Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters

Christian Helm
The University of Tokyo
Tokyo, Japan
christian@eidos.ic.i.u-tokyo.ac.jp

Soramichi Akiyama
The University of Tokyo
Tokyo, Japan
akiyama@ci.i.u-tokyo.ac.jp

Kenjiro Taura
The University of Tokyo
Tokyo, Japan
tau@eidos.ic.i.u-tokyo.ac.jp

***Abstract*—The memory controller of a processor translates the physical memory address to hardware components such as memory channels, ranks, and banks. This DRAM address mapping is of interest to many researchers in the fields of IT security, hardware architecture, system software, and performance tuning. However, Intel processors are using a complex and undocumented DRAM addressing. The addressing can be different for every system because it depends on many aspects such as the processor model, DIMM population on the motherboard, and BIOS settings. Thus an analysis for every individual system is necessary. In this paper, we introduce an automatic and reliable method for reverse engineering the DRAM addressing of Intel server-class processors. In contrast to existing approaches, it is reliable, measurement errors are unlikely to occur, and can be detected if they occur. Our method mainly relies on CPU hardware performance counters to precisely locate the accessed DRAM component. It eliminates the problem of wrong attribution that is common in timing based approaches. We validated our method by reversing engineering the DRAM addressing of a diverse set of Intel processors. This set includes Broadwell, Haswell, and Skylake micro-architectures, with various core counts, DIMM arrangements, and BIOS settings. We show the correctness of the determined addressing functions using micro-benchmarks that access specific DRAM components.**

***Index Terms*—DRAM, Reverse Engineering, Address Mapping, Performance Counters**

I. Introduction

The memory subsystem of a modern computer is complex. The memory is split into different channels to provide higher bandwidth. Organization of DRAM chips in bank groups and banks provide the opportunity for pipelining requests. This has led to increased throughput of DRAM systems over the years without a significant performance increase in the single DRAM cell [1]. The memory controller must interface with those different DRAM components and address them individually. The memory controller receives requests to load data at specific physical addresses. From the physical address, the DRAM controller must determine the channel, rank, bank, row, and column in which the data is stored. The definition of how addresses are translated is called DRAM address mapping.

If the DRAM address mapping is known, it enables a wide range of applications. In hardware architecture and system software research, approaches for better usage of memory channels and banks are being explored. For example, application-aware memory channel partitioning [2], adapted page sizes for better usage of row buffers [3], efficient use of new hybrid memory technology [4], effects of unreliable memory [5], or DRAM layout aware memory allocators [6], [7], [8]. For the evaluation of such new concepts, simulated hardware is often used. If the address mapping is known, such a system can be implemented and evaluated using real processors and applications. The few cases where evaluations are done using real hardware rely on specific processor models where the address mapping is documented or require manual reverse engineering effort. For example, Chandru and Mueller [8] use a Tilera processor, Pan et al. [7] use an AMD Opteron 6128 processor, and PALLOC [6] is implemented on a Xeon W3530 and Freescale P4080. The above-mentioned concepts also showed performance gains which we cannot utilize on other machines without knowing the address mapping.

Researchers in IT security are also interested in the DRAM address mapping to evaluate their concepts. Covert communication channels across CPUs were demonstrated by Pessl et al. [9]. A variation of Rowhammer [10] attacks was introduced by Gruss et al. [11]. And Song et al. show a method for hiding rootkits [12].

The address translation is done in hardware, it is different for every system, and it is mostly undocumented except for a few specific processor models. For example, the documentation of the outdated Intel Xeon 5500 contains a description of the address mapping [13]. However, for newer generations of Intel processors, this information is not published. Up to a certain extent, the mapping is configurable in hardware. At the startup of the system, the BIOS reads the DIMM configuration and programs the configuration registers in the processor. After the initialization, the configuration cannot be changed. The mapping can also be influenced by the BIOS settings. For example, the activation of on-chip NUMA domains changes the addressing [14]. Thus the address mapping depends on many factors and may be different for every system. A general addressing function, for example for a specific processor generation, does not exist.

We introduce an automatic and reliable method for reverse engineering the DRAM addressing on Intel processors. Our tool can automatically find the addressing functions of mem-

This work is partially supported by a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

ory channels, ranks, bank groups, and banks. It is available online at https://github.com/helchr/reMap. The main idea is to directly and reliably measure the number of accesses to each component (e.g., bank), unlike existing approaches that leverage unreliable numbers such as access latency or the number of bit-flips by rowhammer. With a smart selection of probed addresses, we can determine the addressing functions in a short time. Because we use boolean algebra to resolve the addressing functions, we can differentiate measurement errors from insufficient sampling. In the case of asymmetric DIMM population, we gather additional information from configuration registers. Our tool supports server-class processors with a large amount of memory and also supports asymmetric DIMM population. We demonstrate that we can reverse engineer the address mapping on an Intel Haswell, two Broadwell, and a Skylake system. They are equipped with up to 2TB of RAM and include a system with asymmetric memory channel population. Based on benchmarks that access certain components of the DRAM using the reverse engineered address mapping, we confirm that our method can find the correct address mapping on recent Intel server-class processors.

II. DRAM Fundamentals

Modern DRAM is organized in a hierarchical arrangement of channels, ranks, bank groups, and banks as shown in Figure 1.

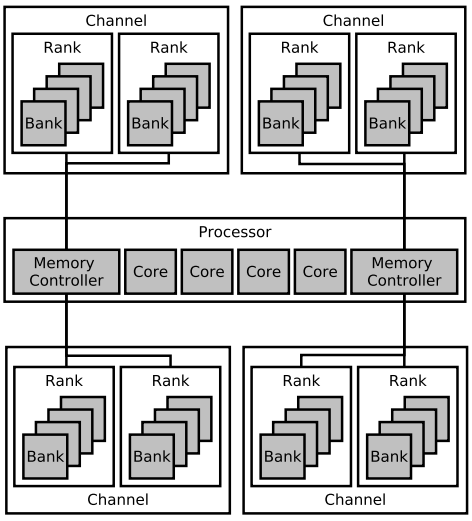

Figure 1: High level DRAM system organization.

A processor has a memory controller for interfacing the DRAM system. There can also be multiple memory controllers within one CPU. Each of them can have multiple channels. Often, only the total number of channels, but not the separate memory controllers are mentioned in hardware descriptions. E.g. the system in Figure 1 could be seen as a processor with 2 controllers, each with two channels or a processor with four channels. The memory channels can be accessed in parallel. It is the highest degree of parallelism in a DRAM system.

Each channel consists of one or more ranks. A rank consists of multiple banks. Each bank can be used at the same time that other banks are being used. DDR4 RAM consists of 16 banks that are organized in four bank groups, each with four banks. A bank consists of rows and a row buffer as shown in Figure 2. A row is a group of storage cells that are activated in parallel.

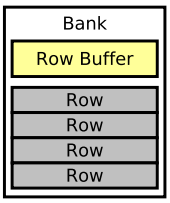

Figure 2: A DRAM bank consists of rows and a row buffer.

A row can only be accessed from the row buffer. The row buffer is essentially a cache that can hold a single row. There are three different states upon a row access. First, if the requested row is cached, it can be accessed immediately. Second, if the row buffer is empty, the requested row needs to be loaded into the buffer before the access is possible. This increases the access latency. Third, if the row buffer is occupied with a row different from the requested row, the currently cached row needs to be written back first. Then the requested row can be loaded into the buffer. This further increases the access latency. A DRAM row is also called a DRAM page.

The memory controller in the CPU is responsible for enabling and addressing those components based on the incoming physical memory address. First, the memory controller selects a channel based on the address mapping and uses the channel's own address and data lines. The ranks and banks within a channel share the same address and data lines. The correct rank must be activated by using additional rank and bank activation signals. Again, the mapping function defines which rank and bank is activated when accessing a physical address. Except for the configuration at system startup, the mapping of physical addresses to components is static. The address mapping influences how well parallelism and pipelining opportunities can be used [15, Section 13.3]. If we want to implement a custom mapping on existing hardware, we need to know the hardware mapping function and use the physical addresses that result in an access of the desired component.

In multi CPU systems, the DRAM can be organized as NUMA aware or NUMA unaware (i. e. interleaved) mode. In NUMA aware mode, the OS sees each processor with its own distinct DRAM, and the OS (or application) can explicitly access the memory of a CPU. In the NUMA interleave mode, the OS sees only one memory space and the hardware will interleave accesses to all memories with a similar address mapping as for the other DRAM components.

III. Related Work

Seaborn [16] shows manual reverse engineering of the address mapping of a Sandy Bridge processor. First, the author

uses information about the DIMM configuration from the Serial Presence Detect (SPD) ROM stored on the DIMMs themselves to build a hypothesis about the mapping. Then, the author uses a rowhammer tool that causes bit flips in the RAM. Such bit flips can be caused in neighboring rows that are in the same bank. This approach has the following disadvantages. First, the sample generation is not accurate. Bit flips are not guaranteed to occur and may also occur in rows that are not next to each other but further apart. The author describes that this occurred in the experiment and it required manual detection and removal of the outliers. Second, there is no algorithmic method for determining the addressing function. The author manually analyzes the reported addresses of successful bit flips to determine the addressing functions. While it works for this relatively old and simple PC-class processor, it is hardly possible to do a manual analysis on a more modern processor which has more complicated hashing and region based mappings.

A timing-based approach is introduced by Pessl et al. [9]. It is based on the principle that a row buffer hit results in a lower access latency than a row buffer conflict. They use pairs of addresses and repeatedly access the pairs. If both addresses in a pair are in the same bank, alternating accesses will lead to a relatively long delay due to row buffer conflicts. First, these address pairs and timing results are collected. In a second step, the linear xor functions are recovered from the data using a brute force search. They present results for several systems including Sandy Bridge, Ivy Bridge, Haswell, and Skylake, as well as Qualcomm and Samsung mobile processors. They have at most two channels and two ranks. The main disadvantage of this approach is the inaccurate attribution of physical addresses to components. It is based on measuring the timing of accesses, which can be easily disturbed. For example, the processing of other instructions in the pipeline may introduce additional delay. The memory controller is another source of inaccuracies because it can re-schedule DRAM access requests. This changes the timing and can change row buffer access behavior. Despite our best efforts, we could not reproduce the results on our machines. We suspect that such inaccuracies in the measurement lead to the inconsistent results that we have observed.

A performance counter based approach for L3 caches is presented by Maurice et al. [17]. The L3 cache is typically split into slices. The slices are addressed in a similar way as the DRAM components. Each slice has separate access counters, thus for each physical address, it can be determined which slice was accessed. Their approach uses two addresses that differ only by one bit. If the output (accessed cache slice) is the same for both addresses, the bit does not play a role in the result. If the output is different, then this bit is included in the calculation of the cache slice index.

The address mapping is configurable, and there are hardware registers that store the configuration. Reverse engineering of those configuration registers is a method introduced by Hillenbrand [18]. The approach is to change the DIMM configuration of the servers, and then to monitor the changes in the configuration register space. The result is documentation of registers that goes beyond what is officially available by Intel. This study covers Intel Haswell and Broadwell systems.

IV. Methodology

Our reverse engineering approach has two steps. In the first step, pairs of the physical address and accessed component are collected. This is done by picking an address from a pool and then finding the component that this address accesses. The component is identified using performance counters. After this process, there is a list of physical addresses and the component that the address accessed. In the second step, address mapping functions are calculated from this list of samples. The steps are summarized in Algorithm 1. The remainder of this section explains the individual steps.

Algorithm 1 Pseudocode of the reverse engineering method.

```
Allocate pool of memory
while not enough addresses do
    Pick a virtual address from the pool
    Get the physical address
    for all components do
        Set up measurement for component
        Repeatedly access the address
        if counter value > number of accesses then
            Record pair of physical address and component
        end if
    end for
end while
Calculate mapping function from samples
```

A. Memory Allocation

The DRAM address mapping is based on the physical address. If we can control individual bits of the physical address, we can use a structured address selection method, such as the one described in Section IV-B. Only the bits that express offset within a page frame directly translate to bits of the physical address. The bits for the frame number are not under our control. Thus we increase the page size to 2MB, which leads to 21 bits of the physical address being under the direct control of our tool.

B. Address Selection

Theoretically, it is possible to use random addresses from a large pool to gather samples. However it would require the collection of many samples to get enough coverage to reconstruct the addressing function. We want to find out for every bit of the physical address if it influences the accessed DRAM component. Thus addresses that differ in only one bit would allow us to directly judge the influence of this one bit on the result. To generate such addresses we use the following mechanism. First, we take a random address from the allocated memory pool. The next address is generated by changing the least significant bit. The following addresses are generated by reversing the last bit change and then changing the next more

significant bit. In other words, a shifted bitmask with a single one is xored with the initial address. Once we run into the next page frame or out of the boundaries of the address pool, we choose a new random address and start again with modifying the least significant bit.

The memory controller takes physical addresses as the input of the mapping function. Thus we need to gather physical addresses samples. Through the /proc/self/pagemap interface, we can translate the virtual addresses to physical addresses.

C. Performance Counters

For each physical address, we need to know which component is accessed. We use performance counters for each component to measure if they are accessed. Each channel has its own Performance Monitoring Unit (PMU) with its own counters. By selecting the right PMU we count the number of transfers on this channel. For each rank, there is a separate performance event with a separate umask for each bank or bank group. The event definitions can be found in the Intel uncore performance monitoring guide [19]. Each measurement checks one specific channel, rank, and bank. We cycle through all possible components until we found one where the counter value is equal or higher than the number of programmed DRAM accesses. The performance counters that we use require root access or the perf event paranoid flag to be set accordingly. The use of performance counters is a significant difference from a previous timing based approach [9]. The measurement is more reliable and practically eliminates attribution errors. A disadvantage of this method is that only CPUs which have the appropriate performance counters are supported.

D. Enforcing DRAM Accesses

For the measurement, it is required to repeatedly access a certain address, and every access must cause a load from DRAM that we can count. Registers and caches exist to avoid such redundant loads from DRAM. Thus we use the code in Figure 3 with cache line flush and fence instructions that enforce a load from DRAM with every access to the same address.

```
volatile uint64_t *p = (volatile uint64_t *) addr;
for (unsigned int i = 0; i < NUM_ACCESS; i++)
{
  _mm_clflush((void*)p);
  _mm_lfence();
  *p;
  _mm_lfence();
}
```

Figure 3: The C code that enforces memory loads from DRAM when repeatedly accessing the same address.

The performance counters we use count for the whole system, not just for our application. Thus co-running applications or transfers by the OS cause noise. We need to set the NUM_ACCESSES variable high enough so that those other accesses do not disturb our measurements. If the number of accesses is too high, it causes unnecessary delays. In our experience, on a system that is not running other applications than the default OS services, 2000 or more accesses allow accurate measurements. On a system that executes other tasks in the background, a higher value may be required.

E. Computing Addressing Functions

The list of samples that contain physical address and accessed component is not useful on its own. We need to extract an addressing function from those samples. We introduce a novel method to resolve the bits of the physical address that are used for addressing components. It reports exact results for used, unused, and unconfirmed bits, as well as other types of errors.

1) Constructing an Equation System: All previous work [9], [16], [17], [18] indicates that the mapping either uses a single bit or a xor combination of multiple bits of the physical address to calculate the component index. Because of the limitation to xor functions, this problem can be formulated as a boolean equation system consisting of two operations (*xor*, *and*). The idea for the construction of the equation system is as follows. The input is a list of samples as shown in Figure 4.

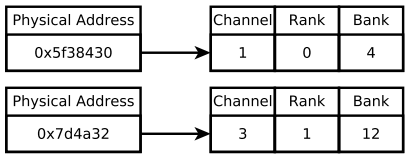

Figure 4: A list of physical addresses and the accessed component.

We split all the collected samples into a one-bit component address. E. g. If there are four memory channels, two bits are needed to address those four channels. We duplicate the list of physical addresses and build two lists of samples, one for the first bit of the channel address and one for the second bit of the channel address. Each of the bits of the physical address could be used for the calculation of the component address bit. Thus we add a switch (the boolean *and* operation) to every bit. This switch can turn the usage of this bit on or off. If there are multiple bits used, we know that they are combined using the xor operation. Thus we add a xor between every bit. The resulting structure is visualized in Figure 5.

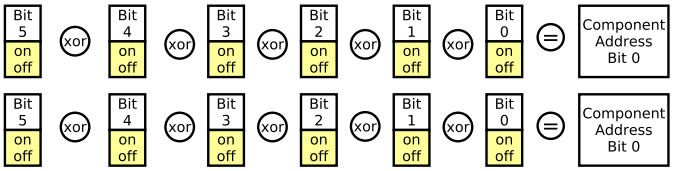

Figure 5: The list of samples converted into equations with bit switches.

The formalization of this concept as an equation system is shown in Equation 1.

$$\begin{aligned} x_{0,0} * b_0 \oplus x_{0,1} * b_1 \oplus \cdots \oplus x_{0,n} * b_n &= c_0 \\ x_{1,0} * b_0 \oplus x_{1,1} * b_1 \oplus \cdots \oplus x_{1,n} * b_n &= c_1 \\ \vdots \\ x_{m,0} * b_0 \oplus x_{m,1} * b_1 \oplus \cdots \oplus x_{m,n} * b_n &= c_m \end{aligned} \tag{1}$$

In Equation 1 the $*$ symbol denotes the *and* operation and $\oplus$ symbol is the *xor* operation. There are n unknown parameters b that express if a bit is used or not. And there are m physical address samples taken with their individual address bits x. The c on the right-hand side represents the one-bit component address.

2) Solving the Equation System: The equation system can be solved in the F_2 space, where *xor* is an addition and *and* is a multiplication, with any equation system solver. We use Gaussian elimination. The usage of established linear equation system mathematics brings the advantage of a clear differentiation of the results.

The equations system either has a solution, is partially solvable, or it has no valid solution. If it is not solvable, there are contradicting equations. This can happen in case of wrong measurements or if a more complex address mapping, such as one with multiple regions, is not correctly considered. Theoretically, it could happen that wrong measurements lead to an equation system that is solvable and produces wrong results. However, this would require a systematic error in the measurement. For example, if a performance counter always reports accesses to a different component than the one specified in the measurement setup. Such an error is unlikely to occur, and we never observed such a case in our experiments.

If there is a solution or a partial solution, for every bit of the physical address there are three possible states. A bit can be used for calculating the index, a bit can be unused, or it is unknown if a bit is used. The unknown state happens if there is a partial solution with dependent equations, and there are no samples that cover this specific bit.

This accurate reporting is an advantage over the brute force solver by Pessl et al. [9] because it can identify wrong measurements and differentiate it from unknown bits caused by too few samples.

F. Region Based Mapping

In the case of asymmetric DIMM population or if the number of DIMMs in a channel is not a power of two, the hardware uses more complex region based address mapping. For example, a two-bit wide channel address, calculated using the xor combination of bits, targets four different channels. A space of three different channels can not be expressed. Thus a region based mapping is used in hardware. The regions are address ranges. For each region, a different addressing function can be used. The regions help to get a balanced distribution of requests over all of the three channels. The regions are defined in registers in the memory controller and set up during system startup. Those registers are mostly undocumented, but Hillenbrand [18] provides the locations and decoding for Intel Haswell and Broadwell systems.

The address sample collection works the same, no matter if regions are used or not. For the calculation of the addressing functions, additional steps are necessary. First, we read the region limit addresses from the registers. Then we group all captured addresses into their respective region. Finally, for each of the regions, the addressing functions can be computed as described in Section IV-E.

As already reported previously [18], on some systems, Linux is not able to access the PCI extended configuration space. The channel region definitions are within an address range that is always accessible. But for the ranks, the registers may not be accessible. We experienced this issue on an Intel Haswell system that is equipped with 6 ranks per channel. A workaround is to use a modified Linux kernel [18]. On our two Broadwell systems, complete configuration space was accessible.

Hillenbrand [18] only describes the registers of Broadwell and Haswell based systems. For Skylake and its successors, the register layout changed and is poorly documented. We cannot read the registers to find the regions for channels or banks. On those newer generation systems, our approach only works for a balanced power of two DIMM population.

V. Results

We reverse engineered the address mapping on four different systems and confirm that the obtained addressing functions are correct.

A. Hardware

Table I lists the basic facts of the server systems. In the following, the servers will be referenced by their name, which is in the leftmost column of Table I.

All of the systems are NUMA aware but do not use on-chip NUMA. This means that the OS sees the different processors with their own explicitly addressable memory but cannot see the different memory controllers within one processor. All of the systems are equipped with DDR4 RAM. DDR4 RAM always has four bank groups, each with four banks. Every system uses only a single type of DIMM. For Arcturus, Rigel, and Spica, the configuration is the same on all sockets. On Comet, the memory setup differs for socket 0 and socket 1. On socket 0, only three out of four channels are active due to a hardware defect. On socket 1, all of the four channels are active. Rigel is a Skylake system, which supports up to six memory channels. Our system is equipped with DIMMs on four of the six channels. The rightmost column ranks in Table I is the number of ranks per channel.

All of the experiments were executed on machines running Ubuntu 18.04, and the benchmarks were compiled with gcc 7.4. We configured our tool to use a 20GB address space that is allocated using 2MB pages. We execute 2000 accesses per test and capture a total of 400 address samples.

Table I: The hardware we used for testing the reverse engineering method.

Name	Architecture	CPUs	Board	DRAM Speed	DIMMs	Number of Channels	Number of Ranks
Arcturus	Broadwell	2x E5-2699v4	Supermicro X10DGQ	2400Mhz	Micron 36ASF4G72LZ-2G3B1	4	4
Comet	Haswell	2x E5-2699v3	Dell 0CNCJW	1867Mhz	SK Hynix HMA84GL7MMR4N-TF	3 and 4	6
Rigel	Skylake	2x Xeon 8176	Supermicro X11DPG-QT	2667Mhz	Samsung M386A8K40BM2	4	4
Spica	Broadwell	4x E7-8890v4	Supermicro X10QBL-4	1600Mhz	Samsung M386A8K40BM1	4	8

B. Addressing Functions

We use a 0 based numbering for address bits. I. e. the bit number 0 is the least significant bit of an address followed by bit 1 and so on. The bank addressing can either be interpreted as 16 different banks, which need 4 bits for addressing. Or it can be seen as four bank groups, each with four banks. Thus the bank group addressing bits are the same as two of the bank addressing bits. There are separate performance counters for the 16 banks and the four bank groups. The reverse engineering is done separately and we report the results for all 16 banks and the four bank groups individually.

Table II shows the addressing function of Arcturus. Channels and banks use xor hashing. The pattern of the mapping is similar to what is reported by Pessl et al. [9]. However, the individual bits used for addressing are different. It highlights the need to study the mapping for every system individually.

Because Comet is equipped with 6 ranks per channel, a region based mapping is used for the ranks. The rank configuration registers are not accessible using a standard Linux kernel. Thus we cannot resolve the rank addressing and subsequently cannot resolve the bank addressing. On socket 0, there is a region based mapping due to the use of 3 channels. It is shown in Table IV. In the first address region, we did not record any memory accesses. We suspect that it is a reserved hardware area that is not mapped to the DRAM. The second region uses interleaving between the two controllers. Within the first memory controller, the channels are interleaved. The second controller needs no further interleaving because only one channel is available. The third region only uses the first controller and interleaves it's two channels. The used bits are different from the second region. To the best of our knowledge, this is the first time a region based mapping was successfully reverse engineered.

Spica and Rigel do not use xor hashing. Instead, only single bits are used as shown in Table V and Table VI. With such a configuration, a performance decreasing imbalanced use of channels, ranks, or banks can easily occur if strided memory accesses happen to all fall into the same channel, rank, or bank.

If we equip Rigel with enough DIMMs for all six channels, a region based mapping will be used. Because we do not know the configuration registers for this architecture, reverse engineering is not possible.

C. Speed of Reverse Engineering

In addition to the reliability of the measurements, timing based approaches also have the disadvantage of long processing time. We compared the time required for reverse

Table II: DRAM address mapping of Arcturus

Component	Index Bit	Physical Address Bits
Channel	0	$8 \oplus 12 \oplus 14 \oplus 16 \oplus 18 \oplus 20 \oplus 22 \oplus 24 \oplus 26$
	1	$7 \oplus 17$
Rank	0	15
	1	16
Bank	0	$6 \oplus 24$
	1	$21 \oplus 25$
	2	$22 \oplus 26$
	3	$23 \oplus 27$
Bank Group	0	$6 \oplus 24$
	1	$21 \oplus 25$

Table III: DRAM address mapping of Comet socket 1

Component	Index Bit	Physical Address Bits
Channel	0	$8 \oplus 12 \oplus 14 \oplus 16 \oplus 18 \oplus 20 \oplus 22 \oplus 24 \oplus 26$
	1	$7 \oplus 17$

Table IV: DRAM channel address mapping of Comet socket 0

Address Region	Component	Physical Address Bits
0 to 1984M	-	-
1094M to 198592M	Controllers	$7 \oplus 17$
	Channels in Controller 0	$8 \oplus 12 \oplus 14 \oplus 16 \oplus 18 \oplus 20 \oplus 22 \oplus 24 \oplus 26$
198592M to 296896M	Channels in Controller 0	$7 \oplus 12 \oplus 14 \oplus 16 \oplus 18 \oplus 20$

Table V: DRAM address mapping of Rigel

Component	Index Bit	Physical Address Bits
Channel	0	8
	1	9
Rank	0	15
	1	16
Bank	0	6
	1	21
	2	22
	3	23
Bank Group	0	6
	1	21

Table VI: DRAM address mapping of Spica

Component	Index Bit	Physical Address Bits
Channel	0	6
	1	7
Rank	0	8
	1	9
	2	10
Bank	0	11
	1	12
	2	13
	3	14
Bank Group	0	11
	1	12

engineering of our approach and the timing based approach of Pessl et al. [9], even though it did not report the correct result. We did the experiment on Spica. It has a large DRAM size of 512GB per socket, and it has 512 addressable sets (16 banks × 8 ranks × 4 channels). Our tool finds the correct addressing for all sets in ten out of ten tests in an average time of 1:04 minutes. In contrast, the timing based approach needs over 51 hours for the complete reverse engineering process. We assume the optimal case when the mapping is found in the first try, which often does not work due to inaccuracies in the timing measurement.

VI. Usage of Addressing Functions

To confirm that the recovered address mappings are correct, we implement micro benchmarks that reproduce known performance effects of bank and channel usage.

Based on addressing functions shown in Section V-B, we implement a benchmark that can access specific memory components. The benchmark first allocates a large array. Then it calculates the array indexes that are on the specified channels, ranks, and banks. Finally, it accesses the calculated array indexes in a parallel for loop. The number of data accesses stays constant, regardless of the configured channels, ranks, banks, and threads.

A. Channels

Figure 6 shows the measured bandwidth on the four different channels of Spica. In this experiment, the benchmark accesses only one memory channel. This can be clearly seen in the diagram. We measure a bandwidth of about 10GB/s on one of the channels but almost no activity on the other channels. Figure 7 shows the bandwidth on Comet with two out of four channels in use. Figure 8 shows the speedup over the sequential version when only one, two, three, or all four memory channels are used. As expected, the speedup is limited by the number of available channels. This experiment was executed on Spica. These experiments demonstrate that our determined addressing functions for the channels are correct.

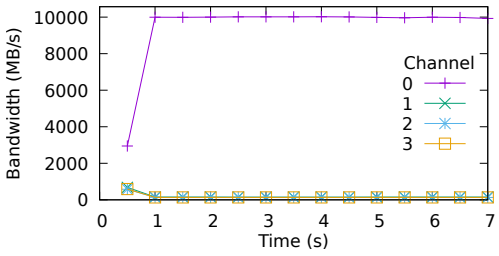

Figure 6: The time resolved bandwidth measured on Spica when memory accesses are limited to one specific channel.

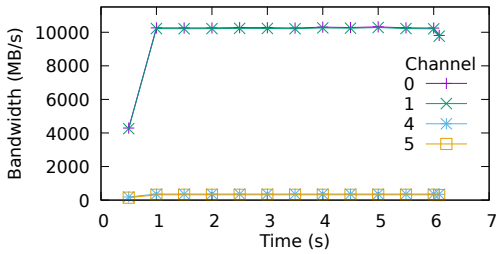

Figure 7: The time resolved bandwidth measured on Comet when memory accesses are limited to two channels.

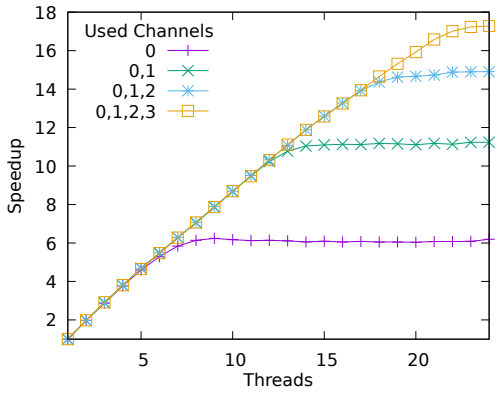

Figure 8: Parallel speedup on Spica when the usage of memory channels is restricted.

B. Banks

It is known that co-running applications or multi-threaded programs, where concurrent threads access different address regions, interfere with each other, and cause increased row buffer conflicts [8], [20], [21]. We implemented a micro-benchmark that reproduces this phenomenon and an optimized version that uses a fixed thread-to-bank assignment. The benchmark reads an array using 16 threads. We limit the access on one channel and one rank so that there are 16 banks available to use. The original version simply accesses the array indexes in ascending order using a loop that is parallelized with OpenMP. Thus each thread accesses different array locations. In the optimized version, we use the same parallel for loop, but each thread accesses only a specific bank. E. g. thread 1 only accesses data stored on bank 1, while at the same time thread 2 only accesses bank 2. We measure the row buffer access status (hit, empty, conflict) as described in the Intel documentation [22].

Figure 9 shows the page hit, page empty, and page conflict ratios measured over time of the original version. We see that, in the original version, after the initialization phase, the page conflicts increase and the page hits decrease. Over 40% of the row accesses result in a page conflict. In contrast, in the optimized version, we can eliminate most of the conflicts and get a high page hit ratio by mapping each thread to a designated bank, as shown in Figure 10.

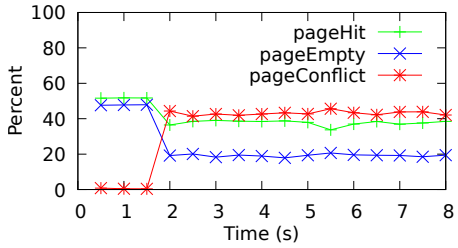

Figure 9: Row buffer hit, empty, and conflict ratio measured over time on Arcturus when using a standard multi-threaded array access.

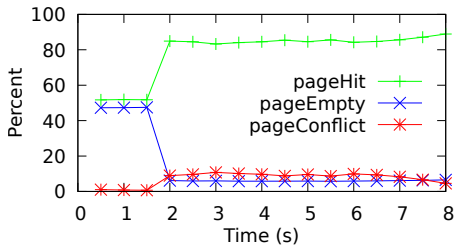

Figure 10: Row buffer hit, empty, and conflict ratio measured over time on Arcturus when using a fixed thread-to-bank assignment.

VII. Conclusion

We present a new automatic method to reverse engineer Intel's undocumented DRAM addressing. With the use of performance counters, we achieve reliable DRAM component attribution. Our evaluation shows that we can resolve the correct address mapping on recent Intel machines with a diverse set of memory configurations. In the future, we want to apply this method to other types of processors.

References

[1] Graham Allan. Ddr4 bank groups in embedded applications. https://www.synopsys.com/designware-ip/technical-bulletin/ddr4-bank-groups.html.

[2] S. P. S. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 374–385, 2011.

[3] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: Increasing DRAM efficiency with locality-aware data placement," in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2010, pp. 219–230.

[4] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row Buffer Locality-Aware Data Placement in Hybrid Memories," *SAFARI Technical Report*, vol. 005, 2011. [Online]. Available: http://www.ece.cmu.edu/{~}safari/tr/tr-2011-005.pdf

[5] S. Akiyama, "A lightweight method to evaluate effect of approximate memory with hardware performance monitors," *IEICE Transactions on Information and Systems*, vol. E102D, no. 12, pp. 2354–2365, 2019.

[6] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *Real-Time Technology and Applications*. IEEE, 2014, pp. 155–166.

[7] X. Pan, Y. J. Gownivaripalli, and F. Mueller, "TintMalloc: Reducing Memory Access Divergence via Controller-Aware Coloring," *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, pp. 363–372, 2016.

[8] V. Chandru and F. Mueller, "Reducing NoC and memory contention for manycores," *Lecture Notes in Computer Science*, vol. 9637, pp. 293–305, 2016.

[9] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," *USENIX Security Symposium*, 2016.

[10] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," *Proceedings - International Symposium on Computer Architecture*, pp. 361–372, 2014.

[11] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in JavaScript," in *Lecture Notes in Computer Science*, vol. 9721, 2016, pp. 300–321.

[12] W. Song, H. Choi, J. Kim, E. Kim, Y. Kim, and J. Kim, "Pikit: A new kernel-independent processor-interconnect rootkit," *Proceedings of the 25th USENIX Security Symposium*, pp. 37–51, 2016.

[13] Intel Corporation, "Intel Xeon Processor 5500 Series Datasheet, Volume 2," 2011.

[14] Fujitsu, "FUJITSU Server PRIMERGY & PRIMEQUEST Memory Performance of Xeon scalable processor (Skylake-SP) based Systems," 2018.

[15] B. Jacob, S. W. Ng, D. T. Wang, and M. Schuette, *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[16] M. Seaborn. How physical addresses map to rows and banks in dram. http://lackingrhoticity.blogspot.com/2015/05/how-physical-addresses-map-to-rows-and-banks.html.

[17] C. Maurice, N. le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters," in *Lecture Notes in Computer Science*, vol. 9404, 2015, pp. 48–65.

[18] M. Hillenbrand, "Physical Address Decoding in Intel Xeon v3/v4 CPUs: A Supplemental Datasheet," 2017.

[19] Intel Corporation, "Intel® Xeon® Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual," Tech. Rep. July, 2017.

[20] W. Mi, X. Feng, J. Xue, and Y. Jia, "Software-hardware cooperative DRAM bank partitioning for chip multiprocessors," *Lecture Notes in Computer Science*, vol. 6289 LNCS, no. 2005, pp. 329–343, 2010.

[21] H. Huang, L. Liu, F. L. Song, and X. Y. Ma, "Architecture supported synchronization-based cache coherence protocol for many-core processors," *Jisuanji Xuebao/Chinese Journal of Computers*, vol. 32, no. 8, pp. 1618–1630, 2009.

[22] Intel Corporation, "Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide," no. March, 2012.