Introducing Approximate Memory Support in Linux Kernel

Giulia Stazi, Francesco Menichelli, Antonio Mastrandrea, Mauro Olivieri
Sapienza University of Rome
Dept. of Information Engineering, Electronics and Telecommunications (DIET)
Rome, Italy
Email: {stazi,menichelli,mastrandrea,olivieri}@diet.uniroma1.it

***Abstract*—This paper describes the implementation of approximate memory support in Linux operating system kernel. The new functionality allows the kernel to distinguish between normal memory banks, which are composed by standard memory cells that retain data without corruption, and approximate memory banks, where memory cells are subject to read/write faults with controlled probability.**

Approximate memories are part of the wider research topic regarding approximate computing and error tolerant applications, in which errors in computation are allowed at different levels (data level, instruction level, algorithmic level). In general these errors are the result of circuital or architectural techniques (i.e. voltage scaling, refresh rate reduction) which trade off energy savings for the occurrence of errors in data processing.

The ability to support approximate memory in the OS is required by many proposed techniques which try to save energy by raising memory fault probability, but the requirements at OS level have never been described and an actual implementation has never been proposed. In this paper we provide an analysis of the requirements and a description of the implementation of approximate memory management. Our approach allows Linux kernel to be aware of exact (normal) and approximate physical memories, managing them as a whole for the common part (e.g. optimization algorithms, page reuse) but distinguishing them in term of allocation requests and page pools management.

The new kernel has been built and extensively tested on a hardware x86 platform, showing the correctness of the implementation and of the fallback allocation policies.

I. Introduction

Approximate computing [1] has become a viable approach to energy efficient design of digital systems. Such an approach relies on the fact that many applications can tolerate a certain degree of approximation in the output data without affecting the quality of results perceived by the user. Approximate computing solutions are also supported by technology scaling factors, due to the growing statistical variability in process parameters which makes traditional design methodologies inefficient [2].

Memory represents a significant contribution to system power consumption in modern digital circuits and, when considering systems that spend most of their time in energy saving states (e.g. standby), it can reach up to 50% of total power consumption [3].

Main memory in modern systems is composed of DRAM cells that store data as charge on a capacitor. Due to leakage currents, the charge must be periodically restored by a refresh operation, which is usually performed in the background by dedicated hardware units (DRAM controller). This operation degrades performance, but also wastes energy, a drawback which is expected to worsen as DRAMs scale to higher capacities and densities.

Many works address the problem caused by DRAM refresh using hardware or software techniques [4], [5], trying to reduce refresh rate while conserving data integrity (e.g. using of error correcting codes, ECC) [6], [7].

In the late period a new approach has been proposed, lying in the general topic of approximate computing. Starting from the consideration that a range of applications (called error tolerant applications) do not require exact computation and storing of data, approximate computing design paradigm try to reach new trade-offs, exchanging energy consumption/performance for increased fault probability. These error tolerant applications are relatively insensitive to errors in large portions of their data, while require correctness only on a subset of data, defined as *critical data* [8].

Specifically for DRAMs, approximate computing techniques experiment the reduction of refresh rate (to the point of allowing controlled error rates in memory cells) saving significant energy. In [9] the authors propose to reduce refresh rate in selected DRAM memory banks by allowing software developers to specify critical and non-critical data in programs; physical memory is partitioned in two banks, one with regular refresh rate for critical data and one with reduced refresh rate for non-critical data. In order to implement the technique, some changes on hardware architecture and on software support are introduced; in particular, software requires OS to be aware of exact and approximate DRAM banks, providing a way to allocate non-critical data in the approximate bank.

Operating system support is a fundamental part of the approach and, while describing the general mechanism of allocating non-critical memory, authors do not investigate or provide implementation details on a specific OS. In this paper we propose and describe the implementation of approximate memory allocation support in the Linux OS kernel. The new OS kernel can be directly run in architectures containing banks of DRAM with reduced refresh rate, but it is not limited to them. In general, it provides support for any kind of system architecture where coexist exact and approximate banks of memory.

978-1-5090-6508-0/17/$31.00 ©2017 IEEE

In the following sections, we provide a description of the physical memory management in Linux OS kernel (Section II); then we describe the extension we have developed to support approximate memory allocation (Section III). We finally provide results, in the form of allocation statistics provided by the kernel, discussing the characteristics of the implementation.

II. Physical memory management in Linux kernel

A. Memory zones

Physical pages are the basic units of Linux Kernel memory management. Linux Kernel does not treat all pages in the same way, but groups pages having similar properties in zones. This partitioning has no physical relevance, but allows kernel to keep track of pages and overcomes hardware limitations (some pages cannot be used for specific tasks because of their physical memory address). There are five primarily zone:

- ZONE_DMA: pages within this zone can be used by DMA hardware.
- ZONE_DMA32: this zone is like ZONE_DMA but it can be accessed only by 32 bit devices.
- ZONE_NORMAL: pages within ZONE_NORMAL are directly mapped by the kernel.
- ZONE_HIGHMEM: this zone corresponds to *high memory* and it is not directly mapped by the kernel.
- ZONE_MOVABLE: Unlike memory zones described previously, zone movable does not have a specific physical range but pages within this zone come from other memory zones. The scope of this virtual memory zone is indeed to avoid memory fragmentation.

Zone attributes are considered hierarchically the order presented, i.e. a page in ZONE_NORMAL could be used to satisfy an allocation request for a lower hierarchy zone as ZONE_HIGHMEM, while a page in ZONE_DMA, being the higher in hierarchy, could satisfy every allocation request. All memory zones are not necessarily present, but depend on the hardware architecture, however ZONE_NORMAL and ZONE_MOVABLE are always enabled in the kernel. Partitioning memory pages into zones allows kernel to satisfy memory allocation requests as needed: some requests (e.g. allocation for DMA-able memory) could require pages only from a specific zone, other allocations instead could get pages from multiple zones.

B. Allocators

All interfaces provided by the kernel to allocate memory are based on a low level algorithm with page size granularity, called Binary Buddy Allocator [10]. According to this algorithm, physical memory is divided in power of two size blocks, when a block of the requested size is not available, a larger block is split in two half (buddies) and the process is iteratively repeated until a block of the requested size is produced (see Fig. 1).

In the Linux kernel, the core routine of buddy allocator is received as parameter, among others, a bit mask (called *gfp_mask*) which is a set of flags that allow the kernel to determine allocator behavior. In particular, these flags allow to specify the zone for the allocation, indicating that the kernel should allocate memory from the requested zone if possible. The allocator considers the zone specified in *gfp_mask* as an indication and, in some cases depending on actual memory utilization and balancing policies, the request could be satisfied selecting pages belonging to hierarchically higher zones.

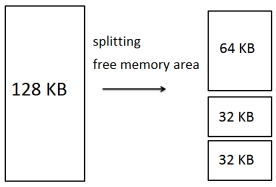

Fig. 1. Buddy system allocator

III. Kernel memory management extension

This paper proposes a new approach to introduce the support for approximate memories in Linux operating system, allowing the system to manage allocation requests of critical data, that must be stored in exact memory, and non-critical data, that can be stored in approximate memory. In this section we describe our extension to Linux kernel memory management, implementing a new memory zone, called ZONE_APPROXIMATE and a new dynamic allocator to select ZONE_APPROXIMATE for allocation.

A. Approximate zone creation

The introduction of ZONE_APPROXIMATE allows the kernel to manage a set of memory pages that will be selected when approximate memory allocations are requested. Pages within ZONE_APPROXIMATE will correspond indeed to a portion of physical memory where it has been allowed a certain amount of data corruption probability in order to save energy. The physical memory could be, for example, DRAM banks with reduced refresh rate, as described in Section I.

When we create the ZONE_APPROXIMATE region, we define it as the last memory zone because of kernel *fallback* mechanism. According to this policy, when an allocation request is scheduled, Linux kernel checks if the selected zone is suitable to satisfy the request; if the zone is not suitable, the kernel allocator falls back to a hierarchically higher zone. In other words, if an allocation requests the ZONE_APPROXIMATE region, but memory pages in this zone are not available, the request would be satisfied by one of the hierarchically higher zones (e.g. ZONE_NORMAL). This would result in storing approximated data in exact memory and cancel possible energy savings, but the functionality of the application would be not compromised. Moreover, ZONE_APPROXIMATE must be the last one in hierarchy also considering an allocation request for exact memory (ZONE_NORMAL or ZONE_DMA). When a normal (i.e. exact) allocation request is scheduled, but pages in ZONE_NORMAL cannot satisfy the request, kernel must never select ZONE_APPROXIMATE pages. In this way critical

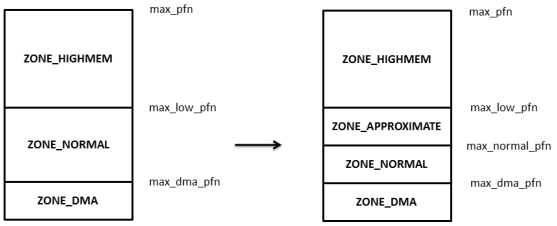

Fig. 2. Memory zones layout

data, that must be exact, will be excluded from approximate memory.

Finally, another step is required to complete the creation of ZONE_APPROXIMATE: the association of a physical address range to this zone. This step, as well as the layout of each memory zone, is architecture dependent, so we may have different sizes and layouts of approximate memory zone, depending on the architecture.

Summing up, as shown in Fig. 2, a new memory zone for storing approximated data is introduced in the Linux kernel. Fig. 2 shows the general layout where all zones are active, but we recall that the actual layout may differ since, depending on the architecture, some zones could be not present.

B. Approximate memory allocation

The next step is to implement a custom allocator to dynamically allocate non-critical data on ZONE_APPROXIMATE. As described previously, all internal functions provided by the kernel to allocate memory pages get a mask of *gfp_flag*s as parameter, which allows to drive the behavior of the allocator. In order to be able to specify ZONE_APPROXIMATE as favorite zone for allocation, we have defined a specific *gfp_flag* called GFP_APPROXIMATE: when this flag is set, the kernel tries to allocate memory using ZONE_APPROXIMATE pages.

Another important optimization mechanism inside the kernel that must be taken into account is *allocation fair policy*. According to this policy, kernel tries to balance allocation request by interleaving them between enabled zones, avoiding that a zone is saturated before other zones. Leaving the original policy, requests for ZONE_APPROXIMATE memory could be diverted to ZONE_NORMAL or ZONE_DMA even before ZONE_APPROXIMATE is full, resulting in sub-optimal allocation strategy. In order to properly handle allocation requests for ZONE_APPROXIMATE, we chose to disable kernel allocation fair policy for ZONE_APPROXIMATE (i.e. when GFP_APPROXIMATE flag is set).

The definition of GFP_APPROXIMATE flag plus changes to fallback mechanism and fair allocation policy, allows to block all allocations of ZONE_APPROXIMATE pages apart from explicit requests: if the kernel allocator routines do not get GFP_APPROXIMATE flag as parameter, ZONE_APPROXIMATE will never be selected as memory zone to satisfy the allocation request.

Eventually, since *gfp_flag*s are not defined outside kernel space, we implemented a new kernel space function dedicated to approximate memory allocation. The kernel space routine is then exported to a user space routine called *approx_malloc*, that allows user space applications to request pages from ZONE_APPROXIMATE.

TABLE I
HARDWARE SETUP: MEMORY ZONE LAYOUT

Zone	Physical range
ZONE_DMA	0-16M
ZONE_NORMAL	16M-456M
ZONE_APPROXIMATE	456M-896M

```
cat /proc/dmesg
...
895MB LOWMEM available
mapped low ram: 0 - 37f14000
low ram: 0 - 37f14000
Zone ranges:
DMA [mem 0x0000000000001000-
0x0000000000ffffff]
Normal [mem 0x0000000001000000-
0x000000001bf89fff]
Approximate [mem 0x000000001bf8a000-
0x0000000037f13fff]
...
```

Fig. 3. Output of *dmesg* command

IV. RESULTS

In this section we describe the experimental setup used to evaluate our Linux operating system with approximate memory support, showing results corresponding to memory allocations on ZONE_APPROXIMATE.

A. Hardware setup

The whole implementation work started from the mainline Linux kernel, version 4.3. We extended it with approximate memory support and, for the hardware dependent part, using the x86 32-bit architecture. In particular, since in this architecture only four memory zones can be enabled at the same time, we defined ZONE_DMA, ZONE_NORMAL, ZONE_MOVABLE and ZONE_APPROXIMATE. The extension of each zone is described in Table I.

We compiled and used our kernel to boot Linux on a x86 PC; in this setup the actual DRAM memory (exact) covers both the ZONE_NORMAL and ZONE_APPROXIMATE address spaces.

B. Allocation tests

As first test, after system boot, we examined the output of kernel ring buffer (Fig. 3) in order to check x86 RAM memory mappings and zone ranges. These logs come from the *dmesg* command and show that ZONE_APPROXIMATE has been properly created: our zone is the last kernel memory zone, after ZONE_DMA and ZONE_NORMAL, and it is mapped in the physical memory in the range 0x1bf8a000-0x37f13fff.

For evaluating our custom allocator, we wrote a test-bench application that calls the *approx_malloc* function to request a block of 1000 memory pages (about 4MB) from ZONE_APPROXIMATE. Using the *zoneinfo* system command

```
cat /proc/zoneinfo
...
Node 0, zone Approximate
pages free 114570
spanned 114570
present 114570
nr_dirtied 0
nr_written 0
...
```

Fig. 4. ZONE_APPROXIMATE statistics after boot

```
cat /proc/zoneinfo
...
Node 0, zone Approximate
pages free 113570
spanned 114570
present 114570
nr_dirtied 0
nr_written 0
...
```

Fig. 5. ZONE_APPROXIMATE statistics after *approx_malloc* call

we get the information printed in Fig. 4, which shows statistics about ZONE_APPROXIMATE before the allocation request. These lines show that ZONE_APPROXIMATE before allocation has 114570 free pages, considering that each page on x86 architecture is 4KB, we get confirmation that ZONE_APPROXIMATE size is about 440MB (see Table I).

Moreover we can see that statistics about ZONE_APPROXIMATE are all zero, confirming that kernel has not used approximate pages to allocate its data structures. This is a relevant aspect because pages in approximate region, being subject to faults, must not store critical kernel data and should never be allocated by kernel for its internal purposes.

Fig. 5 shows the same statistics after the allocation request. We can see now that ZONE_APPROXIMATE has 113570 free pages confirming that our custom malloc allocated exactly 1000 pages. We can also get kernel allocator information from the *vmallocinfo* command. Fig. 6 shows that the kernel allocator has requested and obtained 1000 pages in virtual address range 0xfc401000-0xfc7ea000.

In order to assess stability we extensively tested the system with allocation benchmarks programs, filling the whole ZONE_APPROXIMATE page set. As expected from that point on further requests of ZONE_APPROXIMATE pages caused allocations in ZONE_NORMAL, confirming that the fallback mechanism is working properly.

V. Conclusion

In this paper we presented an extension to Linux Kernel memory management in order to dynamically allocate non-critical data in a specific portion of memory, separated from exact RAM memory. This extension involved the creation of ZONE_APPROXIMATE, where approximated pages containing non-critical data can be grouped, and the implementation of a custom allocator to request pages within this zone. Pages within ZONE_APPROXIMATE are supposed to tolerate a certain amount of data corruption, allowing to trade off more relaxed specifications on data integrity with energy saving measures.

```
cat /proc/vmallocinfo
...
0xfb332000-0xfb336000 16384
n_tty_open+0x11/0xc0 pages=3 vmalloc
0xfb33d000-0xfb33f000 8192
bpf_prog_alloc+0x25/0x80 pages=1 vmalloc
0xfc401000-0xfc7ea000 4100096
SyS_approxvmalloc+0x23/0x90 pages=1000
...
```

Fig. 6. Output of *vmallocinfo* command

We tested our prototype kernel on a real x86 platform, showing that we are able to exercise dynamic memory allocation on ZONE_APPROXIMATE pages in a stable way.

We are currently working on the development of a hardware emulator for architectures containing approximated memories, based on QEmu [11]. An emulator with approximated memory, together with our Linux kernel extension, would allow indeed the execution of actual error tolerant applications, where faults could be injected at run time with predefined statistical properties. In this way it would be possible to evaluate output degradation on real applications and study the relationships between tolerable output error and energy savings.

Another important step would be the design of a hardware platform with approximated DRAM cells (e.g. DRAM banks with a slowed down refresh period). Booting the kernel on such platform would expose ZONE_APPROXIMATE pages to real hardware faults allowing to experimentally validate the whole technique against power consumption reduction.

References

[1] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18th IEEE European Test Symposium (ETS)*. IEEE, 2013, pp. 1–6.

[2] A. Mastrandrea, F. Menichelli, and M. Olivieri, "A delay model allowing nano-cmos standard cells statistical simulation at the logic level," in *Ph. D. Research in Microelectronics and Electronics (PRIME), 2011 7th Conference on*. IEEE, 2011, pp. 217–220.

[3] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 1–12.

[4] C. Isen and L. John, "Eskimo-energy savings using semantic knowledge of inconsequential memory occupancy for dram subsystem," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 337–346.

[5] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-aware placement in dram (rapid): Software methods for quasi-non-volatile dram," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*. IEEE, 2006, pp. 155–165.

[6] P. G. Emma, W. R. Reohr, and M. Meterelliyoz, "Rethinking refresh: Increasing availability and reducing power in dram for cache applications," *IEEE micro*, vol. 28, no. 6, 2008.

[7] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 83–93.

[8] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*, vol. 46. ACM, 2011, pp. 164–174.

[9] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving dram refresh-power through critical data partitioning," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 213–224, 2012.

[10] K. C. Knowlton, "A fast storage allocator," *Communications of the ACM*, vol. 8, no. 10, pp. 623–624, 1965.

[11] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.