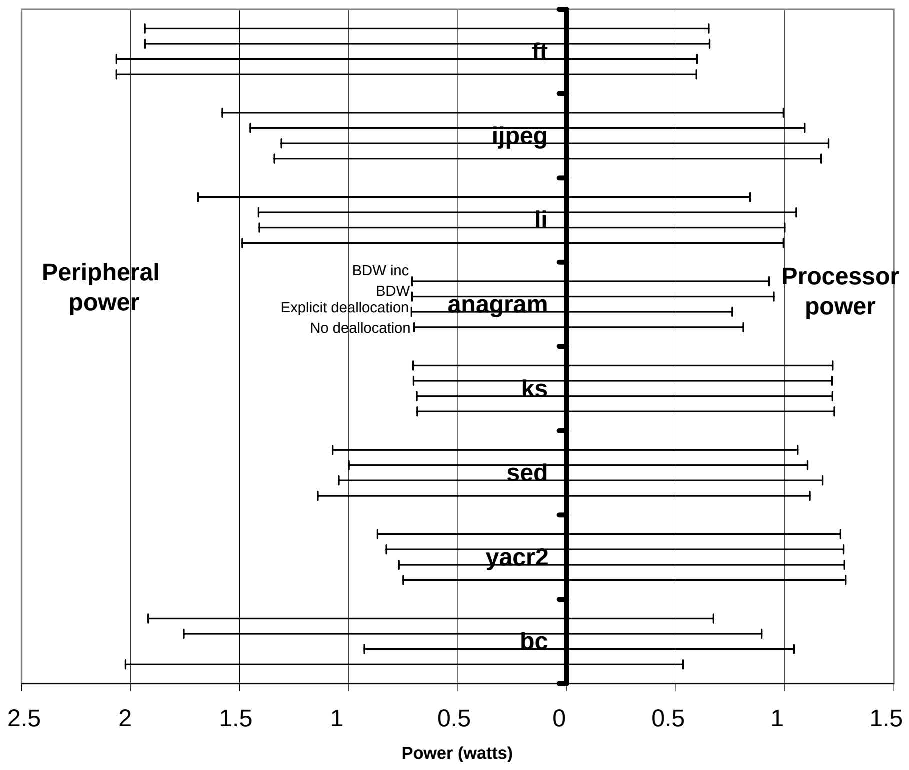

Figure 6: Power consumption (Energy/Time) by processor and peripherals in Watts. The four lines for each benchmark are (from top to down): BDW inc, BDW, Explicit deallocation, and No deallocation

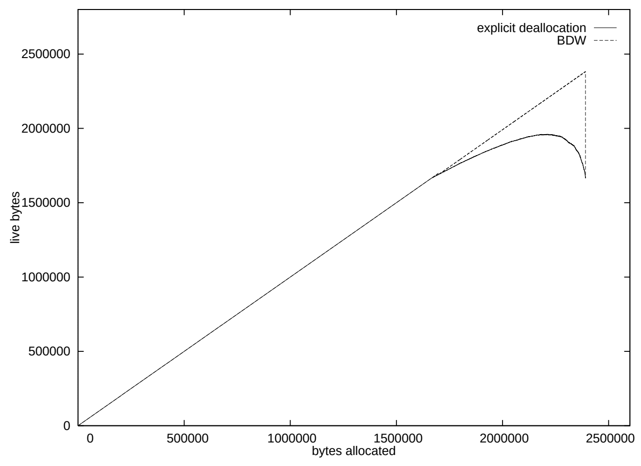

Figure 8: Memory behavior of `ft` running on a Compaq Alpha Workstation

Each spike in the memory behavior curve represents the point at which garbage collection either began or ended.

We see that explicit deallocation is quite efficient at managing the memory of `bc`: even though the program is allocating at a fast rate, it is also explicitly deallocating at a fast rate and that objects are deallocated soon after they become useless. Garbage collection, in contrast, has a much bigger memory requirement since it doesn't deallocate objects immediately: garbage collection is invoked only periodically but when it is invoked, it is just as effective as explicit deallocation in reclaiming objects.

If we use the no-deallocate memory manager with *bc* then it will use a lot of memory (up to 23MB) and thus incur potentially more cache misses and write-buffer activity, which are all expensive in terms of energy. Thus, *no deallocation* is much worse than explicit deallocation for energy consumption. However, *no deallocation* is still better than either garbage collection strategy since the overhead of tracing live objects during frequent garbage collections dominates the benefits due to a smaller working-set size. Further, comparing the data in Table 2 to that in Figure 4, we see that the time spent in the garbage collector is much greater than the full running time of the original program!

The memory usage of *ft* is in stark contrast to that of *bc* (Figure 8): all the objects allocated by *ft* remain live until nearly the end of the run at which point many objects are explicitly freed. Garbage collection is unable to free any objects in *ft*. However, despite the ineffectiveness of garbage collection, the garbage collected configurations run faster and consume less energy than explicit or *no deallocation*.

From Table 2 we see that even though garbage collection is ineffective in reclaiming objects in *ft*, garbage collection consumes only 1.1% of total execution time. Because *ft* does not deallocate any memory and it spends little time in the garbage collector, the differences in run time and energy between explicit deallocation and garbage collection are due to the time spent doing allocation. The BDW allocator is optimized to handle many objects of the same size and since most of *ft*'s objects are one of three sizes, BDW may be more efficient than the standard allocator.

4.4 Summary of Results

In this section we demonstrated that the choice of memory manager affects the energy consumption of a program on the Itsy. Also, in order to understand the energy consumption of a program or configuration, it is misleading to measure the energy consumption of just the processor: one must measure the energy consumption of the entire system.

5 Related Work

There are two bodies of related work: work in the area of energy-efficient computing, and work in the area of memory management.

Chen et al. [7] describe the effect of variations in mark-and-sweep garbage collection algorithm in a severely resource limited environment. Chen et al. also propose optimizations for turning off memory banks and demonstrate, via simulation, that they are effective at reducing energy consumption. Chen's environment is a microSPARC-IIep based 100MHz system with 128 KB of ROM. Our work differs from their work in two major ways. First, we report results from actual measurements of energy consumption rather than simulations. Second, while our environment (the Itsy pocket computer) is resource limited compared to a desktop, it is not as severely limited as the handheld that Chen et al. explore. hence, our results are more representative of emerging mobile computers, which are powered by high-performance yet low-power microprocessors with significant amounts of DRAM

A crucial first step in producing energy efficient systems is the use of code optimizations and compilers that seek to reduce the number of instructions executed by an application, thereby reducing the time required to execute it and the energy required [16]. There has also been considerable work on designing the hardware used by such systems to be energy efficient and energy conscious. Much of this work has focused on circuit-level [23] design issues or CAD tools [20]. More recently, work has focused on microarchitectural features to reduce power. This work ranges from characterizing power use or the potential power reduction in systems [29, 16, 18] to designs that optimize the memory subsystem [15, 21], reduce waste from speculative execution [22] or use a combination of techniques such as clock gating and voltage scaling [31, 6, 3, 5].

Relating specifically to memory-system issues, there are many papers that compare the power efficiency of different static organizations of the memory hierarchy, particularly the caches and TLB, for specific workloads [21] or using analytical models [20]. However, there is very little published material that compares the power efficiency of the memory system beyond the primary or secondary caches. In one of the few references we have found, da Silva [19] compared different dynamic memory allocators used in an ATM switching network.

There has been much work on comparing different memory management strategies [11, 36, 35, 26] and on measuring the memory system performance of different memory management strategies [24, 34, 33, 25, 12, 13].

6 Future Work

In undertaking the evaluation we report on here of the energy impact of various memory management strategies, we have identified a number of areas worthy of future study.

First, the choice of garbage collector clearly has an impact on the energy consumed by an application. Because the power consumed while accessing memory has a significant impact on the overall power consumption of a pocket computer, we are considering garbage collectors that have better memory behavior. Examples of these include copying collectors, which change the layout of objects in memory, and generational [30] or more general age-based collectors [27], which focus collection efforts on a small region of memory.

Second, while the applications we have chosen for the basis of our study are representative of those that are likely to be run on pocket computers, users of such computers will likely run more than one application at a time. We are thus investigating how multiprogramming impacts the choice of memory management strategy. We are also investigating Java-based applications, and multimedia applications. One challenge in investigating multiprogramming and these more media-rich applications is that there is considerable inter-run variance for a given application, and thus, it is more difficult to get repeatable runs for different memory management schemes.

Third, we are investigating optimizations that directly affect the energy consumption of applications. For example, the Itsy is an example of a system in which decreasing the clock speed of the processor does not result in a corresponding increase in the access time of the memory system, since memory access delay is dominated by DRAM precharge and other external factors.

Table 4 shows the power, energy and execution time for the "ijpeg" application in which this effect can be leveraged to reduce the energy consumed when executing it. The processor clock was reduced from 206MHz to 118MHz for the duration of garbage collection. The overall execution time increases by less than 5%, while the power consumed decreases by about 3%. From this data, we draw two conclusions. If this change in clock speed was accompanied by a reduction in voltage, the energy saved by the voltage reduction could offset by a significant amount the approximate 2% increase in energy consumption brought about by the longer run time. Thus, a net energy saving could be achieved. Secondly, the lower rate of energy consumption may increase the battery life without significant user delay. These outcomes are possible because the application was memory bound. We are investigating garbage collection algorithms that take advantage of this property of systems, and how it may be more generally applied.

7 Conclusions

In modern pocket computers energy is one of the most important resources–a pocket computer is useless if its battery has been exhausted. This paper demonstrates that the choice of memory management strategy affects energy consumption of programs and thus battery life of pocket computers.

We conducted our study by measuring the energy consumption of eight C programs with four different memory management strategies each. The memory management strategies are: no deallocation, explicit deallocation, conservative mark-and-sweep garbage collection, and conservative mark-and-sweep incremental garbage collection. We measured the energy consumption on an Itsy, a prototype of a modern pocket computer developed at Compaq Computer Corporation's Palo Alto Research Labs.

Our measurements show that different memory management strategies have very different energy requirements. In the most extreme case, one program

Program	Execution Time	Average Power	Total Energy
No clock scaling	15.47	1.67	25.83
Clock Scaling	16.18	1.63	26.31

Table 4: Performance of the "ijpeg" benchmark with the BDW-inc collector

consumed 40 times as much energy with incremental garbage collection than with explicit deallocation. We also show that processor and peripheral energy consumption compliment each other: programs that have a lower rate of processor energy consumption often have a higher rate of peripheral energy consumption (and vice versa). In addition to execution time, the behavior of programs with respect to how they use objects also affects the energy consumption of programs. Overall we found that explicit deallocation usually used the least energy, closely followed by no deallocation. Both garbage collection based memory managers typically consumed the same or more energy than explicit or no deallocation.

References

[1] Todd Austin. Pointer-intensive benchmark suite. http://www.cs.wisc.edu/~austin/ptr-dist.html.

[2] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 1988.

[3] Thomas D. Burd and Robert W. Brodersen. Processor design for portable systems. *Journal of VLSI Signal Processing*, 13(2/3):203–222, August 1996.

[4] Carla-Fabiana Chiasserini and Ramesh R. Rao. Pulsed Battery Discharge in Communication Devices. In *The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 88–95, August 1999.

[5] Edwin Chan, Kinshuk Govil, and Hal Wasserman. Comparing algorithms for dynamic speed-setting of a low-power cpu. In *Proceedings of The First ACM International Conference on Mobile Computing and Networking*, Berkeley, CA, November 1995.

[6] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, 1992.

[7] G. Chen, R. Shetty, N. Vijaykrishnan, M. M. Irwin, and M. Wolczko. Tuning garbage collection in an embedded environment. In *Symposium on High Performance Computer Architecture (HPCA)*, 2002. To appear.

[8] Compaq Computer Corporation. http://athome.compaq.com/showroom/static/iPaq/3835.asp.

[9] Daniel F. Zucker and Shawn Barnett. Enterprise Technology for the Palm Computing Platform. *Pen Computing Magazine*, 7(32):16–30, February 2000.

[10] David Linden (editor). *Handbook of Batteries, 2nd ed.* McGraw-Hill, New York, 1995.

[11] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado, 1993.

[12] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with copying garbage collection. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming languages*, Portland, Oregon, January 1994. ACM.

[13] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 1995.

[14] Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, , and Jennifer Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Proceedings of the ACM SIGMETRICS '00 International Conference on Measurement and Modeling of Computer Systems*, 2000.

[15] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce McGaughy, and David Patterson. The Energy Efficiency of IRAM Architectures. Technical report, May 1997.

[16] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[17] William R. Hamburgen, Deborah A. Wallach, Marc A. Viredaz, Lawrence S. Brakmo, Carl A. Waldspurger, Joel F. Bartlett, Timothy Mann, and Keith I. Farkas. Itsy: Stretching the bounds of mobile computing. *IEEE Computer*, 34(4):28—36, April 2001.

[18] Lorch J. A complete picture of energy consumption of a portable computer. Master's thesis, University of California, Berkeley, 1995.

[19] Julio L. Da Silva Jr, Francky Catthoor, Diederik Verkest, and Hugo De Man. Power exploration for dynamic data types through virtual memory mangement refinement. In *IEEE Symposium on Low Power Electronics*, pages 311–316. IEEE Symposium on Low Power Electronics, 1995.

[20] M.B. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In *ACM/IEEE International Symposium on Low-Power Electronics and Design*, August 1997.

[21] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 184–193, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[22] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings 25th Annual Annual International Symposium on Computer Architecture*, Barcelona, Spain, June 1998. ACM.

[23] W. Nebel and J. Mermet (Eds.). *Low Power Design in Deep Submicron Electronics*. Kluwer, 1997.

[24] Chih-Jui Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Department, University of Wisconsin-Madison, July 1989.

[25] Mark B. Reinhold. *Cache Performance of Garbage-Collected Programming Languages*. PhD thesis, Laboratory for Computer Science, MIT, September 1993.

[26] Frederick Smith and Greg Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *International Symposium on Memory Management*, pages 68–78, October 1998.

[27] Darko Stefanovic, Kathryn McKinley, and Eliot Moss. Age-based garbage collection. In *ACM 1999 SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 370–381, Denver, CO, November 1999.

[28] David Tarditi and Amer Diwan. Measuring the cost of memory management. *Lisp and Symbolic Computation*, 1996.

[29] V. Tiwari and et al. Instruction-level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13:223–238, 1996.

[30] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[31] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of The ACM*, 36:74–83, July 1993.

[32] Paul R. Wilson. Uniprocessor garbage collection techniques logic programminglanguages, 1992.

[33] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *1992 ACM Conference on Lisp and Functional Programming*, pages 32–42, San Francisco, California, June 1992.

[34] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, May 1991.

[35] Benjamin Zorn. The measured cost of conservative garbage collection. In *Software Practice and Experience*, pages 733–756, July 1993.

[36] Benjamin Zorn and Dirk Grunwald. Empirical measurement of six allocation-intensive C programs, CU-CS-604-92. Technical report, University of Colorado at Boulder, July 1992.