Hardware Profile-guided Automatic Page Placement for ccNUMA Systems *

Jaydeep Marathe Frank Mueller

Department of Computer Science, North Carolina State University, Raleigh, NC 27695-7534
e-mail: mueller@cs.ncsu.edu

Abstract

Cache coherent non-uniform memory architectures (ccNUMA) constitute an important class of high-performance computing platforms. Contemporary ccNUMA systems, such as the SGI Altix, have a large number of nodes, where each node consists of a small number of processors and a fixed amount of physical memory. All processors in the system access the same global virtual address space but the physical memory is distributed across nodes, and coherence is maintained using hardware mechanisms. Accesses to local physical memory (on the same node as the requesting processor) results in lower latencies than accesses to remote memory (on a different node). Since many scientific programs are memory-bound, an intelligent page-placement policy that allocates pages closer to the requesting processor can significantly reduce number of cycles required to access memory. We show that such a policy can lead to significant savings in wall-clock execution time.

In this paper, we introduce a novel hardware-assisted page placement scheme based on automated profiling. 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. Our method operates entirely in user space, is widely automated, and handles not only static but also dynamic memory allocation.

We evaluate our framework with a set of multi-threaded benchmarks from the NAS and SPEC OpenMP suites. We investigate the use of two different hardware profile sources with respect to the cost (*e.g.,* time to trace, number of records in profile) *vs.* the accuracy of the profile and the corresponding savings in wall-clock execution time. We show that long-latency loads provide a better indicator for page placement than TLB misses.

Our experiments show that our method can efficiently improve page placement, leading to an average wall-clock execution time saving of more than 20% for our benchmarks, with a one-time profiling overhead of 2.7% over the overall original program wallclock time. To the best of our knowledge, this is the first evaluation on a real machine of a completely user mode interrupt-driven profile-guided page placement scheme that requires no special compiler, operating system or network interconnect support.

* This work was supported in part by NSF grants CAREER CCR-0237570, CNS-0410203 and CCF-0429653. This research used resources of 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.

1. Introduction

Cache-coherent non-uniform memory architectures (ccNUMA) constitute an important subset of current high performance computing platforms. Contemporary ccNUMA platforms, such as the SGI Altix, consist of a large number nodes, where each node has a small number of processors and a fixed amount of physical memory. All processors can access the same global virtual address space, but the physical memory is distributed across the entire system and coherence is maintained using hardware mechanisms.

In a ccNUMA system, accesses to virtual memory mapped on the same node as the requesting processor typically experience much shorter latencies than accesses to physical memory on a different node. We constructed an OpenMP micro-benchmark to evaluate access latency on our target platform, the SGI Altix. The program counts the processor cycles required to access physical memory on the local and remote nodes. The results are shown in Table 1. We see that, on average, it takes more than twice as long to load from remote memory than from memory on the local node.

Table 1. Access latencies on the SGI Altix

Access Type	Average Latency (Cycles)	Standard Deviation
Local Node Memory	207	121
Remote Node Memory	430	176

In this paper, we focus on multi-threaded OpenMP benchmarks. Many of these programs are memory bound *i.e.*, the overall wall-clock execution time of the program is significantly affected by the performance of the memory hierarchy. If the physical page placement is sub-optimal, *i.e.*, the bulk of the accesses are to pages whose physical memory has been allocated on a remote node, the program will take much longer to execute. On the other hand, an intelligent page-placement scheme, that allocates physical memory on nodes closer to the processors with most frequent accesses to a page, can reduce the average access latency leading to potentially significant wallclock time savings.

To effect this intelligent page placement, we must efficiently determine the overall memory access pattern of the program. In practice, even for reasonably-sized programs, it is difficult for programmers to know the best page placement for each page. Furthermore, on systems using "first-touch" page allocation, compulsory initialization of data elements (*e.g.*, from a file) in one thread can cause the page to be allocated permanently on a particular node. We have encountered OpenMP programs that have not been specifically tuned for ccNUMA environments and often initialize all data elements in the master OpenMP thread. This causes 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. Finally, even programs that specifically initialize ("touch") data in parallel on multiple threads can still achieve sub-optimal page allocation. This commonly occurs when the number of accesses to a particular page during the stable execution phase (*e.g.*, a single timestep) of the program may indicate a better page placement than the one effected by the parallel initialization with multiple threads.

To tackle these problems, we need an efficient whole-program analysis tool that considers the overall run-time memory access pattern of the program during its stable execution phase and uses this information to decide the best page placement. In this paper, we contribute precisely such a scheme.

Our scheme works as follows. First, we execute a truncated one-timestep version of the program. We use the performance monitoring capabilities in existing microprocessors to efficiently extract an approximate trace of the memory accesses from all the active processors during this partial (truncated) run. We then use this access information to decide the best page placement, *i.e.,* the physical node on which a particular virtual page should be allocated ("affinity hints"). Finally, we run the complete program and use the affinity hints to allocate pages on the assigned physical node. The allocation is achieved by "touching" the target page from a processor on the assigned node, *i.e.*, by leveraging the default "first-touch" page allocation policy of the operating system. Our method handles both statically defined and dynamically allocated regions of memory. For statically defined memory regions (in the `bss` segment), the page touch takes effect at startup. For dynamically allocated regions, we delay the page touch till the region has been allocated.

Overall, we show that long-latency loads provide a better indicator for page placement than TLB misses and result in average wall-clock execution time savings of greater than 20% over all benchmarks, with a average one-time profiling overhead of 2.7% over the wallclock time for the complete original program. These results may make automatic page placement a cheap commodity without requiring user intervention.

The paper is structured as follows. First, we describe our page placement mechanism in detail. Then, we evaluate the placement mechanism with respect to the profile collection cost, the quality of the collected profile and the performance impact on the target program execution. We explore the use of two different profile sources, namely TLB misses and long latency loads, and the impact of different sampling intervals. Finally, we contrast our approach to related work and summarize our contributions.

2. Profile-guided Page Placement

Figure 1 shows our scheme. There are 3 distinct phases — *profile generation*, *affinity decision* and *profile-guided page placement*. In the profile-generation phase, we run a truncated version of the multi-threaded program (*e.g.*, a single timestep) and collect information about the memory access pattern for each thread. For the experiments in this paper, we explicitly bind each OpenMP thread to a different processor using the `sched_setaffinity` primitive. We also intercept and log any dynamic allocation requests generated by the program. 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, we re-run the application and use the affinity information to force allocation of pages on their assigned affinity nodes.

We have to automated our approach extensively 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 profile-guided runs and initializes the hardware performance monitor during the profile 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 and use that to guide page placement decisions. Next, our framework is described in more detail.

3. Profile Generation

We want to capture 2 types of profile information — memory accesses of each thread and calls to dynamic memory allocation.

Capturing Memory Accesses: We leverage the capabilities of the Itanium-2 performance monitoring unit (PMU) to capture an approximate trace of the memory accesses. We use the `libpfm` library to access the hardware counters of the processor [5]. The PMU operation is described in detail elsewhere [6]. In this paper, we use the PMU 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. In this mode, the PMU supports selective tracking of load instructions based on a latency threshold. However, the PMU does not capture all long-

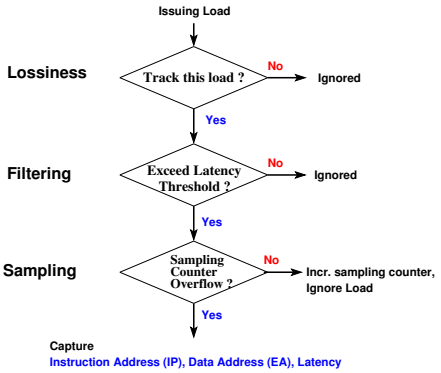


Figure 2. Simplified PMU Operation

latency loads that exceed the latency threshold. There are two reasons for this. First, due to hardware restrictions, 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 PMU uses *randomization* to decide whether or not to track an issuing load instruction. Due to these reasons, the load miss trace that can be captured is *lossy*.

If a PMU-tracked load exceeds a user-configured latency threshold value, it qualifies for capture, otherwise it is ignored (*Filtering*). Since the access latencies increase monotonically for cache levels further away 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, with a lower bound of 4 cycles (*i.e.*, valid threshold values are 4 cycles, 8 cycles, 16 cycles, etc.) .

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 processor architectures can give misleading instruction addresses for the missing load due to superscalar issue, deep pipelining and out-of-order execution [6].

The mechanism for capturing DTLB misses is similar, though there is no support for the latency threshold. Various specific sub-types of the DTLB misses can be captured (described in more detail in [6]). In this paper, we enable all types of DTLB misses for capture 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 profile source includes DTLB misses

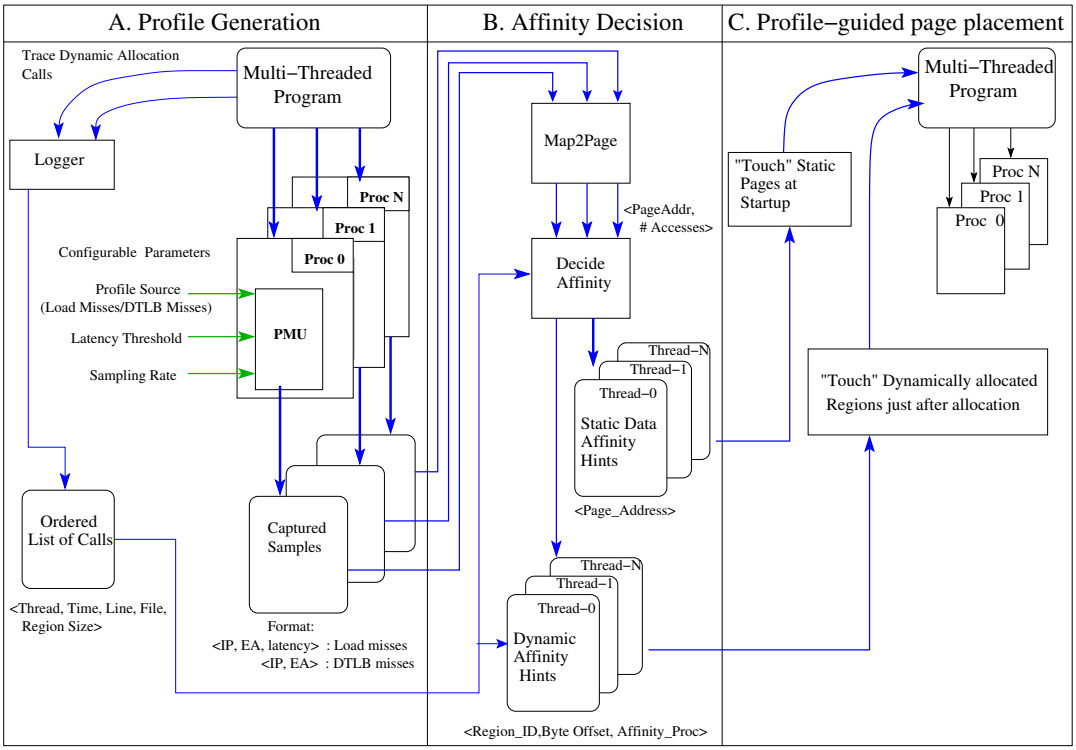


Figure 1. Automatic Profile-guided Page Placement

caused by both loads and stores while the the long-latency capture mechanism described earlier only monitors load instructions.

Capturing Dynamic Allocation Information: For profile-guided page placement, we leverage the "first-touch" allocation policy that is used by SGI's Linux version for the Altix to allocate physical memory pages. To "touch" a particular virtual page address, we need to know the earliest point in the program at which such an address becomes valid. This requires the logging of memory allocation calls for each OpenMP thread.

The logger intercepts the program calls to `malloc`, `calloc` and `free`. We also log executions to Fortran `allocate` statements. The Itanium architecture has a high-resolution timer called the "interval timer counter" (itc). By logging the `itc` timestamp for each call and knowing the skew between the itc registers of each processor, a post-processing tool builds a unified ordering of allocation calls across all the threads.

4. Affinity Decision

Next, the approximate memory access trace and the dynamic memory allocation information is used to determine page affinity, *i.e.,* to decide the node on which physical memory should be allocated for a particular virtual memory page. The affinity decision module currently uses a simple decision criterion for mapping the page — the page should be allocated on the node that had the maximum number of accesses to that page. The idea is that by allocating pages closer to the most active requesting processors, the average latency of access can be reduced.

First, the accesses are grouped by page address, and the total accesses from all threads to each page are calculated by `map2page`, as shown in Figure 1. On our target platform (SGI Altix), each node has two processors that have identical latencies when accessing local physical memory. So the affinity decision module groups the accesses by each processor to calculate the per-node access count for each page. The page is recommended for allocation on the node with the maximum number of accesses to that page. 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 profile-guided program run.

Dynamically allocated regions pose an additional challenge since 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. During the profile-guided run, just after the region is allocated, the affinity offsets will be used to "touch" the pages on the appropriate nodes.

Other programs had a large number of calls clustered in time, each allocating a very small region of memory (*e.g.*, NAS-2.3 MG). For such cases, we rely on the fact that these regions will most often be allocated contiguously in space. Since the memory access trace is *lossy*, we observe in practice that we do not have even a single access record for many small allocated regions ("silent regions"). But we still handle these regions correctly because physical memory is allocated on *page* granularity, and we have trace records for other small regions whose page tends to include the silent region.

5. Profile-guided Page Placement

In this final phase, we re-run the original program and use the affinity information we generated in the earlier phase to guide our page placement decisions. At the time of writing, the operating system (SGI's Linux version for the Altix) does not support dynamic page

migration at all. Instead, we leverage the existing "allocate-on-first-touch" policy to effect our 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. Thus, to force page placement on a particular node, we "touch" the page by executing a load followed by a store to an address in the particular page from a processor on that node.

For this mechanism to succeed, we must touch the page before any other processor accesses that page. 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. Since the static allocation is done only once, at startup, it has minimal execution overhead.

The process for dynamically allocated regions is similar, except that we must delay the page touch till the target memory region is allocated. In this case, we know that, in a legal program, no other processor can access the allocated region before the allocation function (*e.g. malloc*) has completed. We take advantage of this to ensure that our first-touch scheme will effect the page allocation we want 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 profiling 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, during the profile-guided runs, the wrapper uses affinity hints to effect page allocation as follows.

When the wrapper is invoked, it 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. After all affinity hints for all processors for the dynamic region are processed, the thread re-schedules itself on its original processor.

This scheme is completely transparent to the user-level program, except for the addition of the wrapper function. However, it has high execution overhead. 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). We found during our evaluation (Section 6) that this scheme has substantial execution overhead, which erases 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 the dynamically allocated region before the touching has occurred (to prevent inadvertent page allocation). We leave this idea for future work.

There is one additional issue for dynamically allocated regions. In the benchmarks we evaluate, programs allocate memory only at the start of the program and do not free it till the end of execution. If, on the other hand, 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 is a limitation of using the first-touch mechanism. This issue could be solved if our operating system (Linux) supported dynamic page *migration*, which is not available at the present time.[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.

6. Evaluation Framework

We described the scheme for profile-guided page placement in the previous section. In the following we present a cost versus benefit analysis as we vary the configurable parameters shown in Figure 1.

There are two configurable parameters that we shall vary — the choice of the *profile source* and the *sampling interval* for capturing memory access samples. The hardware provides two profile sources — long-latency loads and DTLB misses. The sampling interval provides a method to trade-off sampling overhead *vs.* the amount of profile data collected. For each profile source, we experiment with different sampling interval values (Sections 7 and 8).

As we vary these two parameters, the amount and type of profile information that we collect will change. How good are the affinity hints generated using these profiles? How effective are the affinity hints in reducing the overall wallclock execution time? To answer these questions, we need to compare the performance of these profiles with respect to the performance of affinity hints based on a *reference* profile. We call this reference profile the "maximum information profile". The reference profile answers this question: What affinity hints will be generated, if we knew as much as possible about the memory access pattern of the program? How much improvement in performance can be achieved using these hints? By comparing our profiles against the results achieved with the maximum information profile, we can clearly evaluate the tradeoff between profile collection cost and the optimization benefit.

Initially, we experimented with a software memory tracing tool to capture all memory loads and then used this access trace as the maximum information profile. However, this method had too much execution overhead for the benchmarks we evaluated. Instead, we configure the PMU with the lowest latency threshold setting (4 cycles) and the highest sampling frequency (1) and use the collected trace as the maximum information profile. 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" refers to the affinity hints generated using the maximum information profile. Similarly, the "target profile results" denote the affinity hints generated by the profile being evaluated.

Our evaluation has three aspects - the *profiling cost*, the *quality* of the collected profile, and the resulting *execution benefits*. The profiling cost is the cost of collecting the access trace, which is determined by the *size of the profile* and by the *execution overhead* inflicted on the benchmark during the profile collection phase. As the sampling interval is increased, both the profile size and the profiling overhead are expected to decrease.

For evaluating the quality of the profile, we shall compare the target profile results to the reference results using three different metrics. **Coverage** denotes the fraction of the pages in the *reference results* for which we have an affinity hint in the target profile results.

[1] Draft APIs for manual page migration have been proposed, and are expected to be available with future Linux systems.

The affinity node values for the page do not need to be the same between the reference and target profile results. **Accuracy** denotes the fraction of the pages in the *target profile results* that have the same affinity hint node value as the reference results. If the coverage value is low, it indicates that for large number of pages we simply do not have enough information in the target profile to generate an affinity hint. If the coverage value is high, we are confident that the target profile contains affinity hints for almost the same number of pages as the reference profile (though the affinity node *values* might be different).

In contrast, *accuracy* measures the stand-alone usefulness of the target profile. It answers the question: If the target profile were to be used to generate affinity hints, what fraction of the affinity hints are identical to those present in the reference results? If the accuracy value is high, it indicates that the target profile is as useful as the reference profile (though at a potentially much reduced overhead). On the other hand, a low accuracy value indicates that the target profile is potentially misleading in the sense that the affinity hints do not match the hints in the reference results.

Finally, the **Useful Fraction** metric combines the information from these two metrics. It measures the fraction of the affinity hints in the target profile that are not only present in the reference trace, but also have the *same* affinity node value.

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

Ref = # hints in reference results;

Targ = # hints in target profile results;

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

A = # hints in target profile 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 profile characteristics. For example, a high accuracy value might still not indicate an *effective* profile if the coverage is low. This is because we simply will not have affinity hints for many pages (low coverage), but the few hints that we do generate are accurate (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 the need for all three metrics.

So far, we have seen the metrics for *cost* and *profile quality*. For assessing *profile benefit*, we measure two things - the net reduction in *remote accesses* and 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 the program run with our profile-guided page placement scheme.

The evaluation process works as follows. First, the target program is run for one time step and profile data is collected. This profile data is used to compute the affinity hints and the entire program is re-run using this profile data. Thus, the *profile cost* is the cost to capture the samples over one timestep of the program. On our platform, there is no easy way to measure the number of remote memory accesses generated by the program. Instead, we present an *approximate* measure of the reduction in remote memory accesses as follows. We set the PMU latency threshold to 512 cycles[2] and count the number of accesses that exceeded this threshold for the original program. 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 1). Then, we run the program with our page placement scheme and count the number of accesses exceeding the latency threshold (512) as before. The difference between the two values provides an approximate measure of the net reduction in remote memory accesses. In practice, we have found this value to be quite consistent across multiple runs.

When comparing wallclock execution time, we compare the wallclock time for the complete run of the original program to the wallclock time of the program with profile-guided page placement, including the overhead of the page touching mechanism. During our experiments, we noticed 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 profile-guided page placement (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.

In order to account for this variability in execution time, we ran each benchmark for 6 times (5 times for BT). Each time, the profile-guided runs and the non-profile 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: We use a set of 9 OpenMP benchmarks. This includes 7 out of the 8 NAS-2.3 benchmarks (excluding EP). The NAS benchmarks are C versions of the original NAS-2.3 serial benchmarks [2], provided by the Omni Compiler group [1]. We do not evaluate EP since it does not have significant sharing of data [7]. In addition, we also evaluate the 320.equake and 332.ammp benchmarks from the SPEC OMPM2001 benchmark set. These benchmarks have significant dynamic memory allocation, thereby putting our 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 our current platform, there are two processors per 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. Our 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.

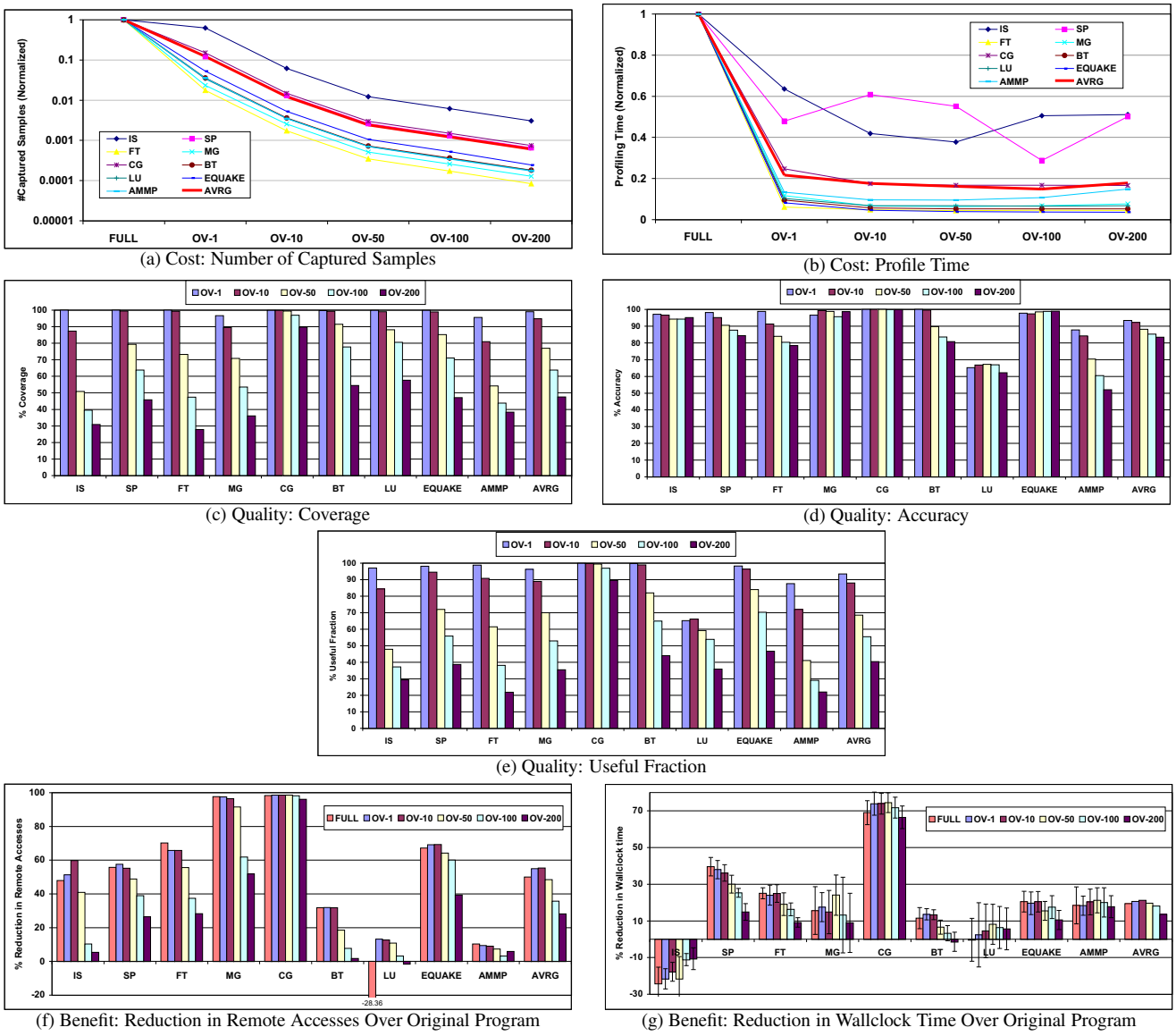
For each program, we inserted markers delineating the start and end of the timestep. For 332.ammp, we disabled the pre-existing round-robin allocation of the "atom" element for the profile-related runs. However, we still compare the benefit metrics (wallclock time, number of remote accesses) against the original program. For the IS benchmark, we perform a one-time dynamic allocation for the `prv_buff1` array 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, IS — utilize dynamic memory allocation. The remaining benchmarks operate with statically declared global arrays.

7. Evaluation with Long-latency Load Profiling

We evaluated the performance of our page-placement scheme with long-latency loads as the profile source. Figure 3 shows the perfor-

[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 1.



(a) Cost: Number of Captured Samples

(b) Cost: Profile Time

(c) Quality: Coverage

(d) Quality: Accuracy

(e) Quality: Useful Fraction

(f) Benefit: Reduction in Remote Accesses Over Original Program

(g) Benefit: Reduction in Wallclock Time Over Original Program

Figure 3. Evaluation with Latency threshold=128, Profile Source=Long Latency Loads

mance using the cost / quality / benefit approach that was described in the last section.

For these experiments, we fix the latency threshold in the PMU to 128 cycles. This filters out most of the load accesses that hit in the L1D, L2 and L3 caches. We select sampling intervals of 1, 10, 50, 100, 200 (OV-1 to OV-200 in the graphs).

Profiling Cost: The graph for cost comparison shows the cost for the "maximum information profile" (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 profile values.

Number of Captured Samples: The number of accesses captured at OV-1, depicted in Figure 3(a), is about an order of magnitude lower than the FULL profile for most benchmarks (except IS). By keeping the latency threshold much higher (128 cycles instead of 4 cycles for the FULL profile) we filter out most of the loads that hit in cache. 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.

Profiling Execution Overhead: The absolute execution overhead for profiling is extremely low, since it is sufficient to only a single timestep for the benchmarks that we considered, *i.e.*, the partial execution saves significant overhead over an execution of the entire benchmark without any loss in accuracy for the benchmarks studied.

On average, over all benchmarks, we measured the execution overhead for profiling a single timestep at OV-1 to be 2.7% of the overall original program execution time.

The *relative* profiling execution overhead (compared to `FULL`) is shown in Figure 3(b). We see that the overhead *flattens* out with increasing sampling intervals. This indicates that the profile

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, profiling execution overhead at OV-1 is about 20% of the FULL profile cost, with the exceptions of `SP` and `IS` that have lower savings.

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

Coverage: Figure 3(c) depicts the coverage results for different sampling intervals. At OV-1, the average coverage is 99% indicating that we have affinity hints for almost all of the FULL profile pages. The OV-10 coverage still remains high at 94%. After that, we observe a noticeable decline in coverage at sampling intervals of OV-50 and beyond. 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, we simply do not have enough profile data to generate affinity hints for page placement for a significant number of pages.

Accuracy: The accuracy values, depicted in Figure 3(d), 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 profile 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 profile affinity hints that are present and have the same affinity node value in the target profile results. A high useful fraction indicates that we are obtaining almost the same results as the FULL profile results, with much smaller profile input data.

The average useful fraction, depicted Figure 3(e),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`.

Profile Benefits: We explore the impact of the page placement scheme on 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 3(f) shows the net reduction in the number of remote accesses for the full-program run using automatic profile-guided page placement *vs.* the original program. The figure compares the reduction in remote accesses using the FULL profile, *vs.* the reduction achieved at latency threshold 128, and the different sampling intervals.

For all but one case (LU:FULL profile), 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 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 profile. 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 profile-guided placement compared to the original unmodified program. Figure 3(g) 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 profile-guided page placement. We determined that 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 we bind each OpenMP thread to a different processor. Hence, we refer 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, we see less degradation in wallclock execution time 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 profile do not match the affinity hints generated by the other profiles (OV-1 to OV-200). This causes low accuracy and useful fraction values, as seen in Figures 3(d) and 3(e). Furthermore, using the full profile leads to an *increase* in the number of remote accesses (Figure 3(f)) 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-profile results (0% improvement).

The underlying cause is as follows. The affinity node values differ between the full profile and the OV-1 profile (and higher sampling interval profiles) for parts of the large `rsd` global static array. The full profile 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 profiles, most of the accesses on the affinity node given in the full-profile are hits in the local caches. Hence, the affinity decision is different from the OV-1 profile-based decision (which filters out the cache hits). First, we observe that loads which are hits in cache will not be affected by page placement decisions. Second, the full-profile 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-profile results compared to the OV-1 based experiment. Thus, the average wallclock time improvement is lower for full-profile than for OV-50 in this case.

Conclusions: Long-Latency Profiling: 1) Overall, we observe that the size of the profile data at OV-1 is one-tenth the size of the FULL profile on average. With increasing sampling intervals (OV-1 to OV-200), the profile size decreases linearly. 2) For

most benchmarks, the execution overhead of profile 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 profile collection. 3) With increasing sampling intervals, the coverage drops significantly, which indicates insufficient profile information to generate affinity decisions for many pages. 4) Nevertheless, the *accuracy* of the profile 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 profile-guided page placement. However, for programs with dynamic allocation, the page touching mechanism is expensive 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 profile (FULL) actually resulted in a *degradation* of performance. For this benchmark, the *filtering effect* of the high latency threshold used by the target profiles (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 profile almost always had the maximum (or close to maximum) performance benefits, *i.e.*, reduction in remote accesses and wallclock time.

8. Evaluation with Data TLB Misses Profiling

Figure 4 depicts the results using data TLB misses as the profile source obtained with PMU support. We evaluate results for sampling intervals values of 1, 2, 4, 8 and 16 (denoted OV-1 to OV-16 in the graphs). For the discussion below, we shall refer to the results presented in the last section using long-latency loads as the profile source as the *load-based results*. In the following, we describe the DTLB miss results and contrast them with the load-based results.

Profile Cost: As before, the cost metrics are compared against the cost incurred for the "maximum information profile" (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 profile, as seen in Figure 4(a). 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 profile data at OV-1 compared to FULL while IS has almost the same number of samples as FULL.

Profiling Execution Overhead: The results for the relative profile overhead, depicted in Figure 4(b), are similar to load-based results. The average execution overhead for trace collection at OV-1 is 18% of the FULL profile's cost. With increasing sampling intervals (OV-1 to OV-16), the execution overhead is not significantly reduced.

Profile Quality: As before, we evaluate the three quality metrics of *coverage*, *accuracy* and *useful fraction* shown in Figures 4(c), 4(d) and4(e), respectively.

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 4(c). 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 we have insufficient information to generate page affinity hints 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 4(d) 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 profile trace.

We also observe sharply lower accuracy for FT, BT, LU and AMMP compared to the load-based results. 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 sharply 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 profile 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.

Profile Benefits: We have seen that the coverage, accuracy and useful fraction for DTLB-based results are significantly lower than their load-based counterparts for most benchmarks. This will impact the performance benefits obtainable with profile-guided page placement. Figures 4(f) and 4(g) show the reductions in remote accesses and overall wallclock execution time, respectively.

Reduction in Remote Accesses: As before, the reduction in remote accesses using profiles obtained at different sampling intervals is compared to the reduction obtained with results based on the full profile (marked FULL) seen in Figure 4(f).

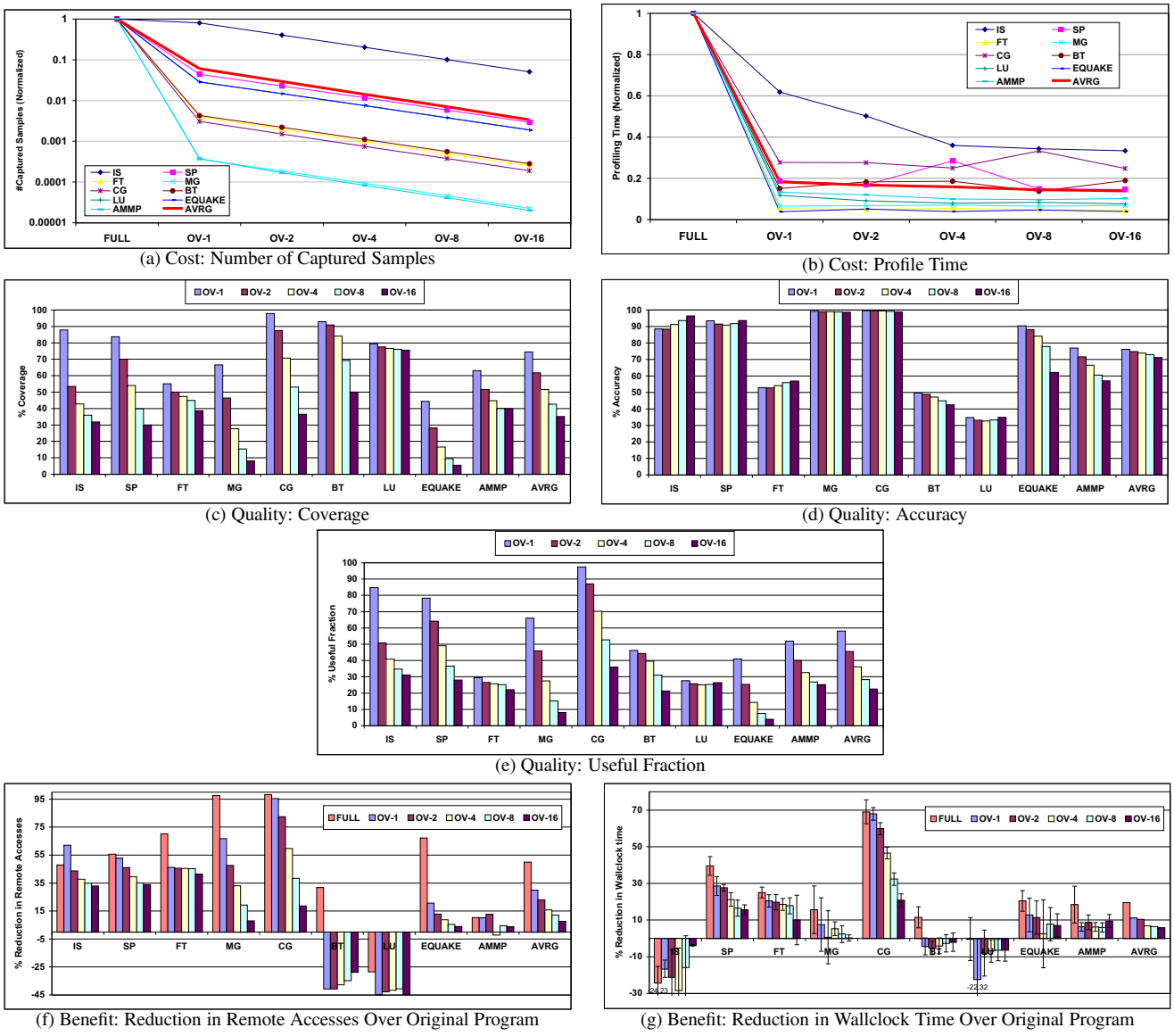
Two benchmarks, BT and LU, experience an *increase* in remote accesses with DTLB-guided page placement. The increases are significant (more than 30%) and occur with all sampling intervals. In comparison to the load-based results, the reduction in remote accesses is much lower for many benchmarks, especially 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 wallclock reduction at OV-1 is 11% for DTLB misses (see Figure 4(g)) *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%), Ammp (18% *vs.* 6%) compared to load-based results at OV-1.

Conclusions for DTLB-Profiling: 1) Overall, the cost of profile 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 for DTLB-based results 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 profile. 4) The performance benefits (reduction in remote accesses and wallclock time) are also much lower for DTLB-based results. 5) The profile costs for both DTLB misses and long-latency loads are similar, but the quality of the profile and the resulting performance benefits are much larger for long-latency load-based profiles compared to the DTLB miss based profiles.

We conclude that DTLB misses are not a good candidate to decide page placement. This shows that, for the benchmarks we considered, DTLB misses do not correlate well with the relative



(a) Cost: Number of Captured Samples

(b) Cost: Profile Time

(c) Quality: Coverage

(d) Quality: Accuracy

(e) Quality: Useful Fraction

(f) Benefit: Reduction in Remote Accesses Over Original Program

(g) Benefit: Reduction in Wallclock Time Over Original Program

Figure 4. Evaluation with Profile Source=Data TLB Misses

volume of loads from a processor to a particular memory page. This could occur, for example, 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 we only consider the DTLB misses (since repeated accesses to the same page will tend to hit in the DTLB). [3]

9. 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 [11]. 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 profile-guided page placement leveraging the simpler "first-touch" page allocation policy of the operating system. [4]

Our method uses a different profile 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

[3] 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.

[4] In the future, our approach can be extended in a straightforward way to eliminate the need for a separate profiling run by *migrating* pages. This depends on proposed future extensions in Linux to support dynamic page migration under user control.

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 till 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 [8, 9]. 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 don't 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).

Verghese *et al.* describe a simulation-based kernel-level implementation of dynamic page migration [12]. They consider both number the of load-misses to a page and the number of data TLB misses as profile sources. In our work, we found data TLB misses to be less effective for deciding the best page placement, which confirms results presented in their work.

Other approaches to kernel-level dynamic page migration and replication are discussed in Noordergraaf *et al.* [10] and in Bolosky *et al.* [3]. In contrast, we operate completely in user-space 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 [4]. 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 [6]. 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 [7].

10. Conclusion

In this work, we developed and evaluated a low-cost whole-program analysis tool that considers the overall run-time memory access pattern of the program during its stable execution phase. It uses this information to decide the best page placement. The novelty of our work also lies in the exploration of hardware-assisted performance monitoring techniques, completely in user mode, without any special compiler, operating system or network interconnect support.

Our technique operates as follows. First, we execute a truncated one-timestep version of the program. We leverage performance monitoring capabilities in existing microprocessors to efficiently extract an approximate trace of the memory accesses from all the active processors during this partial (truncated) run. We then use this access information to decide the best page placement, *i.e.,* the physical node on which a particular virtual page should be allocated ("affinity hints"). Finally, we run the complete program and use the affinity hints to allocate pages on the assigned physical node. The allocation is achieved by "touching" the target page from a processor on the assigned node, *i.e.*, by leveraging the default "first-touch" page allocation policy of the operating system. Our method handles both statically defined and dynamically allocated regions of memory. For statically defined memory regions (*i.e.*, the `bss` segment), the page touch is effected at startup. For dynamically allocated regions of memory, we delay the page touch till the region has been allocated.

Our framework is currently constrained to work with the "first-touch" page placement policy, as dynamic page migration is not supported on Linux at the present time. Due to this, we cannot do effective page allocation for programs whose memory access patterns *change over time*, *e.g.*, adaptive mesh refinement (AMR) codes, and programs with multiple execution phases. Also, the first-touch based scheme would lose effectiveness on programs which frequently allocate and free memory during the stable execution phase (none of the programs in this study show this behavior). When page migration support is added to Linux, we shall overcome both these limitations.

Overall, we show that long-latency loads provide a better indicator for page placement than TLB misses that results in average wall-clock execution time savings of more than 20% over all benchmarks. with an average one-time profiling cost of 2.7% over the overall original program wallclock time. The low overhead may make automatic automatic page placement a cheap commodity without requiring user intervention.

References

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

[2] 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*, 5(3):63–73, Fall 1991.

[3] W. Bolosky, M. Scott, R. Fitzgerald, R. Fowler, and A. Cox. NUMA policies and their relation to memory architecture. In *Proceedings of the fourth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 212–221, 1991.

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

[5] Hewlett-Packard. *Perfmon project*.

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

[7] 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.

[8] 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*, pages 95–103, Aug. 2000.

[9] 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*, pages 85–99, 2000.

[10] L. Noordergraaf and R. Zak. Performance experiences on Sun's Wildfire prototype. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999.

[11] M. M. Tikir and J. K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 46, Washington, DC, USA, 2004. IEEE Computer Society.

[12] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on ccNUMA compute servers. In *Proceedings of the seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, 1996.