# Asymmetric NUMA: Multiple-memory management for the rest of us.

**Paul Mundt**
Renesas Technology
2-6-2 Otemachi, Chiyoda-ku, Tokyo, 100-0004
paul.mundt@renesas.com

October 16, 2007

**Abstract**

Embedded processors have long shipped with small blocks of on-chip memory for fine-tuned and hand-optimized applications. With the move towards smaller processes, these blocks are becoming both increasingly larger in capacity (128k, 256k, 512k, etc.) and increasingly underutilized. Static utilization has been the common approach for handling these blocks, but this is highly inflexible, does not lend itself to kernel/application transparency, and largely side-steps the existing VM infrastructure.

With the growing trend of multi-core CMPs in the embedded space, complex multiple-memory hierarchies are becoming common place. In these cases, page locality, worst-case allowable latency, and the effects on caching behaviour can make or break an application. This paper discusses the existing, on-going and future work for generalizing NUMA support in the kernel. Management of small asymmetric nodes in both UP and SMP configurations are discussed, as well as the impact this has for memory-aware applications aimed at modern embedded CPUs.

## 1 Introduction

Non-Uniform Memory Access (NUMA) has traditionally been a characteristic of large-scale systems, employing both many different memories and many CPUs. A NUMA system is comprised of logical "*nodes*", with each node typically containing both memory and CPUs, generally in symmetric configurations. Additionally, access times to memory on different nodes typically varies, often due to physical location, interconnect latency, and so on. This characteristic of varying access cost is defined as the node "*distance*", and is a fundamental aspect of the NUMA topology.

More recently, nodes consisting of only memory or only CPUs have started to appear, demonstrating many of the same characteristics as embedded systems with multiple memories, and furthering the need to re-evaluate node symmetry assumptions within the kernel.

### 1.1 Memory models

The kernel supports many different memory models depending on the architecture. Conventional embedded systems with only one range of physically contiguous memory and consistent access times typically use the "*flatmem*" memory model, resulting in the least amount of overhead[1]. NUMA requires the use of either the "*discontig*" memory model (now obsoleted) or the newer "*sparsemem*" memory model, the latter being the only one capable of more esoteric things, such as memory hot-add. It should be noted that while NUMA depends on one of these models, the models themselves do not necessarily imply NUMA characteristics.

### 1.2 Memory policies

Memory policies are the kernel's way of determining which node (or nodes, via a node list) to allocate memory from. NUMA-aware applications can define

[1]While *flatmem* results in the lowest amount of overhead of any of the memory models, *sparsemem vmemmap* is an alternate low-overhead solution for the sparsemem model, it is slowly being adopted by sparsemem platforms for mapping the mem map with large TLBs.

their own memory policy by using Andi Kleen's *libnuma*[1] and its corresponding system calls, or by using *cpusets*[2]. Additionally, applications that only wish to have certain data backed by specific nodes can use *tmpfs*[3] mounted with a specific memory policy.

Several different memory policies exist, these include:

- MPOL_DEFAULT - System default policy, using node-local page placement.
- MPOL_BIND - Allocations restricted to a node list, error case if allocation can not be met by the specified nodes.
- MPOL_INTERLEAVE - Spread allocations evenly across a node list.
- MPOL_PREFERRED - Prefer allocations from specified node list, fall back on other nodes if allocation can not be satisfied by the specified nodes.

This only aims to be a rough overview of the type of capabilities available to NUMA-aware applications, details on the default system-wide policies are covered in more detail later. For more specific information on individual policies, it is recommended to consult the *libnuma* documentation, particularly the *mbind()* man page.

## 2 SRAM

Embedded CPUs are more regularly containing larger blocks of on-chip SRAM. As silicon processes get smaller, on-chip SRAM capacities grow. What used to be a 4kB block of scratch space for high-speed buffers in fine-tuned and extensively profiled applications has turned in to a 128kB block of largely under-utilized memory. As these blocks grow exponentially in size, static instrumentation by applications remains unchanged.

One other item that is often overlooked are the caching implications for SRAM-backed allocations. While special placement of pages for a specific workload are crucial for cutting down on cacheline bouncing and other common afflictions, on-chip SRAM often has the characteristic of bypassing the cache controller completely. This not only allows for a larger working set to be and remain D-cache resident while doing meaningful work in SRAM, it also reduces pressure on the snoop controller, subsequently reducing contention both on the CPU interconnect and the system bus.

### 2.1 Existing Management Schemes

Schemes to manage this type of memory have been both numerous, and creative. Current management approaches typically do one or more of the following:

- Special device node, application mmap()'s most of the block
- Home-grown allocator
- Special system calls
- Some sort of horrible sysfs thing
- Having Kconfig select individual functions to place in a special linker section (!)

Additionally, each one of these approaches generally expects userspace or the kernel to be using this area exclusively. For those implementations that attempt to support both, static utilization is the norm, with only elaborate home-grown allocators even attempting to support dynamic allocations.

While this approach may have worked for small individual blocks in the past, it is not a scalable solution. With the gradual shift to SMP multi-cores (and each core containing local memory), these existing approaches need to be continually reworked, particularly as individual cores start utilizing CPU-local allocations for per-CPU data while still attempting to get reasonable general-purpose utilization out of the block. As a result of this, application portability is virtually non-existent, and likewise, needs to be continually reworked for even the most minor changes in utilization.

### 2.2 Why NUMA?

> "*... superh is starting to use NUMA now, due to varying access times of various sorts of memory, and one can envisage other embedded setups doing that.*"
> \- Andrew Morton, on linux-mm.

While NUMA might seem like an unusual match for embedded CPUs, the different costs of access (both in terms of access time and caching behaviour) fit the memory distance model quite well. Memory policies are also a well-established interface for application developers and the kernel alike, which helps to avoid architecture-specific implementation divergence. Additionally, reinventing the wheel is rarely a good idea in software development, particularly when the new wheel lacks functionality compared to the existing one, looks completely different from all of the other wheels, and needs to be re-aligned every time you want to use it.

## 3 Kernel Semantics

With the traditional node split being symmetric, the kernel has a strong preference for trying to spread allocations evenly across online nodes at system initialization time, rather than simply consuming the first node's memory. While for some nodes (particularly those with large amounts of memory) node-local page placement is a both a clear performance win and strongly desired behaviour, smaller nodes do not desire such behaviour implicitly. This results in a few different areas that require re-eximaniation and extending beyond the system defaults. The primary items in this area are:

- System initialization memory policy.
- System-wide default memory policy.
- SLAB/SLOB/SLUB allocators.

These items are further discussed individually.

### 3.1 System initialization

At system initialization time allocations are spread across a specific node map, using an *interleave* policy. In older kernels mm/mempolicy.c*:numa_policy_init()* was simply a reference to the online node map:

```
void __init numa_policy_init(void)
{
        ...
        /* Set interleaving policy for system init. This way not all
           the data structures allocated at system boot end up in node zero. */
        if (do_set_mempolicy(MPOL_INTERLEAVE, &node_online_map))
                printk("numa_policy_init: interleaving failed\n");
}
```

This had the side-effect of spreading allocations to every node in the system, regardless of size. Subsequently the SLAB and kernel data structure allocations would consume small nodes (up to 1MB) in their entirety, simply with basic accounting, and before userspace or kernel drivers could access any of the memory. Worse still, all of the small node's memory would be consumed by SLAB caches, which are generally not of any interest to small nodes specifically.

In order to solve this particular problem, a new node-size heuristic was added to *numa_policy_init()*, forbidding the inclusion of nodes smaller than 16MB in the node interleave map used for system initialization. In the event that all nodes are smaller than that, as may be the case with tiny systems or NUMA emulation, the largest available node is used as a fallback. This is visible in the refactored code, which is now the kernel default, and which has no functional differences for larger symmetric configurations:

```
void __init numa_policy_init(void)
{
        nodemask_t interleave_nodes;
        unsigned long largest = 0;
        int nid, prefer = 0;
        ...
        /*
         * Set interleaving policy for system init. Interleaving is only
         * enabled across suitably sized nodes (default is >= 16MB), or
         * fall back to the largest node if they're all smaller.
         */
        nodes_clear(interleave_nodes);
        for_each_online_node(nid) {
                unsigned long total_pages = node_present_pages(nid);
                /* Preserve the largest node */
                if (largest < total_pages) {
                        largest = total_pages;
                        prefer = nid;
                }
                /* Interleave this node? */
                if ((total_pages << PAGE_SHIFT) >= (16 << 20))
                        node_set(nid, interleave_nodes);
        }
        /* All too small, use the largest */
        if (unlikely(nodes_empty(interleave_nodes)))
                node_set(prefer, interleave_nodes);
        if (do_set_mempolicy(MPOL_INTERLEAVE, &interleave_nodes))
                printk("numa_policy_init: interleaving failed\n");
}
```

While the 16MB cut-off works well in practice, it is a stop-gap solution that will likely be refactored using the new node states in later kernels, subsequently enabling memoryless nodes to customize the interleave nodemask in the same path.

### 3.2 Default memory policy

The system-wide default memory policy is to prefer node-local placement. This can be overridden by applictions that set an explicit policy for a memory range, file systems with an explicit policy, or by using *cpusets*.

Additionally, to best exploit the linear scanning and node-placement heuristics throughout the kernel, it is recommended that all systems place system memory in node zero when using asymmetric configurations! This is especially true on UP configurations, where *numa_node_id()* (as used by the page allocator for default page placement when no node id is explicitly provided), deriving an index from *smp_processor_id()*, will always equate to 0.

### 3.3 SLAB allocators

Each of the SLAB allocators contains special hooks for NUMA support, both in terms of API extension, and indirectly following the system-wide default memory policy by way of the page allocator. The basic in-kernel NUMA APIs focus around *kmalloc()* and *kmem_cache_create()* variants, these are *kmalloc_node()* and *kmem_cache_create_node()* respectively. Each one of these variants takes an explicit node id to perform the allocation on, a lack of a node specifier will prefer the current node (or node zero in UP configurations).

Optionally, a GFP flag (GFP_THISNODE) also exists for node-local placement when working with page allocator functions that take a *gfp_t* directly.

### 3.3.1 SLAB

With SLAB being gradually moved off of in newer kernels, the particulars of SLAB and NUMA interaction are not discussed in this paper.

### 3.3.2 SLUB

Basic support for asymmetric configurations is possible utilizing the existing heuristics, and additional patches[4] exist to keep all slab caches off of special nodes. However, SLUB still ends up taking quite a bit of space on small nodes for basic accounting, making this only an attractive option for larger nodes, or configurations where a node will not be used for any kernel allocations and can therefore be excluded from the node map.

Without patches on a 128kB node:

```
/ # cat /sys/devices/system/node/node1/meminfo
Node 1 MemTotal:          128 kB
Node 1 MemFree:            64 kB
Node 1 MemUsed:            64 kB
Node 1 Active:              0 kB
Node 1 Inactive:            0 kB
Node 1 Dirty:               0 kB
Node 1 Writeback:           0 kB
Node 1 FilePages:           0 kB
Node 1 Mapped:              0 kB
Node 1 AnonPages:           0 kB
Node 1 PageTables:          0 kB
Node 1 NFS_Unstable:        0 kB
Node 1 Bounce:              0 kB
Node 1 Slab:                8 kB
Node 1 SReclaimable:        0 kB
Node 1 SUnreclaim:          8 kB
Node 1 HugePages_Total:     0
Node 1 HugePages_Free:      0
```

roughly half of the node is available, with SLUB only utilizing 2 pages directly. Patching SLUB and excluding the small node from the nodes which SLUB touches results in:

```
/ # cat /sys/devices/system/node/node1/meminfo
Node 1 MemTotal:          128 kB
Node 1 MemFree:            72 kB
Node 1 MemUsed:            56 kB
Node 1 Active:              0 kB
Node 1 Inactive:            0 kB
Node 1 Dirty:               0 kB
Node 1 Writeback:           0 kB
Node 1 FilePages:           0 kB
Node 1 Mapped:              0 kB
Node 1 AnonPages:           0 kB
Node 1 PageTables:          0 kB
Node 1 NFS_Unstable:        0 kB
Node 1 Bounce:              0 kB
Node 1 Slab:                0 kB
Node 1 SReclaimable:        0 kB
Node 1 SUnreclaim:          0 kB
Node 1 HugePages_Total:     0
Node 1 HugePages_Free:      0
```

This works out to roughly the amount of space that is consumed by the node-local *pgdat* and pages for the bootmem map. However, this approach forbids SLUB from touching the node completely, meaning that kernel allocations will also be unspported. For this reason it is recommended to use SLOB instead.

### 3.3.3 SLOB

SLOB was lacking in several areas, namely a lack of both *sparsemem* and NUMA support, though was otherwise a much better fit for the problem space. As of 2.6.23, SLOB supports both of these, and is the recommended allocator for embedded systems seeking an asymmetric NUMA configuration. SLOB also results in the smallest amount of per-node overhead (equal to that of SLUB with patching and without the need for node list exclusion), making this the best fit for tiny nodes.

The NUMA support in SLOB is however not without its drawbacks, as noted in the source:

> NUMA support in SLOB is fairly simplistic, pushing most of the real logic down to the page allocator, and simply doing the node accounting on the upper levels. In the event that a node id is explicitly provided, alloc_pages_node() with the specified node id is used instead. The common case (or when the node id isn't explicitly provided) will default to the current node, as per numa_node_id().
>
> Node aware pages are still inserted in to the global freelist, and these are scanned for by matching against the node id encoded in the page flags. As a result, block allocations that can be satisfied from the freelist will only be done so on pages residing on the same node, in order to prevent random node placement.

in practice placement on a global freelist is not a significant performance problem with small nodes, but is still a common contention point on SMP systems, due to the CPUs contending for the single spinlock protecting the freelist. A large number of partial free pages will also result in additional overhead when linearly scanning for available pages on a specific node.

While this is something that is fairly easily corrected, it is not something that has occured yet, so this is something that will have to be taken in to consideration when choosing between available SLAB allocators.

## 4 Working with Memory

While applications have many different ways to get at memory from specific allocations, it should be noted that this type of memory is usually a scarce resource, which should only be used sparingly by carefully profiled applications. These memories are generally too small for the system to attempt to automatically balance workloads relative to CPU distance, as is usually done on symmetric configurations.

Applications mixing and matching various types of memory also need to take special care to observe access patterns, particularly to avoid situations where cachelines are bounced between CPUs or routinely refilled from and written back to memory with differing node distance.

On SMP systems, CPU placement must also be considered, as a page with CPU locality will by definition have a shorter distance in terms of latency than memory on an alternate CPU, where the access first has to contend on the CPU interconnect. For this reason, task-specific cpumasks derived from a maximum-allowable latency should also be considered when deploying NUMA-aware applications.

## 5 Future work

*cpusets* currently has some limitations in that it depends on SMP due to its utilization of scheduler domains, this is likely to be resolved in the near future. Many nodes will be too small to fully bind a task to using a self-contained cpuset, making sporadic *libnuma* control and *tmpfs*-backed data the only way forward for applications.

In addition to that, *page migration* is of limited use in these small asymmetric configurations. While tasks can isolate themselves to a specific range of memories, application pages alone may quickly OOM a small node when the pages are lazily migrated from a larger node, especially if the pressure on the target node is increased before the migration occurs. For this reason, page migration should only be used carefully between suitably sized nodes, or disabled completely.

Memory policies are a target for rework, including being made more fine-grained, something of benefit to applications wanting to use allocations bound to several different nodes without requiring a separate *tmpfs* mount for each different policy.

SLOB scalability is another target for future work, though the NUMA scalability work is something trivially modelled on top of SMP scalability work. This primarily involves the removal of the single freelist and its corresponding lock, splitting it out per-CPU and per-node, in order to remove the remaining contention points, as well as greatly reducing the additional overhead in linear scanning from list bloat.

## 6 Conclusions

While NUMA is a good fit for managing memories with differing costs, it does not come without a size cost of its own. Both a *pgdat* and a bootmem map are usually placed at the beginning of each node in order for the kernel to manage it, for some nodes ($<$ 128kB in size) this amount of overhead likely outweighs any benefits to be derived from dynamic utilization. However, this is an area where more improvement can be made, and it is expected that even smaller nodes will be easily adapted in to this model with minimal accounting overhead.

While a node could place each of these on node zero to reduce node-local overhead, this would significantly reduce any performance benefit, as slower memory would have to be accessed in order to dereference a *struct page* before access to faster memory can be made (also creating an additional contention point on SMP)! A node-local *pgdat* could be maintained while relocating the bootmem map to node zero in order to reduce overhead down to a single page frame or less (depending on whether the early allocator does pfn rounding for the *pgdat* allocation or not), this is not something that has been experimented with at the time of this writing.

## 7 Availability

With the exception of the future work items, all of the changes discussed in this paper were merged during the 2.6.23 merge window, after having been in -mm for some time. They have been part of all of the 2.6.23-rc releases since then. There are presently no outstanding patches.

## Acknowledgements

The author would like to thank Christoph Lameter, Matt Mackall, and Nick Piggin for all of the patient patch review, feedback, and most importantly, existing infrastructure!

## References

[1] Andi Kleen's libnuma, *ftp://ftp.suse.com/pub/people/ak/numa/*

[2] *Documentation/cpusets.txt*

[3] *Documentation/filesystems/tmpfs.txt*

[4] SLUB node exclusion patches, `http://marc.info/?l=linux-mm&m=118127688911359&w=2`