ESKIMO - Energy Savings using Semantic Knowledge of Inconsequential Memory Occupancy for DRAM subsystem

Ciji Isen
ECE Department
University of Texas at Austin
isen@mail.utexas.edu

Lizy John
ECE Department
University of Texas at Austin
ljohn@ece.utexas.edu

ABSTRACT

Dynamic Random Access Memory (DRAM) is used as the bulk of the main memory in most computing systems and its energy and power consumption has become a first-class design consideration for modern systems. We propose *ESKIMO*, a scheme where when the program or operating systems memory manager allocates or frees up a memory region, this information is used by the architecture to optimize the working of the DRAM system, particularly to save energy and power. In this work we attempt to have the architecture work hand in hand with information about allocated and freed space provided by the program. We discuss multiple ways to use this information to reduce the energy and power consumption of the memory and present results of this optimization. We evaluate the energy and power benefits of our technique using a publicly available, hardware-validated, DRAM simulator, DRAMsim [1]. Our current studies show very promising results with energy savings on average of 39%.

Categories and Subject Descriptors

B.3.1 [**Memory Structures**]: Semiconductor Memories – *power and energy savings.*

General Terms

Management, Performance, Design, Experimentation.

Keywords

Memory power and energy, cross-boundary or cross-layer architecture optimizations, allocated and freed memory states, program semantic aware architecture.

1. INTRODUCTION

Dynamic Random Access Memory (DRAM) is used as the bulk of the main memory in most computing systems. In both the server and mobile space, both power and energy has been a first class design constraint [2, 3]. According to observations made at a recent International Solid-State Circuits Conference (ISSCC), Sun Microsystems revealed that the power consumption of DRAM in the UltraSPARC T1 ("Niagara") systems running SPECjbb was approximately 60 watts [30]. This amounts to nearly 22% of the total system power, in other words almost as much as the processor cores. Observations like this brought light to the power and energy consumption of DRAM memory, and made it a first-class design consideration for modern systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MICRO'09 December 12–16, 2009, New York, NY, USA.
Copyright © 2009 ACM 978-1-60558-798-1/09/12…$10.00.

With the need to reduce the power/energy consumption of the DRAM subsystem of modern systems in mind, we discuss some program semantics aware optimizations to the microarchitecture. Although there have been optimizations such as cache-locking, cache-bypass, prefetching etc, most of the modern optimizations done in the microarchitecture and memory subsystem tend to be agnostic of the program semantic. We focus our attention on our knowledge about program behavior and attempt to have the microarchitecture work with the information provided by the program. With *ESKIMO*, we propose to take advantage of the program's semantic notion of valid and invalid state for the memory regions. When a program or operating systems memory manager allocates or frees up a memory region, this program semantic information can be used by the architecture to optimize the working of the memory system.

In order to understand the optimization we discuss later, it is necessary to understand how the typical memory management works [4, 5, 6]. We make use of the program semantic observations resulting from memory management. Most program languages provide means for dynamic memory allocation (implicitly or explicitly). Considering its wide use and understanding, we will use C language constructs and assumptions for the purpose of examples. One of the commonly used ways for dynamic allocation in C is the malloc function. Its function prototype is

*void *malloc(size_t size);*

which allocates *size* bytes of memory. If the allocation is successful, a pointer to the block of memory is returned. If it fails a null pointer is returned. The pointer returned by malloc is a void pointer (void *), indicating the lack of any known data type. This pointer can be cast to the necessary type and assigned to a pointer variable. The memory allocated via malloc is persistent, i.e. it will continue to exist until it gets explicitly deallocated by the programmer (in the code) or the program terminates. The explicit deallocation (freeing the block of memory) is done with the help of the function "free". Its prototype is

*void free(void *pointer);*

which releases the block of memory pointed to by *pointer*. The address, *pointer* must have been returned previously by memory allocation functions such as malloc, calloc, or realloc etc. Once the address is *freed* any access to this memory location is incorrect and its behavior is undefined. Hence the programmer will not be using this location after it is freed and many modern programming

languages have mechanisms inbuilt to prevent such erroneous accesses. Consider this example program:

```
1.  int main(void){
2.    const char *p1 = "hello";
3.    char *p2;
4.    /* point zero */
5.    p2 = malloc(strlen(p1) + 1);
6.    if (p2 == NULL) {
7.        fprintf(stderr, " malloc failed.\n");
8.        exit(); }
9.    strcpy(p2, p1);
10.   /* point one */
11.   free(p2);
12.   /* point two */
13.   return 0; }
```

The diagrams (Figure 1.) show the situation at point zero (line 4), point one (line 10) and point two (line 12). At point zero (line 4), the pointer p2 has no memory allocated to it. After this, at point 1 (line 10), the pointer p2 has memory allocated to it as well data copied into the allocated space. At point two (line 12) we note that the pointer variable p2 still contains the address of a byte in the free store but that there is no longer an allocated block at that address. It would be a serious error to actually use that address. Thus, according to program semantics the address pointed to by p2 is not expected to contain useful data after point two. Similarly, if the pointer p2 were read before the strcpy function (line 9) one would get random data and even cause an error. So before strcpy function writes data to the allocated memory it contains random data and the program does not read from it without writing to it first. In both these cases the data present is invalid hence inconsequential to the program execution.

Hence we can have inconsequential memory accesses, i.e. the transfer of data between the caches and the DRAM that has not been initialized by the program, or has already been released by the program. The hardware transfers data based on demand, regardless of memory state. During a typical program execution all structures such as the stack and heap contain inconsequential data until they are allocated to some data structures and are initialized for sure. Figure 2 illustrates the different states a memory location can be in as a result of memory management. An unallocated-invalid memory location becomes allocated-invalid after allocation. It remains in this state until the memory location is written to, causing it to transition to allocated-valid state. From this point on that memory location can not be considered to be inconsequential. Eventually when the program is done with the memory location a free operation causes it to be returned to the heap; i.e. the location transition from an allocated-valid state to an unallocated-invalid state.

Research work that has attempted to exploit this information is sparse; Lewis et al. [32] being the closest. Although Lewis et al. explored using part of this semantic information (information relating to allocation and resulting allocated-invalid state) to optimize the cache to improve performance we are unaware of any work that has employed and analyzed it for optimizing DRAM power. In *ESKIMO*, we focus both on allocated-invalid and unallocated-invalid states to reduce power and energy consumption of the DRAM (shaded area in Figure 2). We present 4 techniques in section 3. Although Lewi's did not extend their work to save energy we extend Lewis's technique for DRAM to serve as a competition and call it Lewis++ (marked by dotted line in Figure 2).

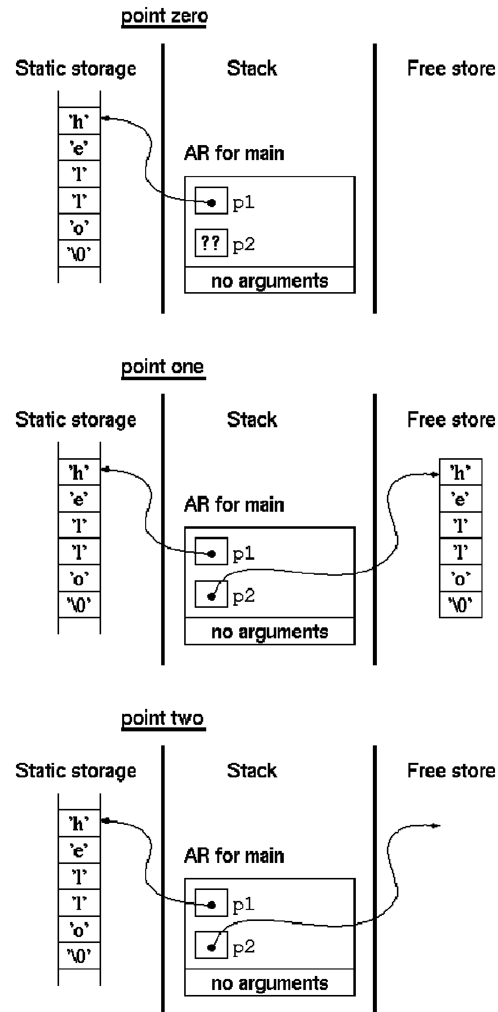

Figure 1. Changes to pointer status due to malloc & free [7]

The primary contributions of this paper are:

1. An insight into the usefulness of memory management semantic information from the program towards saving power and energy in the microarchitecture.
2. Employing known DRAM refresh technology to save power and energy by leveraging memory allocation/de-allocation semantic information.
3. Memory access optimizations (write-back, write-miss and virgin access) to the DRAM subsystem leading to reduction in energy consumption, and hence showing the potential of a program semantic aware memory controller.
4. Analysis Lewis et al.'s work on reducing write-miss if extended to DRAM, Lewis++.

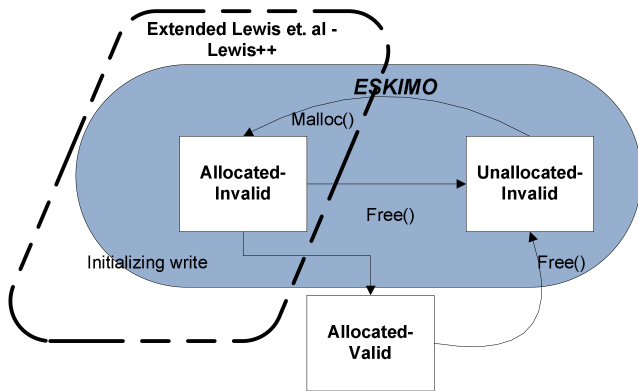

Figure 2. Inconsequent memory state transitions covered by ESKIMO vs. Our extension to Lewis et al. [32] (Lewis++)

2. ORGANIZATION

The rest of the paper is organized as follows. Section 3 discusses the various optimizations (4 in total) we propose to leverage the memory inconsequential state knowledge using *ESKIMO*. Section 4 discusses the details of mechanisms necessary to achieve this. Section 5 analyzes the allocation and de-allocation behavior of the benchmark suite we use. Sections 6 and 7 discuss our experiments and evaluation of both our system and the extended version of Lewis et al. we implement. We discuss related works in section 8 and summarize the paper in section 9.

3. ESKIMO BASED ADAPTATIONS

3.1 Semantics Aware DRAM Refresh

Due to the dynamic nature of a DRAM cell, periodic refresh operations are required for keeping the data stored. Even in standby mode, such regular refreshes account for a large energy consumption in DRAMs. Some studies have shown that even in the lowest power mode, the power needed to keep the DRAM contents is about a third of the total power dissipated. Factors such as the memory vendor and the design technology affect the refresh rate. A refresh interval of 32ms means, a refresh operation takes place every 32ms. A refresh operation fundamentally involves reading the DRAM cell out and writing back to the same cell. Although this refresh operation consumes, power and bandwidth, it is inevitable for the sake of data correctness.

We propose a technique to eliminate unnecessary refreshes by leveraging the program semantics information. The regions of memory that are marked as free (unallocated-invalid) or freshly allocated (allocated-invalid) have no concern related to correctness since the data stored is of no consequence to the correct execution of the program. Thus, not refreshing the data stored in these regions would not be a concern. Hence we propose to avoid refreshing the rows that are known to be "Inconsequent" from the program semantics. There are several technologies proposed in literature [31] as well as patents [35,36] that allow selective refresh of DRAM rows. We propose to employ these technologies along with knowledge of memory state to save power and energy.

3.2 Write Back Optimization

The cache hierarchy of a processor may operate on write-through or a write-back policy. Most modern caches tend to be write-back. In a write-back cache, writes are not immediately stored into the memory. Instead, the cache keeps tracks of the locations that have been written over (marks them as dirty). The data in these locations is written back to the memory when the data is evicted from the cache due to a new piece of data. Hence a read miss in a write-back cache, i.e. replacing a block with another, will often require two memory accesses to be serviced – one to fetch the necessary data for the read and one to write the replaced data from the cache to the store.

Programs typically allocate memory regions to compute and store data that may persist across different parts of the program in some what temporal fashion. Data stored in those allocated region might appear as dirty data but once the temporary use of the memory region is over (marked by a call to free), the program has no expectations of the data stored in these memory address. We call the state as "Inconsequent" state since the data stored in a location that has been freed is of no consequence to the correct execution of the program. We propose to use this to our advantage by storing this information in the cache lines. We would need a bit in addition to the dirty bit per cache line to mark the cache line. The implementation of the free function would mark and address range as free. If a cache line holds data for a location that has been freed, i.e. will not have any consequence on the execution of the program, we mark that cache line as "Inconsequent" via the dedicated bit.

In the context of the program semantic status of valid and "Inconsequent" cache lines one can perform this write-back more optimally. For example, some of the cache lines that have been marked dirty could become free and hence is in an "Inconsequent" state (i.e. unallocated-invalid in Figure 2). When a read miss occurs requiring an eviction of this cache line, one can avoid writing the replaced data to the next level. Hence have just one access to be serviced in that case as opposed to two as in regular caches. By doing this we are reducing the number of accesses to the DRAM resulting in reduction in the number of pre-charge discharge cycles as well as more opportunity for the DRAM to enter power saving mode. Hence one can reduce power as well as save on energy. This optimization is not applicable to all the cache lines that need a write-back since we can only avoid the write-backs to cache lines that are "Inconsequent". The data in these cache lines are not required by future state of the application since it was freed by the program. Thus not reflecting the modified value in the main memory is of no consequence to the program. Since this assumption is based on the program semantic and compiled code the system is not introducing any additional error.

3.3 Write Miss Optimization

A write miss occurs in a cache when a store attempts to store data and none of the cache ways contain the necessary cache line. In a cache that is write allocate, such a miss would result in a corresponding fetch from memory. The necessary cache line is fetched from memory and brought to the cache so that the write to the cache can proceed. This is important for correctness, particularly for partial writes (writes a fraction of the cache line in size). If this data was not brought from the main memory and then

the data write executed, the data would have to be written to the cache line with the rest of the cache line filled up with random data. Since this cache line would be marked as dirty after a write operation a future eviction to this cache line would result the whole cache line(including the random data) to be written back to the main memory, resulting in loss of data. For this reason and others a write-miss requires a corresponding read to memory.

The memory management library allocates and frees data for the program and these operations on the heap are done on a contiguous memory range. A write to a newly allocated memory space resulting in a write-miss though, has no useful data to gain by fetching the data from memory since the data present in this address range is random/"inconsequent". We propose to avoid this "inconsequent" read operation. We propose to achieve this by tracking the "inconsequent" address ranges at a per row granularity. We use the counter mechanism described in Section 4.1 and Figure 3. Using this, we can identify memory segments that have been freshly allocated were the data currently residing in the segment is of no consequence to the program. When a write-miss occurs, a fetch to the memory is issued. This fetch operation can be quickly resolved to be one to an "Inconsequent" memory segment. If so, this can be signaled back to the cache allowing it to proceed with the partial write to the cache line and intentionally disregard the random data occupying the rest of the line, i.e. the part of the cache line that was not written to. Hence with a small list of memory segments that are "Inconsequent" can help in avoiding a read operation to the main memory. By doing this we are reducing the number of accesses to the DRAM resulting in reduction in the number of pre-charge discharge cycles as well as more opportunity for the DRAM to enter power saving mode. Hence we can reduce power as well as save on energy.

3.4 Virgin Segment Optimization

This optimization is an extension of write-miss optimization. It is conceptually the same, but extended to apply to memory segments that are allocated by the operating system. When programs start up the operating system allocates a set of virtual memory pages for its use. Part of this space is populated by code and other data. We use the same tracking system discussed in Section 4.1 and Figure 3, to track segments allocated by the operating system too. Freshly allocated segments contain random data, which is of no consequence to the correct execution of the program. These segments are conceptually similar to an allocated memory (e.g. by malloc) with the only difference that, this space is allocated by the operating system and tends to be in much larger granularity. Until data is written to these segments, they can be assumed to contain random data that is of no consequence. Once data is written to these pages this assumption does not hold true.

As in the previous case, when a write-miss occurs, the fetch that will be issued can be quickly resolved to be one to an "Inconsequent" memory segment. If it is one such segment, this can be signaled back to the cache allowing it to proceed with the partial write to the cache line and intentionally disregard the random data occupying the rest of the line(the part of the cache line that was not written to). Similar to write-miss optimization this too can reduce the power consumed as well as result in energy savings.

4. ESKIMO IMPLEMENTATION

4.1 Allocation and De-allocation Tracking

The memory management library allocates and frees data for the program and these operations on the heap are done on a contiguous memory range. We propose to track the "inconsequent" address ranges at a per row granularity. Using this, we can identify memory rows that have been freshly allocated where the data currently residing in the page is of no consequence to the program. We could use a per row set of counters to track the malloc and free operations but we find empirically from our analysis that a limited set of counters would suffice (1k to 2k). We optimize this implementation by using a thousand counters which have a tag to indicate the row it is tracking. When the counter reaches full value indicating the whole row is inconsequential, a single bit flag at a row granularity is set. This can be stored inside the DRAM (option used for results) or a separate bit array large enough to contain 1 bit per row. The status of this is used to determine if the row is inconsequential and also used control refresh in our selective refresh policy.

Based on the DRAM configuration we assume, we need a counter that count up to 1k, i.e. 10 bits long. We also try a granularity of 4k (standard linux page size) which would need a 12bit counter but the results presented are for a 10bit counter. A *malloc* that is aware of the potential of semantic information would pass the address allocated to the system, which in turn can identify the row, lookup the 10 bit counter corresponding to it and increment it by the allocated size. Figure 4 is an example illustration. When the counter value for a row reaches the necessary size (1k or 4k), the memory segment can be considered to be "Inconsequent" as long as no writes have happened to it. For correctness, any write operation (besides inconsequent write-back) to an address inside a memory segment in the memory will reset the counter and hence it's "Inconsequent" status. When a row is found to be inconsequent, a bit corresponding to that row is set in the bit array. This frees up the counter to be reused. If a row that has its bit set in the bit array to indicate its inconsequential state, loses its inconsequential state due to a write operation or eviction or any other operation, the bit is reset.

Counter # 1	DRAM row 1 tag
Counter # 2	DRAM row 2 tag
	. . .
	. . .
Counter # 1000	DRAM row n tag

Figure 3. Per Row counter structure – 10 bit counter per row

4.2 Selective DRAM refresh

Due to the dynamic nature of a DRAM cell, periodic refresh operations are required for keeping the data stored. Even in standby mode, such regular refreshes account for a large energy consumption in DRAMs. Some studies have shown that even in

the lowest power mode, the power needed to keep the DRAM contents is about a third of the total power dissipated. Factors such as the memory vendor and the design technology affect the refresh rate. A refresh interval of 32ms means, a refresh operation takes place every 32ms. A refresh operation fundamentally involves reading the DRAM cell out and writing back to the same cell. Although this refresh operation consumes, power and bandwidth, it is inevitable for the sake of data correctness. There are several technologies proposed in literature [31] as well as patents [35,36] that allow selective refresh of DRAM rows. We propose to use the implementation suggested by Ohsawa et al. [31]. Figure 4-a illustrates how a flag is added to each row and how the value is set according to the status of the row, i.e. depending up on the presence of significant data. Figure 4-b shows the DRAM cell logic as illustrated by Ohsawa et al.

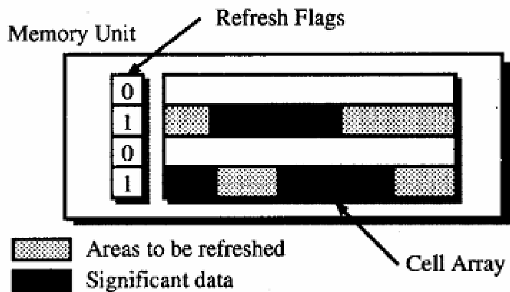

Figure 4-a. Per row flag to store refresh states[31]

4.3 Adaptation in ISA

Since any information that is parlayed between the program and the microarchitecture is done via the ISA, taking advantage of this program semantic information requires the help of the ISA. Most modern ISAs do have ISA instruction that could be modeled around to convey information about freed or allocated address space. For example, the x86 ISA has the INVD (invalidate data cache) and WBINVD (write back and invalidate data cache) instructions while PowerPC ISA has DCBI (data cache block invalidate) and ICBI (instruction cache block invalidate) instructions. We propose an extension to the ISA on similar lines that conveys information about addresses ranges that are allocated or freed. It is obvious from the optimizations discussed before that we need a mechanism in the ISA to pass information about allocated and freed memory segment. We propose two instructions modeled around some of the cache invalidate instructions.

Inconsequent on free - INQF *addr size*: This instruction is meant to be invoked by the free call and it is aimed at reporting the address range that has been freed. INQF tells the system that the address range starting at addr of size size has been freed. The system can now safely assume this address range to contain inconsequent data. Alternatively this instruction could take the form INQF addr where the size of the address range is implicit rather than explicit.

Inconsequent on malloc - INQM *addr size*: This instruction is meant to be invoked by the malloc call and it is aimed at reporting the address range that has been allocated. INQM tells the system that the address range starting at addr of size size has been allocated. The system can now safely assume this address range currently contains inconsequent data, i.e. until it is changed by some part of the program. Alternatively this instruction too could take the form INQM addr where the size of the address range is implicit rather than explicit.

4.4 Overhead

We now consider the storage overhead for the counter structure. We need to maintain 1000-2000 counter tag pairs. Each counter tag pair needs 10 bits for counting and 48 bits (current virtual address range) i.e. a total of 58 bits. Thus we need approximately 116 KB of storage which amounts to less than 0.01% of the main memory we model. Previous studies such as that of Ohsawa et al. [31] have analyzed the cost of implementing the selective refresh logic in hardware and have found it to be very small. Their estimate based on silicon modeling attributes less than 5% area overhead to incorporate all the necessary storage and control logic.

5. DYNAMIC MEMORY ALLOCATION

We can observe the dynamic memory activity to the heap by tracking invocations of malloc() and free() routines (along with other related routines such as realoc()). In Table 1 we summarize the percentage of procedure calls for allocation and percentage of space allocated according to different allocation sizes. For example astar has 21.68% of allocation calls invoked to allocate memory regions of size smaller than 64 bytes. The total number of allocation calls for astar is 1117*1000. Similarly 97.1% of the total 545MB space allocated by hmmer was given to data chunks of size ranging between 64 bytes and 2k bytes. Similarly in Table 2 we summarize the percentage of procedure calls to free and the percentage of space freed up according to different allocation sizes. For example gobmk has 71.94% of calls to free memory freeing up data of size less than 64 bytes; in total gobmk had 104 calls to free data. The same benchmark has 98.25% of the space allocated being freed up in chunks of size ranging from 2k to 256k.

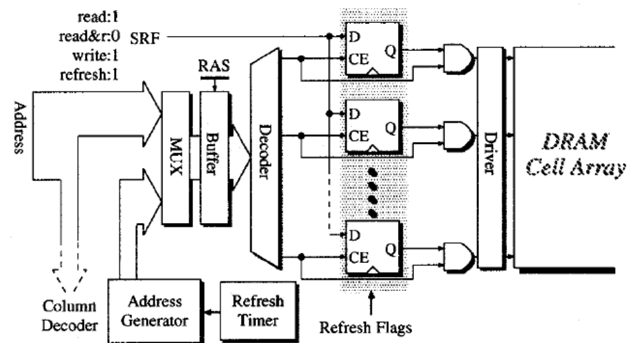

Figure 4-b. DRAM cell array - selective refresh [31]

There are some interesting observations we make from this data. There is a clear difference in the data chunks the allocation calls target and the data chunks that make up most of the allocation. For example, *gcc* has 64.81% of its allocation function calls targeted at chunks of size less than 64 bytes but only 0.82% of the total allocated bytes are made up of this chunk size. The chunk size of 2k-256k range has 93.29% of the memory allocated even though only 10.27 % of the calls target this range. A similar observation applies to dynamic de-allocation too. For example,

Table 1: Dynamic memory allocation behavior

Benchmark	% of Allocation Calls						% of Allocated Bytes					
	<64	<2K	<256K	<16M	>=16M	Total (1k)	<64	<2K	<256K	<16M	>=16M	Total(MB)
astar	22	77	2	0	0	1117	1	82	9	8	0	997
bzip2	0	0	46	43	11	<1	0	0	0	4	96	628
dealII	92	8	0	0	0	153873	49	24	5	8	15	10819
gcc	65	25	10	0	0	2920	1	3	93	3	0	6634
gobmk	74	0	26	0	0	119	1	0	80	19	0	123
h264ref	5	89	5	1	0	105	0	4	23	73	0	1026
hmmer	3	97	0	0	0	1000	0	97	0	2	0	545
lbm	0	50	0	0	50	<1	0	0	0	0	100	409
libquantum	29	6	9	40	16	<1	0	0	0	39	61	1486
mcf	0	40	0	40	20	<1	0	0	0	0	100	1676
Milc	0	0	0	62	38	7	0	0	0	35	65	84226
Namd	0	42	57	1	0	1	0	1	28	71	0	45
omnetpp	19	81	0	0	0	267065	3	97	0	0	0	42503
perlbench	18	64	15	3	0	22917	0	0	68	32	0	592549
Povray	96	4	0	0	0	2462	43	21	37	0	0	114
Sjeng	0	20	0	20	60	<1	0	0	0	7	93	172
Soplex	1	2	90	7	0	9	0	0	9	22	68	3186
Sphinx3	66	17	17	0	0	14225	1	21	67	10	0	15398
xalancbmk	67	25	7	0	0	135184	3	44	54	0	0	59352

Table 2: Dynamic memory de-allocation (free) behavior

Benchmark	% of Free Calls						% of Freed Bytes					
	<64	<2K	<256K	<16M	>=16M	Total (1k)	<64	<2K	<256K	<16M	>=16M	Total(MB)
astar	22	77	2	0	0	1117	1	82	9	8	0	997
bzip2	0	0	50	50	0	<1	0	0	1	99	0	25
dealII	92	8	0	0	0	153873	49	24	5	8	15	10819
gcc	65	25	10	0	0	2876	1	2	93	3	0	6521
gobmk	72	0	28	0	0	104	1	0	98	0	0	93
h264ref	5	89	5	1	0	105	0	4	23	73	0	1026
hmmer	3	97	0	0	0	1000	0	98	0	2	0	542
lbm	0	50	0	0	50	<1	0	0	0	0	100	409
libquantum	39	4	9	26	22	<1	0	0	0	6	94	935
mcf	0	40	0	40	20	<1	0	0	0	0	100	1676
milc	0	0	0	62	38	6	0	0	0	35	65	83613
namd	0	42	57	1	0	1	0	1	28	70	0	45
omnetpp	19	81	0	0	0	266999	3	97	0	0	0	42499
perlbench	20	78	3	0	0	18643	0	6	82	12	0	23231
povray	98	2	0	0	0	2427	67	13	20	0	0	73
sjeng	0	100	0	0	0	<1	0	100	0	0	0	<1
soplex	3	3	85	9	0	4	0	0	15	31	54	1160
sphinx3	65	17	17	0	0	14024	1	21	67	10	0	15358
xalancbmk	67	25	7	0	0	135184	3	44	54	0	0	59352

xalancbmk has 66.47% of the de-allocation calls for chunk size of <64 bytes amounting to only 2.79 % of the total data de-allocated; while 7.18% of the de-allocation calls to chunk size 2k-256k amount to 53.51% of the total data freed. We also that some of the benchmarks do have a significant percentage of data allocated and freed in chunk sizes less than 64 bytes. For example *dealII* and *povray* have 49 and 42.5% of data allocated while 49 and 66.6% of data de-allocated less than 64 bytes. When tracking the allocation and de-allocation behavior in programs we need to be able to track smaller granularities too for benchmarks like this. These examples show how diverse the benchmarks are with respect to the chunk sizes they concentrate on during allocation and de-allocation. This makes any assumption about the predominant chunk size difficult. Very interestingly, for some benchmarks both the allocation and de-allocation patterns follow very closely. For example *astar* has 21%, 76% and 1.8% of the allocation invocation for types <64, <2k and <256k respectively. The same distribution is observed by *astar* for de-allocation/free calls. For *astar* the observation can be made for the dynamic bytes allocated and freed.

6. EXPERIMENTAL SETUP

Our experiments use a cache simulator built on top of PIN [9, 10] and a DRAM simulator, DRAMsim [1] that models both power

and latency. The DRAMsim is a hardware-validated, public-domain DRAM system simulation code that was developed by members of the Systems and Computer Architecture Lab (SCAL) in the Department of Electrical and Computer Engineering at the University of Maryland. We use the DRAM simulator to model DRAM power and focus our analysis on results based on this. The cache simulator assumes a simplistic CPU architecture close to that of an Intel ATOM processor, an in order processor with a simplified pipeline and system architecture. The important assumptions about the memory subsystem are summarized in Table 3. Our simulator uses x86 binaries of the benchmarks and can simulate the allocation and de-allocation behavior in C and C++ benchmarks. We use a subset of SPEC CPU 2006 benchmarks and inputs based on the suggestions of Phansalkar et al. [11] to evaluate our optimizations. A subset of the benchmark suite was picked to reduce simulation time as well as avoid benchmarks which could not be simulated (particularly fortran code). We simulate 1 billion instructions for each benchmark and believe it is sufficient to prove the usefulness of our techniques. We fast forward the initial 500 million instructions to avoid startup behavior and simulate the next 500 million instructions. During the fast forward stage we do keep track of all allocation and de-allocation behavior to ensure correctness.

Table 3: DRAM Module and L2 Cache Configuration

Parameter	Value	Parameter	Value
Type	DDR2	*Columns*	1024
Size	1GB	*Data Width*	72 bits
Rows	16384	*Refresh Interval*	32ms
Frequency	667*(MHz)*	*L2 cache size*	1MB
Banks	8	*L2 cache way*	8way
Ranks	2		

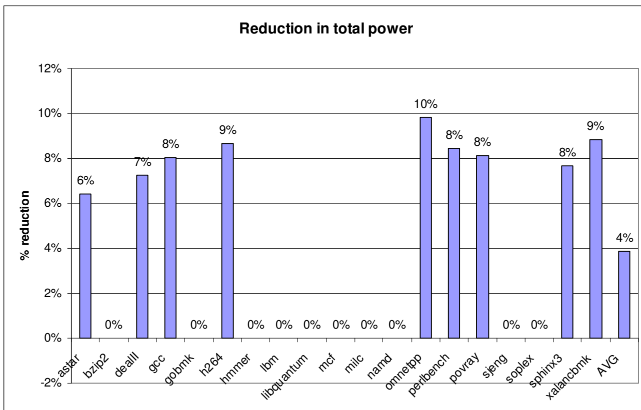

Figure 5. Savings in Total power due to ESKIMO based DRAM refresh optimization

7. RESULTS

In this section we will present and discuss the results of ESKIMO. Although Lewis et. al [32] did not extend their work to save energy we extend Lewis's technique to save energy and apply it to DRAM to serve as a competition. We call this extension ***Lewis++*** and present its results along with ***ESKIMO*** when applicable.

7.1 Semantic Aware Dram refresh

We implemented the refresh optimization based power saving technique in our simulator and model the power savings for it. We compare the power of our scheme to that of the baseline and compute the savings in power. In Figure 5 we present the savings in total power arising from the semantic aware DRAM refresh. . Note that, the refresh power of a DRAM is only a part of the power consumed by the DRAM. Benchmarks such as astar, dealII, gcc, h264, omnetpp, perl, povray, sphinx3 and xalanc are able reduce total power by a significant amount. The benchmark to demonstrate the best power savings is omnetpp, with a 10% reduction in total power consumed by the DRAM. The savings in power among the benchmarks with some impact ranges from 6% to 10%; the others have no impact.

7.2 ESKIMO Write Miss vs. Lewis++ Store Miss Optimization on Dram

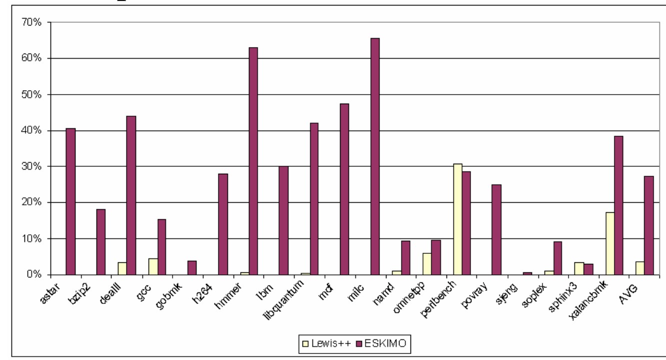

Figure 6-a. Reduction in memory cycles in the DRAM

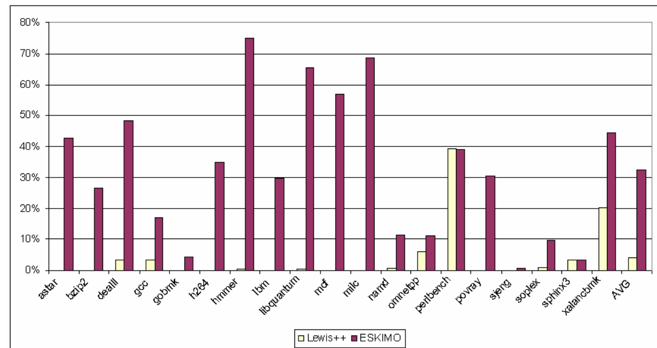

Figure 6-b. Reduction in energy in the DRAM

Figure 6 shows the comparison between the *Lewis++* and *ESKIMO* with only write-miss optimization in place. *ESKIMO* significantly out performs *Lewis++* for energy and memory cycles. The reduction in memory cycle is directly related to the amount of write-misses that can be avoided. The work by Lewis et al. was done on SPEC cpu 2000 benchmarks which made less aggressive use of memory management. In fact, in SPEC cpu2000 has only one C++ benchmark while most of the benchmarks in our suite are C++ programs. These programs tend to make more aggressive and dynamic use of allocation and de-allocation as is evident from Table 1 and Table 2. Our ability to use counters and off-load filled up counters as flag bits per row allows us to get far better reach for our tracking mechanism. Clearly write-miss

optimization results in significant reduction in energy consumption by the DRAM.

7.3 Virgin Segment Optimizations

Virgin segment optimization typically is caused by OS or library activity and tends to be much clustered. In Figure 6 we present the reduction in energy consumed by DRAM we observe due to this optimization. We observe a reduction of up to 47% for lbm. A few other benchmarks such as gobmk, h264, hmmer etc have significant reduction while the others gain very little from this optimization. On an average we observe an improvement of 9%. In our studies we found that *Lewis++* performs as well as *ESKIIMO* in tracking this artifact. Since these allocations arise from the OS of the library, it is easy to separate them from regular allocation and de-allocation and hence avoid its polluting effect. We present only one graph (Figure 7) for this reason and make no distinction.

7.4 Total Energy Savings

In this section we present data regarding the total energy savings the DRAM achieves as a result of all four of our proposed optimizations. Figure 8 has the total energy savings achieved by *ESKIMO* and *Lewis++*. In the extension we also include virgin segment tracking using their scheme and include its savings towards their total savings. *ESKIMO* gets an average savings of 39% in energy while *Lewis++* gives us an average of 13% savings. The savings we obtain varies widely from benchmark to benchmark. In general, benchmarks which exhibit large clustered allocation and de-allocation benefits significantly from both the write-miss and virgin segment optimization. These two optimizations make up for the bulk of the energy savings we observe. For *lbm*, which does exceptionally well, close to 100% of the 0.5GB of data allocated occur in large chunks of size >= 16M. This pattern makes it amenable to write-miss as well as virgin segment tracking provided the tracking system is able to filter off 50% of the allocation calls which result in very little of the data allocated. Similar pattern can be observed for *mcf* and to some extend *milc*. It is possible that further tuning could improve the performance of Lewis et al.'s work which would give further credibility to the usefulness of tracking allocation and de-allocation patterns to save energy in the DRAM system.

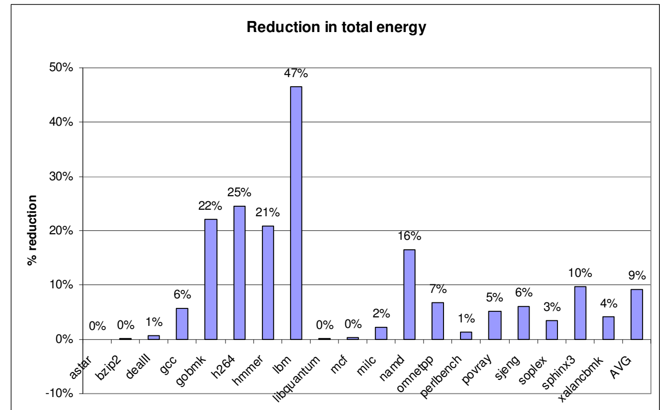

Figure 7. Virgin segment optimization – energy savings

8. RELATED WORK

Prior work in making informed discussions based on the block status was done both in software and hardware. Some techniques have tried to identify this at the software level [12,13] while others have attempted to do the same at the hardware level [14,15,16,17]. Most of these techniques try to identify blocks that are not likely to be used in the near future. Software solutions do this by passing hints about blocks that are thought to be not likely to be used in the near future to the hardware – based on inferences from profiling or compiler analysis [12,13]. Hardware solutions employ predictors to predict those blocks that are not likely to be used in the near future. The predictor does this based on the data address [15] or the program counter [14,16,17]. All these approaches differ significantly from our approach in the fact that they predict the likelihood of usage and attempt to use that while our work bases its self on the knowledge from program semantics about validity of a block from the programs perspective.

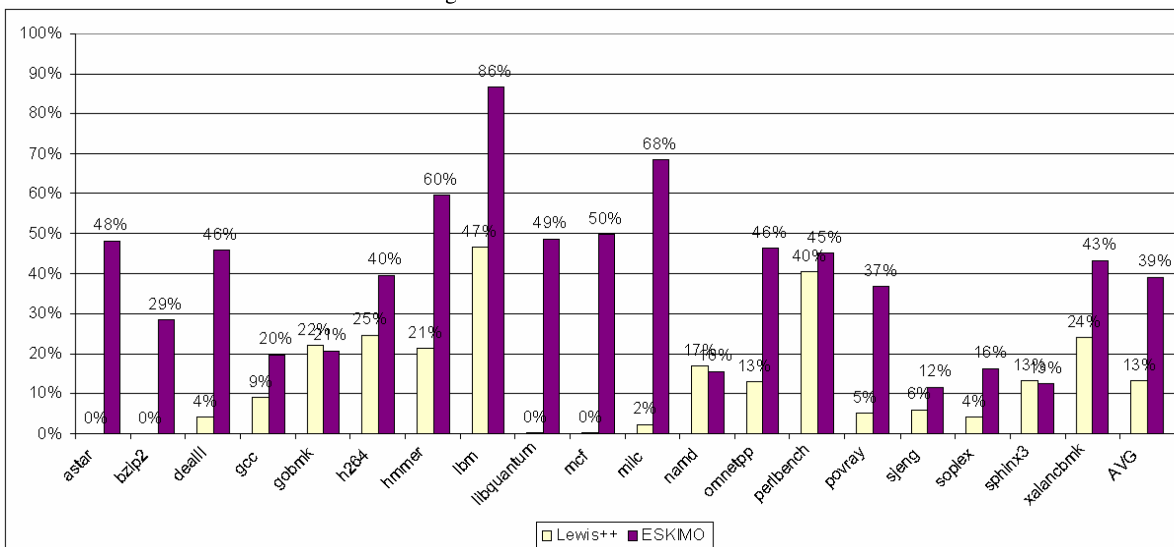

Figure 8. Total energy savings from all optimizations

This is different from predicting how likely a block is to be used in the near future and there by predicting how useful it is to keep a certain block in the cache.The knowledge based on program semantics allows us to optimize the DRAM subsystem to avoid some of the operations without fear of incorrectness; the same can not be done with predictive techniques. Lewis et al.[32] explored using program semantic information about allocated space for caches and at cache block granularity to improve performance; ours is for DRAM and at DRAM row granularity. Additionally, energy savings were never explored in Lewis et al.

The application of this knowledge has some similarity to existing uses of block lifetime prediction and a few others which are unique to block semantics invalidity. Some of the uses that have also been sighted in prior work related to block lifetime prediction are perfecting [17,15,17], replacement [16], bypassing [18,19,20,21,22], coherence protocol optimizations [23,24,25] and to a limited extend power reduction [26,14,27]. Works by Lai et al [17], Hu et al [15], Ferdman *and* Falsafi [17] use the predictions of the block lifetime to trigger pre-fetches; Lai et al [17], Hu et al [15], pre-fetched into the L1 data cache while Ferdman and Falsafi did the same from off-chip to on-chip memory. Kharbutli and Solihin [16] used the knowledge of block lifetime to improve the LRU algorithm by replacing dead blocks first and also bypassing the cache. Cache coherence protocols have also been tuned to take advantage of block lifetime prediction to maintain or avoid status updates. Wood proposed a reduction in cache coherence protocol overhead by invalidation some of the shared cache blocks early [23]. Lai and Falsafi [24] employed a predictor based on program counter to predict last-touch and decide when blocks should be invalidated. PC-traces are used to identify last stores to a cache block in Somogyi et al's work [25]. Power saving techniques employ different hardware techniques to save power by turning off (Kaxiras et al[26]) or gating/putting to sleep (drowsy caches [27]) that are predicted to be not useful in the near future (i.e. based on block lifetime prediction). Venkatesan et al. in [28] introduced a retention-aware placement algorithm which tries to reduce the refresh operations by experimentally identifying that, different rows require different refresh times. Mrinmoy et al [29] suggested a technique to identify rows that have been refreshed by a memory access and avoid refreshing those rows when possible. Murakami [31] presents the benefits of selective DRAM refreshing using OS or compiler, however they do not describe how exactly this is done. Theirs essentially is a limit study evaluating the benefits of capturing all condition where refresh can be avoided. In our paper we are describing and evaluating mechanisms to achieve part of the benefits. Jouppi [33] investigated a cache policy, 'write-validate", which does word-level sub-blocking [34]. In this policy data for the write is not fetched but rather written directly to the cache line with the valid bits turned off for all but the data being written. This could potentially eliminate all write-misses; but the implementation overhead of this scheme is significant. Wulf and McKee proposed having a "first write" instruction to bypass cache stall due to write-miss. The PowerPC has an instruction *dcbz* geared towards this end. Our work uses two instructions whose application goes beyond write misses and helps us track several different artifacts and reap benefits from them.

ESKIMO relies on semantic information available from the program allowing for the system to act with out fear about correctness. In this paper we study the impact of these optimizations on power and energy consumption. *ESKIMO* in many ways can work in a complementary fashion with most of the previous power saving techniques. It could also be applied to other areas were block lifetime prediction has been put to use to but the converse is not true since it requires accurate information. We do not study such where block lifetime prediction have been used for in this paper since we focus primarily on power and energy.

9. SUMMARY

In the current world where power is a first class design constraint for system architects (both in the server and mobile space), the consumption of power by the memory subsystem is of particular importance. Observations about the memory power being comparable to that of the core power, points to need to focus our attention on the energy consumed by the memory subsystem. In our paper we propose some enhancements to the microarchitecture and memory subsystem. *ESKIMO* works by taking advantage of some information one can exploit by virtue of the program semantics. We propose adjustments to the ISA in the form of two instructions in order to pass information about addresses and size that are *allocated* or *freed* from the program to the architecture. With the help of these two instructions, we propose mechanisms to reduce the power consumption as well as energy. *ESKIMO* reduces the amount of energy consumed by refresh cycles and the amount of write-backs and reads caused by write-misses by being cognizant about the inconsequent nature of a memory address range. These techniques reduce the pressure on the memory and the amount of charging and discharging of lines required there by reducing the power and energy consumed by the memory subsystem. We evaluate the energy and power benefits of our technique using a publicly available, hardware-validated, DRAM simulator. We also extend Lewis et al.'s work for DRAM and compare it to your scheme. For the benchmarks we simulated, the savings in energy consumption of the benchmarks range from 12% to 86% with an average of 39%.

10. ACKNOWLEDGMENTS

The authors are supported in part by NSF grant 0702694 and an IBM Faculty award. Ciji Isen is also supported by an IBM PhD Fellowship. Any opinions, findings and conclusions expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF) or other research sponsors.

11. REFERENCES

[1] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, Bruce Jacob, DRAMsim: a memory system simulator, ACM SIGARCH Computer Architecture News, v.33 n.4, November 2005

[2] T. Mudge. "Power: A first class design constraint," Computer, vol. 34, no. 4, April 2001, pp. 52-57.

[3] Barroso, L., The Price of Performance, ACM Queue, Volume 3, Number 7, September 2005.

[4] A. S. Tanenbaum, Modern Operating Systems, 1st ed. Englewood Cliffs, NJ: Prentice-Hall, Feb. 1992.

[5] Bruce Eckel. Thinking in C++: Introduction to Standard C++, volume 1. Prentice Hall, 2000.

[6] Bruce Eckel. Thinking in C++: Practical Programming, volume 2. Prentice Hall, 2003.

[7] http://enel.ucalgary.ca/People/Norman/engg333_fall1996/stat_dyn/

[8] Micron. Various Methods of DRAM Refresh. http://download.micron.com/pdf/technotes/DT30.pdf

[9] Chi-Keung Luk , Robert Cohn , Robert Muth , Harish Patil , Artur Klauser , Geoff Lowney , Steven Wallace, Vijay Janapa Reddi , Kim Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, June 12-15, 2005, Chicago, IL, USA

[10] Pin Website: http://www.pintool.org/

[11] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in ISCA '07:Proceedings of the 34th Annual international symposium on Computer architecture. New York, NY, USA: ACM Press, 2007, pp. 412–423.

[12] J. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang. Cooperative caching with keep-me and evict-me. In The 9th IEEE Annual Workshop on the Interaction between Compilers and Computer Architectures, 2005.

[13] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, 2002.

[14] J. Abella, A. Gonz`alez, X. Vera, and M. F. P. O'Boyle. IATAC: a smart predictor to turn-off L2 cache lines. ACM Transactions on Architecture and Code Optimization (TACO), 2(1):55–77, March 2005.

[15] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: Predicting and optimizing memory behavior. In Proceedings of the 29th International Symposium on Computer Architecture, 2002.

[16] M. Kharbutli and Y. Solihin. Counter-based cache replacement and bypassing algorithms. IEEE Transactions on Computers, 2008.

[17] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead block correlating prefetchers. In Proceedings of the 28th Annual International Symposium on Computer Architecture, pages 144–154, 2001.

[18] A. Gonz´alez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In Proceedings of the 9th International Conference on Supercomputing, 1995.

[19] J. Jalminger and P. P. Stenstr¨om. A novel approach to cache block reuse prediction. In Proceedings of the 2003 International Conference on Parallel Processing, 2003.

[20] T. L. Johnson, D. A. Connors, M. C. Merten, and W. W. Hwu. Run-time cache bypassing. IEEE Transactions on Computers, 1999.

[21] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing resue information in data cache management. In Proceedings of the 12th International Conference on Supercomputing, 1998.

[22] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In Proceedings of the 28th Annual International Symposium on Microarchitecture, 1995.

[23] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 48–59, 1995.

[24] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In Proceedings of the 27th Annual International Symposium on Computer Architecture, pages 139–148, 2000.

[25] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Memory coherence activity prediction in commercial workloads. In 3rd Workshop on Memory Performance Issues, 2004.

[26] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In Proceedings of the 28th Annual International Symposium on Computer Architecture, 2001.

[27] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002.

[28] R. Venkatesan, S.Herr, and E. Rotenberg. Retention-Aware Placement in DRAM (RAPID):Software Methods for Quasi-Non-Volatile DRAM. In Proceedings of the Twelfth Annual Symposium on High Performance Computer Architecture, pages 155.165, Nov. 2006.

[29] Mrinmoy Ghosh , Hsien-Hsin S. Lee, Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs, Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, p.134-145, 2007

[30] Laudon, J., UltraSPARC T1: Architecture and PhysicalDesign of a 32-threaded General Purpose CPU, Proceedings of the ISSCC Multi-Core Architectures, Designs, and Implementation Challenges Forum, 2006.

[31] Ohsawa, T.; Kai, K.; Murakami, K., Optimizing the DRAM refresh count for merged DRAM/logic LSIs. Proceedings. 1998 International Symposium on Low Power Electronics and Design, vol., no., pp. 82-87, 10-12 Aug 1998

[32] Lewis, J.A.; Black, B.; Lipasti, M.H., Avoiding initialization misses to the heap, Proceedings. 29th Annual International Symposium on Computer Architecture, vol., no., pp.183-194, 2002

[33] Jouppi, N.P., Cache Write Policies And Performance, Proceedings of the 20th Annual International Symposium on Computer Architecture, vol., no., pp.191-201, 16-19 May 1993

[34] Cragon, H.G. "Memory Systems and Pipelined Processors". Jones and Bartlett Publishers, Inc., Sudbury, ME, 1996.

[35] US Patent 6542958 - Software control of DRAM refresh to reduce power consumption in a data processing system

[36] US Patent 6094705 - Method and system for selective DRAM refresh to reduce power consumption