Looking Inside Memory

Tooling for tracing memory reference patterns

Ankita Garg, Balbir Singh, Vaidyanathan Srinivasan
IBM Linux Technology Centre, Bangalore
{ankita, balbir, svaidy}@in.ibm.com

Abstract

Memory is a critical resource that is non-renewable and is time consuming to regenerate by reclaim. While there are several tools available to understand the amount of memory utilized by an application, there is presently little infrastructure to capture the physical memory reference pattern of an application on a live system. This knowledge would enable the software developers and hardware designers to not only understand the amount of memory used, but also the way the references are laid out across RAM. The temporal and spatial reference patterns can provide new insights into the benchmark characteristics, which would enable memory related optimizations. Additional tools could be developed on top to extract useful data from the reference information. For example, a tool to understand the working set size of an application, and how it varies with time. The data could also be used to optimize the application for NUMA systems. Kernel developers could use the data to check fragmentation and generic data placement issues.

In this paper, we introduce a memory reference instrumentation infrastructure in the Linux kernel that is built as a kernel module, on top of the trace framework. It works by collecting memory reference samples from page table entries at regular intervals. The data obtained is then post processed to plot various graphs for visualization. In this paper, we would provide information on the design and implementation of this instrumentation, along with the challenges faced by such a generic memory instrumentation infrastructure. We will demonstrate few additional tools built on this infrastructure to obtain interesting data collected from several benchmarks. The target audience are people interested in kernel based instrumentation, application developers and performance tuning enthusiasts.

1 Introduction

Typically, application developers are abstracted from the physical view of the memory by the memory management subsystem of the operating system. An application would request for memory using several well-defined APIs, which is then serviced by the operating system. The OS uses sophisticated algorithms to ensure that the memory allocation for a particular application takes place from the most appropriate location, for example, on a NUMA system, memory requests should be satisfied from a local node to reduce overhead in accessing remote memory. However, a developer looking for optimizations, can greatly benefit by having a good understanding of the way the memory is being used by the application. Finding a program's memory usage on Linux is complex. However, utilities like `ps`,`free`, `vmstat`, and proc filesystem interfaces like `/proc/meminfo` etc, provide a goof estimate of the various memory usage statistics.

Most of the existing tools, however, generally only provide information about the memory usage statistics of an application and/or system. There is very little information available on the way the memory is being referenced or the classification of accesses into kernel, user and buffers. Also, it is difficult to obtain read/write access patterns. All this information can however be obtained from hardware simulation but at a much slower speed. The infrastructure we propose here enables gathering the above missing pieces of information on a live system, running real-world applications.

2 Memory Organization

Memory is typically organized in a hierarchical manner in modern systems to have the right balance of access speed and cost. Accesses to memory addresses or pages are first looked up in TLBs (or page tables) to get the

physical address (which is needed for physically tagged cache) and then searched in several levels of caches (L1, L2 and/or L3). If the addresses are not found, then the right cache line is looked up in the main memory and loaded into the cache and registers for use. A hierarchy of page tables or similar address translation mechanisms are used by the hardware memory management unit (MMU) in the processor to map a virtual address to a physical address in memory. On most architectures, a page table entry (PTE) consists of mainly the page's physical address, and other attributes like access permissions and a **referenced** or **accessed** bit. The processor sets this bit when the page is accessed. Once set, the processor itself does not clear the bit. Software is expected to clear this bit to record new accesses. PTE also typically contains another bit called the **dirty** or **changed** bit, which indicates whether a page has been written to or not.

Memory pages can be broadly classified into :

- Kernel pages – This corresponds to the kernel code including device drivers and its data
- User pages – The application pages (both code and data)
- Page cache or buffer pages – Indirectly used by applications (unless mapped)

3 Memory Reference Pattern Instrumentation Infrastructure

The objective of the instrumentation is to track references to every page of memory over a given time interval. As explained before, memory is referenced using the page tables. Using these page table entries, memory references could be tracked in a number of ways.

3.1 Tracking Page Faults

One of the approaches to capture references is to modify the PTE entries such that a page fault is generated at every access. The fault could then be handled in a custom page fault handler routine. This could be achieved fairly easily for user pages. One could also use this data to find the exact task/routine referencing the memory. The reference data obtained thus would be very accurate. However, the disadvantages of the approach out-weigh the benefits:

- Page fault for every memory reference would slow down the system tremendously, resulting in difficulty in running long-running, real-world benchmarks and also interfere with the benchmark execution itself.
- Reference data obtained would be voluminous and cumbersome to post process unless this level of accuracy is needed
- Complex to capture kernel page reference pattern with this approach, as all the kernel pages are present in memory and not demand paged in or out. Most parts of the kernel will expect the translation to be available and do not expect a page fault.

Alternatively, to get an estimate of the reference pattern and to reduce the overhead, we could periodically reclaim or unmap several pages and follow what gets faulted in. Also, by setting up a trace event in the page fault handler (to be triggered occasionally) and forcing reclaim to reduce the RSS, we can get information on the address range of pages that are being referenced by an application. However, this technique would not be useful in getting system-wide memory reference data. While this approach can estimate the RSS and virtual address range, it cannot estimate the per-page reference rate over a short sampling interval compared to just sampling the page reference bit as explained below in section 3.3

3.2 Performance Counters

Performance counters are special hardware registers available on most modern platforms. These registers keep a track of the count of certain types of hardware events, like, instructions executed, cache misses suffered, or branches mis-predicted. Since the counting is done in the hardware, it does not slow down the kernel or applications. The Linux Performance Counter subsystem provides an abstraction of these hardware capabilities, which would work depending on the support in the underlying hardware platform.

We could use the hardware cache miss counter to approximate the memory accesses, since a cache miss implies a memory reference. However, the absolute number of cache misses does not reflect the distribution of physical memory accesses, as a memory reference could happen without a miss as well (when the page is in

cache). Besides, as noted before, it might not be possible to use this approach on platforms with no such counter. Further, it might not be possible to classify accesses into read/write.

3.3 Sampling Using PTE Reference bit

As explained in section 2 the referenced bit of the PTE tracks references to pages. In this method we adopt a sampling based approach, in which, at every sampling interval, we scan all the PTEs, looking for all the reference bits that were set to one since the previous sampling interval, indicating that those pages were accessed. We make a note of all such PTEs, along with their physical addresses. The reference bit is then cleared[1], to enable data capture for the next sampling interval. This process is repeated at every sampling interval. The particular page tables that are scanned define the type of pages that were referenced. For user space page references, the page tables of either a given task or of all the tasks are sequentially scanned for the reference bits To capture kernel page reference pattern, different approaches are needed for different platforms. On x86 systems, the `pgd` field in `struct mm` of the `init_task` points to the kernel page table directory. Using this pointer, the kernel page tables can be walked in a manner similar to user space pages. However, on Power platform, the page tables are handled differently. The hypervisor maintains a hash table of the page table entries. The entries corresponding to the kernel are bolted in the hash table. For every address, a key is generated, called `vsid`, that is used to hash into the table to obtain the PTE.

It is important to note here that if page access is served from the hardware cache, the reference bit in the PTE would still be set by the hardware and not strictly based on cache miss and memory access.

3.4 Design Overview

We have adopted the approach described in section 3.3 in our implementation. It can be easily seen that if the data is captured for every single page in the system, a lot of memory would be consumed in just capturing the information and also would pose difficulties in post-processing. Thus, instead of gathering data for every page, we group the pages in chunks and collect data for the group instead of individual pages. If any page within a group was accessed, the entire group of pages is marked as having been accessed in a particular sampling interval. On Power systems, we use the logical memory blocks (LMBs) as a means to group pages. LMBs are groups of contiguous memory, usually of 64MB or 128MB (tunable), that the hyperviser uses to give out memory to the partitions. On x86 systems however, there is presently work in progress [5] to create a notion of LMBs, in the absence of which, contiguous pages are internally grouped for the purpose of collecting data. The granularity of this group determines the accuracy of the reference pattern captured. The smaller the group size, the more accurate the data.

[1]The disadvantage of this approach is that it could interfere with the LRU algorithm that the reclaim subsystem uses. We therefore run these tests in a system that has sufficient memory to not enforce reclaim of pages, while we run our tracing framework

Seq ID	Phys Addr	Kernel	User	Page Cache	R/W

Figure 1: Output Data Format

At every sampling interval, the data that is captured is illustrated in Figure 1

The `sequence id` is a unique identifier associated with a sampling interval. The `physical address` corresponds to the physical address of the first page in a given group. The next three fields, `kernel, user and page cache`, indicate the count of the number of pages referenced in each types of page within the group. This helps in estimating the working set size of an application and also how it varies with time. We can derive more information by classifying the references into kernel, page cache or user pages. The last fields, `read and write`, indicate the number of pages in the group that had reads and writes, which is detected using the dirty or the changed bit in the PTE.

By default, the page tables of all the processes in the system are scanned for references, in addition to the kernel page tables. The user can also specify the pid of a task for restricting the PTE scanned.

3.5 perf Integration

To make it easy to use the memory reference pattern tracing infrastructure, we have integrated it with the `perf(1)` [2] framework. A trace event called `memref`

```
memref-2680  [005]  3549.112460: memref_log_data: 1230 268435456 12 0 0 1
memref-2680  [005]  3549.112461: memref_log_data: 1230 301989888 20 0 0 0
memref-2680  [005]  3549.112461: memref_log_data: 1230 335544320 0 0 0 3
memref-2680  [005]  3549.112462: memref_log_data: 1230 369098752 0 56 0 0
memref-2680  [005]  3549.112462: memref_log_data: 1230 402653184 0 10 0 1
memref-2680  [005]  3549.112463: memref_log_data: 1230 436207616 19 34 0 0
memref-2680  [005]  3549.112463: memref_log_data: 1230 469762048 0 0 0 3
memref-2680  [005]  3549.112464: memref_log_data: 1230 503316480 0 45 0 0
```

Figure 2: memref, integration with perf

is defined, which when enabled, starts capturing the reference pattern data. The `pid` is specified by echoing a value in the `memref_pid` file, created under the `tracing` directory. The binary data obtained can then be post-processed to obtain useful information, as discussed in the next subsection. Figure 2 shows a sample output from the `memref` trace event.

The first 3 columns indicate the process running, CPU number and time when the trace was captured. `memref_log_data` is the name of the trace event. Fifth column indicates the sequence number associated with the particular sampling interval. The next column indicates the physical address (corresponds to the start of the page group or a LMB). The next three columns are for kernel, user and page cache respectively. The value of the column indicates the number of pages of a particular type referenced with the group. The last column provides information on whether the access was read (0), write(1) or none (2). Also, within a group, there would be both reads and writes. However, here we trade off accuracy for the ease of data capture, by marking the whole group as having write reference.

3.6 Data Representation

The data obtained, either in raw binary or ASCII format, needs to be post-processed to obtain useful information about the reference pattern. A few useful representations of the data are as follows:

- Temporal Reference Pattern – With sampling time plotted on the X-axis and the total number of LMBs referenced on the Y-axis, we get the temporal reference pattern plot, which indicates the total amount of memory referenced at a particular time.
- Spatial Reference Pattern – The spatial reference plot indicates the number of times a given LMB was referenced over the period for which the data was collected. The LMBs are plotted on the X-axis and the reference count on the Y-axis. The plot also helps in understanding how the memory accesses are laid out in the RAM.

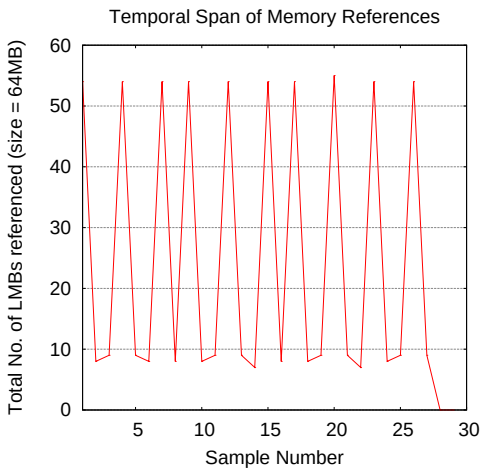

Figure 3: Temporal Reference Pattern for pagetest

- Working Set Size – The working set size of any process is defined as the set of all pages referenced by it in a given time interval. From the instrumentation data, we can compute the working set size of a process for a given sampling interval, by summing up the total number of pages referenced in each group of pages within that sampling interval, for a particular type of reference. Assuming that the page references are stable across sampling interval, one could extend it further by grouping samples to form a large time interval for computing working set size. It is important to note here that if a page is being referred to by both user and the kernel space, the kernel and user references would be accounted for separately for that page.

4 Data Verification

In order to verify the functionality of the framework, we obtain the memory reference pattern for the `pagetest` program. `Pagetest` allocates a chunk of memory, as specified by the user, using either malloc, anonymous mmap, file mmap or shared memory. It then performs either a `write` or a `read` operation on every single page that was allocated, for a specified number of iterations. To verify that the instrumentation indicates all the memory touched by the program as having been referenced, we tweak the test program to make it sleep for a

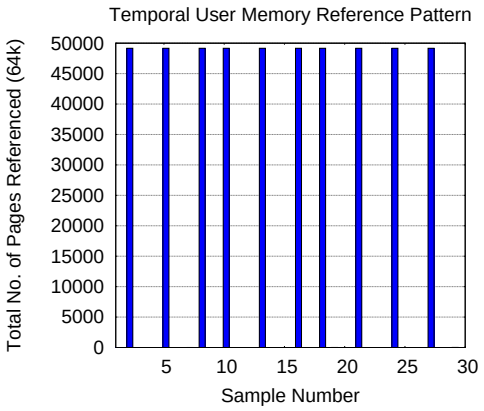

Figure 4: Temporal User Memory Reference Pattern for pagetest

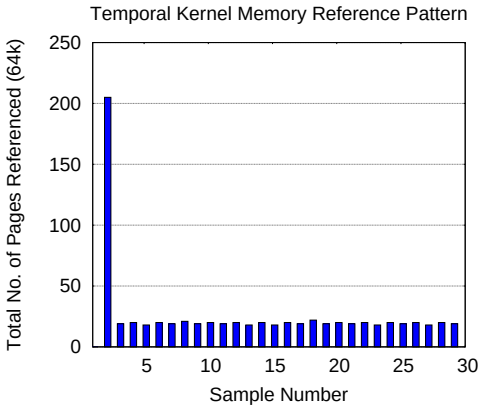

Figure 5: Temporal Kernel Memory Reference Pattern for pagetest

fixed amount of time after every iteration of touching all the pages. We run the instrumentation in parallel, at an interval that ensures that the reference data is captured after an iteration of `pagetest` is complete or in between iterations. This ensures that the instrumentation does not interfere with the reference information of the benchmark (solely for the purpose of verification). We run the benchmark to allocate 3GB of memory using `malloc`. We let the program run for 10 iterations. The temporal reference pattern span is as shown in Figure 3.

From the graph we can see that there are 10 spikes, with a height of about 52 LMBs, corresponding to the 10 iterations and usage of 3GB in each iteration. Flat lines at the bottom indicate periods when the benchmark was sleeping and no references were generated to its pages. We also verify the data obtained regarding classification

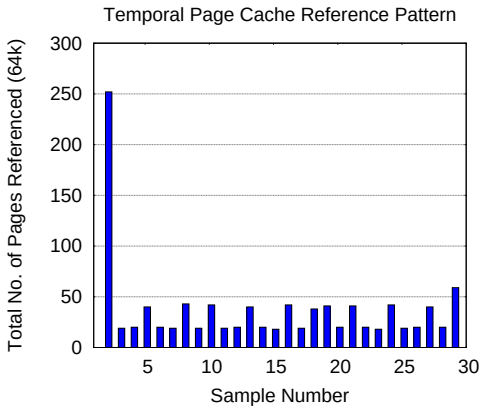

Figure 6: Temporal Page Cache Memory Reference Pattern for pagetest

of accesses into user, kernel and page cache. Figure 4 indicates the number of user pages accessed in a given sampling interval. About 49165 pages (each of size 64k) were marked as being referenced. This equals 3GB. Similarly, Figure 5 and Figure 6 indicate the number of kernel and page cache pages that were referenced respectively. Both the kernel and page cache account for less than 2MB of references each.

5 Errors & Approximations

The proposed method of aggregating memory reference patterns has known approximations and errors as detailed below:

- Effect of cache hierarchy – The primary mechanism through which the reference pattern is aggregated is the reference bit in the hardware memory management unit's page table entry. This bit is updated (or set) whenever there is a memory access to the region translated by this page. That region of memory could have been in the caches as well. Basically the reference bit is updated independent of whether the actual access was a cache hit or a cache miss. Hence if we have been looking to model bus traffic or traffic at memory controller level, then this metric needs to be correlated with last level cache miss counts to get a reasonable estimate.
- Effect of sampling – As described earlier, this method is based on periodic sampling and hence the resolution of information obtained depends on

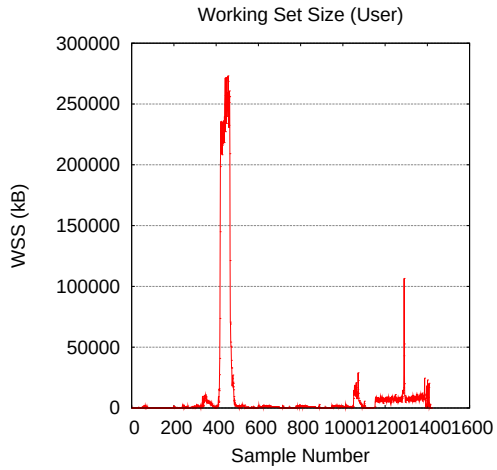

Figure 7: User Memory Working Set Size of kernbench with 64 threads

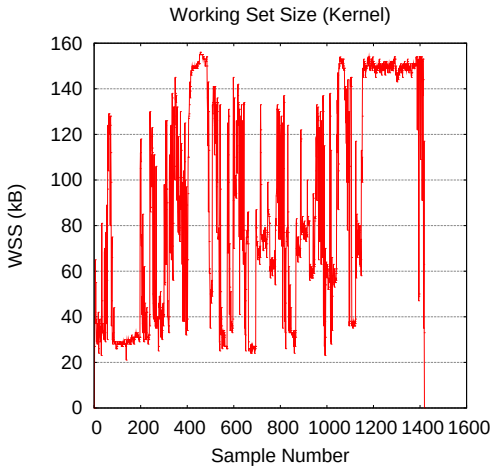

Figure 8: Kernel Memory Working Set Size of kernbench with 64 threads

the sampling interval. There are severe constraints in the lower bound of the sampling interval. Based on the size of the system and amount of data to be collected, the sampling interval may have to be larger leading to further approximation of the reference data. However one can assume that references with high temporal and spatial locality will most likely be cached and hence will not actually hit the bus and memory chips, leaving scope for optimization.

- Missing reference information – The memory regions used to store the page tables themselves may not be marked as referenced by the MMU hardware, though they may have significant reference rate. Similarly, IO DMA activities and other direct memory manipulation that does not traverse the processor MMU unit may not have been captured.

6 Sample Use Cases

The reference pattern information can be useful in several scenarios. Below we present information on some sample use cases for the different type of data obtained.

6.1 Working Set Size of an Application

Working Set Size (WSS) of an application is the amount of memory that is required to be present in the main memory at any time during its execution. The working set size could vary at different times of execution, depending on the application design and the amount of data being worked upon. If the WSS for an application is much bigger than the memory present on the system, some of the application pages would be swapped out to accommodate newer pages. This could lead to reduced performance as the swapping activity increases.

Accurately finding out the total memory that is being used by a process is complex. There is very little tooling that exists which can indicate the WSS of an application. An estimate of the instantenous WSS of an application could be obtained using the memory reference pattern data. The WSS information can be derived from the data about the number of pages accessed within a sampling interval. Ideally, the sum of the total number of kernel, user and page cache pages would be the WSS of the application for a given period. However, we present the data for each category separately, to highlight the capability of the framework. Figure 7, Figure 8 and Figure 9 show the WSS we obtained for the `kernbench` benchmark, running 64 threads on a machine with 16GB RAM and 8 processors. The WSS for user memory remains below 1MB for most times, however, hits a maximum of about 250MB. On the other hand, the maximum amount of kernel memory referenced is around 150-160KB, throughout the benchmark execution time. Page Cache memory references constitute to most of the memory references, the maximum instantaneous WSS being slightly over 500MB. This in-

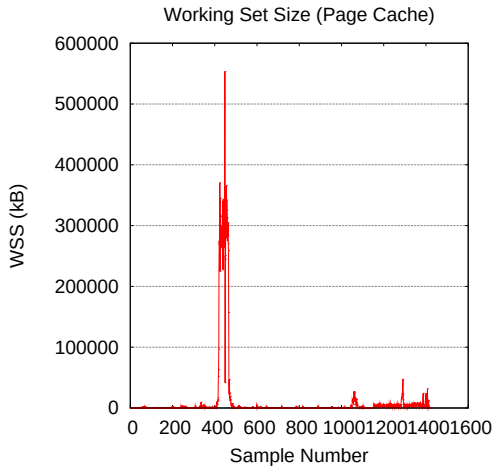

Figure 9: Page Cache Memory Working Set Size of kernbench with 64 threads

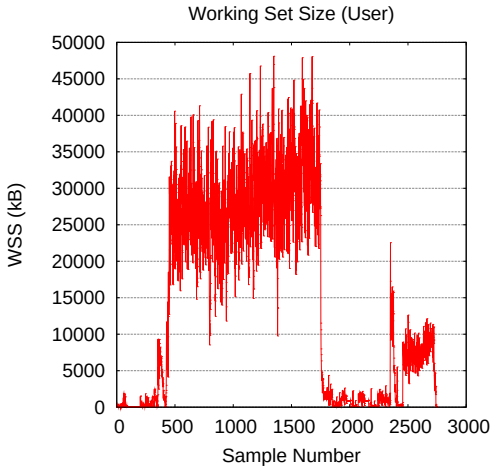

Figure 10: User Memory Working Set Size of kernbench with 16 threads

dicates that if the performance of kernbench has to be improved from memory perspective, its the optimizations in the page cache layer that would have the biggest impact. Besides, this information also indicates that if 64 threads of kernbench are to be run on this platform, maximum memory requirement would be atleast 500MB, for minimal performance impact.

Contrast this observation with the graphs in Figure 10, Figure 11 and Figure 12, which corresponds to kernbench run with only 16 threads on the same machine. It can be seen that the time taken for the benchmark is almost 50% more than the previous run. The amount of kernel memory referenced remains the same. User and page cache memory referenced are below 40MB.

6.2 Usage of Large pages

A virtual address in the virtual memory is translated into the physical address by a combination of hardware and software operations. A page is the smallest entity of address translation. Page tables store the mapping of virtual addresses to physical page addresses. There is some overhead involved in a single page translation. The total number of translations required for a program depends on the number of pages that are accessed by it. The overhead increases as the number of pages accessed are increased. Translation Lookaside Buffers (TLB) are used to reduce the overhead, by caching the frequently used page table entries. Depending on the workload, the TLB may not be sufficient to cache all the translations

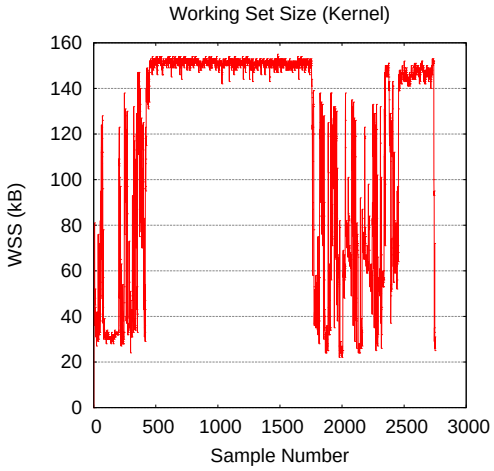

Figure 11: Kernel Memory Working Set Size of kernbench with 16 threads

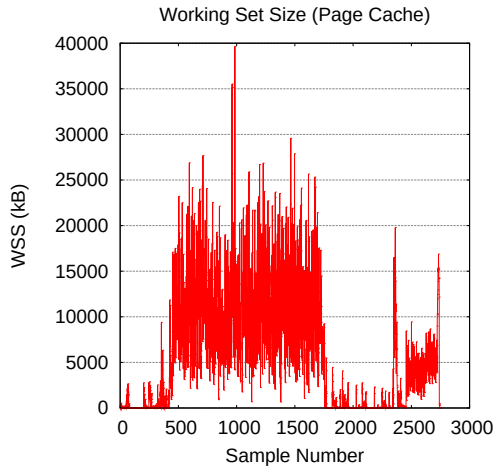

Figure 12: Page Cache Memory Working Set Size of kernbench with 16 threads

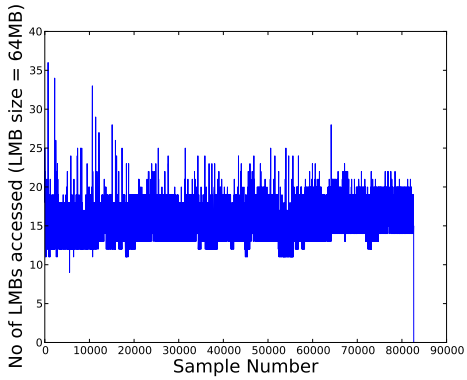

Figure 13: Temporal Reference Pattern for a JAVA Benchmark

needed by the application. Large page [4] support was thus developed to improve the performance for such applications. Large pages enable fewer TLBs to translate larger address ranges, thus allowing more entries to fit into the TLBs.

Use of large pages benefits applications that have a dense memory reference pattern. If the workload reference pattern is very sparse and making a small number of references, usage of large pages might in fact negatively impact performance, depending on the number of large page entries supported by the TLB and also due to memory fragmentation. Thus, the memory reference pattern would enable the application developer to estimate if the usage of large pages would yield performance improvement. Figure 13 indicates the temporal reference pattern of a JAVA benchmark. The graph shows a dense reference pattern and on an average, the memory reference span is about 10 LMBs, which equals about 640MB of memory. Thus, it can be inferred that this benchmark could benefit from usage of large pages.

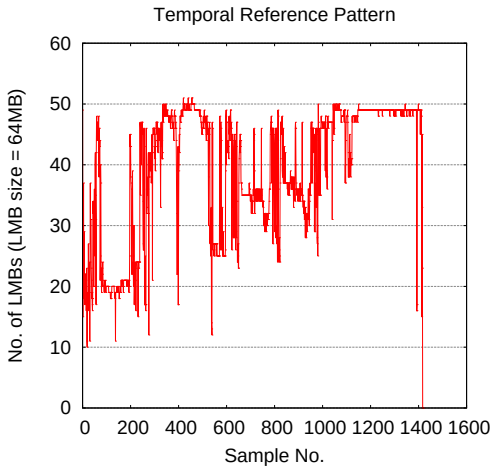

Figure 14: Temporal Reference Pattern for kernbench run with 64 threads

As an example of an application that would not benefit from using large pages is kernbench. From the Figure 14, we see that its reference pattern is not dense and varies with time. Usage of large pages could potentially lead to memory fragmentation. This is a hypothetical analysis, but could serve as a good starting point when analyzing benchmark performance issues.

6.3 Understanding Memory Usage in NUMA Systems

On a NUMA system, memory allocation becomes slightly more complex due to the presence of local and remote node. There are a number of NUMA policies that determine where the memory for a particular process will be allocated from. For example, by default, memory is allocated on the node of the CPU that triggered the allocation. It is important to determine which NUMA allocation policy works best for a given application, since a wrong policy could lead to performance degradation as there is an additional overhead incurred when accessing memory from a remote node. The memory reference instrumentation framework can aid in understanding the way memory is utilized by an application on a NUMA system. Figure 15 indicates the spatial reference plot for `kernbench`, run on a JS22 blade

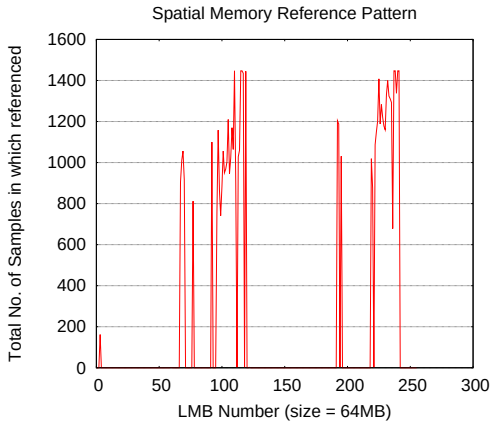

Figure 15: Without NUMA Biasing: Spatial Memory Reference Pattern of kernbench

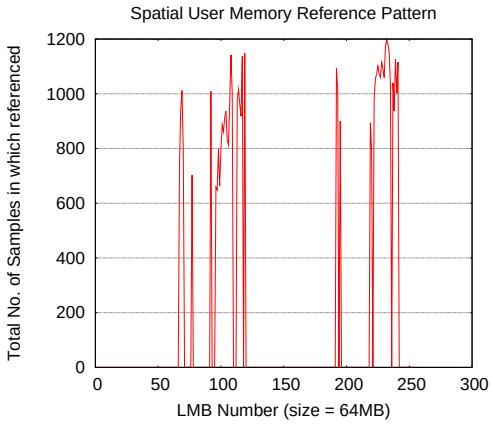

Figure 16: Without NUMA Biasing: Spatial User Memory Reference Pattern of kernbench

POWER6 blade, which had 16GB of RAM (8GB on each node). The LMB size used was 64MB. Thus, about 128 LMBs correspond to one NUMA node. It can be seen that the memory references come from both the NUMA nodes. Figure 16, Figure 17 and Figure 18 show the way references are spread across for user, kernel and page cache memory. However, when `kernbench` threads were tied to processors belonging to only a single NUMA node, we can see from Figure 20 that the user memory references originate from that node only. However, kernel and page cache references still spread across the nodes, as seen from Figure 21 and Figure 22, still giving an overall memory reference pattern as in Figure 19, similar to the one in Figure 15.

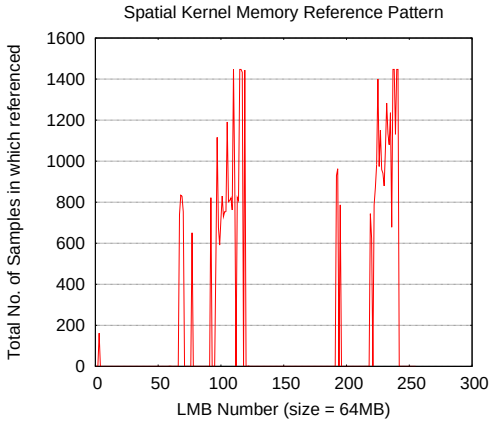

Figure 17: Without NUMA Biasing: Spatial Kernel Memory Reference Pattern of kernbench

Today, one could use `numastat`[1] to obtain information on NUMA access statistics that are obtaied from hardware and maintained by the kernel, like `numa_hit, numa_miss, local_node, other_node`, etc. When an application is run in isolation, these counters would be a representative of the application itself. Also, `/proc/<pid>/numa_maps` gives information about how the process user pages are laid out across the NUMA nodes. However, with the help of the proposed instrumentation, we can also classify the NUMA accesses to kernel, user and page cache and also into read/write accesses. Such information would prove helpful to developers, chasing the cause of peculiar benchmark performance issues on NUMA platforms. This also serves as an effective tool to evaluate NUMA policy implementation in the Linux kernel.

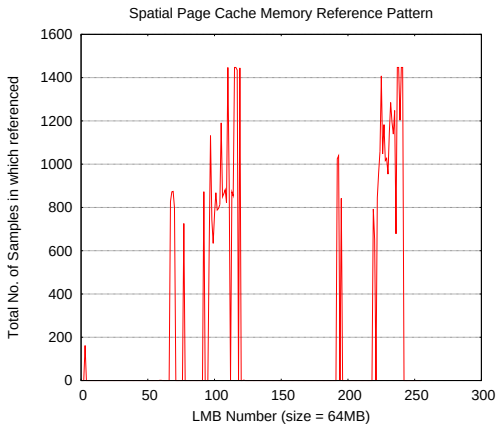

Figure 18: Without NUMA Biasing: Spatial Page Cache Memory Reference Pattern of kernbench

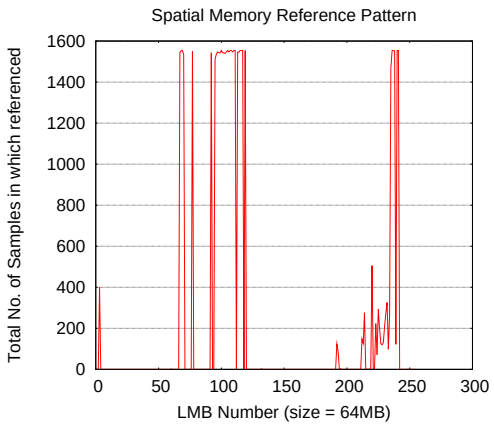

Figure 19: Biased to a NUMA Node: Spatial Memory Reference Pattern of kernbench biased to one NUMA node

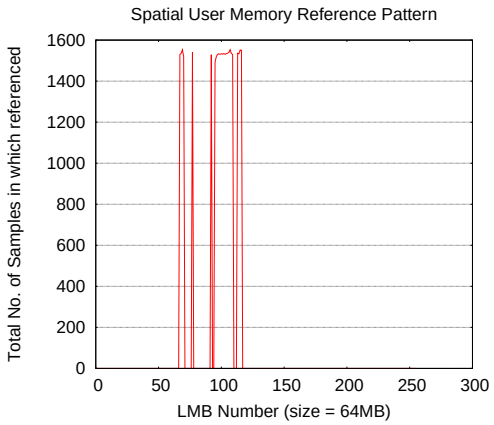

Figure 20: Biased to a NUMA Node: Spatial User Memory Reference Pattern of kernbench biased

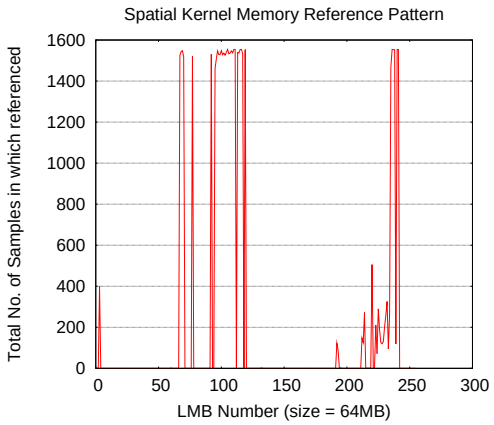

Figure 21: Biased to a NUMA Node: Spatial Kernel Memory Reference Pattern of kernbench

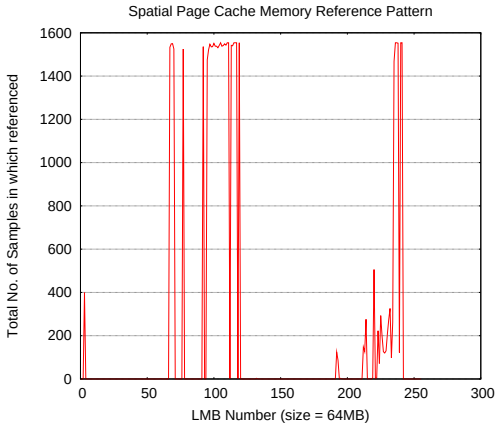

Figure 22: Biased to a NUMA Node: Spatial Page Cache Memory Reference Pattern of kernbench

7 Challenges

The goal of the instrumentation is to ensure that the data is captured with minimal impact on performance and no interference with the benchmark data. Some of the challenges that we faced are as follows:

- Missing information in Software – It is important to understand that memory references, either from caches or from main memory, are transparent and concurrent to the operating system. Due to this, the precise count of the number of times a particular page was referenced cannot be obtained from software. Hardware support in the form of counters that maintain a per-page reference data could improve the accuracy, as in System-Z [3]. Thus, from inside the software, we can only obtain information about whether a page was referenced or not within a sampling interval.

- Indeterminate run time of the instrumentation – The size of the kernel page table remains almost a constant during system runtime. Thus the time taken to scan it also remains a constant. However, the amount of time taken to scan the user page references is directly proportional to the number of processes active and their memory footprint. The larger the number of such processes, or larger their memory footprints, the greater the time required to collect the reference samples and vice versa. As a result, the run time of the scanning framework increases, leading to fewer samples being collected.

The data thus obtained may not be a true representation of the actual memory reference pattern.

- Concurrent execution of software and instrumentation – The reference data that is being obtained is simultaneously being updated or changed by software running on other CPUs. This also has an effect on the software where one of the software threads are delayed due to reference collection kernel thread thereby potentially delaying other software and affecting normal program execution like inducing lock contention.

8 Future Enhancements

Based on the limitations and challenges listed above, we have a good list of things to work on. Potential future improvements are:

- Compress or reduce data that is logged – Basically use simple encoding techniques to reduce the amount of memory used and data transferred to user space.
- Adaptively sample interesting areas of memory – Start with scanning full memory and page tables, but if we can quickly figure out the stale areas, i.e, memory areas that are rarely being accessed, we can make a note of them and scan them less frequently. We keep updating our statistics of such areas.

9 Conclusions

The rapidly developing Linux runtime tracing framework enables low overhead, complex and intrusive instrumentation to get interesting data and facilitate deeper insights into application behavior. Application memory reference tracing is one example where a combination of known techniques can be easily packaged and aptly used to get new insights into the way memory is accessed by an application, that could lead to optimizations.

10 Acknowledgements

The authors wish to thank their team at Linux Technology Centre, IBM and the management for their encouragement and support during the creation of the instrumentation framework and the paper. We would like to thank all colleagues who reviewed the patches and gave valuable feedback. Special thanks to Dipankar Sarma for his guidance all throughout.

The authors wish to thank the IBM management who generously provided an opportunity to work on this feature and paper, without which its presentation at the Linux Symposium 2010 wouldn't have been possible.

11 Legal Statement

References

[1] numastat. Linux kernel Documentation, src/linux/Documentation/numastat.txt.

[2] Performance counter frameowrk for linux. http://perf.wiki.kernel.org/.

[3] System z performance counters. .

[4] Large page support in the linux kernel, August 2002. http://lwn.net/Articles/6969/.

[5] Use lmb with x86, June 2010. http://lkml.org/lkml/2010/6/16/32.

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada

Conference Organizers

Programme Committee

Proceedings Committee