NUMA Distance for Heterogeneous Memory

Sean Williams
New Mexico Consortium
swilliams@newmexicoconsortium.org

Latchesar Ionkov
Los Alamos National Laboratory
lionkov@lanl.gov

Michael Lang
Los Alamos National Laboratory
mlang@lanl.gov

ABSTRACT

Experience with Intel Xeon Phi suggests that NUMA alone is inadequate for assignment of pages to devices in heterogeneous memory systems. We argue that this is because NUMA is based on a single distance metric between all domains (i.e., number of devices "in between" the domains), while relationships between heterogeneous domains can and should be characterized by multiple metrics (e.g., latency, bandwidth, capacity). We therefore propose elaborating the concept of NUMA distance to give better and more intuitive control of placement of pages, while retaining most of the simplicity of the NUMA abstraction. This can be based on minor modification of the Linux kernel, with the possibility for further development by hardware vendors.

CCS CONCEPTS

• **Software and its engineering** → *Memory management*; *Allocation / deallocation strategies*;

ACM Reference Format:
Sean Williams, Latchesar Ionkov, and Michael Lang. 2017. NUMA Distance for Heterogeneous Memory. In *Proceedings of MCHPC'17: Workshop on Memory Centric Programming for HPC (MCHPC'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3145617.3145620

1 INTRODUCTION

We are chiefly concerned with the application developers' experience when interacting with heterogeneous memory. Of course, this is a problem about the future, and the future is manifold, so really we will discuss one likely configuration of complex memory: heterogeneous NUMA, which is not coincidentally the approach used by Intel for exposure of the MCDRAM on Xeon Phi [4].

Nonuniform memory access (NUMA) is an abstraction commonly deployed for multisocket machines, in which each socket has an associated memory controller. Because of the relative distances of processors to memory controllers, *access* to memory is nonuniform, but the memory itself is assumed to be homogeneous. This introduces several subservient abstractions: the NUMA node or domain, which contains processing and memory elements that "go together," for example, a processor and its closest memory controller; the distance between any pair of NUMA nodes; and policies that use distances to decide placement of allocations onto nodes.

The traditional theory of distance is that, since memory devices themselves are homogeneous, NUMA can be treated as though it were a network, such that distance implies the number of hops or the request latency between a sender and a receiver. This also implies a simple strategy to get reasonable performance from a multisocket NUMA machine: pin a process to a single NUMA node, and allocate memory in such a way as to minimize distance. The latter part is, naturally, the default memory policy on Linux.

Another assumption behind this formulation of NUMA is that different memory devices are preferential for different processes. That is, in the multisocket machine with homogeneous memory devices, there is no particular trade-off at play: allocations should, in general, always stay within their node[1]. This is another factor in why a simple default policy works out well, because no optimization is needed, as there are no competing goals.

We see cracks in this formulation when memory devices are inhomogeneous. When a Xeon Phi is configured to expose its MCDRAM as an explicit NUMA node, its devices are assigned to nodes such that processors are paired with ordinary DRAM, and MCDRAM resides in one or more memory-only nodes. Distances are assigned such that local DRAM is closest, then all other DRAM nodes have a next closest and equal distance, then local MCDRAM is next closest, then other MCDRAM nodes have furthest and equal distance. This may seem strange, since MCDRAM is in some respects "better" than DRAM yet is considered further away, but it makes perfect sense on closer inspection: First, MCDRAM is only better in some respects, and second, it is a limited resource, so a more sensible default is for allocations *not* to reside in MCDRAM. This configuration turns MCDRAM into an "opt-in" device, and NUMA is used for implementation.

This is a very reasonable thing to do in the short term, but we see several long-term problems. First, this is unportable, as opting an allocation in to the MCDRAM requires knowing that this is the game being played, and also knowing the meaning of the NUMA node ID numbers. Second, with the existing memory policies, this works because there are only two nodes of interest (i.e., the local DRAM and local MCDRAM nodes). There is a memory policy to handle this situation, in which placement on an explicit, user-selected node is preferred, and if that cannot be accommodated, then placement falls back to the default policy.

In our view, the real culprit here is the implicit view that one distance metric adequately captures users' needs toward and beliefs about heterogeneous memory devices. As stated above, this view includes an assumption that memory devices are themselves homogeneous, and the strangeness surrounding Xeon Phi reflects

[1] There is another, less common use: if one is more concerned about bandwidth than latency, or one does not expect use of an allocation to be confined to one process or processor, then spreading an allocation across NUMA node can be useful. Fittingly, there is an interleave memory policy that handles this situation.

MCHPC'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5131-7/17/11…$15.00
https://doi.org/10.1145/3145617.3145620

a violation of that assumption. Further, it seems plausible both that NUMA will continue to be used in this fashion, and that the number and diversity of devices exposed through NUMA will increase. Therefore, we believe that one important step in accommodating heterogeneous memory into the user's experience is an extension of the existing distance system.

2 RELATED WORK

Other work in this area includes the Advanced Configuration and Power Interface (ACPI), which is an open industry specification codeveloped by Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba [1]. ACPI has coverage of power management, as well as processor and memory topology tables. The System Resource Affinity Table (SRAT) provides information about closeness of memory regions to processors. Additionally it has flags on whether the memory is hot-pluggable and/or nonvolatile. The Platform Memory Topology Table (PMTT) describes the memory subsystems in further detail, specifying relations between CPU sockets, memory controllers, and DIMMs. The memory controller information includes read/write latencies and bandwidths, as well as optimal access units. The System Locality Information Table (SLIT) provides distance between proximity domains. Although optional, the SRAT is usually present, and is used by the Linux kernel to populate the NUMA distance table. Unfortunately PMTT and SLIT are not normally supported by vendors.

Hierarchical NUMA [6] is a recently-suggested set of extensions to NUMA and the Linux kernel. The author proposes many masks for memory policy, including latency, density, bandwidth, power, and device-compute capability. Defining how these values are determined can be problematic, as they depend on data size, stride, focus on writes versus reads, and congesting traffic. Additionally, the proposal doesn't allow for a user-defined ordering of memory pools.

Other related work that exposes the memory hierarchy to programmers includes hwloc [2], which focuses on the topology of the memory hierarchy and pinning of processing elements to NUMA domains. hwloc uses libnuma [7]. memkind [3] is a library that supports memory allocations, using jemalloc [5] as its allocator. memkind and numactl/libnuma both are concerned only with NUMA (i.e., standard memory devices). Additionally, memkind provides only limited ability for determining the intention behind a NUMA node, and otherwise is a tool for managing NUMA policy and userspace heap management.

3 CONCEPT

3.1 Memory Semantics

Most application developers want easy-to-use, high-level abstractions that are suggestive of some underlying mechanics. This is already present in the design of NUMA, in which a node "corresponds to" some hardware in a generally-reasonable way, but abstracts most of the details. When it comes to heterogeneous memory, we believe this strategy should be expressed through the usual language of memory, e.g., bandwidth, latency, capacity. This then results in developers thinking about allocations in terms of where they fit within their concepts of memory, for example, "this allocation is bandwidth-sensitive," or "this allocation doesn't demand high performance."

NUMA already has much of the groundwork for this kind of approach, in the form of memory policies. On one hand, we therefore see the problem of heterogeneous NUMA as a problem of the inadequacy of existing memory policies. On the other, projects like memkind already facilitate attaching heap allocators to memory policies, so for the particular problem we've outlined, more robust memory policies would finish the story began by NUMA and memkind. Thus, the solution we outline is one in which memory policies are enhanced with the ordinary semantics of memory.

3.2 The Need for Kernel Hacking

One objection immediately presents itself: modifying the Linux kernel is invasive and can be difficult to deploy. Nonetheless, we believe that kernel modifications are the best solution, because they do the best to facilitate the "spilling" of allocations between nodes. Our expectation is that memory devices tuned for performance will also have limited capacity, and in that sense will be oversubscribed.

What should happen if a developer requests an allocation on performant memory, that exceeds the available capacity of the best device? Ideally, one wants a contiguous virtual address space that is backed by the best collection of physical pages, i.e., the allocation should back pages on the best device as long as it has free pages, then pages should come from the second-best device, and so on. This is impractical in userspace since reports of available capacity are unreliable, not least because pages can be backed at any time so there's opportunity for "races" to back pages on particular devices.

Within the kernel itself, decisions about where to back pages can be made quickly and in a serialized fashion. Furthermore, as implied above, memory policies are implemented in the kernel, so to the extent that memory semantics can be encoded in memory policies, they provide a straightforward path for implementing such modifications to the kernel. Finally, the changes required to add more memory policies are fairly minor and isolated.

3.3 Distance and Ordering

A distance metric always implies a partial ordering, i.e., an ordering in which elements can be indifferentiable, or, in which elements can occupy the same position in the ordering. NUMA distance is only a relative distance—it only implies relations between nodes—so it is only reliable for the purpose of creating such an ordering. Indeed, the default memory policy in Linux is one in which allocations are placed based on the ordering implied by NUMA distance.

Thus, in spite of the title of this paper, we argue that the problem of heterogeneous NUMA is better-treated as a problem of ordering, rather than a problem of distances. As we will argue later, it is still perfectly reasonable to deploy the same gambit, i.e., to develop distance metrics for the purpose of developing partial orders. However, most of the rest of this paper will concern problems of ordering.

This ultimately brings us to our proposal: we altered the Linux kernel to add a number of explicit NUMA node orderings and corresponding memory policies. That is, for each NUMA node, we define one or more orderings of NUMA nodes, and each ordering is associated with a policy, and the policy is to place pages on nodes

```
if (pol->mode == MPOL_HBM_ORDERING) {
  int nid, i, j;
  struct zonelist *zl;
  struct zone *z;
  struct page *page;
  long fp, tp;
  int thiscpu = raw_smp_processor_id();
  int thisnid = cpu_to_node(thiscpu);
  for(i = 0; i < MAX_NUMNODES; i++) {
    nid = numa_hbm_ordering[thisnid][i];
    if(nid < 0) break;
    zl = node_zonelist(nid, gfp);
    z = zonelist_zone(&zl->_zonerefs[0]);
    fp = zone_page_state(z, NR_FREE_PAGES);
    tp = fp;
    for(j = NR_ZONE_INACTIVE_ANON; j <= NR_KERNEL_STACK_KB; j++)
      tp += zone_page_state(z, j);
    if(fp > tp / 10) {
      page = __alloc_pages(gfp, order, zl);
      if(page)
        return page;
    }
  }
}
```

Figure 1: Linux kernel source code for allocating a page using an ordered memory policy. This code lives in the `alloc_pages_vma` function in `mm/mempolicy.c`, and must be provided (in some form) for each ordering.

according to the ordering. So if a machine has four nodes, and node 0 has a (e.g.) high-bandwidth ordering of $(2, 0, 1, 3)$, then an allocation using the high-bandwidth policy will place pages on node 2 until it is full, then on node 0 until it is full, and so forth.

NUMA distance is always specified from a source, typically the NUMA node containing the CPU that is running a querying process, to a destination, typically the NUMA node containing a memory device that is a candidate for allocation. Similarly, our proposal maps a source, a CPU running a querying process, to an ordered list of NUMA nodes. Hence the remaining discussion will always have this implicit relativity to it: everything is done with respect to a source CPU, and the full collection of orderings is two-dimensional.

4 IMPLEMENTATION

4.1 Memory Policies

Memory policies are applied by two system calls: in terms of virtual memory areas with `mbind`, and in terms of processes with `set_mempolicy`. From the user's perspective, memory policies are a pair of enumerations, on the kernel side in `include/uapi/linux/mempolicy.h` and on the user side in `include/numaif.h` (i.e., part of libnuma). In the kernel, this is actually an enumeration type, and we can add policies by adding more entries (noting that policies should be added before `MPOL_MAX`), while on the user side the policies are literals named by `#define`. Of course, one must ensure that the enumerations are equivalent in both files.

Several memory policies have node masks associated with them; for example, the `MPOL_PREFERRED` policy must indicate which node is preferred, which is implemented using a bit mask (with width `MAX_NUMNODES`, i.e., the kernel configuration parameter for the maximum number of NUMA nodes). Ordering policies do not need any data associated with *particular instances of* a policy, which we need to keep in mind when creating data structures for them.

Instantiation of an ordering memory policy within the kernel requires a couple minor changes to `mm/mempolicy.c`: we must bypass a check for a nonempty nodemask in `mpol_new`, and likewise, we need to exit early from `mpol_set_nodemask`. Both changes relate to the fact that the new policies should not have nodemasks (unlike the other nondefault policies), but cannot use the default policy path, since the policy data structure is `NULL` in the default case. Ultimately, the memory policy itself is still just a position in an enumeration, so this will ultimately create a `struct mempolicy` with its `mode` value set accordingly.

4.2 Orderings

As mentioned in Section 3.3, the orderings in question are two-dimensional, being a mapping from a source NUMA node to an ordered list of nodes. Thus, each ordering requires a matrix of integers of size `MAX_NUMNODES` × `MAX_NUMNODES`. Each source node has a row of the matrix, and each row is a list of nodes, with the end of the list indicated by a sentinel value, e.g., -1.

The actual implementation of these orderings resides in `drivers/base/node.c`, which defines the core behavior of NUMA nodes. However, we should now remark that this aspect, in our current work, is more prototypical: we will describe here what we have done, which may be adequate for the future problems, or if not, is a reasonable foundation for future work.

Initialization of the orderings is done in the `register_one_node` function, for now just by assigning any orderings[2]. Each NUMA node then exposes each of its orderings through the kobject/sysfs mechanism, as a read-write file in `/sys/devices/system/node/nodeN/`. Being writable, this file is currently how one configures the orderings: we implemented the write callback to parse a space-delimited list of numbers, implicitly adding the sentinel value to the end. Reading the file, likewise, produces a space-delimited listing, e.g.,

```
$ cat /sys/devices/system/node/node3/hbm_ord
3 0 1 2
$ echo 3 2 1 0 >
    /sys/devices/system/node/node3/hbm_ord
$ cat /sys/devices/system/node/node3/hbm_ord
3 2 1 0
```

4.3 Page Allocation

The meat of all this, now, resides in the `alloc_pages_vma` function in `mm/mempolicy.c`, which actually provides a page when needed. This function receives a virtual memory area structure, which itself contains a memory policy structure. Source code is provided in Figure 1; this prototype implementation demonstrates what we are trying to accomplish here.

We begin by determining which NUMA node the current process is running on, and then iterate through its row in the correct ordering matrix. If we find a negative value (suggesting a sentinel, i.e., the end of the ordering), we break from the loop, which will ultimately result in the page coming from an ordinary (fallback) allocation mechanism.

We then attempt to compute the node's free memory, by getting a zone data structure for it and iterating through its page counts. The node is only used if this computation indicates it is at most 90% utilized, in which case we attempt to allocate a page from this node. If the node is more than 90% used or the allocation fails, the next node is tried.

4.4 Userspace

With all this in place, use of this technique is very simple. For a complete example:

```
#include <numaif.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define SZ 4000000000

int main() {
  set_mempolicy(MPOL_HBM_ORDERING,
    NULL, 0);
  char *blob = malloc(SZ);
  size_t i;
  for(i = 0; i < SZ; i += 4096)
    blob[i] = 0;
  char path[100];
  sprintf("cat /proc/%d/numa_maps",
    getpid());
  system(path);
}
```

[2]We use a default ordering in which the local node is first, then all other nodes follow in numeric order by ID number, followed by a sentinel of −1. This is fine as a proof of concept, but obviously is not a long-term solution.

The most important line, of course, is `set_mempolicy`; the remaining code just touches all the pages to ensure they are physically-backed, then prints the `numa_maps` to verify that the pages are where they should be. This could also be done using `mbind`, though we expect that most users will prefer dealing with `set_mempolicy`.

5 FUTURE WORK

Most of our future work concerns the problem of deciding the orderings, both the number and names of orderings, and also deciding how devices should be ordered on actual machines. For the first problem, orderings in this sense are equivalent to categories, specifically, categories of intentions for usage of memory. There are several obvious categories—latency, bandwidth, capacity, and normality—with some obvious caveats about how to handle accelerators, but this issue might be worth further exploration, in case we missed something.

The issue of deciding on positions of devices in orderings is more interesting, and worthier of a lengthy discussion here.

5.1 Empiricism

Keeping in mind that for our purposes distance and (partial) order are equivalent, one might ask, why not just measure properties of interest and construct orderings from them? That is, use something like STREAM to estimate peak bandwidth, and order NUMA nodes by the numbers given by that algorithm? But the flaw with this plan is exposed by this question, as most people appreciate that STREAM is not indicative of actual usable bandwidth. Why is that?

The broader flaw with empiricism is, generalizing from specific experiences takes for granted that the total present conditions of the specific experience will either be substantively similar in later situations, or that the external contingencies of the present situation have little bearing on the generalization. This latter position is often invoked/hoped for in the expression "all things equal," and refuted by the expression "past performance is not indicative of future results." Hence the mixed feelings about STREAM represent unease about the generalizability of bandwidth measurement in high-performance computing.

Thus, we expect that walking an empirical road would lead to unending hand-wringing about testing methodology and propriety of generalizations. This could nonetheless be pursued, but only if one does so with eyes wide open.

5.2 System Architect

Another approach would be for the orderings to be part of system configuration, such that system architects could use their expectations about the use of a system's hardware to decide the various orderings by hand. This assumes that system architects themselves have a solid grasp on how application developers want to experience heterogeneous memory, but we view that as more plausible than a general mechanical decision process (that does not, itself, reiterate the problems of empiricism).

This would, in practice, limit the usefulness of this work to high-performance computing, which might make these changes trickier to deploy into the mainstream Linux kernel. Of course, these changes could be maintained as an explicit patch, to be deployed as needed on supercomputers with on-node heterogeneous memory. This would sidestep nearly all of the really serious problems, except that the extra workload on system architects would require them to consider this problem and these solutions worth the trouble. Whether that represents a lower or higher bar is unclear.

5.3 Vendor Involvement

A third approach is for vendors to provide usable specifications in hardware, accessible through, for example, ACPI. Indeed, some initial investigations on our part show that the ACPI specifications outline entries for device latency and bandwidth, but we are uncertain about the universality of support for these kinds of data. Nonetheless, if vendors of heterogeneous memory hardware exposed their intentions for these devices through a mechanism like ACPI, then many of these issues would be readily solved.

6 CONCLUSION

We have outlined a way forward for the user experience of heterogeneous memory that is exposed through the NUMA abstraction. Our solution extends the existing concepts of NUMA distance and memory policy to provide a simple, fast, easy-to-use mechanism for assigning allocations to memory semantics like "high bandwidth" and "low latency." This approach involves modification to the Linux kernel, but the modifications are limited in scope and extent, only extending features that already exist. Open questions remain about how to match devices with semantics, but we believe this work serves as a valuable foundation for addressing those issues.

REFERENCES

[1] 2013. Advanced Configuration and Power Interface. (2013). http://www.acpi.info/spec50a.htm

[2] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. 2010. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 180–186.

[3] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. 2015. *memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies.* Technical Report. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States).

[4] George Chrysos. 2014. Intel® Xeon Phi coprocessor-the architecture. *Intel Whitepaper* 176 (2014).

[5] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDCan Conference, Ottawa, Canada.*

[6] Anshuman Khandual. 2017. Hierarchical NUMA. In *Proceedings of the Linux Plumbers Conference.*

[7] Andi Kleen. 2005. A NUMA API for linux. *Novel Inc* (2005).