Feedback-Directed Page Placement for ccNUMA via Hardware-generated Memory Traces

Jaydeep Marathe, Vivek Thakkar and Frank Mueller

***Abstract*— Non-uniform memory architectures with cache coherence (ccNUMA) are becoming increasingly common, not just for large-scale high performance platforms but also in the context of multi-cores architectures. Under ccNUMA, data placement may influence overall application performance significantly as references resolved locally to a processor/core impose lower latencies than remote ones.**

This work develops a novel hardware-assisted page placement paradigm based on automated tracing of the memory references made by application threads. Two placement schemes, modeling both single-level and multi-level latencies, allocate pages near processors that most frequently access that memory page. These schemes leverage performance monitoring capabilities of contemporary microprocessors to efficiently extract an approximate trace of memory accesses. This information is used to decide *page affinity*, *i.e.*, the node to which the page is bound. The method operates entirely in user space, is widely automated, and handles not only static but also dynamic memory allocation.

Experiments show that this method, although based on lossy tracing, can efficiently and effectively improve page placement, leading to an average wall-clock execution time saving of over 20% for the tested benchmarks on the SGI Altix with a 2x remote access penalty and 12% on AMD Opterons with a 1.3-2.0x access penalty. This is accompanied by a one-time tracing overhead of 2.7% over the overall original program wallclock time.

***Index Terms*— Hardware performance monitoring, NUMA, trace-guided optimization, page placement**

I. Introduction

Non-uniform memory architectures with cache coherence (ccNUMA) represent an increasingly popular design for commodity and high-performance computing systems alike. The ccNUMA paradigm has spread from traditional installations, such as the SGI Altix, to AMD's Opteron x86 architecture and is now spreading to homogeneous multi-core architectures in general. The latter trend is driven by a need for high-speed interconnects, such as AMD's Hypertransport and Intel's Common System Interconnect (CSI) / QuickPath. In such systems, processors access the same global virtual address space but the physical memory is distributed across nodes and coherence is maintained using hardware mechanisms. Accesses to physical memory local to a processor (on the same node/attached to the current processing core) result in lower latencies than accesses to remote memory (on a different node/core). Similarly, accesses that hit in local caches impose a lower latency than those resolved by cache accesses to a remote node or a different core.

An OpenMP micro-benchmark was constructed to evaluate access latency on an SGI Altix machine, which represents a typical ccNUMA platform. The program counts the processor cycles required to access physical memory on the local and remote nodes. The results are shown in Table I. On average, it takes more than twice as long to load from remote memory than from memory on the local node. While this observation generalizes to ccNUMA machines, actual overheads depend on the interconnect and the number of nodes (sockets). For instance, HP reports for its Itanium-class servers latencies ranging from 185ns to 395ns for remote references for a maximum of 8 and 64 processors, respectively [1], depending on the number of hops required in their crossbar memory interconnect. Relative to local references, we measured latencies of up to 1.6x for a four-socket single core AMD Opteron system with a Hypertransport interconnect (see Section IX).

Authors' address: J. Marathe, V. Thakkar and F. Mueller, Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534, e-mail: mueller@cs.ncsu.edu, phone: +1.919.515.7889

TABLE I
Access latencies on the SGI Altix

Access Type	Average Latency [Cycles]	Standard Deviation
Local Node Memory	207	121
Remote Node Memory	430	176

The focus of this work is on multi-threaded OpenMP benchmarks for scientific computing, yet the general paradigm applies to any threaded code executed on ccNUMA machines, though experiments for non-scientific code are beyond the scope of this paper. OpenMP programs from the domain of scientific applications are often memory bound, *i.e.*, the overall wall-clock execution time of the program is significantly affected by the performance of the memory hierarchy. Any physical page placement that is sub-optimal may result in significantly longer wallclock execution time. Thus, if the bulk of the accesses are to pages whose physical memory has been allocated on a remote node, considerable performance potential may be unnecessarily sacrificed. An intelligent page-placement scheme that allocates physical memory on nodes closer to the processors with most frequent accesses to a page, in contrast, can reduce the average access latency leading to potentially significant reductions in wallclock execution time.

The objective of this work is to steer page placement intelligently. Yet, this requires us to determine the overall memory access pattern of the program and to do so efficiently since the cost of program analysis needs to be amortized by the speedups that later result from the chosen data layout optimizations. Programmers often find it difficult to reason about the best page placement for each page and would rather prefer to delegate this task to the operating or runtime system. Furthermore, system-specific details are often hard to track. For example, until now, Linux on the SGI Altix utilizes a "first-touch" page allocation policy, *i.e.*, a page is placed in the local memory of that processor that first accessed this page. Hence, compulsory initialization of data elements (*e.g.*, from a file) in a single thread (as often performed by the master thread under OpenMP) can cause the page to be allocated permanently on a particular node. Several OpenMP programs were encountered that had not been specifically tuned for ccNUMA environments and often initialized all data elements in the master OpenMP thread. This caused the bulk of the data space to be allocated in physical memory on only a single node, thereby drastically increasing the number of memory load instructions that access remote memory. But even when programs specifically initialize ("touch") data in parallel on multiple threads, sub-optimal page allocation may be observed. Such behavior was observed by determining the number of accesses to a particular page during the stable execution phase (*e.g.*, a single timestep) of an application for each thread. This data is subsequently used to derive a better page placement than the original one.

To determine the locality of memory references to pages, an efficient whole-program analysis tool is required. Such a tool monitors the memory accesses during the stable execution phase of an application. The information is subsequently utilized to derive the best page placement in terms of reducing the number of remote references to a given page. Depending on the interconnection topology, remote accesses amongst multiple nodes may incur constant or variable latencies. We handle both of the above cases in our work.

The general approach is as follows. First, a truncated one-timestep version of the target application is executed. Exploiting performance monitoring capabilities in existing microprocessors, an approximate trace of the memory accesses from all the active processors during this partial (truncated) run is efficiently extracted. This access information is subsequently used to decide the best page placement, *i.e.*, the physical node on which a particular virtual page should be allocated. This information is denoted as the "affinity hint" per page. Finally, the entire application is executed, yet with transparent wrappers that allocate pages on the assigned physical node based on the affinity hints.

The first scheme simplifies NUMA systems in that it assumes a constant latency for remote references. It leverages the default "first-touch" page allocation policy of operating systems, such as implemented on SGI IA64 Altix systems. For this approach, allocation is realized within wrappers by "touching" the target page from a processor on the assigned node. Both statically defined and dynamically allocated regions of memory are handled by initialization and allocation wrappers, respectively. Statically defined memory regions (in the `bss` segment) defer to page touching upon application startup while the page touch is delayed for dynamically allocated regions to allocation time. Experimental results on the first ccNUMA platform, an SGI Altix, show that long-latency loads provide a better indicator for page placement than TLB misses. Overall, an average wall-clock execution time savings of greater than 20% was observed over all benchmarks. The average one-time tracing overhead amounted to 2.7% of the wallclock time for the complete original NAS and SPEC OpenMP application benchmarks.

The second scheme targets ccNUMA platforms where remote memory references depend on the hop count over the interconnect. This is the case for most NUMA platforms, including HPs Integrity line, SGI's IA64 Altix machines and AMD's Hypertransport for the Opteron architecture. For the latter, the Opteron architecture, a different sampling mechanism was developed as AMD has not publicly revealed any hardware sampling support for memory references yet in contemporary Opteron systems. Instead of obtaining traces on the Opteron architecture, the hardware capabilities of Intel's performance monitoring unit on recent Pentium4/Xeon processors is utilized. These traces, based on the execution of a common application binary for both Intel and AMD processors, are utilized for deriving page affinity information as before. The page hints are subsequently driving page placement for an application running on an Opteron ccNUMA platform. The intent was to demonstrate the *portability* of the developed processor-centric scheme and also to evaluate the impact of page placement for the benchmark set on the widely used Opteron/Hypertransport architecture, which is similar to Intel's forthcoming Common System Interconnect (CSI) / QuickPath for larger multi-core architectures. In the course of this work, a novel and more aggressive page affinity policy was developed. The policy exploits the fact that remote node access latencies on NUMA systems, such as the Opteron, are not uniform but vary by the distance (hop count) between the source and target nodes. Experimental results from the Opteron platform indicate that this scheme results in an average wallclock time saving of 12%. This is lower than the wallclock time reduction observed on the SGI Altix. The difference can be attributed to the fact that remote accesses are relatively less expensive on the Opterons than on the Altix (depending on the number of hops, 1.3x/1.6x/2x on the Opterons but 2x on the Altix). Overall, the developed schemes make automatic page placement a cheap commodity that is widely transparent to the user. This result is unprecedented in its low overhead of the scheme, its elegant exploitation of hardware monitoring, operating system and runtime system support and coverage of not only static arrays, as in past work, but also heap-allocated regions.

II. Overview of the Framework

The overall framework for trace-guided page placement is depicted in Figure 1. It consists of three distinct phases — *trace generation*, *affinity decision* and *trace-guided page placement*.

During trace generation, a truncated version of the multi-threaded program consisting of a single timestep is executed. During execution, information about the memory access pattern for each thread is collected. Each OpenMP thread is explicitly bound to a different processor using the `sched_setaffinity` primitive for the experiments discussed in Section VI. Furthermore, dynamic allocation requests issued during execution are intercepted and logged. The collected information is used during the *affinity decision* phase to choose the most favorable mapping of pages to nodes, *i.e.*, the page affinity. Finally, the entire application is re-executed, this time in its non-truncated, full version, and the affinity information is utilized to force allocation of pages on the respective nodes derived from the affinity information.

The approach has been extensively automated such that user interaction is only required in three steps. First, a special header file transparently *wraps* allocation functions like malloc with calls to handler functions. Second, a call to an initialization function is placed at the very start of the program. This function effects page placement for statically-defined memory regions during trace-guided runs and initializes the hardware performance monitor during the trace collection run. Third, the user must identify the *stable execution phase* of the program and mark the phase with calls to handler functions. For example, in time-stepped programs, the stable execution phase is a single timestep. The idea is to collect a snapshot of the program's memory access patterns during a snippet of its stable execution phase, which becomes the basis for guiding page placement decisions. In the following section, a detailed description of the framework is given.

III. Trace Generation

Two types of trace information need to be captured, namely memory accesses and calls to dynamic memory allocation, each on a per-thread basis. For the former, tracing of individual accesses and misses in data translation-lookaside buffer (DTLB) are further distinguished.

The Itanium-2 performance monitoring unit (PMU) provides hardware support for capturing memory traces [2]. To this end, the `libpfm` library is utilized as it provides an interface to access the hardware counters of the processor in a convenient manner at the user level. The details of the Itanium PMU functionality

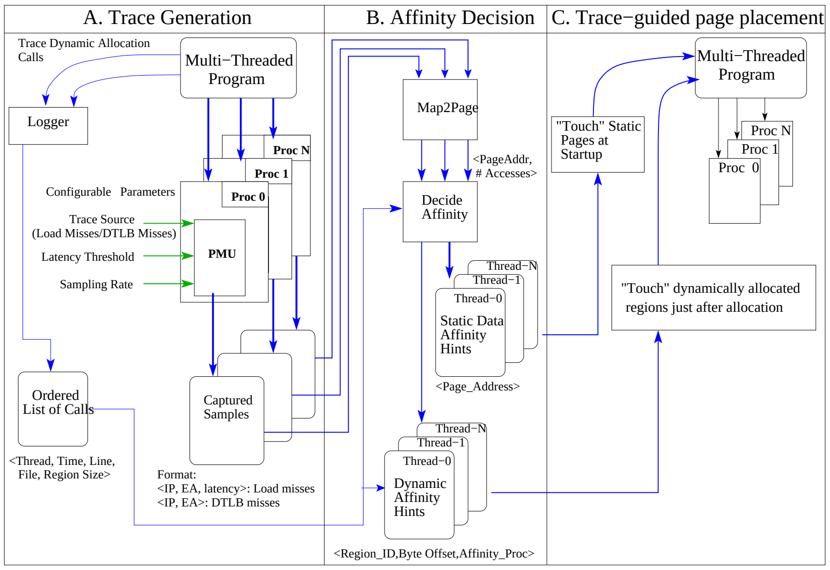
A. Trace Generation
B. Affinity Decision
C. Trace−guided page placement
Trace Dynamic Allocation Calls
Multi−Threaded Program
Logger
Proc N
Proc 1
Proc 0
Configurable Parameters
Trace Source (Load Misses/DTLB Misses)
Latency Threshold
Sampling Rate
PMU
Ordered List of Calls
<Thread, Time, Line, File, Region Size>
Captured Samples
Format: <IP, EA, latency>: Load misses <IP, EA>: DTLB misses
Map2Page
<PageAddr, # Accesses>
Decide Affinity
Thread−N
Thread−1
Thread−0
Static Data Affinity Hints
<Page_Address>
Dynamic Affinity Hints
<Region_ID,Byte Offset,Affinity_Proc>
"Touch" Static Pages at Startup
"Touch" dynamically allocated regions just after allocation

Fig. 1. Automatic Trace-guided Page Placement

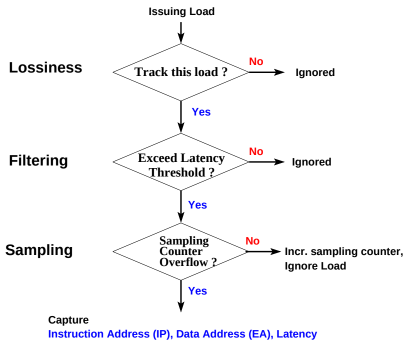
Issuing Load
Lossiness
Track this load ?
No
Ignored
Yes
Filtering
Exceed Latency Threshold ?
No
Ignored
Yes
Sampling
Sampling Counter Overflow ?
No
Incr. sampling counter, Ignore Load
Yes
Capture
Instruction Address (IP), Data Address (EA), Latency

Fig. 2. Simplified PMU Operation

are described elsewhere [3]. In this work, the PMU is utilized to capture two different types of memory access data — long latency loads and data translation lookaside buffer (DTLB) misses. A simplified view of the PMU operation for capturing long-latency loads is shown in Figure 2. When sampling long-latency loads, the PMU supports selective tracking of load instructions based on a latency threshold and optionally provides reduced statistical sampling and constraints on memory ranges of sampled addresses.

A. Capturing Memory Accesses

One pitfall common for hardware-based sampling on other architectures as well is that the PMU does not capture all long-latency loads, which exceed the latency threshold, but just of subset of them. There are two reasons for this. First, due to hardware restrictions, such as the depth of the pipeline and the ability to track only one load at a time within the pipeline, the PMU can only track one load at a time out of potentially many outstanding loads. Second, in order to prevent the same data cache load miss from always being captured in a regular sequence of overlapped cache misses, the Itanium PMU uses *randomization* to decide whether or not to track an issuing load instruction. Due to these reasons, the load miss trace is considered *lossy* when obtained by hardware tracing. In Section VII, hardware-based tracing is compared with non-lossy software-instrumented tracing to gauge the trade-offs between accuracy and efficiency.

The Itanium PMU further allows *filtering* for traces. Specifically, if a PMU-tracked load exceeds a user-configured latency threshold value, it qualifies for capture, otherwise it will be ignored, *i.e.*, the load will no longer be tracked within the pipeline. Since the access latencies increase monotonically for cache levels that are more distant from the processor, the latency threshold allows selective capturing of the load miss stream (L1-D misses, L2 data load miss stream, etc.). Due to hardware limitations, the latency threshold can only be set in in powers of 2 on the Itanium. A lower bound is given by 4 cycles.

Each filtered load increments the PMU overflow counter. By appropriately initializing this counter, the user can vary the *sampling rate* for the captured long-latency load stream (*Sampling*). The Itanium-2 has special support to capture the exact instruction address (IP) and the corresponding data address being loaded (EA) for the sampled long-latency load. In contrast, counter-overflow based sampling on other architectures can give misleading instruction addresses for the missing load due to superscalar issue, deep pipelining and out-of-order execution [3].

B. Capturing Data TLB Misses

DTLB misses can also be captured exploiting PMU functionality, yet there is no support for specifying a latency threshold. Various specific sub-types of the DTLB misses can be captured (described in more detail in [3]). In this work, all types of DTLB misses are captured by selecting the corresponding `libpfm` event `DATA_EAR_TLB_ALL`. Each captured sample contains the address of the memory access instruction that caused the TLB miss and the accessed data address. This trace source includes DTLB misses caused by both loads and stores. In contrast, the long-latency capture mechanism described earlier only monitors load instructions. Furthermore, once a DTLB miss has occurred, subsequent accesses to the page whose entry was brought into the TLB become invisible, irrespective of whether or not they miss in the L2 cache. Hence, DTLB misses are a means to capture the first access to a page efficiently, but this efficiency comes at the price of accuracy. Section VIII evaluates these trade-offs.

C. Capturing Dynamic Allocation Information

The layout of statically allocated data can be steered at initialization time of the program. Dynamically allocated data pages, however, cannot be forced onto certain nodes at initialization time as their memory becomes available only during execution. Recall the approach of trace-guided page placement. The “first-touch” allocation policy, used by SGI’s Linux version for the Altix, also the default under contemporary x86 Linux systems, is leveraged to allocate physical memory pages on the intended node. To “touch” a particular virtual page address, the earliest point in the program needs to be known at which such an address becomes valid. This requires the logging of heap memory allocation calls on a per-thread basis.

During execution of the truncated program, calls to `malloc`, `calloc` and `free` are intercepted and logged. Notice that our approach is constrained to heap allocations within the truncated program. Any subsequent allocations will not be represented in the trace, which reduces page affinity information. Hence, the approach promoted in this work is more suitable for applications that pre-allocate large arrays prior to computation, as is common for most scientific codes — with the exception of dynamically changing computational methods, such as adaptive mesh refinement (AMR). This includes calls resulting from Fortran `allocate` statements. The Itanium and x86 architectures have a high-resolution timer called the "interval timer counter" (itc) or "time-stamp counter" (tsc), depending on the architecture. A post-processing tool builds a unified ordering of allocation calls across all the threads. The ordering is derived from the logged per-call timestamps and compensates for the clock skew between different processors.

IV. Affinity Decision

After obtaining per-thread memory traces, the per-node page affinity is determined, *i. e.*, it is decided on which node a physical memory page should reside for a particular virtual memory page.

Based on the approximate memory access trace and the dynamic memory allocation information, the affinity decision module currently supports two metrics, one for uniform remote latencies and one for hop-sensitive remote latencies.

The *uniform latency policy* allocates a page on the node that issues the maximum number of accesses to that page.

$$p_i \rightarrow n_j \Leftrightarrow rw_{i,j} = max_k(rw_{i,k}) \quad (1)$$

This allocation rule requires that page p_i is allocated ($\rightarrow$) in the local memory of node n_j iff the number of read/write references $rw_{i,j}$ within page p_i issued from node n_j is maximal within the read/write references issued by any node for this page, where m is the total number of nodes. Intuitively, the average latency of access can be reduced when a page is allocated closer to the processor that issues the largest amount of requests.

The page affinity decision process consists of a number of steps. Initially, accesses are grouped by page address, and the total accesses from all threads to each page are calculated. This is depicted as `map2page` in Figure 1. Here, accesses are grouped by processor to calculate the per-node access count for each page.

In practice, remote access penalties are not uniform but rather vary with the distance (number of hops over the interconnect) to the target node. The *hop-sensitive latency policy* allocates a page on the node that has the lowest aggregate access cost for references to this page issued from any node.

$$p_i \rightarrow n_j \Leftrightarrow \sum_{l=1..m} rw_{i,l} \times w_{j,l} = min_k(\sum_{l=1..m} rw_{i,l} \times w_{k,l}) \quad (2)$$

This allocation rule requires that page p_i on node n_j iff the aggregate number of read/write references $rw_{i,l}$ for this page issued from all nodes, weighted by the hop-sensitive cost $w_{j,l}$ relative to local allocation on this node, is minimal within all aggregate weighted costs of any node allocations of this page. Intuitively, the average latency of access can be reduced when a page is allocated close to all processors that issue large amounts of requests, *i.e.*, this metric takes references from multiple nodes into account instead of using the winner-takes-all paradigm.

The page affinity decision process consists of again of grouping accesses by page address and by thread (node), as indicated by `map2page` in Figure 1. Yet, the page cost for an allocation is then calculated for a hypothetical allocation to each node. This cost is the additive weighted number of accesses issued for any node for this mapping. The weight $w_{i,l}$ denotes the latency (cost) of resolving a reference from node n_l that is locally mapped onto node n_i. Such pair-wise latencies can be experimentally obtained once and for all for a given architecture (see Section IX).

The affinity decisions are generated differently for statically defined and dynamically allocated regions of memory. Statically defined memory (*i.e.*, the `bss` segment) contains space for uninitialized global variables. The starting address and extent of the static region is determined at link time. The affinity decision module simply generates a per-node list of page address offsets that have affinity to that node. The first logical processor in a node is responsible for using these page offsets to issue the actual "first-touch" page placements during the final trace-guided program run.

A more sophisticated scheme is required for dynamically allocated regions. Here, the starting address of the allocated region can and does change over multiple runs of the same program. For the benchmarks evaluated, two distinct dynamic memory allocation patterns were observed. Many programs had a small number of calls, each of which allocated a large chunk of contiguous memory. For such cases, we adjust the affinity page offsets relative to the starting address of the region. The affinity offsets will be used to "touch" the pages on the appropriate nodes during the trace-guided run just after the region is allocated.

A second dynamic allocation pattern was observed for programs issuing a large number of calls clustered in time, each allocating a small region of memory (*e.g.*, NAS-2.3 MG). The resulting heap regions are mostly allocated contiguously in space. However, due to the lossiness of memory access traces, many small allocated regions ("silent regions") are not represented by even a single access record in the trace. By inspecting trace records for other small regions allocated close by, these silent regions are allocated in their vicinity (on the same page) since physical memory is allocated on *page* granularity.

V. Trace-guided Page Placement

Once page affinity decisions have been made, the original program is executed again in its entirety. The affinity information generated in the earlier phase then drives the page placement decisions. At the time of writing, the SGI's Linux version for the Altix, as currently installed by HPC centers, does not support dynamic page migration (even though there is forthcoming support promised for the next release). Instead, the existing "allocate-on-first-touch" policy is leveraged to effect page placement. This policy allocates physical memory for the virtual page on the node that first accesses ("touches") a data element on that specific node. A page is "touched" by executing a load followed by a store to an address in the target page from a processor on a particular node in order to force page placement on that node.

Care must be taken to touch a page on the respective node before any other processor accesses that page, or an unintentional placement may result. For static regions of memory, each processor reads its static affinity file on program startup and touches all the page address offsets listed in that file, as shown in Figure 1. All processors synchronize at a barrier after the touching phase to ensure that no processor accesses a statically-defined page before the affinity hint for the page has been applied. This scheme has minimal execution overhead since the static allocation is done only once, at startup.

A similar approach is employed for page placement of heap allocated regions. One difference here is that the page touch is delayed until the target memory region is allocated. In a legal program, no other processor can access the allocated region in any case before the allocation function (*e.g.*, *malloc*) has completed. This fact is exploited to ensure that the developed first-touch scheme will effect the intended page allocation before any other processor touches the memory region. The idea is to insert a *wrapper* around the allocation call. The behavior of the wrapper is controlled by an environment variable. During unoptimized runs of the program, the wrapper does no work. During the tracing phase, the wrapper records the allocation request parameters (size of region, starting address, thread id, timestamp). The affinity generation phase tags each allocation request with affinity hints. Finally, the wrapper uses affinity hints to effect page allocation as follows during the trace-guided runs.

Upon its invocation, the wrapper first calls the real allocation function. The dynamic affinity hints provide information about which parts of this dynamically allocated regions should be allocated on which target processors. This information consists of a list of processor identifiers and the address offsets that need to be touched on these processors. For each such processor, the current thread reschedules itself on the target processor by using the `sched_setaffinity` function call. When the `sched_setaffinity` call returns, the thread is now executing on the target processor. Then the touch mechanism "touches" the given list of address offsets, thereby causing page allocation on the physical memory in the target processor's node. The thread eventually re-schedules itself on its original processor after all affinity hints for all processors for the dynamic region are processed.

To the user-level program, this approach is nearly completely transparent with the exception of the wrapper function. Nonetheless, the execution overhead of the wrapper approach can be high. For every allocation request for which there are affinity hints for *n* processors, there are *n+1* context switches (one switch to every target processor, and the final switch back to the original processor). The experiments in Section VI reveal that this scheme has substantial execution overhead, which diminishes the gain due to reduction in remote accesses for several benchmarks. The overhead can be reduced by a less transparent scheme that involves more effort on part of the user. A simple way to reduce overhead would be to *group* the touching effort for multiple dynamically allocated regions. For each group, there would be only one context switch for each processor in the list of affinity hints. The user would also need to insert additional synchronization to ensure that no thread begins accessing dynamically allocated regions before the touching has occurred (to prevent inadvertent page allocation), an idea left for future work.

The approach for page placement employed for dynamically allocated regions still is subject to one caveat. While the benchmarks of the experimental sections always allocate memory only at the start of the program and do not free it until the end of execution, such behavior is not guaranteed. If programs repeatedly allocate and free memory in the stable execution phase, then the effectiveness of the "first-touch" scheme would be reduced. This occurs because portions of the virtual address space may be "recycled" by the allocation function after they were initially freed, but the *physical* memory will only be allocated once on the node where the page of virtual memory was first touched. This presents a limitation of using the first-touch mechanism. The issue is being resolved as the operating system (Linux) supports dynamic page *migration* in the latest kernels. [1] With migration support, the virtual pages in the dynamically allocated region could be simply *migrated* to the target processor given by the affinity hint at the additional cost of dynamic page migration overhead. This is another topic of future work.

[1]Draft APIs for manual page migration have been proposed and are being incorporated into future Linux releases for the Altix. They are already present in the latest experimental Linux kernels for the x86 architecture.

VI. Evaluation Framework

The previous section described a scheme for trace-guided page placement. In the following, a cost versus benefit analysis is presented as the configurable parameters shown in Figure 1 are varied. More specifically, two parameters are subject to change, (1) the choice of the *trace source* and (2) the *sampling interval* for capturing memory access samples. The hardware provides two trace sources, namely long-latency loads and DTLB misses. The sampling interval allows a trade-off of sampling overhead *vs.* the amount of trace data collected. For each trace source, different sampling interval values (Sections VII and VIII) are investigated.

By changing these parameters, the amount and type of trace information collected will change. This poses the questions of (1) how these traces affect the quality of the affinity hints and (2) how the affinity hints influence changes in overall wallclock execution time. To answer these questions, a comparison between the performance of these traces is needed with respect to the performance of affinity hints based on a *reference* trace. This reference trace is referred to as the "maximum information trace". A comparison with the reference trace helps answer the following question: What affinity hints will be generated given a complete memory access pattern of the program? How much improvement in performance can be achieved using these hints? By comparing results obtained from partial hardware traces against the results achieved with the maximum information trace, the tradeoff between trace collection cost and the optimization benefit can be clearly attributed.

The maximum information trace was originally obtained by a software memory tracing tool to capture all memory loads. However, this method had too much execution overhead for the benchmarks evaluated. Instead, the PMU was configured with the lowest latency threshold setting (4 cycles) and the highest sampling frequency (1). The resulting trace was utilized as the maximum information trace. Since the L1 cache hit access latency is 1 cycle, this corresponds to capturing a fraction of all accesses that miss in the L1 data cache. In the discussion below, the "reference results" refer to the affinity hints generated using the maximum information trace. Similarly, the "target trace results" denote the affinity hints generated by the evaluated trace.

The evaluation has three aspects: (1) the *tracing cost*, (2) the *quality* of the collected trace, and (3) the resulting *execution benefits*. The tracing cost is the cost of collecting the access trace, which is determined by the *size of the trace* and by the *execution overhead* inflicted on the benchmark during the trace collection phase. As the sampling interval is increased, both the trace size and the tracing overhead are expected to decrease.

The quality of the trace was evaluated by comparing the target trace results to the reference results using three different metrics. (1) **Coverage** denotes the fraction of the pages in the *reference results* for which an affinity hint exists in the target trace results. The affinity node values for the page do not need to be the same between the reference and target trace results. (2) **Accuracy** denotes the fraction of the pages in the *target trace results* that have the same affinity hint node value as the reference results. (3) The **Useful Fraction** metric combines the information from these two metrics. It measures the fraction of the affinity hints in the target trace that are not only present in the reference trace but also have the *same* affinity node value.

These metrics should be interpreted as follows. A low *coverage* value indicates that, for a large number of pages, not enough information is embedded in the target trace to generate affinity hints. A high coverage value increases confidence that the target trace contains affinity hints for almost the same number of pages as the reference trace (though the affinity node *values* might be different). In contrast, *accuracy* measures the stand-alone usefulness of the target trace. It answers the question: If the target trace were to be used to generate affinity hints, what fraction of the affinity hints are identical to those present in the reference results? A high accuracy value indicates that the target trace is as useful as the reference trace (though at a potentially much reduced overhead). A low accuracy value, in contrast, indicates that the target trace is potentially misleading in the sense that the affinity hints do not match the hints in the reference results.

The coverage, accuracy and useful fraction are computed as follows. Let

Ref = # hints in reference results;

Targ = # hints in target trace results;

C = # hints in target trace results that are also present in the reference results (though the affinity node values might not match);

A = # hints in target trace results that are also present in the reference results AND the affinity node values match. Then,

$Coverage = \frac{C}{Ref} * 100\%$

$Accuracy = \frac{A}{Targ} * 100\%$

$UsefulFraction = \frac{A}{Ref} * 100\%$

These three metrics each provide a different understanding of the trace characteristics. For example, a high accuracy value might still not indicate an *effective* trace if the coverage is low. This can be the case when a large number of pages lacks affinity hints (low coverage), yet the few hints generated are correct (high accuracy). Similarly, a low useful fraction value could be either due to low coverage or to low accuracy of the hints. Thus, there is a need for all three metrics.

The metrics discussed so far cover *cost* and *trace quality*. For assessing *trace benefit*, two metrics are utilized: (1) the net reduction in *remote accesses* and (2) the reduction in *wallclock execution time*. The net reduction is compared by taking the difference between the metric values (remote accesses, wallclock execution time) between the original unmodified program run and a run with the new trace-guided page placement scheme.

The evaluation process works as follows. First, the target program is run for one time step and trace data is collected. This trace data is used to compute the affinity hints and the entire program is re-run using this trace data. Thus, the *trace cost* is the cost to capture the samples over one timestep of the program. On the Altix/Itanium and Opteron platforms, there is no easy way to measure the number of remote memory accesses generated by the program. Instead, an *approximate* measure of the reduction in remote memory accesses is employed on the Altix platform. To this effect, the PMU latency threshold is set to 512 cycles,[2] and the number of accesses that exceeded this threshold for the original program is counted. The high latency threshold ensures that almost all loads that hit in cache or in local memory will be filtered out (though some remote loads may also be filtered out, as indicated by the latencies in Table I). Then, the program is run with the affinity-enhanced page placement scheme, and the number of accesses exceeding the latency threshold (512) is counted as before. The difference between the two values provides an approximate measure of the net reduction in remote memory accesses. In practice, this value was found to be quite consistent across multiple runs.

[2]Due to PMU limitations, the latency threshold can only be set in powers of 2. The next lower threshold (256 cycles) would not filter out a significant fraction of local memory loads, as indicated by the latencies in Table I.

In the experimental evaluation, the wallclock time for the complete run of the original program is compared to the wallclock time of the program with trace-guided page placement including the overhead of the page touching mechanism. During the experiments, it was observed that the execution time of the program varied measurably across runs. This may be due to several reasons. First, the difference in scheduler allocation of processors for the batch runs affects the degree of benefit obtained with trace-guided page placement, *e.g.*, the benefit will be less if the allocated processors are closer. Second, all operating system calls on the Altix must go through a small collection of CPUs in the interactive login partition. Thus, the load on the interactive nodes affects the performance of the jobs running on the batch nodes. This is especially significant for the dynamic page touching mechanism, which potentially involves multiple context switches for a single affinity hint.

To compensate for this variability in execution time, each benchmark was executed six times, except for only five executions of BT. Each time, the trace-guided runs and the non-trace guided runs were executed on the same scheduler-assigned processor allocation. The wallclock execution time graphs show the average benefit obtained with each sampling interval. The error bars denote the confidence interval range for a 95% confidence interval.

Benchmarks: Nine OpenMP benchmarks were experimentally evaluated. This includes seven out of the eight NAS-2.3 benchmarks (excluding EP). The NAS benchmarks are C versions of the original NAS-2.3 serial benchmarks [4] provided by the Omni Compiler group [5]. EP is not evaluated since it does not have significant sharing of data [6]. In addition, the 320.equake and 332.ammp benchmarks from the SPEC OMPM2001 benchmark set were assessed in the results. These benchmarks have significant dynamic memory allocation, thereby putting the dynamic touching mechanism to the test.

All programs were compiled at the -O2 optimization level. All NAS benchmarks use Class C data sets while the SPEC benchmarks use the reference data set. All experiments were carried out on a non-interactive (batch) allocation of eight processors. On the Altix platform, two processors always share one node. A total of four nodes were used. All programs were run with eight OpenMP threads. Each thread is bound to a separate processor using the `sched_setaffinity` primitive. OpenMP thread scheduling was set to static. This hardware platform has Itanium-2 processors running at 1.5GHz, each with a 6 MB L3 cache, 256 KB L2 cache and 16 KB L1D cache.

For each program, markers were inserted to delineate the start and end of the timestep. For 332.ammp, the pre-existing round-robin allocation of the "atom" element was disabled for the trace-related runs. However, the benefit metrics (wallclock time, number of remote accesses) are still compared against the original program. For the IS benchmark, a one-time dynamic allocation for the `prv_buff1` array is issued since the program failed to execute with the default stack allocation for this variable. Out of the 9 benchmarks, 4 benchmarks — MG, 332.ammp, 320.equake,

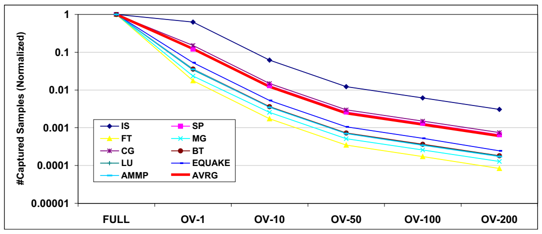

Fig. 3. Cost: Number of Captured Samples

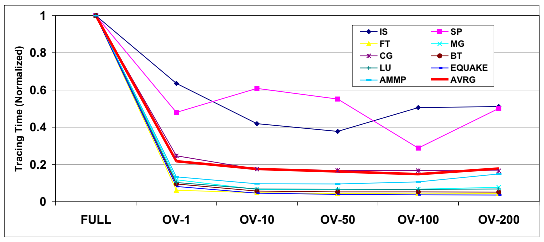

Fig. 4. Cost: Trace Time

IS — utilize dynamic memory allocation. The remaining benchmarks operate with statically declared global arrays.

VII. Evaluation with Long-latency Load Tracing

The performance of the developed page-placement scheme was assessed for long-latency loads as the trace source using the cost, quality and benefit approach that was described in the last section. For these experiments, the latency threshold in the PMU was set to 128 cycles. This filters out most of the load accesses that hit in the L1D, L2 and L3 caches. The sampling intervals were selected as 1, 10, 50, 100, 200 (OV-1 to OV-200 in the graphs). Pages were allocated to nodes using the uniform latency policy (Eq. 1).

Tracing Cost: The graph for cost comparison shows the cost for the “maximum information trace” (denoted as `FULL` in the graphs) and the results for each of the reduced sampling intervals. The reduced sampling results are normalized to the FULL trace values.

Number of Captured Samples: The number of accesses captured at OV-1, depicted in Figure 3, is about an order of magnitude lower than the FULL trace for most benchmarks (except IS). By keeping the latency threshold much higher (128 cycles instead of 4 cycles for the FULL trace), most of the loads that hit in cache are filtered out. These loads can be ignored since they do not propagate past the cache to memory. Hence, they will not be affected by page placement. With increasing sampling intervals, the total number of samples captured decreases linearly. At OV-200, the average number of accesses in the trace has been reduced by 1000 times over the FULL trace.

Tracing Execution Overhead: The absolute execution overhead for tracing is extremely low since it is sufficient to execute only a single timestep for a benchmark, *i.e.*, partial execution significantly reduces overhead over an execution of the entire benchmark without any loss in accuracy for the benchmarks studied. On average, over all benchmarks, the execution overhead for tracing a single timestep at OV-1 was 2.7% of the overall original program execution time.

The *relative* tracing execution overhead (compared to `FULL`) is shown in Figure 4. The overhead flattens out with increasing sampling intervals. This indicates that the trace collection cost does not dominate the time to execute the timestep. The results show that OV-1 or OV-10 are the “sweet spot” values for the sampling interval, since increasing sampling intervals beyond that point does not reduce overhead by much. On average, tracing execution overhead at OV-1 is about 20% of the FULL trace cost, with the exceptions of `SP` and `IS` that have lower savings.

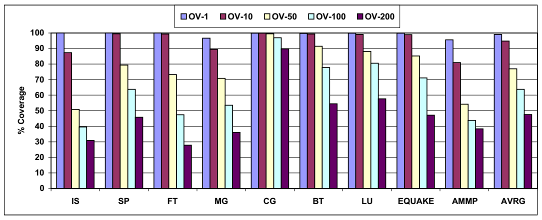

Fig. 5. Quality: Coverage

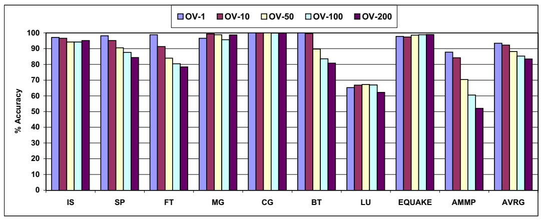

Fig. 6. Quality: Accuracy

Trace Quality: As the sampling intervals increase, the size of the trace collected will tend to decrease. This has an effect on the quality of the trace, *i.e.*, the coverage, accuracy and useful fraction metrics. The maximum values of all these metrics is 100%.

Coverage: Figure 5 depicts the coverage results for different sampling intervals. At OV-1, the average coverage is 99% indicating that affinity hints exist for almost all of the FULL trace pages. The OV-10 coverage still remains high at 94%. After that, a noticeable decline in coverage at sampling intervals of OV-50 and beyond is observed. The average coverage falls from 94% at OV-10 to 76% at OV-50 and finally to 47% at OV-200. Thus, at OV-50 and higher, the trace data is insufficient to generate affinity hints for page placement for a significant number of pages.

Accuracy: The accuracy values, depicted in Figure 6, are very close across sampling intervals for each benchmark. Also, accuracy remains uniformly high across increasing sampling intervals for all benchmarks (except for LU). This is very encouraging as it indicates that even with a reduced number of accesses, the affinity node recommendations match the recommendations given by the FULL trace for most of the affinity hints generated. LU’s behavior is explored in more detail later.

Useful Fraction The useful fraction is the fraction of the FULL trace affinity hints that are present and have the same affinity node value in the target trace results. A high useful fraction indicates that almost the same results were obtained as the FULL trace results, albeit with much smaller trace input data.

The average useful fraction, depicted in Figure 7, is high for OV-1 (93%) and OV-10 (87%). From OV-50 to OV-200, the metric degrades from 68% to 40% on average. This trend occurs because the *coverage* values fall with increasing sampling intervals while the accuracy remains steady. The degradation is much more pronounced for benchmarks like `IS`, `FT` and `MG` whereas there is almost no degradation for `CG`.

Trace Benefits: The impact of the page placement scheme is explored for two metrics: (1) the number of remote accesses generated by the program and (2) the wallclock execution time of the program.

Reduction in Remote Accesses: Figure 8 shows the net reduction in the number of remote accesses for the full-program run using automatic trace-guided page placement *vs.* the original program. The figure compares the reduction in remote accesses

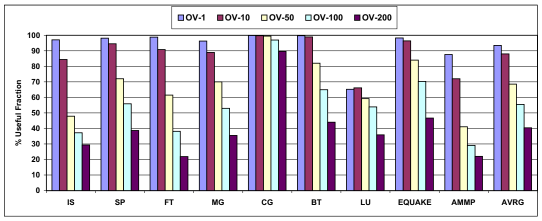

Fig. 7. Quality: Useful Fraction

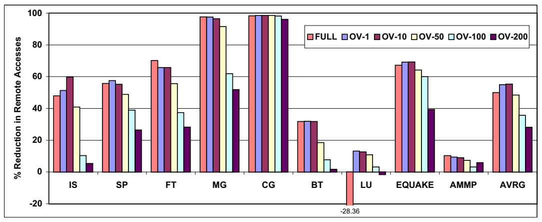

Fig. 8. Benefit: Reduction in Remote Accesses over Original Program

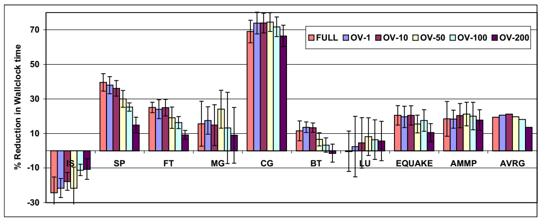

Fig. 9. Benefit: Time Savings over Original Program & Arithmetic Mean

using the FULL trace *vs.* the reduction achieved at latency threshold 128. It also depicts the different sampling intervals.

For all but one case (LU:FULL trace), there is a net reduction in the number of remote accesses. Almost all the remote accesses for CG and MG are eliminated as shown by a 98% and 97% reduction at OV-1 for CG and MG, respectively. Other benchmarks also have a significant reduction in remote accesses. The average reduction at OV-1 is 60% and decreases significantly from OV-50 (48%) to OV-200 (28%). OV-10 appears to be the sweet spot. The average reduction is, in fact, slightly higher for OV-10 (55%) than OV-1 (54%). Only LU shows a 28% *increase* in remote accesses when using the full trace. This anomaly of LU is discussed in more detail later.

Reduction in Wallclock Execution Time: This is the most important measure to assess the overall benefit as it indicates the performance improvement of an application with trace-guided placement compared to the original unmodified program. Figure 9 shows the improvement in wallclock time. As described before, the error bars represent the 95% confidence interval range. The ranges for MG, LU and IS are large indicating that these programs have more variable execution times.

Except for IS, every other benchmark shows a reduction in wallclock execution time. The average reduction is 21% at OV-1. CG achieves exceptionally large savings with over 73% shorter executions at OV-1. Many other benchmarks (SP, FT, MG, Equake) also achieve greater than 15% reductions.

With increasing sampling interval, the wallclock improvements tend to decrease though the magnitude of decrease is program-dependent. CG does not show much degradation with increasing sampling intervals, but there is a noticeable degradation with SP between OV-10 and OV-200.

IS represents an exceptional case where the wallclock execution time *increases* with trace-guided page placement. The cause of the degradation is the cost of the page-touching mechanism for dynamically allocated regions. Each hint on a dynamically allocated region potentially represents at least two context switches: one to switch to the target processor and “touch” the page and the other to switch back to the processor that originally requested the allocation. (Note that each OpenMP thread is bound to a different processor. Hence, the discussion refers to processors instead of threads here.) With increasing sampling intervals, fewer dynamic hints are generated (as coverage falls). This reduces the overall overhead on the target. Thus, less degradation in wallclock execution time is observed for IS with increasing sampling intervals.

Similar to IS, the potential wallclock savings for other programs with dynamic memory allocation (MG, Equake, AMMP) are also affected by the overhead of the touching mechanism. Given that over 98% of the remote accesses for MG are eliminated by page placement, the wallclock reductions for MG would increase even further with a more optimized touch mechanism.

The LU Anomaly: LU represents an anomalous case. For this benchmark, the affinity hints generated by the full trace do not match the affinity hints generated by the other traces (OV-1 to OV-200). This causes low accuracy and useful fraction values, as seen in Figures 6 and 7. Furthermore, using the full trace leads to an *increase* in the number of remote accesses (Figure 8) while OV-50 leads to a 10% decrease in remote accesses. The corresponding wallclock time reduction is *higher* for OV-50 (8% improvement) than that of the full-trace results (0% improvement).

The underlying cause is as follows. The affinity node values differ between the full trace and the OV-1 trace (and higher sampling interval traces) for parts of the large `rsd` global static array. The full trace uses the lowest possible latency of 4 cycles to sample the address trace. This captures all possible loads, irrespective of whether the loads hit in cache or not. For the pages of the `rsd` array that have different affinity hints in the full and OV-1 traces, most of the accesses on the affinity node given in the full trace are hits in the local caches. Hence, the affinity decision is different from the OV-1 trace-based decision (which filters out the cache hits). First, loads that hit in cache will not be affected by page placement decisions. Second, the full-trace based page placement, in fact, *worsens* the average access latency for cache misses since the corresponding pages are allocated on a node that only has infrequent cache misses for those pages. This explains the *increase* in the average number of remote accesses for the full-trace results compared to the OV-1 based experiment. Thus, the average wallclock time improvement is lower for full-trace than for OV-50 in this case.

Conclusions: Long-Latency Tracing: 1) Overall, the size of the trace data at OV-1 is one-tenth the size of the FULL trace on average. With increasing sampling intervals (OV-1 to OV-200), the trace size decreases linearly. 2) For most benchmarks, the execution overhead of trace collection decreases sharply from FULL to OV-1, yet it does not decrease significantly with larger sampling intervals (OV-10 to OV-200). Thus OV-1 or OV-10 appears to be the *sweet spot* for trace collection. 3) With increasing sampling intervals, the coverage drops significantly, which indicates insufficient trace information to generate affinity decisions for many pages. 4) Nevertheless, the *accuracy* of the trace information does not degrade significantly with increasing sampling intervals. 5) A significant reduction in the wallclock execution time and the number of remote accesses is possible with trace-guided page placement. However, for programs with dynamic allocation, the page touching mechanism is expensive

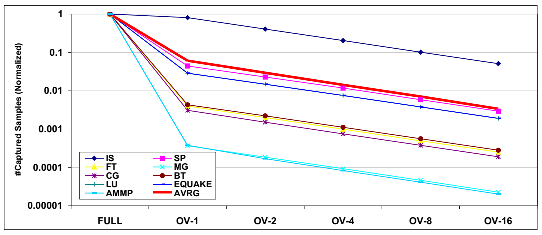

Fig. 10. Cost: Number of Captured Samples

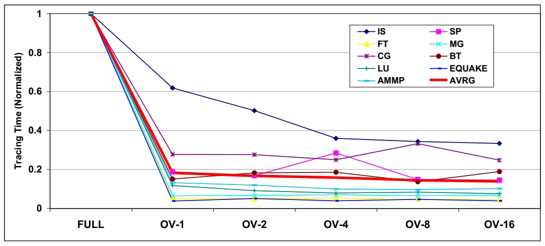

Fig. 11. Cost: Trace Time

and adversely affects wallclock execution time. A more optimized touching scheme should lead to even better wallclock reductions for these programs. 6) For one benchmark (LU), using the reference trace (`FULL`) actually resulted in a *degradation* of performance. For this benchmark, the *filtering effect* of the high latency threshold used by the target traces (128 cycles) removed loads that hit in the cache and resulted in a more accurate picture of which pages are frequently accessed by which processors. Thus, using the full memory access trace may actually result in sub-optimal page placement in rare cases. For all other benchmarks, the reference trace almost always had the maximum (or close to maximum) performance benefits, *i.e.*, reduction in remote accesses and wallclock time.

VIII. Evaluation with Data TLB Misses Tracing

Next, the results based on data TLB misses as the trace source obtained with PMU support are presented. Experiments are conducted for sampling intervals values of 1, 2, 4, 8 and 16 (denoted OV-1 to OV-16 in the graphs). Pages were allocated to nodes using the uniform latency policy (Eq. 1). For the discussion below, the results presented in the last section using long-latency loads are refereed to as the trace source as the *load-based results*. In the following, the DTLB miss results are described and contrasted with the load-based results.

Trace Cost: As before, the cost metrics are compared against the cost incurred for the "maximum information trace" (denoted as `FULL` in the graphs).

Number of Captured Samples: The average number of samples captured at OV-1 is less than one-tenth of the number of samples in the full trace, as seen in Figure 10. With increasing sampling intervals (OV-1 to OV-16), the number of captured samples decreases almost linearly. In contrast to the load-based results, the difference between FULL and OV-1 tends to be program-dependent. Ammp and MG have more than 1000 times less trace data at OV-1 compared to FULL while IS has almost the same number of samples as FULL.

Trace Execution Overhead: The results for the relative trace overhead (Figure 11) are similar to load-based results. The average execution overhead for trace collection at OV-1 is 18% of the FULL trace's cost. With increasing sampling intervals (OV-1 to OV-16), the execution overhead is not significantly reduced.

Trace Quality: As before, the three quality metrics of coverage, accuracy and useful fraction are evaluated as shown in Figures 12, 13 and 14, respectively.

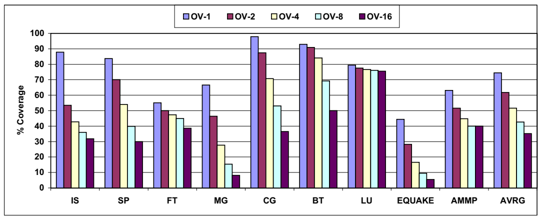

Fig. 12. Quality: Coverage

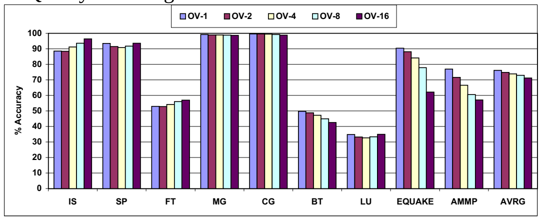

Fig. 13. Quality: Accuracy

Coverage: The average coverage at OV-1 (74%) is sharply lower than the average coverage at OV-1 in the load-based results (99%), as depicted in Figure 12. This is due to significantly lower coverage values for FT, MG, LU, Equake and Ammp, as compared to the load-based results. With increasing sampling intervals, the coverage begins to degrade significantly, except for LU. Coverage falls from 74% at OV-1 to 35% at OV-16.

The low coverage values indicate that the information to generate page affinity hints is insufficient for a significant number of pages. The problem is more acute for the DTLB case than for the load-based results, as indicated by the lower coverage values. Low coverage lessens the effectiveness of the page-placement scheme resulting in a reduced potential for performance benefits.

Accuracy: The results in Figure 13 indicate that accuracy is benchmark-dependent. For most benchmarks (except Equake and Ammp), the accuracy values for increasing sampling intervals are similar. This indicates that accuracy is less sensitive to reduction in the size of the trace. Compared to the load-based results, a significantly lower accuracy is observed for FT, BT, LU and AMMP. This indicates that page-affinity decisions based on DTLB misses do not agree with affinity decisions based on the FULL trace or long-latency load-based results.

Useful Fraction: Due to the lower coverage (FT, MG, LU, Equake, Ammp) and lower accuracy (FT, BT, LU), the useful fraction values are also significantly lower than for the load-based results. The average value at OV-1 is 58% compared to 93% at OV-1 with long-latency loads as the trace source. With increasing sampling intervals, the useful fraction value tends to fall significantly for most benchmarks. The average useful fraction degrades from 58% at OV-1 to 22% at OV-16.

Trace Benefits: The coverage, accuracy and useful fraction for DTLB-based results were observed to be significantly lower than their load-based counterparts for most benchmarks. This will impact the performance benefits obtainable with trace-guided page placement. Figures 15 and 16 show the reductions in remote accesses and overall wallclock execution time, respectively.

Reduction in Remote Accesses: As before, the reduction in remote accesses using traces obtained at different sampling intervals is compared to the reduction obtained with results based on the full trace (marked FULL) seen in Figure 15. Two benchmarks, BT and LU, experience an *increase* in remote accesses with DTLB-guided page placement. The increases are significant (more than 30%) and occur with all sampling intervals. In comparison to

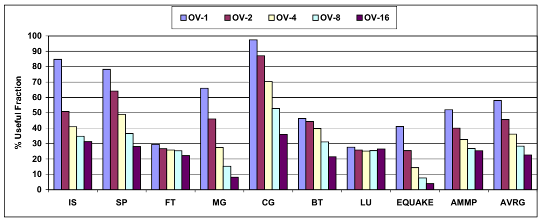

Fig. 14. Quality: Useful Fraction

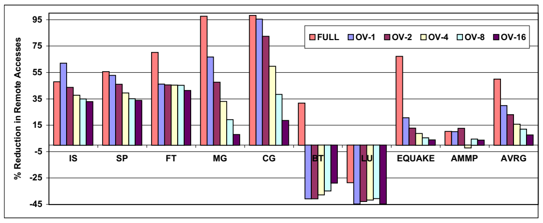

Fig. 15. Benefit: Reduction in Remote Accesses over Original Program

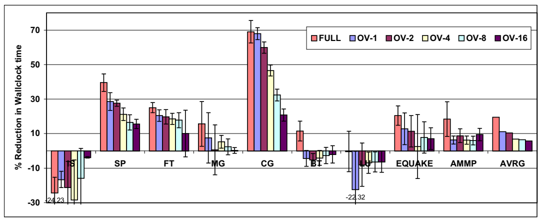

Fig. 16. Benefit: Time Savings over Original Program and Arithmetic Mean

the load-based results, the reduction in remote accesses is much lower for many benchmarks, especially for MG (98% *vs.* 67%) and Equake (69% *vs.* 20%). The average reduction of remote accesses is 29% at OV-1, which is much lower than the 54% average reduction at OV-1 for the load-based results.

Reduction in Wallclock Execution Time: As with remote accesses, the DTLB miss-based scheme generally performs worse than the long-latency load-based mechanism. The average wall-clock reduction at OV-1 is 11% for DTLB misses (see Figure 16) *vs.* 20.6% for the load-based results. IS, LU and BT show an increase in execution time with DTLB-guided feedback. CG has the maximum improvement (67%), while improvements reduce sharply for MG (17% *vs.* 7%) and Ammp (18% *vs.* 6%) compared to load-based results at OV-1.

Conclusions for DTLB-Tracing: 1) Overall, the cost of trace collection is similar for both DTLB misses and long-latency load-based schemes. 2) The coverage and accuracy for DTLB-based results are significantly lower compared to the long-latency load-based results. 3) Due to sharply lower coverage and accuracy, the useful fraction values are also low. This indicates that DTLB-based affinity decisions are not representative of decisions that would be made with the full trace. 4) The performance benefits (reduction in remote accesses and wallclock time) are also much lower for DTLB-based results. 5) The trace costs for both DTLB misses and long-latency loads are similar, but the quality of the trace and the resulting performance benefits are larger for long-latency load-based traces compared to DTLB miss-based traces.

In conclusion, DTLB misses are not a good candidate to decide page placement, which can be explained as follows. For the considered benchmarks, DTLB misses do not correlate well with the relative volume of loads from a processor to a particular memory page. This could occur, *e.g.*, if the program has few DTLB misses but a large number of cache misses going to memory. Then, the information about the *frequency* of accesses to each page is lost if only the DTLB misses are considered (since repeated accesses to the same page will tend to hit in the DTLB). Another possible scenario is a large number of DTLB misses with few cache misses going to memory. In this case also, the DTLB trace will not be representative of the relative distribution of load requests to a page from each processor. In general, TLB misses are not indicative for the number of remote reference issued per memory page. They only indicate that the page is referenced for the first time (cold miss in TLB) or that the number of references in the working set exceeds the number of TLB entries (for capacity misses). A sufficiently accurate cost/benefit model for page placement cannot be derived from such coarse-granular information. Lossy reference traces or long-latency accesses are a much better indicator, as demonstrated in Section VII.

IX. Page Placement Exploiting Hardware PEBS Traces

In the preceding sections, the developed approach was evaluated using Itanium2-specific hardware traces for automated page placement. In this section, a similar approach shall be evaluated using a completely different hardware tracing mechanism and NUMA platform. The basic idea of exploiting *processor-centric* hardware support for user-level page placement is shown to be portable and widely applicable across multiple platforms (irrespective of interconnect topologies).

Instead of the Itanium architecture, the target platform is the widely used x86 platform. The objective of this work is to perform page placement on a ccNUMA multiprocessor Opteron system available from AMD [7]. In this system, each processor directly accesses (using an on-chip memory controller) a fixed amount of local physical memory. Communication with other processors and their attached physical memories is achieved over the point-to-point *HyperTransport* connection network [7]. Systems exploit a bus-based MOESI coherence protocol instead of the directory-based coherence present in SGI's NUMALink fabric. Processors can access their local memories faster than memories attached to other processors, and the access penalty increases with the number of hops to reach the remote memory (due the point-to-point interconnect). The experiments will assess the benefits of intelligent page placement on this system. Though the results were obtained on the AMD Opteron, the developed scheme should equally work on future ccNUMA systems from Intel that use the CSI (Common System Interconnect) / QuickPath architecture.

The premise of the developed technique is the ability to obtain hardware-generated traces that efficiently drive the page placement policy. Since current AMD Opterons, prior to the quad-core Barcelona chip, have no published hardware trace capability, the performance monitoring unit built into Intel Pentium4/Xeon/ systems is exploited for this purpose. This hardware is called "Precise Event-Based Sampling" (PEBS). PEBS captures the register state when a specific event, *e.g.*, an L1 cache miss, is detected. By decoding the instruction format and using this register state, the memory address that was accessed can be reconstructed. Details of PEBS are described later on.

This work makes the following contributions:

- The capabilities of PEBS tracing for extracting L1 and L2 load miss traces is assessed. The PEBS mechanism is first evaluated with a micro-benchmark, and the degree of lossiness is characterized.

- A novel cross-platform page placement scheme is demonstrated that uses these traces obtained on the Intel platform for automated page placement on Opteron systems.
- The hop-sensitive page affinity policy from Section III-C that takes into account the *multiple* NUMA penalties is discussed in terms of its implementation and evaluation on Opteron systems that incur distance-dependent latencies.
- The wallclock performance benefit of the developed approach is assessed on Opteron systems by comparing it to the original program runtime and the runtime achieved with an alternative approach that uses the newly available *numactl* library [8] for round-robin page placement.

A. Precise Event-Based Sampling (PEBS)

PEBS is a performance monitoring feature available in Intel Pentium4/Xeon/ processors. It works as follows. The processor can be configured to monitor certain instructions [3] as they flow through the pipeline. If a certain event (*e.g.*, a cache miss) occurs, the causal instruction is *tagged*. When a tagged instruction reaches the head of the retirement queue, PEBS captures the state of the registers immediately before (Xeon) the tagged instruction is retired. This information (register state) is automatically written to a previously set up buffer in virtual memory. When the number of records in this buffer reaches a configurable threshold, an interrupt is triggered. The interrupt service routine saves the content of the buffer to stable storage before tracing is resumed. A sampling mechanism is available to avoid capturing the register state for all tagged instructions. PEBS is described in more detail elsewhere [9], [10].

The saved register state also contains the value of the IP (instruction pointer) at the monitored (cache miss) event. By inspecting the content of the event-triggering instruction, the *format* of the instruction can be deciphered and used for calculating the virtual memory address generated (for load/store instructions). In this work, PEBS is utilized to capture L1 and L2 cache load misses. PEBS allows *precise* knowledge of the exact instruction that caused the cache miss and the memory location that was accessed, similar to the hardware support for Itanium2 that was discussed earlier.

The capabilities of PEBS are evaluated with a microbenchmark to assess the degree of lossiness. The microbenchmark strides over a large array with a 12KB stride in order to defeat the hardware stride prefetcher of the Pentium architecture. The *perfmon2* framework is utilized to access the hardware counters and to collect the PEBS-generated trace [2].

Based on the data size, access pattern and cache parameters, the program is estimated to contain approximately 6 million L1 and L2 load misses. On a different x86 machine, hardware counters reported 6.71 L1 and 6.72 million L2 load misses. Due to the software constraints, these values could not be assessed on the primary machine with PEBS support. For this experiment, the primary machine had an L2 cache of 2MB while the hardware counters were measured on a machine with 1MB L2 cache. However, the measurements on the two machines should be similar because of the inherent structure of the memory accesses, namely widely spaced strides to memory lines that are seldom re-accessed over an area of memory 80MB in size.

[3]Complex instructions are broken down into μops on the x86 architecture so that we are actually referring to μops here (technically speaking).

Figure 17 shows the number of samples collected for L1 and L2 load misses with increasing sampling intervals averaged over 10 runs with a standard deviation of less than 1%. The following

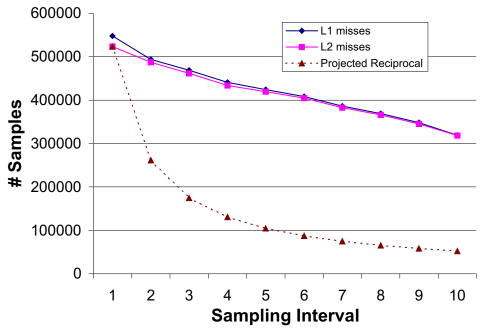

Fig. 17. Evaluation of load miss tracing by PEBS: Intel Xeon

observations were made:

- Both L1 and L2 traces are quite lossy. At the smallest sampling interval (1), less than 10% of the expected L1 or L2 misses are collected.
- The L1 and L2 curves are very close. This is expected because each L1 miss is almost always an L2 miss in the microbenchmark.
- The number of samples does not decrease in linear proportion to the increase in the sampling interval. This is in contrast to an expected decrease in the number of samples depicted by the "Projected Reciprocal" curve.

B. Hop-Sensitive Page Placement

Instead of the uniform latency page placement policy evaluated so far, this section focuses on the implementation of the hop-sensitive page placement policy (Eq. 2). As briefly mentioned, remote access penalties are not uniform but vary with the distance to the target node. To measure the load access latency, the *bplat* microbenchmark [11] is utilized with threads and memory bound to different nodes. The measurements were performed on a four-socket Opteron system with one processor core per node.

Table II shows the reported latencies. The values are the average of 10 runs, and the standard deviation was less than 5%. As can be expected, access to node-local memory is always cheaper. But notice that accesses to non-local nodes take differing amounts of time. *E.g.*, consider the access latencies for CPU on node 1. Normalizing to local node access on the 4-node system, it takes about 30% more time to access memory on nodes 0 and 2, but it takes 60% longer to access memory on node 3. The

TABLE II
LATENCIES 4-NODE OPTERONS [NANOSECS]

CPU on Node	Memory on Node			
	0	1	2	3
0	102	138	**172**	140
1	143	107	141	**172**
2	**179**	141	102	141
3	141	**175**	142	108

Hypertransport interconnect is laid out in a ring (4-node square) topology for a maximum of two hops.

In the hop-sensitive page placement policy, the latencies shown in Table II provide the weights $w_{i,j}$ for placement on node n_i and reference from node n_j (see Eq. 2). To implement the cost-based selection of page placement, a histogram of accesses from every node is again constructed for each page. Consider each node as the candidate affinity node. The values in the latency table and the histogram values are used to compute the *weighted* score that represents the cost of allocating the page on that node. The candidate node with the *lowest* cost wins, and the page is assigned to that node. This approach is portable because it is *observation-based*, *i.e.*, it uses only the *measured* latencies between different nodes without requiring knowledge of the exact architecture and interconnect topology.

C. Evaluation

The same benchmarks as in the previous experiments were used for evaluation, except for 332.ammp_m. Due to memory resource limitations, all benchmarks except IS and LU used the smaller Class B data set instead of the Class C set used before. PEBS-based L1 and L2 load misses were obtained for each benchmark for a truncated program run on a Xeon machine with a sampling interval of ten. (For these experiments, the truncated programs ran longer than the earlier Itanium2-based experiments to allow collection of more trace data.) The traces were processed as described earlier, and affinity hints were generated using the hop-sensitive affinity decision mechanism.

For each program, the wallclock time was measured with trace-guided page placement and compared to the original program's runtime on the 4-node Opteron system. The system was shared but only lightly loaded. Furthermore, a *round-robin interleaving* of the memory pages across the nodes was evaluated, which was obtained through the *numactl* library interface [8].

Figure 18 shows the improvement in wallclock time compared to the original program. The values are an average of 8 runs, and the positive and negative error bars represent one standard deviation each. "L1" and "L2" represent trace-guided page placement with L1 and L2 cache miss traces, respectively. The following observations can be made. The developed trace-guided schemes perform well for 5 benchmarks (SP, FT, MG, CG, BT). Wallclock improvements for the L2 miss trace range from -7% to 30% with an average improvement of 12.2%. Wallclock improvements for the L1 miss trace range are similar, except for LU where a performance loss of 21% is observed. Intuitively, the L2 miss trace filters out loads that hit in L2. Therefore, L2 misses are a better indicator of the true distribution of load requests to a page in memory compared to the L1 miss trace.

The performance improvements obtained with memory interleaving (depicted in Figure 18) indicated that simple round-robin interleaving works almost as well as trace-guided page placement on the small-scale ccNUMA system subject to experimentation,

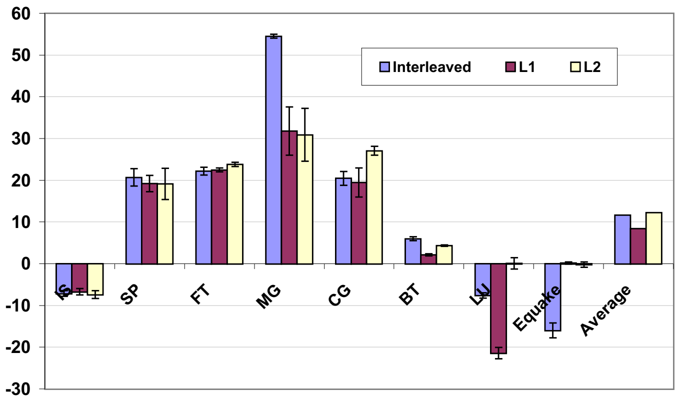

Fig. 18. Time Savings over Original Program and Arithmetic Mean

but this depends in large on the algorithmic properties of the target applications. With MG, the program runtime is very short ($<$ 30 seconds for original program). Apparently, the developed trace-guided scheme is unable to recoup the overhead of forcing page placement within this short time so that interleaving happens to provide a larger relative improvement. On the other hand, interleaving performs badly for equake while the trace-guided scheme shows no net impact. Here, the benefits of page placement seem to be balanced by the overhead of the page touching mechanism. Overall, long latency misses (L2 in this case) provided a uniformly reliable indicator for page placement decisions while interleaved memory allocation occasionally resulted in a significant performance penalty.

Will interleaving still prove as competitive with larger scale ccNUMA systems, where the remote access penalties will be larger? Will it work with benchmarks with larger data sets (instead of class B used in the experiments here)? These questions remain open as input sizes were constrained by the amount of available memory of the test platform.

X. Related Work

Tikir and Hollingsworth describe a dynamic user-level page migration scheme based on an approximate trace of memory accesses obtained by sampling the network interconnect [12]. This is the closest related work. The trace is used for deciding page affinity. Pages are dynamically migrated using the `madvise` system call. In contrast, we focus on trace-guided page placement leveraging the simpler "first-touch" page allocation policy of the operating system. In the future, our approach can be refined to eliminate the need for a separate tracing run by *migrating* pages. Linux recently added support for dynamic page migration user control [13] building on prior NUMA capabilities and scaling considerations [14], [15], [16].

Our method uses a different trace source (long-latency loads or DTLB misses) with varying sampling intervals. Our method is simpler in that it is *processor-centric*. More specifically, we not require special network instrumentation support, we only rely on the ability of the PMU to *time* load accesses. Because their approach is *network-centric*, *i.e.*, the hardware counters are embedded in the network interconnect and do not distinguish between different processes, only one application can use them at a time. In contrast, there is no such restriction with our approach. In addition, our mechanism is interrupt-driven, *i.e.*, the PMU raises an interrupt only when the sampling counter overflows and generates virtual addresses directly. In contrast, their method must *poll* the network interconnect counters to collect a trace of physical addresses, which must subsequently be mapped to virtual addresses using a separate system call.

Finally, our page hints are *abstracted*, *i.e.*, they are relative to the starting address of the region (static or dynamic). Touching is deferred until the region is actually allocated. Thus, the affinity hints are potentially *portable across platforms* in that hints generated on one platform can be used on another if it supports first-touch page placement. We intend to explore this potential in future work.

Nikolopoulos *et al.* describe a user-level dynamic page migration scheme that uses per-page hardware reference counters that capture the frequency of accesses from each node to a particular page [17], [18]. The method depends on the compiler for identifying the pages of virtual memory using whole program analysis. In contrast to our method, they do not handle dynamic memory allocation. In addition, we do not require any compiler or operating system support, and our page-placement mechanism is completely transparent to the target program (*i.e.*, no explicit calls are necessary for page placement).

Chandra *et al.* evaluated a page migration policy based on TLB misses on the Stanford DASH machine [19]. Later, Verghese *et al.* described a simulation-based kernel-level implementation of dynamic page migration [20]. They considered both the number of load-misses to a page and the number of data TLB misses as trace sources. In our work, we found data TLB misses to be less effective for deciding the best page placement, which is in contrast to Chandra's results but confirms results presented in the latter work by Verghese. Yet, our work is neither simulation nor kernel based, it is implemented on contemporary hardware within user space. It focuses on page placement rather than migration and shows that partial, lossy traces are sufficient for page placement.

Other approaches to kernel-level dynamic page migration and replication are discussed in Noordergraaf *et al.* [21] and in Bolosky *et al.* [22]. Bolosky's approach is based on a count-down register in the TLB triggering a trap after k remote reference have been issued. This scheme is compared with others triggered by TLB-misses with freezing of placed pages and optional unfreezing. In other work, Bolosky and Scott provided optimal page placement policies with and without replication based on dynamic programming and derived from complete traces, which were obtained offline through single stepping at the kernel level at a slow-down of over 200x [23]. These policies were subsequently compared to contemporary kernel-based policies. In contrast, we operate completely in user-space, do not propose any hardware extensions, focus on partial traces with a small cost (2.7% of the overall execution time) and leverage the simpler first-touch page allocation policy to steer page placement at region initialization.

Bull and Johnson study the tradeoffs between page migration, replication and data distribution for OpenMP applications on the Sun WildFire system [24]. In their study, they find that page replication performs better than page migration and static data distribution.

Lastly, the hardware mechanism for capturing long-latency loads and DTLB misses is described in the Itanium-2 manual [3]. In previous work, we used this facility in conjunction with software rewriting to efficiently obtain a lossy load/store trace and exploit its information to analyze the coherence behavior of OpenMP programs [6]. The Intel Pentium4/Xeon/Core platforms support the PEBS mechanism that we also used in this paper. PEBS allows capture of the architecture register state when a tagged event (*e.g.*, a cache miss) is detected. The register state is saved to a reserved portion of physical memory without software intervention. PEBS is described in more detail in [9], [10].

XI. Conclusion

This paper introduces a novel hardware-assisted page placement scheme based on lossy tracing. The placement scheme allocates pages near processors that most frequently access that page. The scheme leverages performance monitoring capabilities of contemporary microprocessors to efficiently extract an approximate trace of memory accesses. This information is used to decide page affinity, *i.e.*, the node to which the page is bound. The method is low cost as a lossy trace is obtained just for the stable execution phase, not for the entire program run. The approach operates entirely in user space, is widely automated, and handles not only static but also dynamic memory allocation.

The approach is evaluated with a set of multi-threaded scientific benchmarks from the NAS and SPEC OpenMP suites. Two different hardware trace sources are investigated with respect to the cost (*e.g.*, time to trace, number of records per trace) *vs.* the accuracy of the trace and the corresponding savings in wall-clock execution time. It is shown that just a small subset of traced long-latency loads provides a better indicator for page placement than TLB misses. More specifically, the approach can efficiently improve page placement leading to an average wall-clock execution time saving of more than 20% (SGI Altix) and 12% (AMD Opteron) for the benchmark set with a one-time tracing overhead of 2.7% over the overall original program wallclock time.

In addition, the framework is extended to a different processor and ccNUMA architecture. In this new framework, the traces were obtained on a Xeon processor using the PEBS hardware mechanism available in Intel's Pentium4/Xeon/Core processors. The traces were used as before for page placement decisions, and the page affinity hints were applied to programs running on an Opteron system. This effort targeted three goals. First, the processor-centric instrumentation-based approach was demonstrated to be portable across NUMA platforms and processors, *i.e.*, it exploits long-latency traces without knowledge of the interconnect topology. Second, the impact of page placement was evaluated on the widely used Opteron architecture, and it was demonstrated that significant wallclock improvements are possible with hardware-derived traces on this architecture. Third, the experiments revealed that simple page-level memory interleaving worked almost as well as trace-based page placement on the small-scale Opteron system and small inputs subject to experimentation. Yet, occasionally, interleaved memory allocation may result in a significant performance penalty while long latency misses (L2 in this case) provided a uniformly reliable indicator for page placement decisions.

To the best of knowledge, this is the first evaluation on a real machine of a completely user-mode interrupt-driven trace-guided page placement scheme that requires no special compiler, operating system or network interconnect support, supports low-cost lossy tracing and covers not only static arrays, as in past work, but also heap-allocated regions.

The approach is currently relying on the "first-touch" page placement policy since dynamic page migration is just now entering stable Linux kernels. Hence, programs whose memory access patterns *change over time*, *e.g.*, adaptive mesh refinement (AMR) codes and programs with multiple execution phases, cannot exploit their full potential under the current page allocation scheme. Even though none of the test programs in this study frequently allocated and freed memory during the stable execution phase, the first-touch scheme would lose effectiveness had they done so. With the availability of page migration in the operating system kernels, both these limitations can be overcome, albeit at the cost of migration overhead that first needs to be amortized. The merits of such a dynamic framework is subject to ongoing work beyond the scope of this paper.

Acknowledgments

This work was supported in part by NSF grants CAREER CCR-0237570, CNS-0410203, CCF-0429653 and the Humboldt Foundation. This research used resources of NCSA, UNC Chapel Hill, LRZ Munich, the University of Graz, the Zuse Institute Berlin, and the Center for Computational Sciences at Oak Ridge

National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] *Meet the HP Integrity Superdome with the new HP Super-Scalable Processor Chipset sx2000*, HP, 2006. [Online]. Available: http://h71028.www7.hp.com/ERC/downloads/5982-9836EN.pdf

[2] Hewlett-Packard, "Perfmon project," http://www.hpl.hp.com/research/linux/perfmon/.

[3] Intel, *Intel Itanium2 Processor Reference Manual for Software Development and Optimization*. Intel, 2004, vol. 1.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," *The International Journal of Supercomputer Applications*, vol. 5, no. 3, pp. 63–73, Fall 1991. [Online]. Available: citeseer.ist.psu.edu/article/bailey94nas.html

[5] "C versions of nas-2.3 serial programs," 2003, http://phase.hpcc.jp/Omni/benchmarks/NPB.

[6] J. Marathe, F. Mueller, and B. R. de Supinski, "A hybrid hardware/software approach to efficiently determine cache coherence bottlenecks," in *International Conference on Supercomputing*, June 2005, pp. 21–30.

[7] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, 2003.

[8] A. Kleen, *A NUMA API for Linux*. [Online]. Available: http://www.novell.com/collateral/4621437/4621437.pdf

[9] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide*. Intel, 2007.

[10] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64–71, 2002.

[11] A. Ertl and B. Paysan, "Bplat: A memory latency benchmark," 2004, http://www.complang.tuwien.ac.at/anton/bplat/.

[12] J. H. Mustafa M. Tikir, "Using hardware counters to automatically improve memory performance," in *Supercomputing*, ACM, Ed., 2004.

[13] L. Schermerhorn, "A matter of hygiene: Automatic page migration for linux," in *linux.conf.au*, Jan. 2007.

[14] M. Dobson, P. Gaughen, M. Hohnbaum, and E. Focht, "Linux support for numa hardware," in *Linux Symposium*, July 2003.

[15] R. Bryant and J. Hawkes, "Linux scalability for large numa systems," in *Linux Symposium*, July 2003.

[16] R. Bryant, J. Barnes, J. Hawkes, J. Higdon, and J. Steiner, "Scaling linux to the extreme," in *Linux Symposium*, July 2004.

[17] D. Nikolopoulos, T. Papatheodorou, C. Polychronopoulos, J. Labarta, and E. Ayguade, "User-level dynamic page migration for multiprogrammed shared-memory multiprocessors," in *International Conference on Parallel Programming*, Aug. 2000, pp. 95–103.

[18] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade, "UPMLIB: A runtime system for tuning the memory performance of openmp programs on scalable shared-memory multiprocessors," in *Languages, Compilers, and Run-Time Systems for Scalable Computers*, 2000, pp. 85–99.

[19] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and page migration for multiprocessor compute servers," in *Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 12–24.

[20] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on ccNUMA compute servers," in *Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 279–289.

[21] L. Noordergraaf and R. Zak, "Performance experiences on Sun's Wildfire prototype," in *Supercomputing*, 1999.

[22] W. Bolosky, M. Scott, R. Fitzgerald, R. Fowler, and A. Cox, "NUMA policies and their relation to memory architecture," in *Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 212–221.

[23] W. J. Bolosky and M. L. Scott, "Evaluation of multiprocessor memory systems using off-line optimal behavior," *Journal of Parallel Distributed Computing*, vol. 15, no. 4, pp. 382–398, 1992.

[24] J. Bull and C. Johnson, "Data Distribution, Migration and Replication on a ccNUMA Architecture," in *Proceedings of the Fourth European Workshop on OpenMP*, 2002.

[25] J. Marathe and F. Mueller, "Hardware profile-guided automatic page placement for ccnuma systems," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 2006, pp. 90–99.