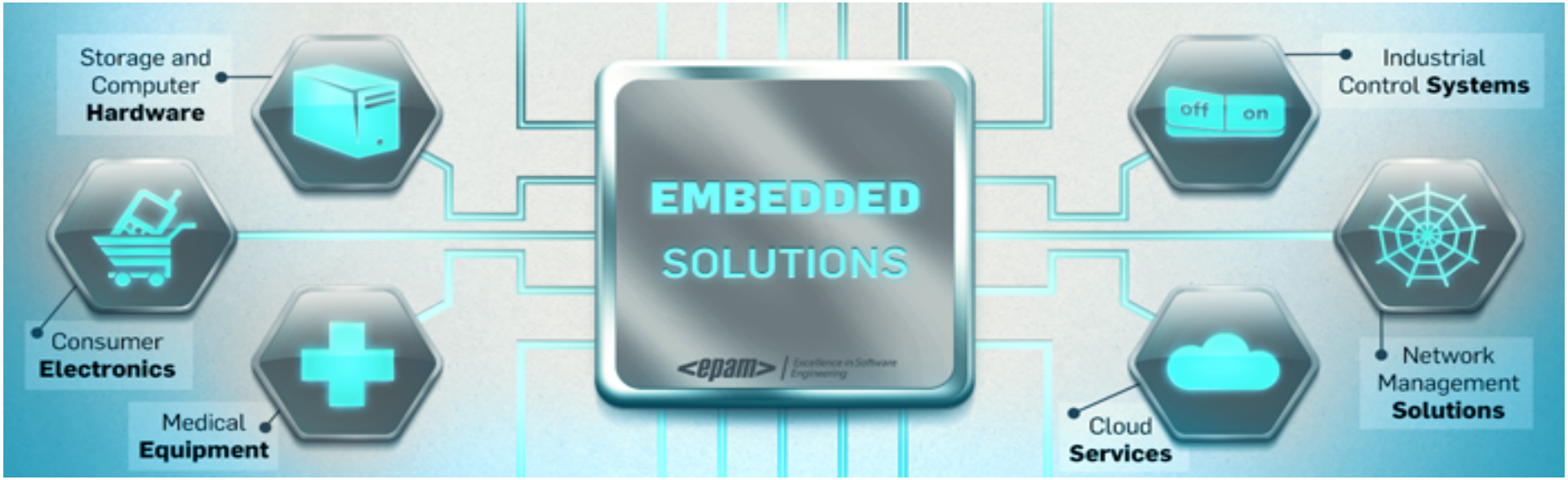
<epam> Excellence in Software Engineering

http://www.epam.com/

#>LLPD

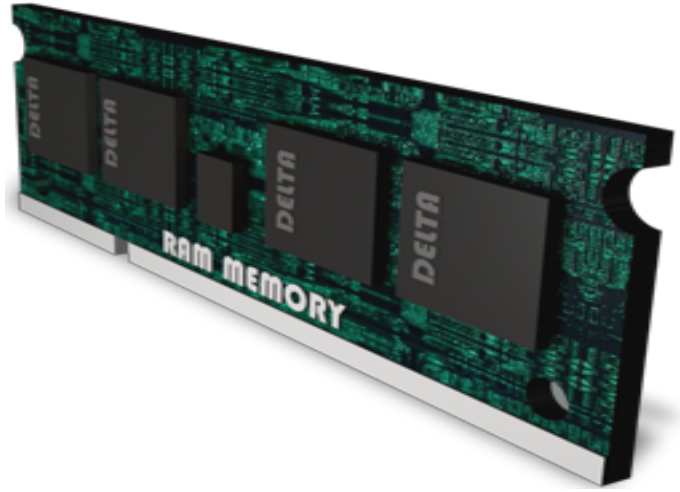


http://www.epam.com/solutions/embedded.html

Ivan Matylitski

https://github.com/buffovich

RAM usage from userspace perspective



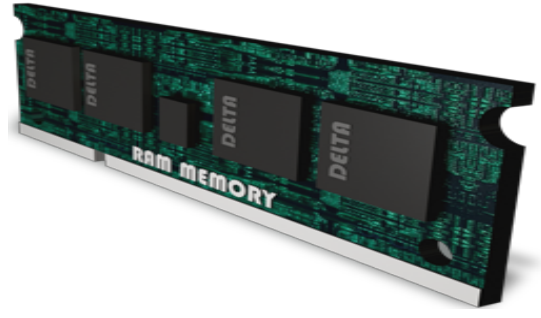
RAM usage from userspace perspective

Modern physical memory architecture

- Basic ideas about how to use it effectively

Memory allocation in code

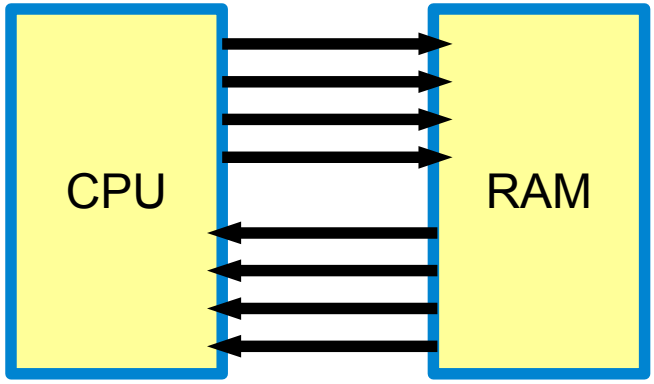
- Classic memory allocation models
  - Static allocation (data segments)
  - Allocation in stack (userspace stack characteristics)
  - Dynamic allocation (jemalloc, tcmalloc, libc6 malloc)
- Object caching (SLAB)
  - Original idea of SLAB
  - Existing slab implementations

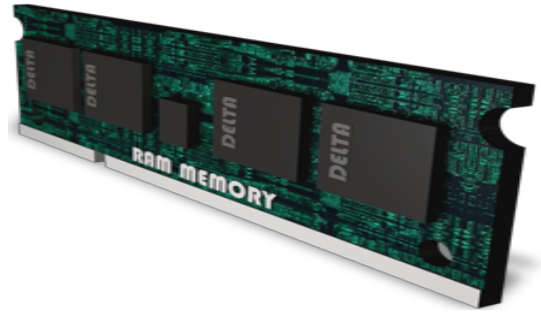


RAM-CPU relationships: historical retrospective

Long time ago...

Processors and memory chips works with the same frequency (actually, with slightly different speeds)



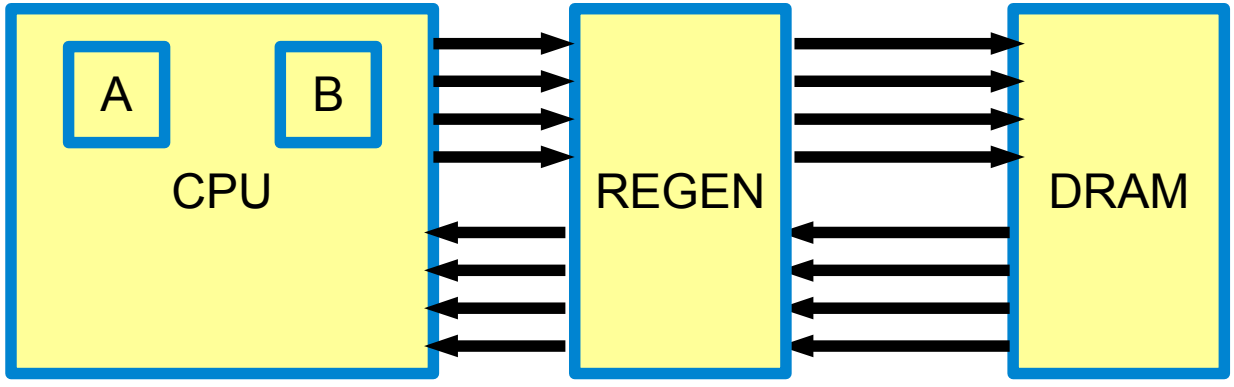


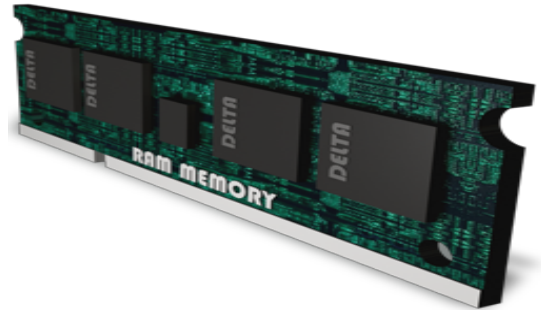
RAM-CPU relationships: historical retrospective

Long time ago...

Were two types of memory:

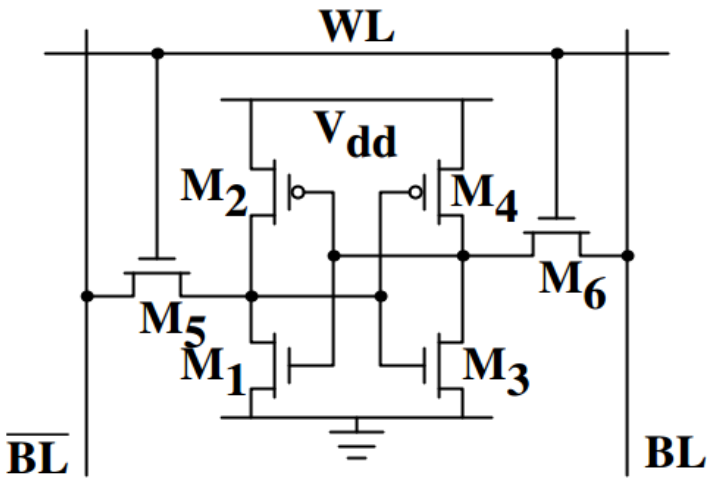
- **Registers based on SRAM**
- **External memory based on DRAM**
- **There was ROM but... You know.**





From now... what about RAM technologies?

SRAM bit cell is based on cross-linked logical gates with positive feedback.

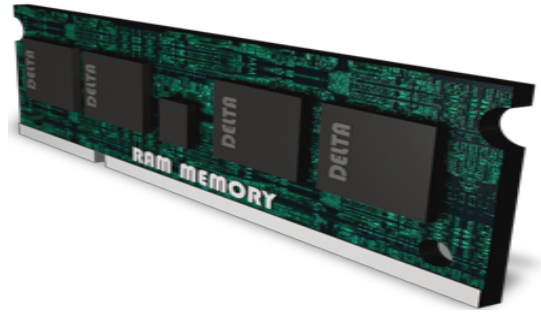


YYES?

- Fast
- Shares the same manufacturing technology with CPU

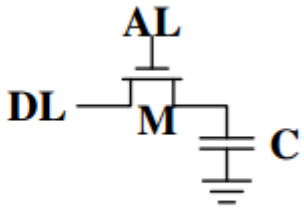
YNOT?

- High power consumption
- Large size



From now... what about RAM technologies?

DRAM bit cell is based on capacitor paired with logic valve.

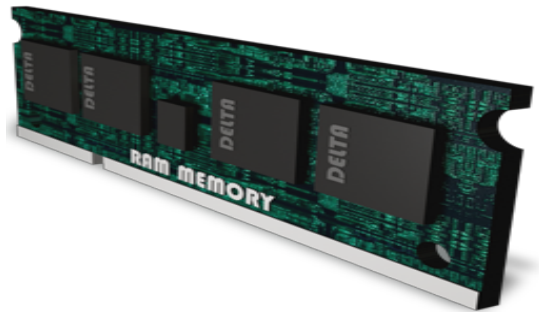


YYES?

- Lower power consumption
- Tiny cell size

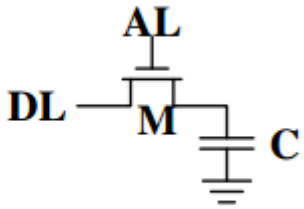
YNOT?

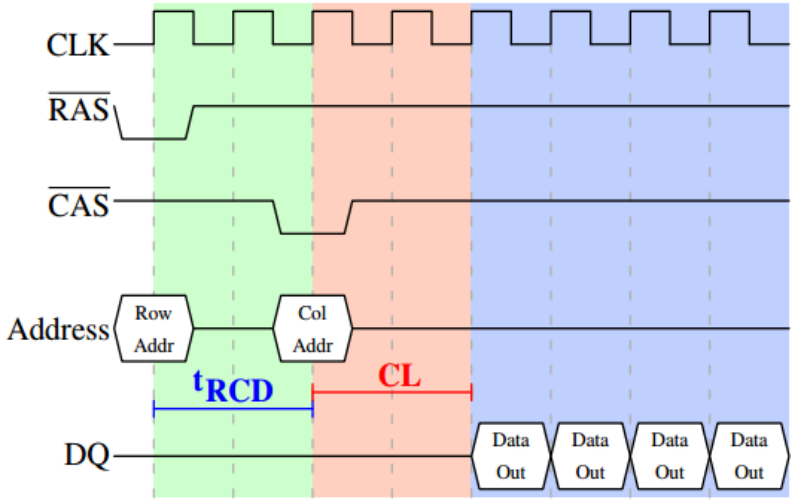
- Needs recharging cycles
- Needs extra machinery serves the needs of regeneration
- Has higher latency and lower speed because of these cycles



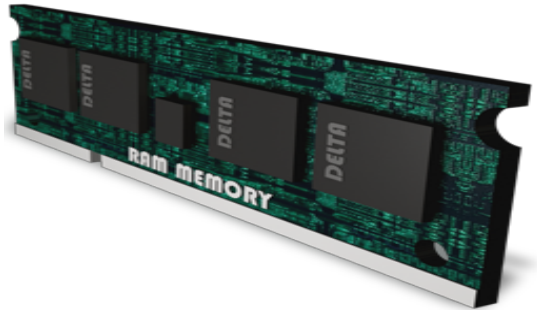
From now... what about RAM technologies?

DRAM is more complicated beast than you might think before...





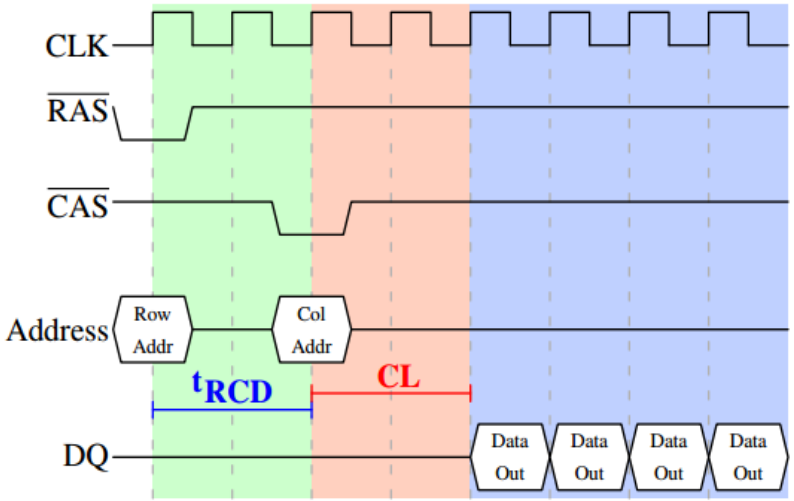
That is how it works through the time scale

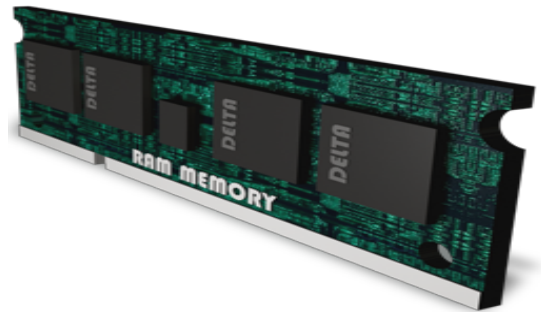


From now... what about RAM technologies?

- OMG! It means that RAM has the same property as spindel drive?

- Well... Each time you want to select new address you need to set up RAM controller to the right position. Random memory isn't so random...





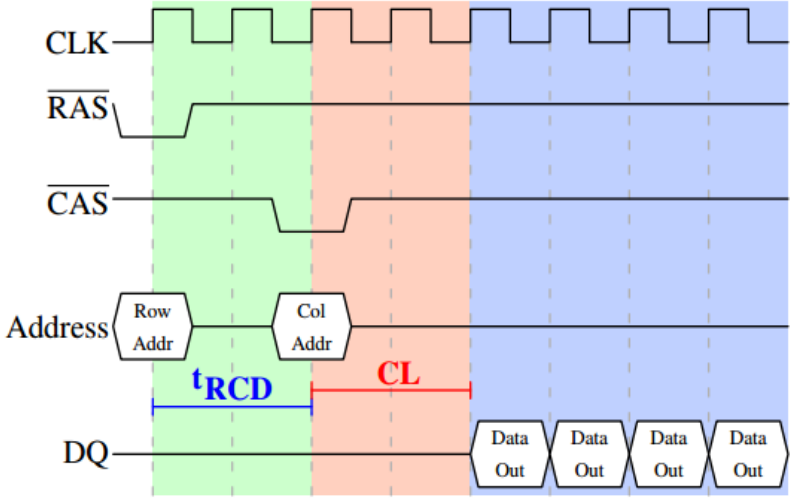
New beast: spatial locality.

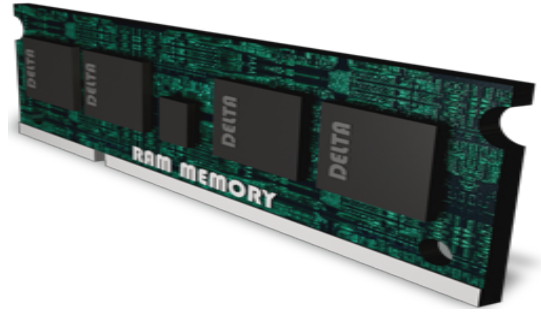
- You know... The fastest way to read a data from the external RAM is to read it sequentially.

- I guess, it means that I should place data used together as close as it possible to each other?

- Definitely.

Sincerely yours, Captain

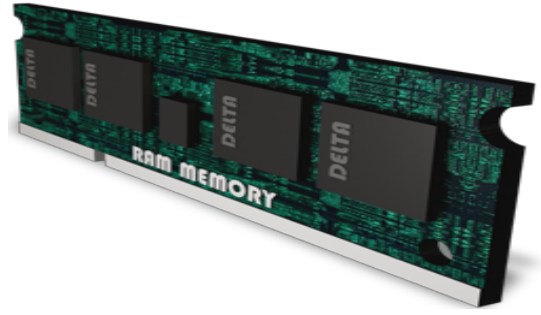




New beast: spatial locality.

BTW, memory defragmentation isn't so weird idea. How would it be nice to have self-defragmenting rope container, for example. Constant-time concatenation and spatial-locality-awareness in the same time...

rope<T, Alloc>

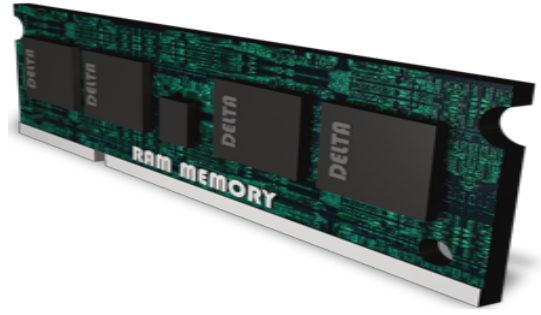


What about contemporary technologies?

So, guys, bad news for you... External RAM isn't rocks anymore...

“From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory speed only improved at 10%.” Wikipedia says. Hundreds of CPU cycles can be wasted in the name of memory access. Moreover, recall spatial and latency issues and you will come up with idea...

?

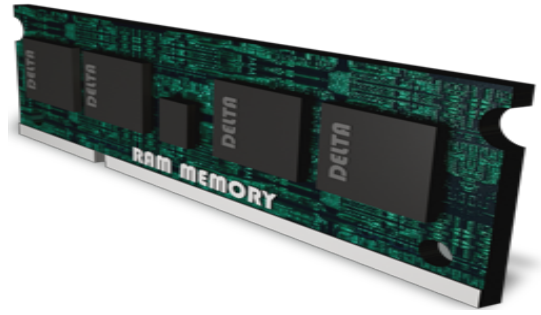


What about contemporary technologies?

We need more registers! They are based on faster SRAM!

- Well modern processors got ones with really ubiquitous names. But not so much as you might think. And you should say now...
-Why?

?

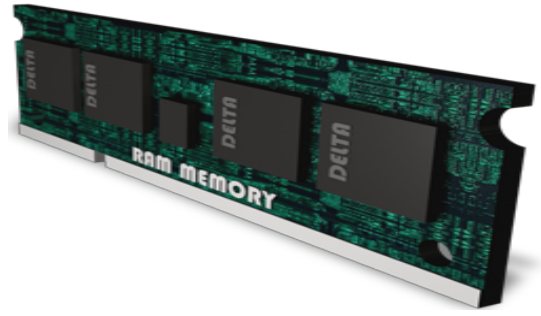


What about contemporary technologies?

Awareness of code about the huge register file will complicate the code and its pipeline processing inside the CPU chip. Imagine 4KB of registers where each one has its own name/index.

The actual answer for the previous question is transparent CPU...

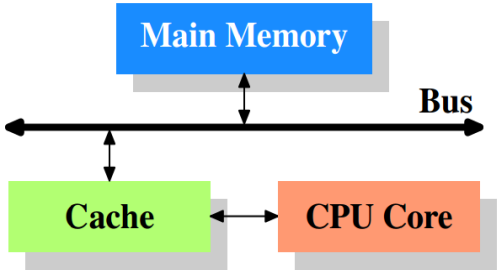
Cache



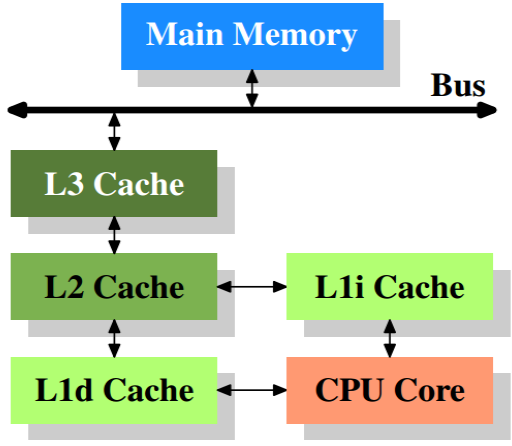
So... what about cache?

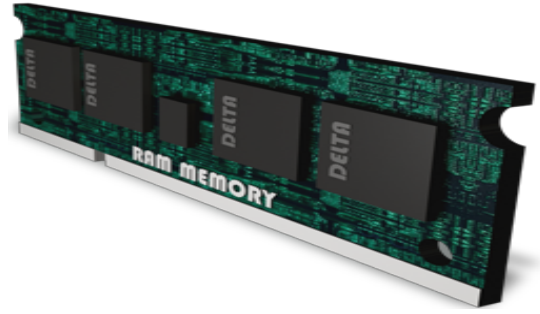
It can be considered as transparent register file based on fast SRAM cells. Actually, there are several levels of cache with different sizes (the more we move away from the CPU core the more latency and register file size become).

It looks like this:



Or even like this:



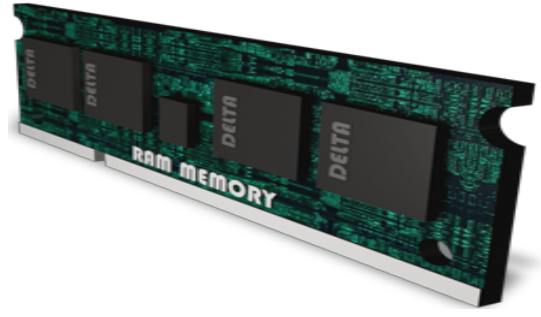


Cache is transparent?

Program code doesn't control cache directly. Cache has its own algorithm coded in logical valves.

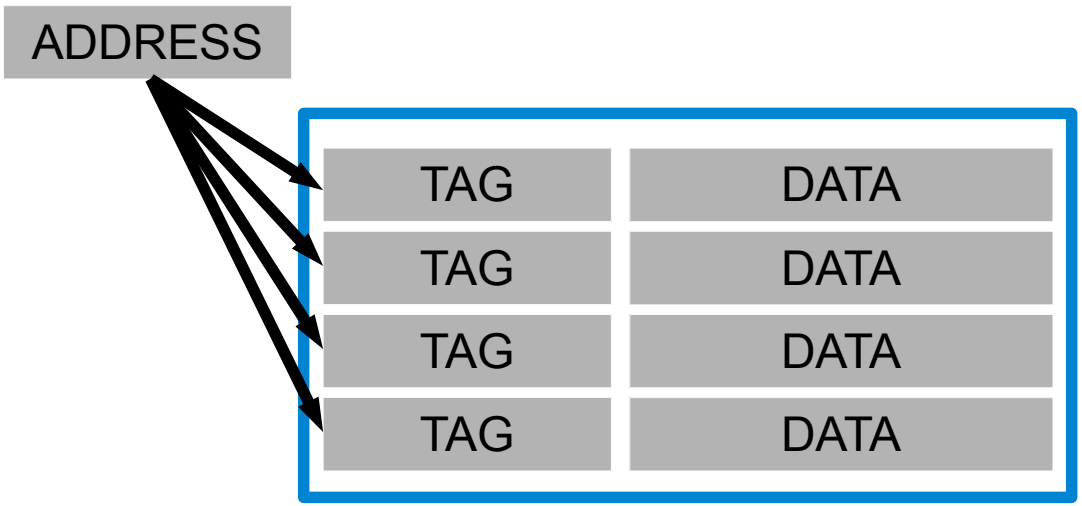
Here are the types:

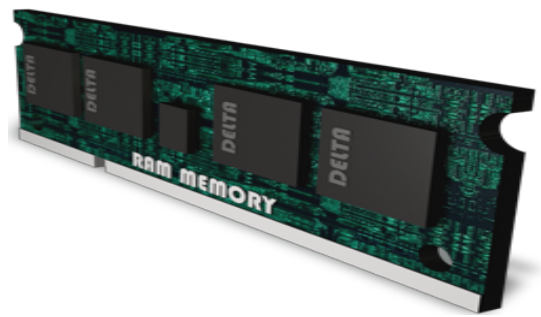
- **Fully-associative (extreme case)**
- **Direct mapped (extreme case)**
- **N-way associative (the one which if frequently used in practice)**



Fully-associative?

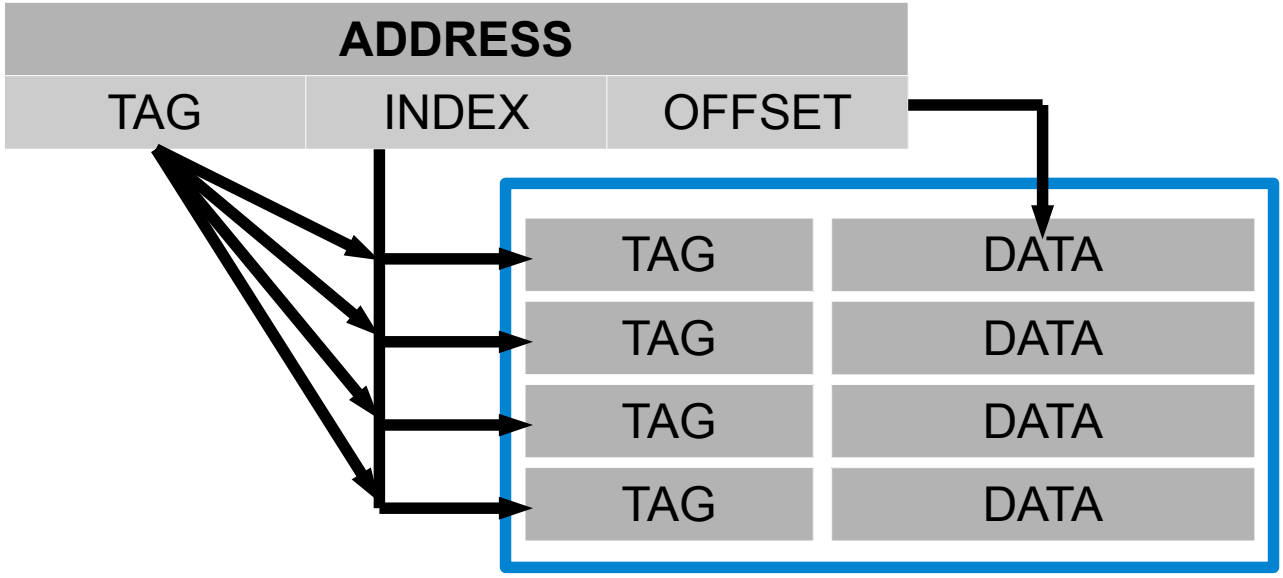
Each entry from underlying external memory may be written to each cache entry. Example: (I/D)TLB.

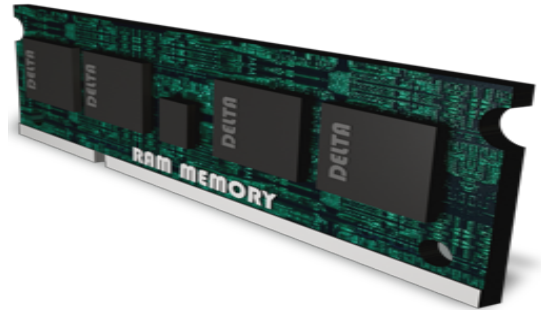




Direct mapped?

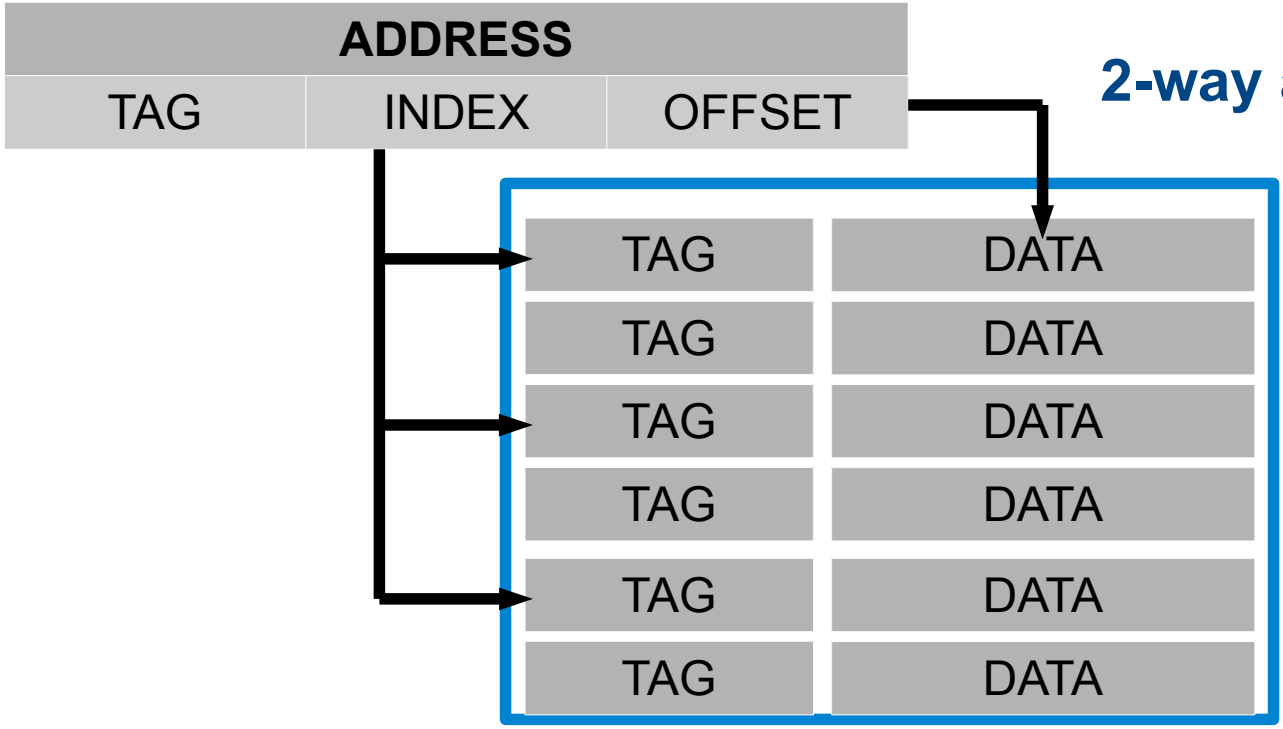
Each entry from underlying external memory may be written to the only one corresponding cache entry. Address is divided into three parts now.



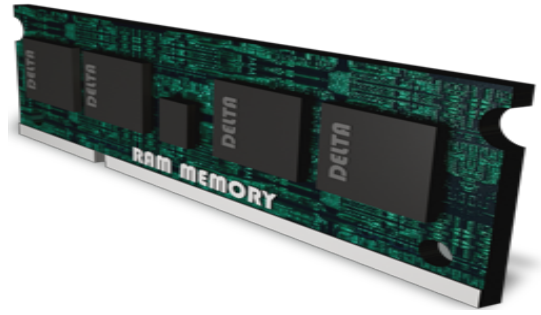


N-associative?

Each entry from underlying external memory may be written to one of entries which is in the group of N entries. Combination of fully-associative and direct mapped cache. Address is divided into three parts as well. Examples: modern L1 and L2 data/instruction caches.

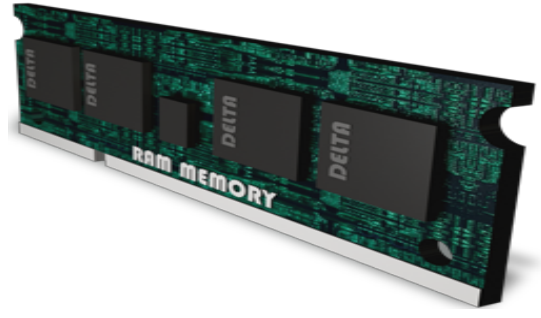


2-way associative



Basically, what does cache provide?

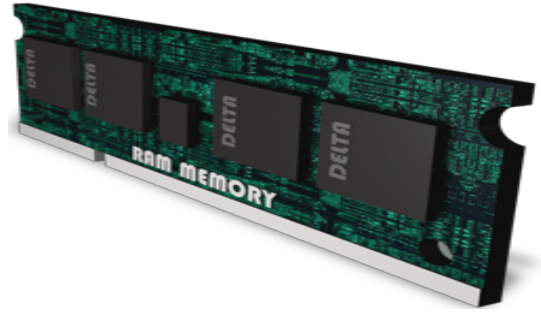
It helps code if code maintains temporal locality. Its new beast which is about re-using data which was used earlier. Example: limit as exit condition in loop. Moreover, it helps with spatial locality because cache entry has particular size and entry can be filled in background in read-ahead manner. In the meantime, code in pipeline can proceed with its execution.



What is boiled down outcome about DRAM and cache?

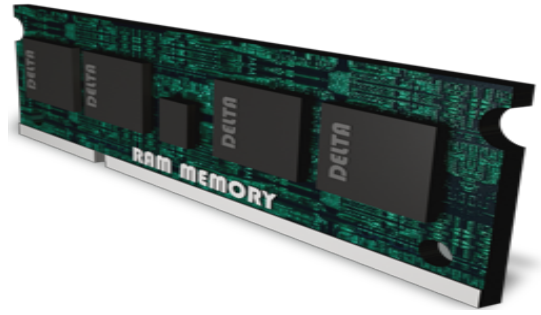
Read/write data sequentially. Re-use data which has been just used. Help cache with read-ahead. Keep data pieces which are used together as close as possible to each other. Use in-lining for functions which performance is crucial.

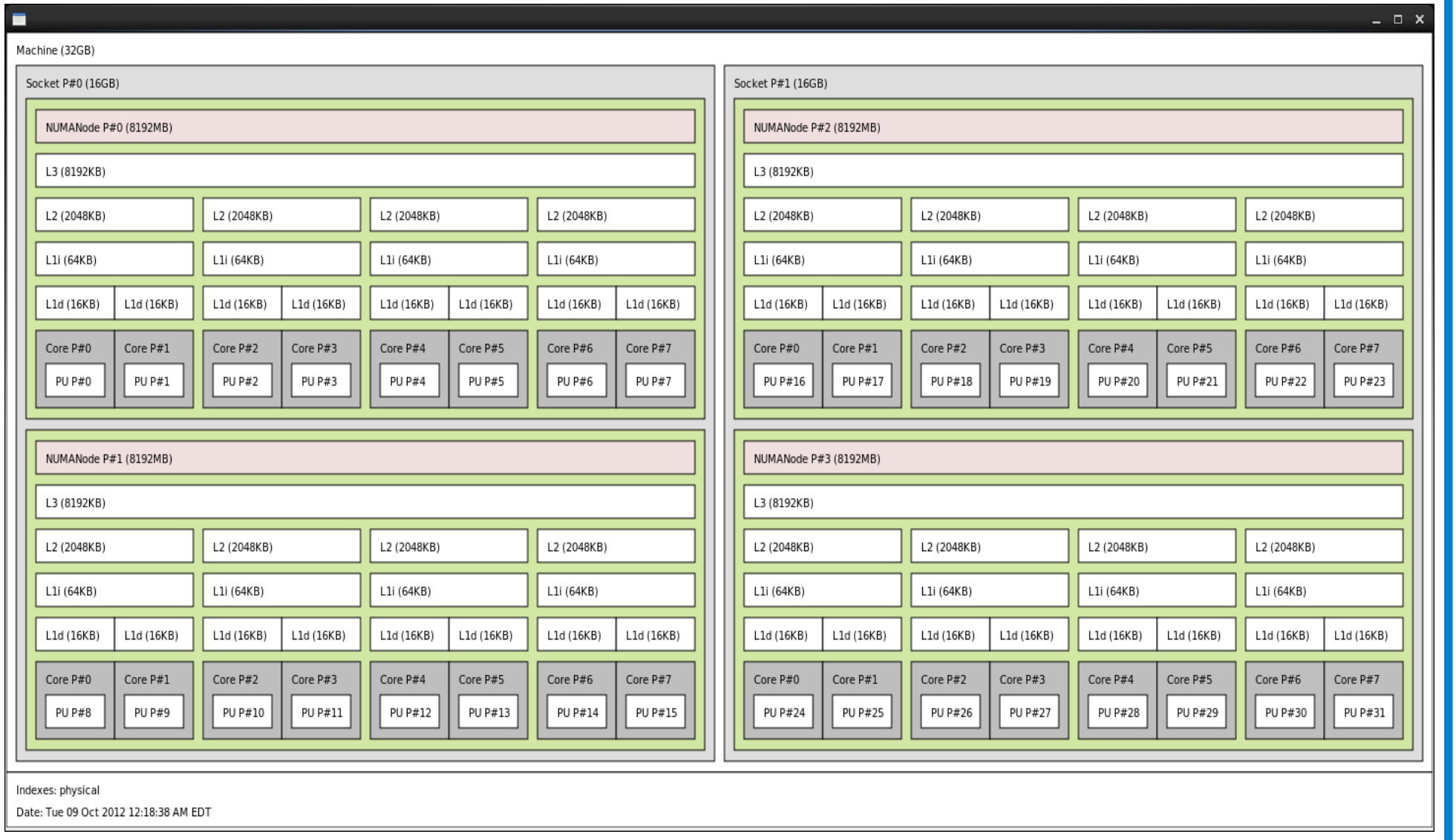
Homework: think about matrix multiplication optimization.



May you tell us about multi-core and multi-processor configurations?

Yep.



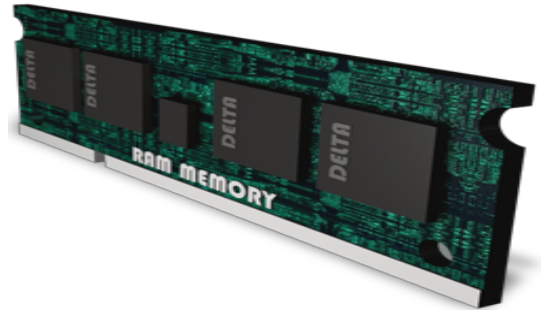

Machine (32GB)
Socket P#0 (16GB)
NUMANode P#0 (8192MB)
L3 (8192KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB)
Core P#0 PU P#0
Core P#1 PU P#1
Core P#2 PU P#2
Core P#3 PU P#3
Core P#4 PU P#4
Core P#5 PU P#5
Core P#6 PU P#6
Core P#7 PU P#7
NUMANode P#1 (8192MB)
L3 (8192KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB)
Core P#0 PU P#8
Core P#1 PU P#9
Core P#2 PU P#10
Core P#3 PU P#11
Core P#4 PU P#12
Core P#5 PU P#13
Core P#6 PU P#14
Core P#7 PU P#15
Socket P#1 (16GB)
NUMANode P#2 (8192MB)
L3 (8192KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB)
Core P#0 PU P#16
Core P#1 PU P#17
Core P#2 PU P#18
Core P#3 PU P#19
Core P#4 PU P#20
Core P#5 PU P#21
Core P#6 PU P#22
Core P#7 PU P#23
NUMANode P#3 (8192MB)
L3 (8192KB)
L2 (2048KB) L2 (2048KB) L2 (2048KB) L2 (2048KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB) L1d (16KB)
Core P#0 PU P#24
Core P#1 PU P#25
Core P#2 PU P#26
Core P#3 PU P#27
Core P#4 PU P#28
Core P#5 PU P#29
Core P#6 PU P#30
Core P#7 PU P#31
Indexes: physical
Date: Tue 09 Oct 2012 12:18:38 AM EDT

Memory management from userspace.

The simplest one:

```
static int I; //first
static buf[] = “123”; //second
static struct { int a; int b } str; //third
```

First and third will come from .bss. The second will come from .data.

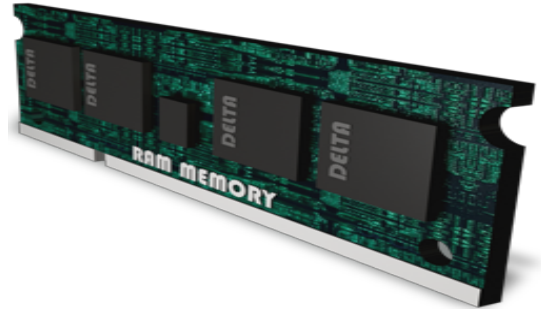


Memory management from userspace.

More “dynamic” in nature (with stack):

```
void f() {
 int I; //first
 buf[] = “123”; //second
 struct { int a; int b } str; //third
}
```

First and third will be uninitialized and nothing will be done for their initialization but extending stack frame (adding sizeof(…) to SP). Second will be derived with frame extending with MOV instruction. Be careful with even userspace stack. For main thread it's just ~4MB long by default.

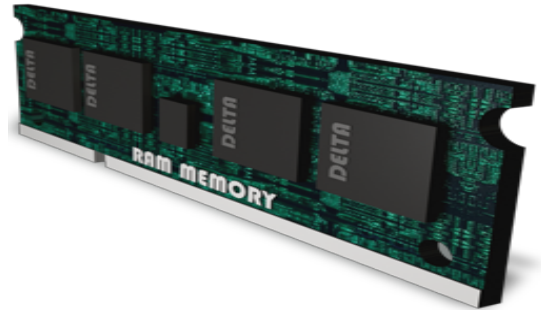


Memory management from userspace.

Even more “dynamic”:

```
void f() {
 int *I = alloca( sizeof( int ) * 10 );
}
```

Expands memory frame further. Still think about limited amount of available stack memory.

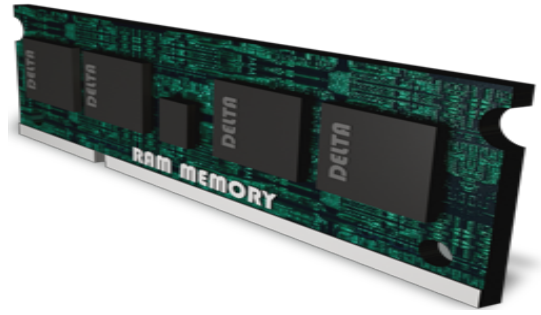


Memory management from userspace.

But what if we can't predict amount of data and this amount can be fairly big? We need heap allocator:

```
void f( size_t s ) {
 int *I = malloc( sizeof( int ) * s );
}
```

Get block from the heap.



Memory management from userspace.

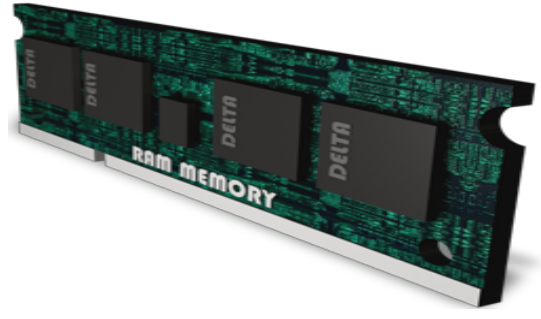
- It's so obvious! Why do you even tell us about this stuff?

- Really? Then, please, answer, why several implementations of dynamic allocator exists? What about allocation algorithms (first-fit, best-fit, buddy-list).

- Oh. Which ones?

The most popular allocators are:

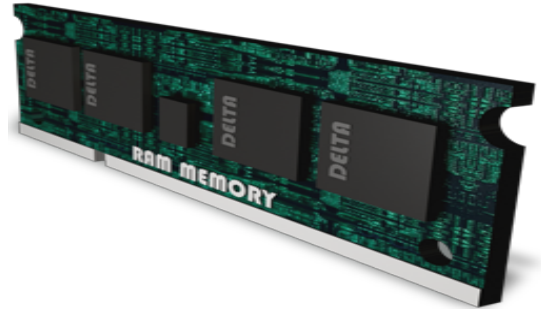
- tcalloc (the one for scalable high-load systems)
- je_malloc (scalable allocator as well as tcalloc; default allocator in freebsd, netbsd)
- ptmalloc (scalable version of dlmalloc for SMP systems; libc6)
- dlmalloc (good at embedded/mobile systems or applications)



Memory management from userspace.

Algorithms of oblivious memory allocators seems heuristic. They don't have any idea about numerical characteristics of particular usage case. How can we hint allocator about our intents and preferences? Moreover, how can we optimize and decrease frequency of object initialization/destruction? We don't need it each time we want to use object for short time.

The answer was delivered by Jeff Bonwick and introduced in the Solaris 2.4 kernel.



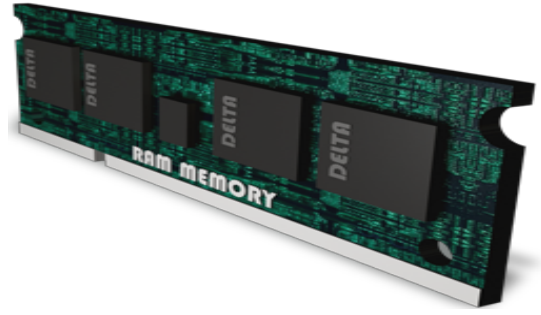
And...?

Jeff Bonwick introduced SLAB allocator. Slab is simple object cache where you can put to and get from already allocated and appropriately initialized structures.

The fact is that you know exact size of crucial data structures in your code. If you need a lot of such structures and need to allocate it dynamically you will definitely extract benefit from usage of SLAB.

What are users SLAB in these latter days?...

It's Linux kernel!



Linux kernel...?

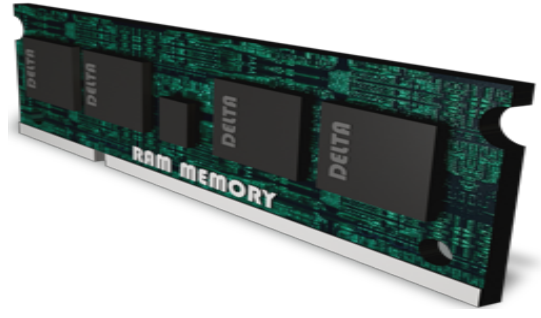
Here is main course:

```
struct kmem_cache *
kmem_cache_create(const char *name, size_t
size, size_t align, unsigned long flags,
void (*ctor)(void *));

void * kmem_cache_alloc(struct kmem_cache
*cachep, gfp_t flags);

kmem_cache_free(task_struct_cachep, tsk);

kmem_cache_destroy(task_struct_cachep);
```



How kernel stuff relates to userspace?

Such approach can be useful in userspace as well. Especially, it's true for object-oriented languages.

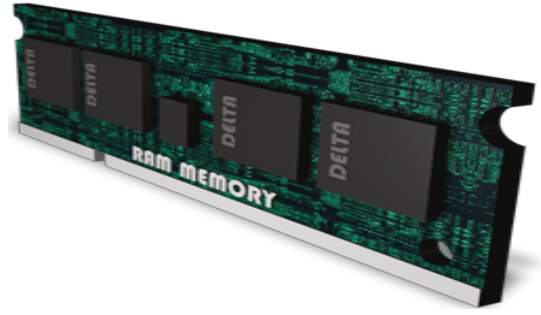
It was considered and there are several ready-for-use implementations:

- **Apache Runtime Library has slab-like machinery**
- **Glib has g_slice**
- **C++ Boost has pool**

And...

I've written another one and named it as libmempool.

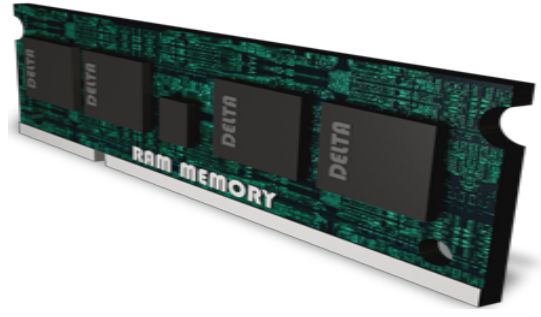
https://github.com/buffovich/libmempool



What? Another one? Who do you think you are? For what reason?

Just few points:

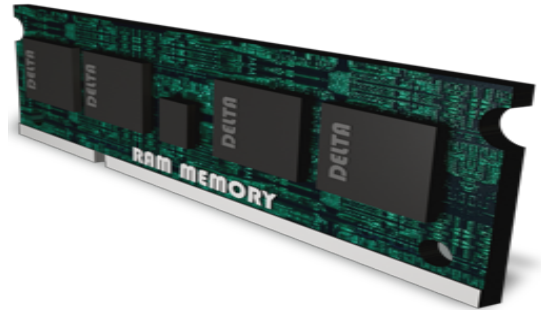
- **Fully statically configurable for big/small memory footprint; for being thread-safe or not (if it's not then a few bytes-per-cache will be saved and there will be no lock/unlock overhead); for being colored or not (a few bytes for slab can be saved)**
- **Project is not attended to particular library/framework/environment; you can use it absolutely separately;**
- **Thread-safety**
- **Optional reference counting for block**
- **Statically configured allocation backends (tcalloc/ptalloc/dlalloc/je_malloc)**
- **Constructors/destructors/”recyclers” support**



Sounds tasty... Something else?

Yep:

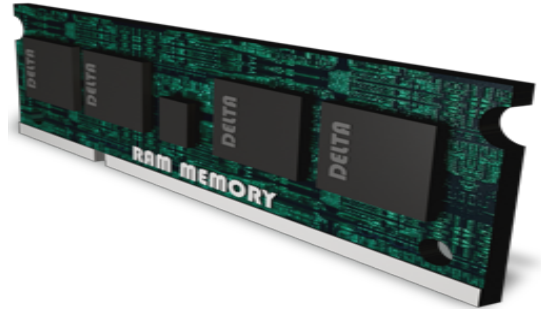
- **Ready-for-run makefile**
- **Doxygen documentation**



What about maturity and unit-tests?

I knew you ask me that.

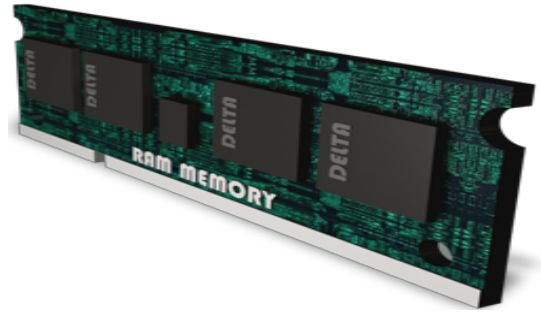
- **Unfortunately, there are no unit-tests yet; it will be implemented, of course; test plan is ready.**
- **Moreover, you may notice the absence of cache coloring; it will be implemented also.**
- **In extreme cases (a huge amount of structures are going to be allocated/deallocated), allocation backend might commit noticeable percentage to the performance graphs. Going to implement “raw” backend based on POSIX mmap.**



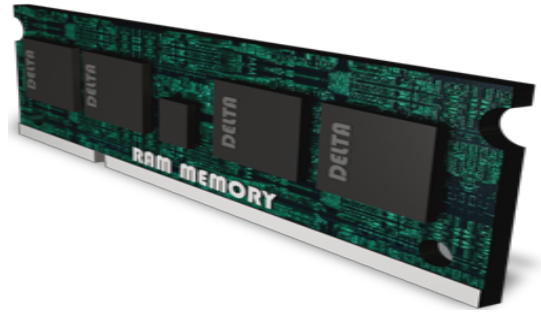
Thanks to ...

Denis Pynkin (EPAM Systems)

Artem Sheremet (EPAM Systems)



Choose your destiny...



...Joking :-)

Ask your questions, please!