# HPMMAP: Lightweight Memory Management for Commodity Operating Systems

Brian Kocoloski and John Lange
*Department of Computer Science*
*University of Pittsburgh*
*Pittsburgh, PA 15260*
{*briankoco,jacklange*}*@cs.pitt.edu*

***Abstract*—Linux-based operating systems and runtimes (OS-/Rs) have emerged as the environments of choice for the majority of modern HPC systems. While Linux-based OS/Rs have advantages such as extensive feature sets as well as developer familiarity, these features come at the cost of additional overhead throughout the system. In contrast to Linux, there is a substantial history of work in the HPC community focused on lightweight OS/R architectures that provide scalable and consistent performance for tightly coupled HPC applications, but lack many of the features offered by commodity OS/Rs. In this paper, we propose to bridge the gap between LWKs and commodity OS/Rs by selectively providing a lightweight memory subsystem for HPC applications in a commodity OS/R environment. Our system HPMMAP provides isolated and low overhead memory performance transparently to HPC applications by bypassing Linux's memory management layer. Our approach is dynamically configurable at runtime, and adds no additional overheads nor requires any resources when not in use. We show that HPMMAP can decrease variance and reduce application runtime by up to 50%.**

***Keywords*-operating systems; high performance computing; cloud computing**

## I. Introduction

While traditional HPC systems have typically followed the practice of providing dedicated systems for a single large scale application, current trends in both cloud environments [1], [2] and supercomputing class systems point to a future where that is no longer the case. Instead, as cloud providers increase their HPC offerings and "in-situ" application architectures [3], [4], [5] become more prevalent, running HPC applications in a consolidated environment concurrently with multiple competing workloads is likely to become a common practice. While this movement towards consolidation offers numerous opportunities for improving access to cloud-based HPC resources [6], [7], [8], [9] and enabling exascale class systems, it introduces a new set of problems to HPC system design in the form of cross-workload interference [10], [11] and resource contention. This issue is exacerbated by the fact that the majority of current and future HPC platforms will rely on some form of a commodity operating system and runtime (OS/R) architecture based on Linux. Such systems are designed from a set of commodity design goals that are fundamentally different from the goals of an HPC system, and often result in behaviors under load that are not aligned with the requirements of an HPC application.

Linux-based OS/Rs have emerged as the dominant environment for many modern HPC systems [12], [13], [14] due to their support of extensive feature sets, ease of programmability, familiarity to application developers, and general ubiquity. Linux environments provide tangible benefits to both usability and maintainability, while generally offering acceptable performance in a dedicated and properly configured HPC system. However, we argue that as HPC systems continue to increase the degree of local workload consolidation, a commodity OS/R architecture is ill-suited to provide an appropriate level of performance for HPC-class applications. This is because commodity systems, and Linux in particular, are designed to maximize a set of design goals that conflict with those required by HPC applications. Specifically, commodity systems are almost always designed to maximize resource utilization, ensure fairness, and most importantly, gracefully degrade in the face of increasing loads. These goals often directly conflict with those of HPC environments that are generally characterized as requiring consistent performance in the face of sustained heavy loads.

The deficiencies of commodity OS/Rs have led to the development of a number of alternative architectures based on a lightweight approach [15], [16]. These lightweight kernel (LWK) OS/Rs provide optimized environments for HPC applications that avoid the pitfalls of commodity architectures. In particular, LWK-based systems are designed to provide consistent performance regardless of the current system load while also removing as much overhead as possible. As a result these systems eschew many of the features present in a commodity system, and instead provide the bare minimum of functionality needed to support a constrained set of HPC applications. Thus, while these environments are capable of providing superior performance at scale [17], [18], [19], they are generally considered difficult to use and limited in their functionality. As such they are not well suited to act as a universal OS/R for a fully consolidated environment executing a mix of HPC applications along with commodity and/or other feature-rich workloads.

In order to provide effective consolidation for an HPC-capable environment it is necessary to support both the performance characteristics required by traditional HPC ap-

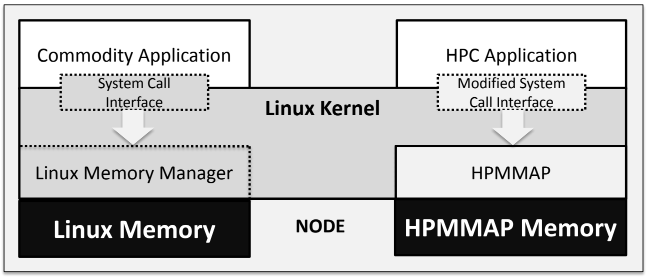


Figure 1: A high-level view of HPMMAP memory partitioning

plications, as well as the extended features needed by both commodity applications and in-situ analysis/visualization workloads. Such an ideal system would combine the features and behaviors of both a commodity and lightweight OS/R into a single environment that supported the full range of HPC and commodity applications. In this paper we focus on providing such an architecture for node level memory management through an OS extension called HPMMAP (High Performance Memory Mapping and Allocation Platform), a secondary memory management layer designed specifically for HPC environments running on commodity OS/Rs.

HPMMAP is based on the memory management design philosophy used by LWK OS/R architectures. Fundamentally, HPMMAP provides the ability to dynamically partition a node's physical memory and independently manage partitions in a separate and isolated resource management layer. As a result, HPC applications running on a consolidated platform are able to bypass the underlying commodity memory management layer and instead use a specialized lightweight memory management framework designed specifically around the requirements of HPC applications. Thus, HPMMAP not only avoids the overheads associated with Linux's commodity memory management architecture, but also is able to isolate HPC applications from interference caused by co-located commodity workloads. Furthermore, HPMMAP is implemented entirely as a kernel module and so does not require modifications to either the commodity OS/R or the applications themselves, and so provides the benefits of a lightweight memory management stack in a way that is completely transparent to HPC applications.

The architecture of HPMMAP is based on the capability, provided by modern Linux kernels, to selectively disable hardware resources. A disabled resource is effectively removed from Linux's resource management subsystems while still remaining accessible to some degree. This not only allows a user to selectively confine Linux into a dynamically configurable hardware partition, but also to assume control of the disabled resources with their own selected management frameworks. HPMMAP specifically utilizes the *memory offlining* capabilities to take control of a physically partitioned region of memory and manage it independently from the rest of the Linux kernel. This allows HPMMAP to not only isolate HPC applications from the effects of competing commodity workloads, but also to provide optimizations unavailable in a commodity system such as low overhead large (2MB/4MB) page allocations.

The contributions of our work are the following:

- We identify several issues in the Linux memory management architecture and examine their effects on HPC application performance.
- We introduce and describe the HPMMAP architecture and demonstrate how it can effectively provide LWK-like memory performance on a commodity OS/R.
- We evaluate HPMMAP on a single node and at scale and show that it is capable of improving performance by up to 50% for a set of widely used HPC benchmarks from the Mantevo[1] and ASC Sequoia[2] suites.

## II. Linux Memory Management

Historically, Linux has taken a somewhat conservative approach to memory management that focuses on the needs of commodity class systems. While features have been added to benefit HPC-class applications, they are designed as secondary components that either operate in the background (Transparent Huge Pages) or require explicit user configuration (HugeTLBfs). Both of these approaches, each of which are discussed in detail in sections II-B and II-C, provide applications with large page (2MB/4MB) memory mappings as opposed to the default (4KB), and so improve performance through shorter page table walks and decreased TLB pressure. While these existing approaches do provide performance benefits, particularly to HPC applications, they still exhibit problematic behaviors, especially when the system is experiencing significant load. We enumerate some of these issues below:

**General Linux Design Issues**

- Processes cannot be isolated from the effects of memory contention, even when mapped by large pages.
- Process address space organization is optimized for small page allocations, which often prevent large page mappings due to alignment issues and permission conflicts.

**Transparent Huge Pages Limitations**

- Merge operations are driven by OS-level heuristics without knowledge of application requirements and can occur randomly during process execution.
- Merge operations are mutually exclusive with other address space operations, requiring all page faults to block until merge completion.
- Memory pinning results in large page "splitting" in which large pages are broken down into small pages.

[1] http://http://mantevo.org/

[2] https://asc.llnl.gov/sequoia/benchmarks/

| Transparent Huge Pages | | | | |
|---|---|---|---|---|
| **Added Load** | **Fault Size** | **Total Faults** | **Avg Cycles** | **Stdev Cycles** |
| No | Small | 136,004 | 1,768 | 993 |
| | Large | 1,060 | 367,675 | 65,663 |
| | Merge | 30 | 1,005,412 | 503,422 |
| Yes | Small | 135,987 | 2,206 | 1,444 |
| | Large | 1,060 | 757,598 | 61,439 |
| | Merge | 45 | 3,360,292 | 4,017,001 |

Figure 2: Cycles needed to handle page faults using Transparent Huge Pages for the miniMD benchmark. “Merge” is a small page fault following a THP “merge” operation

| HugeTLBfs | | | | |
|---|---|---|---|---|
| **Added Load** | **Fault Size** | **Total Faults** | **Avg Cycles** | **Stdev Cycles** |
| No | Small | 1,310 | 1,350 | 1,683 |
| | Large | 84 | 735,384 | 458,239 |
| Yes | Small | 1,777 | 475,724 | 16,387,888 |
| | Large | 75 | 615,162 | 225,726 |

Figure 3: Cycles needed to handle page faults using HugeTLBfs for the miniMD benchmark

**HugeTLBfs Limitations**

- Processes experience significant numbers of page faults despite the presence of preallocated memory pools.
- Page fault handling results in significant overheads when there is memory pressure from competing workloads.
- Process stacks cannot be mapped by large pages.
- Memory pinning is generally not possible for small page regions due to a shortage of conventional memory.

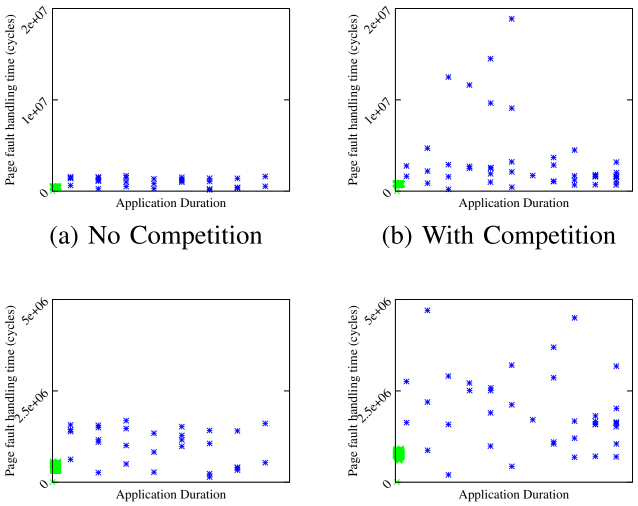
### A. General Linux Design Issues

Linux’s primary purpose as a commodity operating system is evident by its reliance on an entirely demand-paged memory allocation policy. All process memory allocation requests result in the creation of a *virtual memory area* (VMA) to account for the memory region, but do not lead to the allocation of any physical memory. All physical memory allocations are transparently handled by the page fault handler as virtual pages are accessed by the process. The primary design goal of this policy is not to maximize performance, but rather to optimize the utilization of physical memory resources while also eliminating overheads resulting from common commodity application behaviors (e.g. fork/exec). Furthermore, this design yields a system that can adequately support as many concurrent applications as possible while avoiding resource exhaustion. In HPC settings, these considerations are not nearly as important as providing an environment that maximizes performance by being simple and predictable, even if this means that the system cannot overcommit physical resources. As we will show, Linux’s policy decisions can lead to significant overheads for HPC applications whose behavior differs substantially from normal workloads, especially when compared to management schemes found in LWKs that are more concerned with providing consistent performance than maximizing the number of concurrent applications they can support.

Attempts to improve Linux’s memory performance for HPC applications have focused on providing support for large page allocations and page table mappings. It is generally accepted that large pages improve memory performance, particularly for applications that have large memory footprints and exhibit a high degree of memory locality. Large page support is currently implemented using two separate techniques: Transparent Huge Pages (THP) and HugeTLBfs. However, each of these techniques still utilize the demand paging policy from the default kernel architecture, and so are unable to avoid many of the overheads that result from repeated invocations of the kernel’s page fault handler.

Figures 2 and 3 demonstrate this behavior during execution of the miniMD benchmark from the Mantevo suite with both THP and HugeTLBfs. As Figure 2 shows, with THP, page faults resulting in large (2MB) page allocations take over 350,000 more cycles to complete than those handled with small (4KB) pages. Thus, the performance gains that large pages provide are partially offset by the overhead imposed by the page fault handler. This characteristic is exacerbated when pressure is placed on the system from additional workloads. As can be seen, when the system is put under memory pressure from additional processes (in this case, a parallel kernel build) the time that it takes to handle a page fault with a large page nearly doubles. Figure 3 demonstrates that HugeTLBfs suffers similarly from these effects. While the ability to allocate large pages is not negatively impacted by the additional workload, faults that are handled by small pages experience an increase in page fault handling time of roughly 475,000 cycles when a competing workload is present. We will discuss the specific challenges that each of these large page solutions presents in the coming sections, but it is clear in both cases that the advantages of large page allocations come at a significant cost as additional workloads compete with the HPC application for resources.

The use of large pages in Linux is further complicated by the fact that the system is primarily designed to operate using small page allocations. Address space organization and VMA layout decisions are made based on the assumption that the process will be allocating memory at the 4KB granularity. This often results in address space organizations that are incompatible with large page mappings due to alignment issues and permission conflicts, such as different read/write/execute flags assigned to small and/or unaligned VMAs. When the system encounters such configurations it has no choice but to map them using small pages.



(a) No Competition

(b) With Competition

(c) No Competition

(d) With Competition

Figure 4: Impact of competing workloads on the page fault handler using THP during the miniMD benchmark. Figures (a) and (b) show all page faults taken, while Figures (c) and (d) show the lower quarter of Figures (a) and (b), respectively. Blue dots indicate 4KB faults following a THP "merge" operation, while green dots indicate 2MB faults

### B. *Transparent Huge Pages*

THP [20] was introduced to the kernel as a fully automatic large page mechanism requiring no explicit application or administrative cooperation. THP is implemented in two separate components. First, the page fault handler will try to fix faults by allocating and mapping in large pages whenever possible. The success of a large page mapping is largely dependent on the amount of free, contiguous memory in the system, but it also depends on other characteristics, such as the alignment of nearby VMAs in the address space. Accordingly, the page fault handler may fail to fix the fault with a large page, in which case it falls back and allocates a small page to handle the fault. In addition, there is a second component of THP implemented as a background kernel thread, called *khugepaged*, that continuously attempts to allocate a large page. Once successful, *khugepaged* maps the freshly allocated large page into a valid area of the address space of any process in the system that has requested THP support. It should be noted that this "THP merging" operation may require the unmapping of a number of small pages currently mapped into the selected virtual address range.

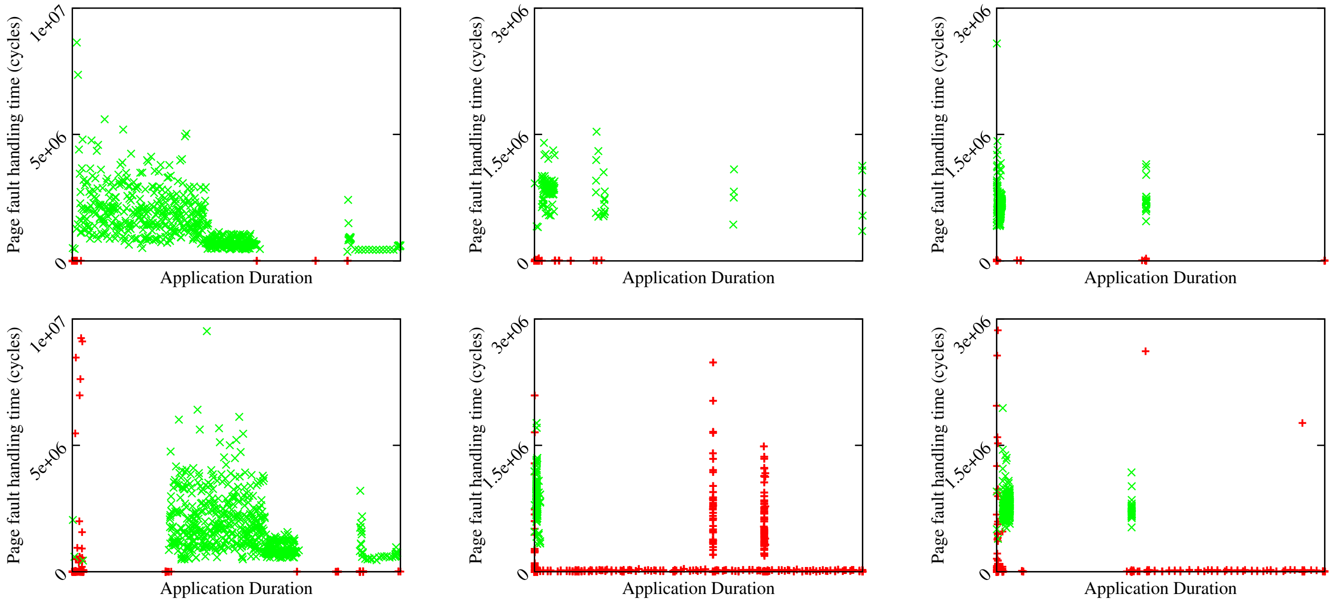
It is well-established in the HPC community that OS noise can have a significant impact on HPC application performance [21], [22]. Thus, while THP may be appropriate for commodity use, its value to HPC applications is limited, particularly due to the noise that can result from merging. Merge operations are driven by OS-level heuristics that are largely unaware of application requirements, and so can have a substantial impact on HPC workloads. When THP allocates a large page and begins to perform a merge, it locks the page tables of the process that it decides to assign the page to. While THP is performing the merge, a relatively long operation compared to a typical page fault, the process receiving the page is prevented from servicing any page faults that occur. Only after the merge has completed may the fault be handled. This behavior, as reported in Figure 2 and illustrated in Figure 4(a), leads to a roughly 1,000x increase in page fault handling time for the miniMD benchmark. Furthermore, when the system is under pressure from additional workloads, these merge delays increase substantially, as demonstrated in Figure 4(b). Finally, these merge operations are unsynchronized across parallelized application ranks, thus introducing a significant source of OS noise into the application's execution.

In addition to the overhead of merging, general memory fragmentation can be problematic for THP as well. An often suggested optimization to provide protection from both fragmentation and the effects of swapping under memory contention is to pin all memory in RAM. Linux provides the `mlock` and `mlockall` system calls that allow a process to lock a specific memory region or its entire address space, respectively, into RAM. However, this optimization is largely incompatible with THP because THP does not support the pinning of large pages. When a user specifies that a region mapped by a large page be pinned in RAM, the page is first split into small pages and then pinned.

### C. *HugeTLBfs*

HugeTLBfs [23], the other large page mechanism, is a RAM-based filesystem that allocates memory for each file using a user-specified page size. HugeTLBfs requires the presence of separate preallocated memory pools that must be explicitly reserved by a system administrator. Access to HugeTLBfs is generally managed through the *libhugetlbfs* library. This library allows the virtual address space, with the exception of the stack, to be mapped using large pages. As previously demonstrated in Figure 3, HugeTLBfs still requires a significant number of page faults to back an application's address space, even though it utilizes its own preallocated memory pools and can essentially guarantee the presence of available physical memory. Also, similarly to THP, HugeTLBfs is constrained by the process address space organization specified by the VMAs.

Figure 5 shows how the addition of competing workloads affects the page fault handler on applications backed by HugeTLBfs. While a lightly loaded system experiences a fairly low amount of overhead with HugeTLBfs, the presence of competing workloads imposes an immediate effect on all applications. The graphs in the top row of Figure 5 show the behavior of the page fault handler for a single workload configuration running an HPC benchmark.



(a) HPCCG (b) CoMD (c) miniFE

Figure 5: Impact of additional workloads on the page fault handler using HugeTLBfs. The y-axis is the number of cycles to handle a fault, while the x-axis is time. Each column is a separate benchmark. The top row is the benchmark running without any additional workloads, while the bottow row is the same benchmark running concurrently with kernel compilations. Faults fixed with small (4KB) pages and large (2MB) pages are red and green, respectively

The graphs on the bottom row show the same behavior for a benchmark co-located with an additional competing workload (parallel kernel build). In each of these figures, we see that the addition of a competing workload results in substantial increases in the time to handle page faults in areas not managed directly by HugeTLBfs. Though this behavior might seem counter-intuitive, it is largely explained by the fact that HugeTLBfs requires a separate memory pool that the default page fault handler is unable to allocate memory from. Although sufficient memory is available through HugeTLBfs, the page fault handler cannot use it as it reserved explicitly for HugeTLBfs allocations. The competing workloads saturate the remaining resources on the system and force the benchmark process to contend for now scarce small pages.

## III. HPMMAP

In this section, we present HPMMAP (High Performance Memory Mapping and Allocation Platform). HPMMAP seeks to provide low overhead memory management for HPC applications by adopting a lightweight design philosophy that bypasses the default memory management system provided by the OS. In addition, HPMMAP is able to provide isolated memory partitioning capabilities that prevent cross-workload interference from affecting the performance of HPC applications. Section III-A will provide a high-level overview of HPMMAP, including a discussion of the theory motivating its design. We will demonstrate how applications make use of HPMMAP and discuss the implementation in section III-B.

### A. *Overview*

In contrast to commodity operating systems, lightweight kernels (LWKs) are specifically built to provide an environment that can optimize HPC application performance. LWKs, such as Kitten [15] from Sandia National Labs and Blue Gene's CNK [16], generally aim to accomplish this by providing low-overhead, consistent, and predictable access to hardware resources. Such goals lead to design decisions that sacrifice things like resource sharing and fine-grained resource allocation, and instead favor more simple, coarse-grained resource management strategies. The goals of the individual subsystems found in lightweight kernels then necessarily conflict with those that exist in commodity operating environments, such as maximizing resource utilization and sharing, fairness, and security.

In this work, our goal is to provide a lightweight memory subsystem that can exist in the context of a full-fledged commodity Linux kernel in a way that can transparently but effectively support HPC application workloads. Our solution takes the form of HPMMAP, which is designed to allow a lightweight kernel memory subsystem to plug into a commodity operating system and support HPC applications. HPMMAP does not attempt to replace or augment any existing memory management techniques, but instead installs an additional lightweight subsystem that can exist in parallel with any commodity subsystems that the OS already employs. Thus, both commodity and HPC workloads can co-exist on the same machine, but need not share a memory management interface that is necessarily better at supporting one than the other.

HPMMAP has been implemented as a Linux kernel module that can be loaded into a running kernel and thus does not require kernel re-compilation or system reconfiguration to install. HPMMAP provides a lightweight memory subsystem and removes overheads at both the software and hardware levels by providing its own virtual memory management and physical memory management layers. HPMMAP's memory management layers borrow heavily from those found in the Kitten LWK. We have shown in previous work that Kitten provides a very high degree of memory performance and isolation and can support HPC applications more effectively than commodity environments [15], [24].

HPMMAP is optimized to support HPC application performance. For example, HPMMAP treats large pages (2MB by default, but up to 1GB where supported by hardware) as the fundamental unit of memory allocation, which allows it to overcome the issues that result from commodity layouts that are optimized for small page allocations. As a result, processes mapped by HPMMAP have their entire address spaces mapped by large pages by default. Furthermore, HPMMAP provides an "on-request" memory allocation policy. As opposed to the demand paging scheme found in Linux and other commodity systems, when a process requests memory from the operating system, all virtual memory regions that are created are immediately mapped to physical memory regions. This means that valid accesses to these virtual regions will generate no page faults, and thus will completely avoid overhead costs imposed by the page fault handler. The combination of default large page allocations and the elimination of page fault handling means that processes mapped by HPMMAP have almost no exposure to any overheads that can be generated from more heavyweight memory management models.

By enabling a Linux configuration feature called *Memory Hot Remove*, HPMMAP can *offline* memory from Linux and impose its own management schemes over it. Offlined memory will never be allocated by Linux, but it remains physically addressable by the CPU. HPMMAP again borrows from Kitten by using Kitten's buddy allocator to manage offlined memory. Memory offlining allows HPMMAP to isolate processes by preventing the effects of memory contention on the commodity memory regions from spilling over into the HPMMAP regions. Furthermore, the fact that offlined memory is available in sufficiently large contiguous blocks (no less than 128MB, and generally much more) ensures that HPMMAP can always allocate large pages, *never* needing to default to a smaller page size.

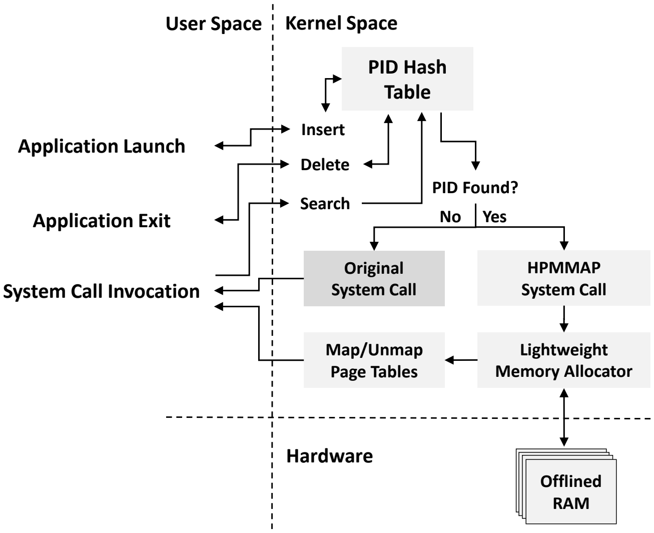


Figure 6: Application Interaction with HPMMAP

## B. *HPMMAP Implementation*

HPMMAP is designed to operate transparently to the applications using it, and thus requires no modifications or special compile time options in order to run. User-level interaction with HPMMAP is diagrammed in Figure 6. The left side of the figure illustrates user-level operations while the right side contains operations executed in the kernel by the HPMMAP subsystem. A special user-level tool is used to both launch and register applications with the HPMMAP service. Registration is done by inserting each process ID number (PID) into an internal hash table. The hash table entry remains valid for the entire lifetime of the process and is only removed when the process exits.

HPMMAP's interface is based on system call interpositioning. When any process makes a system call that requires modifications to its virtual address space, a check is made against the hash table to determine if the process is handled by HPMMAP. If the PID is not found, this indicates that the process has not requested HPMMAP support, and so the default Linux system call handler is invoked as normal. However, if the PID is present, the system call is redirected to an internal implementation provided by HPMMAP. HPMMAP then performs the specified operation on its internal state, allocating and freeing memory from its internal pools as necessary and updating the process page tables directly.

Page table modifications are implemented internally to HPMMAP and are based on a lightweight paging scheme that takes advantage of the fact that processes on 64-bit operating systems generally only use a very small portion of their virtual address space. Typically, at least 256 terabytes of virtual memory are available for the process to use, but in our experience Linux only maps a very small percentage of this memory. HPMMAP locates and maps memory into an unused memory region, with the result that these mappings are entirely independent from the virtual memory state handled by Linux. This ensures that Linux will not interfere either directly or indirectly in the operation of HPMMAP.

Our implementation of the lightweight memory manager consists of roughly 3,000 lines of C code. This includes code that implements the `mmap`, `munmap`, `mprotect` and `brk` system calls,[3] page table management code, and the buddy allocator.

## IV. Experimental Evaluation

To evaluate the efficacy of HPMMAP, we compared the performance of HPMMAP to that of a commodity Linux environment using both Transparent Huge Pages and HugeTLBfs. The goal of these experiments was to evaluate the performance of HPC applications executing concurrently with commodity workloads on the same system. Our evaluation is split into two parts. First, we ran a set of benchmarks on a single node and scaled up the amount of co-located commodity work in order to determine the effects of resource contention on HPC application performance. Second, we performed a multi-node scaling study on an 8-node HPC testbed in which each node executed co-located HPC and commodity workloads to determine whether or not the effects of contention on a single node were serious enough to result in discernible performance impacts at larger scales.

For each experiment we used the same system configuration, changing only the memory manager supporting the workloads. For the THP tests, THP managed both the HPC and commodity workloads. For the HugeTLBfs tests, HugeTLBfs managed only the HPC workload, while THP was disabled and Linux had no large page support for the commodity workload. For the HPMMAP tests, HPMMAP managed the HPC workload while THP managed the commodity workload. For the single node tests we used a dedicated Dell R415 server configured with two 6-core Opteron 4174 CPUs and 16GB of RAM. The memory layout consisted of two NUMA zones equally shared between the processors with memory interleaving disabled. The operating system was a standard Fedora 15 environment running an unmodified 2.6.43.8 (3.3.8) kernel. For the THP tests, the full system memory was available to the operating system to use. For the HugeTLBfs tests, 12GB of the 16GB was reserved at system boot time for large pages. This memory was reserved evenly across the two NUMA zones. For the HPMMAP tests, 12GB of memory was offlined, again split evenly across the two NUMA zones. The scaling tests were conducted on an 8 node experimental cluster located at Sandia National Labs. Each node was configured with two 4-core Intel Xeon X5570 CPUs, 24GB of RAM, and a 1Gbit NIC. The memory layout consisted of two NUMA zones equally shared between the processors with memory interleaving disabled. The operating system on each node was an unmodified 3.5.7 kernel built from source. For the HPMMAP tests, 20GB of memory was offlined, split evenly across the two NUMA zones.

### A. Benchmarks

The benchmarks we selected for our evaluation were taken from the Mantevo MiniApps benchmark suite from Sandia National Labs. These benchmarks are a set of “proxy applications” that exhibit the core kernel behavior common to real world HPC applications. They are generally small but are designed to exhibit the behavior of large-scale HPC applications. Each benchmark was compiled using OpenMPI (version 1.7.2) for parallel execution. We also evaluated the performance of the LAMMPS benchmark from the ASC Sequoia benchmark suite provided by Lawrence Livermore National Lab. We chose the following benchmarks for these experiments:

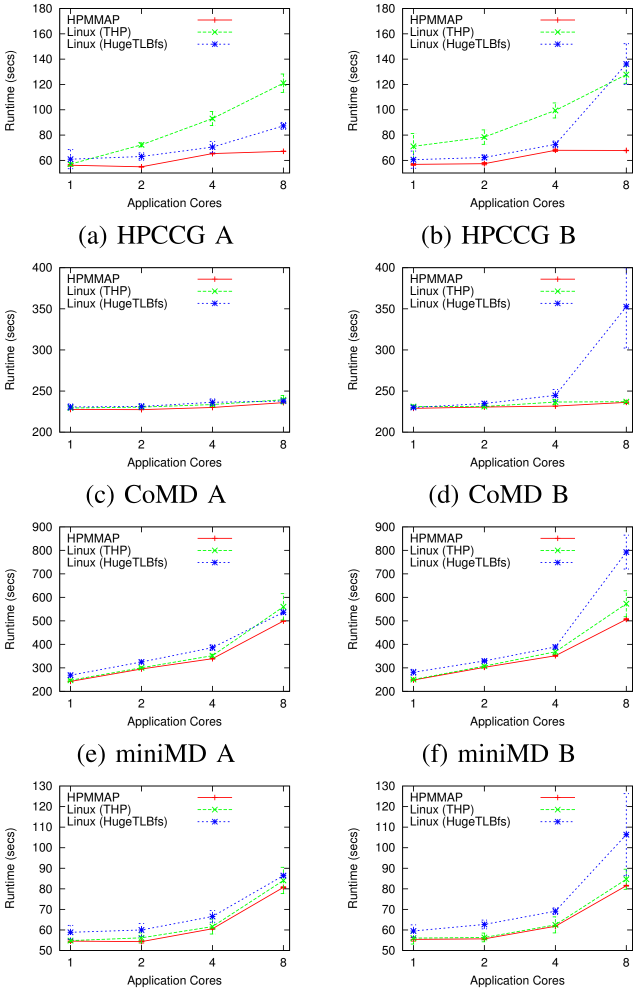
- HPCCG: A conjugate gradient solver whose workload is representative of many HPC applications (single node and scaling study)
- CoMD: A set of classical molecular dynamics algorithms used in materials science (single node)
- miniMD: A proxy for the force computations in a typical molecular dynamics application (single node)
- miniFE: An unstructured implicit finite element code (single node and scaling study)
- LAMMPS: Classical molecular dynamics simulation code (scaling study)

### B. Single Node Experiments

For each experiment, we ran the benchmark in weak scaling mode. We configured each benchmark so that it would use most of the system's resources. Some of the benchmarks require that the number of processors devoted to the application is a power of two, so we limited the scaling to 8 of the system's 12 cores. The input sizes for each benchmark were then set so that roughly 12GB of the 16GB of memory on the system were allocated to the application, as this was the amount reserved for the HugeTLBfs and HPMMAP memory pools.

For each memory manager, each benchmark was executed in two different environments that represent varying levels of competing commodity workloads. The first, which we refer to as *commodity profile A*, consisted of a single parallel kernel build. When the HPC application was running on 1, 2, or 4 cores, this kernel build was configured to run on 8 cores. Conversely, when the HPC application was running on 8 cores, we limited the parallel kernel build to 4 cores so as to not overcommit the cores. However, the second commodity environment we tested, referred to as *commodity profile B*, consisted of a duplicate of the parallel kernel build. In both cases, the kernel builds were not pinned to any memory or

[3] There are additional system calls that modify a process' virtual address space. We have found that HPC applications do not use these calls in practice, so we have not yet implemented them.



(a) HPCCG A

(b) HPCCG B

(c) CoMD A

(d) CoMD B

(e) miniMD A

(f) miniMD B

(g) miniFE A

(h) miniFE B

Figure 7: Results of the single node experiments. The left column shows the benchmarks running with *commodity profile A*, while the right column shows them running with *commodity profile B*

cores, while the HPC application was configured to pin half of its cores on each NUMA zone, while exactly half its memory was allocated from each NUMA zone (for 1 core tests, all memory came from 1 zone).

The results of these experiments are shown in Figure 7. The results of *commodity profile A* are shown in the left column while *commodity profile B* is in the right column. Each data point in these figures reports both the average and standard deviation of 10 runs of the benchmark, with the standard deviation represented by error bars. These figures clearly demonstrate that, for *commodity profile A*, HPMMAP provides superior performance for the HPC benchmarks, reducing the runtime by an average of 15% compared with THP and 9% compared with HugeTLBfs across all benchmarks. In addition, the HPC benchmarks exhibit substantial consistency improvements with runtime variance decreasing by a wide margin. In every experiment we ran, HPMMAP provided better performance than either Linux mechanism, and avoided interference from the commodity workload (seen by the low runtime variance) even as the benchmark scaled up to 8 cores. For these experiments, THP in particular showed substantial degradation as the core count increased, and in each experiment exhibited dramatically less consistency than the other configurations.

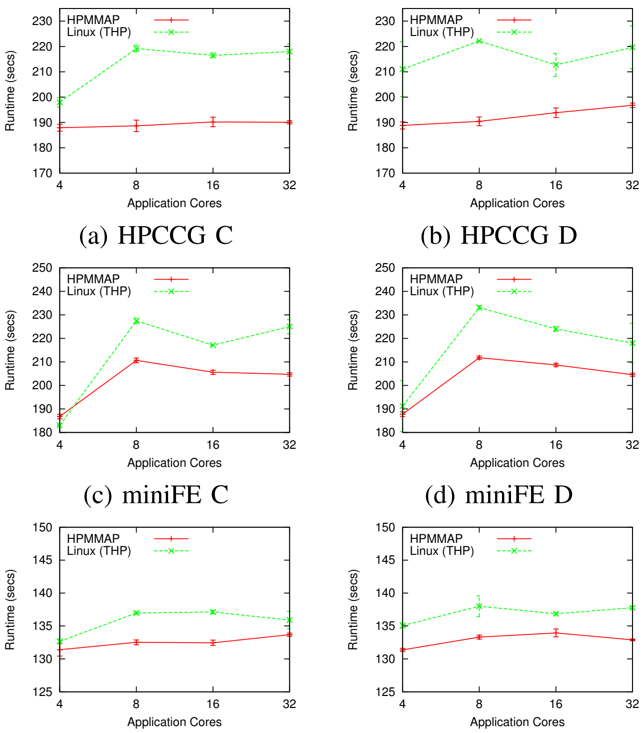
The right column of Figure 7 shows the results of the experiments when running with *commodity profile B*. Again, these results show strong evidence that HPMMAP provides a substantially better environment for HPC applications than both HugeTLBfs and THP. On average HPMMAP improves performance by 16% over THP and 36% over HugeTLBfs, and just as in the earlier experiments the runtime variance is dramatically lower as well. For these configurations HugeTLBfs was the notable outlier in that it showed significant performance degradation as the core count increased to 8. The reason for this is due to the fact that the memory pressure reached a threshold at 8 cores due to the weak scaling properties of each benchmark. As we described earlier, HugeTLBfs is especially susceptible to situations where the system is under significant memory pressure. In this case, even though enough memory was available to satisfy requests (as evidenced by the results of THP and HPMMAP), HugeTLBfs introduced significant overheads due to its effect on the allocation policy.

### C. *Scaling Experiments*

In addition to the single node experiments we conducted on a local machine, we also conducted a set of experiments to determine the impact of memory performance on application scalability. For this study we omitted HugeTLBfs experiments due to the feature not being available in the system's kernel configuration. Nevertheless, given the results from the single node tests, we believe that HugeTLBfs would continue to provide poor results when scaled up to multiple nodes. As in the single node test, we ran each benchmark in weak scaling mode.

Due to the fact that we were constrained to a Gigabit Ethernet network, we chose to limit the application to 4 cores per node instead of 8 to try to reduce the effects that limited network bandwidth would have on the benchmark performance. This allowed us to scale the benchmarks up to 32 ranks (8 nodes with 4 cores each). For these tests, we ran each benchmark with 4, 8, 16, and 32 application ranks across 1, 2, 4, and 8 nodes in order to maximize the memory utilization.

Each benchmark was again executed with two different profiles of competing workload. The first, which we refer to as *commodity profile C*, consisted of a single parallel kernel build that consumed the remaining 4 cores of the node. This workload was executed on each node that was executing the



(a) HPCCG C

(b) HPCCG D

(c) miniFE C

(d) miniFE D

(e) LAMMPS C

(f) LAMMPS D

Figure 8: Results of the scaling experiments. The left column shows the benchmarks running with *commodity profile C*, while the right column shows them running with *commodity profile D*

HPC application. The second commodity environment we tested, referred to as *commodity profile D*, consisted of 2 parallel kernel builds which each ran on 4 cores. In both cases, the kernel builds were not pinned to any memory or cores, while the HPC application was configured to pin 2 cores on each NUMA zone, while exactly half its memory was allocated from each NUMA zone.

The results are shown in Figure 8. For 32 rank runs, HPMMAP improves performance over THP by 12% for HPCCG, 9% for miniFE, and 2% for LAMMPS on *commodity profile C*, as well as by 11% for HPCCG, 6% for minFE and 4% for LAMMPS on *commodity profile D*. Additionally, the variance in performance provided by HPMMAP was generally less than that provided by THP for each of these benchmarks, and significantly less for HPCCG, which suggests that continued scaling of these applications would yield increasing levels of divergence between the memory managers.

The miniFE results are particularly interesting because they demonstrate the effects of single node variability on scaling behavior. As these figures show, on a single node THP performs comparably to HPMMAP, and perhaps slightly better under *commodity profile C*. However, as soon as the application scales past a single node, the improvement in single node consistency that HPMMAP provided translates into a significant improvement in average runtime. Although the benchmark does not scale particularly well from 1 to 2 nodes on either memory manager, this is almost certainly due to the network overhead that is introduced only after the 2nd node is added. Finally, the LAMMPS benchmark results also show steadily superior performance for HPMMAP over THP. In particular, *commodity profile D* begins to show divergence between the memory management layers. We also see that the performance provided by HPMMAP for each of these benchmarks remains very consistent.

## V. Related Work

A number of research projects have provided LWKs to support HPC application execution [25], [16], [24]. While these projects have demonstrated the effectiveness of the LWK approach, our approach is not to replace commodity OSes entirely but rather to replace an individual management layer with a lightweight version. FusedOS [26] attempts to consolidate a lightweight kernel and a fullweight kernel (FWK) on the same node by partitioning the resources of a heterogeneous multi-core system and deploying a LWK to handle HPC workloads, while maintaining the benefits of a FWK environment. Though the high-level motivation for this project is similar to ours, our approach is much more compatible in commodity architectures and cloud-based systems as it is not specifically tailored to supercomputing hardware.

Other research has focused not on replacing commodity OSes but rather on fixing the subsystems that make them problematic for HPC workloads. The ZeptoOS [27] project took the approach of modifying the Linux virtual memory management layer to eliminate the overheads that it imposes. Their solution consisted of preallocating very large regions of memory from the OS at boot time and providing a single HPC-execution environment that maps these regions with large pages. This approach effectively eliminates all overheads associated with virtual memory management, but as the developers themselves admit [12], Big Memory is not a reasonable solution for general purpose OSes. Cray's CNL [13] is another approach focused on optimizing Linux for HPC workloads. CNL provides a runtime environment based on a highly-modified version of the Linux kernel. In comparison to these approaches, our approach is more amenable to commodity environments as it does not modify existing memory management schemes that are optimized for commodity workloads, but rather creates an additional lightweight memory management layer that can exist in parallel with existing subsystems.

## VI. Conclusion

In this work, we proposed HPMMAP (High Performance Memory Mapping and Allocation Platform) and showed that the lightweight memory management it provides is capable of yielding a level of performance typically unattainable in commodity OS/Rs. HPMMAP borrows heavily from the LWK research community to impose a memory management system over partitioned hardware that provides low overhead and consistent access even in the face of significant pressure from competing application workloads. Further, HPMMAP does not modify the Linux memory subsystem but inserts itself in a way that can exist in parallel with Linux. The result is that its installation requires no system re-configuration or application modification. We demonstrated that applications using HPMMAP experience up to **50%** reduction in runtime and execute in a significantly less variable environment in the face of competing commodity workloads.

## References

[1] J. Rehr, F. Vila, J. Gardner, L. Svec, and M. Prange, "Scientific Computing in the Cloud," *Computing in Science & Engineering*, 2010.

[2] J. Napper and P. Bientinesi, "Can Cloud Computing Reach the Top500?" in *Proc. Combined Workshops on Unconventional High Performance Computing Workshop and Memory Access Workshop (UCHPC-MAW)*, 2009.

[3] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova, "In-Situ Processing and Visualization for Ultrascale Simulations," in *Journal of Physics: Proceedings of DOE SciDAC 2007 Conference*, June 2007.

[4] D. Tiwari, S. Vazhkudai, Y. Kim, X. Ma, S. Boboila, and P. Desnoyers, "Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[5] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova, "In-situ Processing and Visualization for Ultrascale Simulations," *Journal of Physics: Conference Series*, 2007.

[6] H. Viswanathan, E. K. Lee, I. Rodero, D. Pompili, M. Parashar, and M. Gamell, "Energy-Aware Application-Centric VM Allocation for HPC Workloads," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing*, 2011.

[7] F. Ferstl, "Consolidate Big Data and HPC via Virtualization - a Good Idea?" Webpage, http://www.wheregridenginelives.com/content/consolidate-big-data-and-hpc-virtualization-good-idea.

[8] A. Gupta, D. Milojicic, and S. Balle, "HPC-Aware VM Placement in Infrastructure Clouds," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, 2013.

[9] A. Gupta, D. Milojicic, and L. V. Kalé, "Optimizing VM placement for HPC in the cloud," in *Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit*, 2012.

[10] J. Brandt, A. Gentile, J. Mayo, P. Pebay, D. Roe, D. Thompson, and M. Wong, "Resource Monitoring and Management with OVIS to Enable HPC in Cloud Computing Environments," in *Proc, 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[11] B. Kocoloski, J. Ouyang, and J. Lange, "A Case for Dual Stack Virtualization: Consolidating HPC and Commodity Applications in the Cloud," in *Proceedings of the third ACM Symposium on Cloud Computing (SOCC)*, 2012.

[12] K. Yoshii, K. Iskra, P. Broekema, H. Naik, and P. Beckman, "Characterizing the Performance of Big Memory on Blue Gene Linux," in *Proc. 2009 International Conference on Parallel Processing Workshops*, 2009.

[13] L. Kaplan, "Cray CNL," in *FastOS PI Meeting and Workshop*, 2007.

[14] X.-J. Yang, X.-K. Liao, Q.-F. Hu, J.-Q. Song, and J.-S. Su, "The TianHe-1A Supercomputer: Its Hardware and Software," *Journal of Computer Science and Technology*, 2011.

[15] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell, "Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing," in *Proc. 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[16] M. Giampapa, T. Gooding, T. Inglett, and R. Wisniewski, "Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK," in *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[17] R. Brightwell, R. Riesen, K. Underwood, T. Hudson, P. Bridges, and A. Maccabe, "A Performance Comparison of Linux and a Lightweight Kernel," in *Proc. IEEE International Conference on Cluster Computing (CLUSTER)*, 2003.

[18] C. Vaughan, J. VanDyke, and S. Kelly, "Application Performance under Different XT Operating Systems," in *Cray User Group Meeting (CUG)*, 2008.

[19] E. Shmueli, G. Almasi, J. Brunheroto, J. Castanos, G. Dozsa, S. Kumar, and D. Lieber, "Evaluating the Effect of Replacing CNK with Linux on the Compute-Nodes of Blue Gene/L," in *Proc. 22nd International Conference on Supercomputing (ICS)*, 2008.

[20] J. Corbet, "Transparent Huge Pages in 2.6.38," Webpage, https://lwn.net/Articles/423584/, 2010.

[21] K. Ferreira, P. Bridges, and R. Brightwell, "Characterizing Application Sensitivity to OS Interference using Kernel-Level Noise Injection," in *Proc. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008.

[22] A. Morari, R. Gioiosa, R. Wisniewski, B. Rosenburg, T. Inglett, and M. Valero, "Evaluating the Impact of TLB Misses on Future HPC Systems," in *Proc. 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[23] M. Gorman, "Huge Pages Part 1 (Introduction)," Webpage, https://lwn.net/Articles/374424/, 2010.

[24] B. Kocoloski and J. Lange, "Better Than Native: Using Virtualization to Improve Compute Node Performance," in *Proc. 2nd International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2012.

[25] S. Thibault and T. Deegan, "Improving Performance by Embedding HPC Applications in Lightweight Xen Domains," in *Proc. 2nd Workshop on System-level Virtualization for High Performace Computing (HPCVir)*, 2008.

[26] Y. Park, E. Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. Ryu, and R. Wisniewski, "FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment," in *Proc. 24th IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2012.

[27] P. Beckman *et al.*, "ZeptoOS Project Website," Webpage, http://www.mcs.anl.gov/research/projects/zeptoos/.