A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures

Xu Liu John Mellor-Crummey

Department of Computer Science MS 132, Rice University
P.O.Box 1892, Houston, TX 77251-1892
{xl10, johnmc}@rice.edu

Abstract

Almost all of today's microprocessors contain memory controllers and directly attach to memory. Modern multiprocessor systems support non-uniform memory access (NUMA): it is faster for a microprocessor to access memory that is directly attached than it is to access memory attached to another processor. Without careful distribution of computation and data, a multithreaded program running on such a system may have high average memory access latency. To use multiprocessor systems efficiently, programmers need performance tools to guide the design of NUMA-aware codes. To address this need, we enhanced the HPCToolkit performance tools to support measurement and analysis of performance problems on multiprocessor systems with multiple NUMA domains. With these extensions, HPCToolkit helps pinpoint, quantify, and analyze NUMA bottlenecks in executions of multithreaded programs. It computes derived metrics to assess the severity of bottlenecks, analyzes memory accesses, and provides a wealth of information to guide NUMA optimization, including information about how to distribute data to reduce access latency and minimize contention. This paper describes the design and implementation of our extensions to HPCToolkit. We demonstrate their utility by describing case studies in which we use these capabilities to diagnose NUMA bottlenecks in four multithreaded applications.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Measurement techniques, Performance attributes; D.2.8 [*Metrics*]: Performance measures.

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

ACM 978-1-4503-2656-8/14/02. . . $15.00.
http://dx.doi.org/10.1145/2555243.2555271

Keywords profiler, threads, NUMA, performance optimization, memory access pattern.

1. Introduction

As microprocessors have become faster and multiple cores per chip have become the norm, memory bandwidth has become an increasingly critical rate-limiting factor for system performance. For multiprocessor systems, scaling memory bandwidth proportional to processing power has led to designs in which microprocessors include memory controllers on chip. As a result, the aggregate memory bandwidth of systems scales with the number of microprocessors.

Multiprocessor systems in which some memory is locally-attached to each processor are known as Non-Uniform Memory Access (NUMA) architectures because it is faster for a microprocessor to access memory that is locally attached rather than memory attached to another processor. Some microprocessors, e.g., IBM's POWER7 [29], have NUMA organizations on-chip as well, with some cores having lower latency access to some directly-attached cache and/or memory banks than others. To simplify discussion of systems where NUMA effects may exist within and/or between microprocessors, we simply refer to cache/memory along with all CPU cores that can access it with uniform latency as a *NUMA domain*. There is not only a difference in latency when accessing data in local vs. remote NUMA domains, there is also a difference in bandwidth: cores typically have significantly higher bandwidth access to local cache/memory than remote.

Systems with multiple NUMA domains are challenging to program efficiently. Without careful design, multithreaded programs may experience significant performance losses when running on systems with multiple NUMA domains if they access remote data too frequently. On such systems, multithreaded programs achieve best performance when threads are bound to specific cores and each thread mostly processes data co-located in the same NUMA domain as the core in which it executes. In addition to latency, contention can also hurt the performance of multithreaded programs. If many of the data accesses performed by a mul-

tithreaded program are remote, contention for limited bandwidth between NUMA domains can be a bottleneck. This problem can be particularly acute if a large data array is mapped to a single NUMA domain and many threads compete for the limited bandwidth in and out of that domain. This situation is more common than one might think. By default, today's Linux systems employ a "first-touch" policy to bind pages of memory newly-allocated from the operating system to memory banks directly attached to the NUMA domain where the thread that first accesses the page resides. As a result, if a single thread initializes a data array, but multiple threads process the data later, severe contention can arise.

Tailoring a program for efficient execution on systems with multiple NUMA domains requires identifying and adjusting data and computation layouts to minimize each thread's use of remote data and avoid contention for bandwidth between NUMA domains. Due to the myriad of opportunities for mis-steps, tools that can provide insight into NUMA-related performance losses are essential for guiding optimization of multithreaded programs.

There are two broad classes of techniques for identifying NUMA bottlenecks in a program: simulation and measurement. Tools such as MACPO [25] and NUMAgrind [32] use simulation to identify NUMA bottlenecks in a program. A drawback of tools that simulate all memory accesses is that they are slow, which makes them of limited use for programs with significant running time. To address this shortcoming, a new class of tools, e.g., ThreadSpotter [26], apply simulation sparingly to selected memory accesses to reduce execution overhead.

In contrast, measurement-based tools, such as TAU [27], Intel Vtune Amplifier [11], IBM Visual Performance Analyzer (VPA) [10], AMD CodeAnalyst [2], and CrayPat [8] can gather data to provide insight into NUMA performance at much lower cost. On today's microprocessor-based systems, such tools can monitor NUMA-related events using a microprocessor's on-chip Performance Monitoring Unit (PMU). These tools measure and aggregate NUMA-related events and associate them with source code contexts, such as functions and statements. We call this approach *code-centric analysis*. A shortcoming of code-centric measurement and analysis is that it often fails to provide enough guidance for NUMA optimization [24]. *Data-centric analysis* tools, which can provide deeper insight into NUMA bottlenecks, use advanced PMU capabilities to gather instruction and data address pairs to associate instructions that access memory with the variables that they touch. Existing data-centric tools, such as HPCToolkit [19–21], Memphis [24] and MemProf [15] can identify both instructions and variables that suffer from NUMA problems.

There are three principal shortcomings of existing tools for measurement-based data-centric analysis. First, since PMU support for data-centric analysis differs significantly across microprocessors from different vendors and even processor generations, most data-centric tools only support only a single family of PMU designs. Second, these tools do not assess the impact of NUMA bottlenecks on overall program performance. Without such information, one may invest significant effort to improve the design of a code for NUMA systems to address measured inefficiencies and net only a small performance gain. Finally, while existing tools can identify NUMA-related bottlenecks, they fail to provide information to guide NUMA-aware code optimization.

To address these shortcomings, we developed a lightweight tool for measurement and analysis of performance problems on multicore and multiprocessor systems with multiple NUMA domains. Our profiler outperforms existing tools in three ways. First, our tool is widely applicable to nearly all modern microprocessor-based architectures. Second, we define several derived metrics that can be computed by tools to assess the severity of NUMA bottlenecks. These metrics can effectively identify whether NUMA-related performance losses in a program are significant enough to warrant optimization. Third, our tool analyzes memory accesses and provides a wealth of information to guide NUMA optimization, including information about how and where to distribute data to maximize local accesses and reduce memory bandwidth contention.

We describe case studies using four well-known multithreaded codes that highlight the capabilities of our tool and demonstrate their utility. For three of the programs, our tool provided unique insights unavailable with other tools. These guided us to code changes that yielded non-trivial performance improvements. In the course of our studies, we found, somewhat surprisingly, that stack variables sometimes play a significant role in NUMA bottlenecks. Another case study demonstrates the utility of our novel metrics for assessing the severity of NUMA bottlenecks.

The rest of the paper is organized as follows. Section 2 describes NUMA problems and introduces NUMA optimization strategies. Section 3 describes hardware support for data-centric measurement in modern microprocessors, which provides the foundation for our tool. Section 4 describes derived metrics our tool computes to quantify the impact of NUMA-related performance losses. Section 5 shows how we attribute metrics with different views for NUMA analysis. Section 6 describes how we efficiently pinpoint locations in the source code that need modification to effect NUMA-aware data distributions. Section 7 describes the implementation of our tool. Section 8 presents case studies that illustrate the use of our tool to identify and fix NUMA-related bottlenecks in four multithreaded programs. Section 9 discusses previous work on tools for NUMA performance analysis and distinguishes our work. Finally, Section 10 summarizes our conclusions and plans for future work.

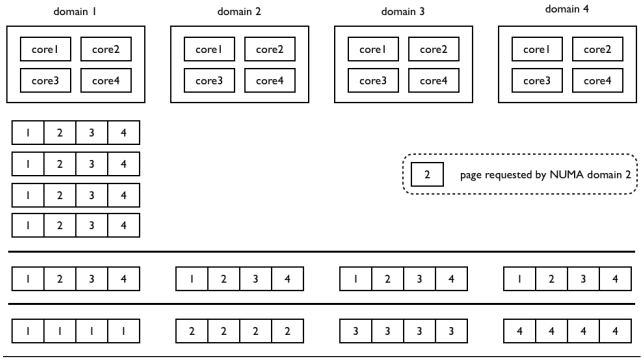


Figure 1. Three common data distributions in a program on a NUMA architecture. The first distribution allocates all data in NUMA domain 1. It suffers from both locality and bandwidth problems. The second distribution maps data to each of the NUMA domains and avoids centralized contention. The third distribution co-locates data with computation, which both maximizes low-latency local accesses and reduces bandwidth contention.

2. NUMA-aware program design

There are two causes of NUMA bottlenecks: excessive remote accesses and uneven distribution of requests to different NUMA domains. In modern processors, remote accesses have more than 30% higher latency than local accesses [28]. If one co-locates data in the same NUMA domain with a thread that manipulates it most frequently, the program can benefit from the fast local accesses. We use the term *co-location* to refer to the process of mapping part or all of a data object and the thread that accesses it most frequently to the same NUMA domain.

On the other hand, uneven distribution of memory accesses to NUMA domains can lead to contention and unnecessary bandwidth saturation in both on-chip and off-chip interconnects, caches, and memory controllers. Contention for interconnect and memory controller bandwidth has been observed to increase memory access latency by as much as a factor of five [7]. Instead of mapping large data objects to a single NUMA domain, in many cases one can reduce contention and associated bandwidth saturation by distributing large data objects across all NUMA domains. We call this optimization *contention reduction*.

Figure 1 illustrates various strategies for mapping data to NUMA domains and discusses the latency, contention, and performance issues associated with alternative distributions. One can identify excessive remote accesses to program data objects by measuring NUMA-related events using techniques described in the next section. One can identify contention by counting requests to different NUMA domains. Co-locating data and computation is the most powerful optimization as it reduces the need for bandwidth to remote domains, reduces bandwidth contention, and reduces latency by exploiting the lower latency and higher bandwidth access to local data. In cases where there is not a fixed binding between threads and data and/or concurrent computations may focus on only parts of a data object at a time, then using memory interleaving to avoid contention for a single NUMA domain may be beneficial. However, in some cases, using interleaving to balance memory requests to NUMA domains may hurt locality and performance [15, 21].

When data objects are not allocated using interleaved memory pages, the Linux "first touch" policy binds a new memory page obtained from the OS to a physical page frame managed by a memory controller when the page is first read or written. To ensure that data will be mapped to the proper NUMA domain, one must carefully design code that first touches each of the principal data structures in a multi-threaded program. To control allocation without completely refactoring an application, one can introduce an initialization pass right after a variable is allocated to control its page distribution.

To help application developers tailor a program for NUMA systems, application developers need tools that

- pinpoint the variables suffering from uneven memory requests, so one can use different allocation methods (e.g., interleaved allocation) to balance the memory requests,
- analyze the access patterns across threads to guide NUMA locality optimization, and
- identify where data pages are bounded to NUMA domains.

To our knowledge, no prior profiling tool provides all of these capabilities. In the next section, we describe address samplng—a key technique needed to build tools with these desired capabilities.

3. Address sampling

Address sampling, which involves collecting instruction and data address pairs to associate memory references with the data that they touch, is essential for profiling NUMA access patterns. PMUs on most recent processors from AMD, Intel, and IBM support address sampling. A processor can support efficient NUMA profiling *iff* it has the following three capabilities.

- It can record the effective address touched by a sampled instruction that accesses memory. It is important for this support to guarantee that memory accesses are uniformly sampled.
- It can sample memory-related events, such as load/store instructions and cache accesses/misses, as well as measure memory access latency. Such information is useful for quantifying NUMA-related bottlenecks.
- It can capture the precise instruction pointer for each sample. Special hardware support is required for correct attribution of access behavior to instructions on processors with out-of-order cores [9].

We know of five hardware-based address sampling techniques used in modern processors: instruction-based sampling (IBS) [9] in AMD Opteron and its successors, marked event sampling (MRK) [30] in IBM POWER5 and its successors, precise event-based sampling (PEBS) [12] in Intel Pentium 4 and its successors, data event address sampling (DEAR) [13] in Intel Itanium, and precise event-based sampling with load latency extension (PEBS-LL) [12] in Intel Nehalem and its successors.

Since support for address sampling is not universally available, e.g., on ARM processors. we developed a software-based strategy for address sampling that we call Soft-IBS. Soft-IBS can simulate address sampling with memory access instrumentation. Using an instrumentation engine based on LLVM [22], we instrument every memory access instruction using a function that captures the effective address and instruction pointer of the sampled instruction. The engine invokes an instrumentation stub function that our profiler overloads to monitor memory accesses. Rather than recording information for each memory access, our profiler records information for every n^{th} memory access, where the value of n can be selected when the profiler is launched.

These six address sampling mechanisms are the foundation for our NUMA profiling tool. The aforementioned hardware and software mechanisms for address sampling each have their own strengths and weaknesses for NUMA profiling. Naturally, the hardware mechanisms are much more efficient than Soft-IBS. In Section 8, we compare the overhead of address sampling with these various mechanisms.

4. NUMA metrics

Using information we gather with address sampling, we compute several metrics to gain insight into NUMA bottlenecks. In the next section, we describe metrics to identify remote accesses and imbalanced memory requests. These metrics can be computed with information gathered using any mechanism for address sampling. In Section 4.2, we describe some additional metrics that can be derived using information available from only some PMU implementations.

4.1 Identifying remote accesses and imbalanced requests

To understand remote accesses, our profiler computes two derived metrics: M_l and M_r. M_l is the number of sampled memory accesses touching the data in the *local* NUMA domain, while M_r is the number of sampled memory accesses touching the data in a *remote* NUMA domain. Unless $M_r \ll M_l$ for a code region, the code region may suffer from NUMA problems. To compute M_l and M_r, our tool needs two pieces of information for each address sample: the NUMA domain that is the target of an effective address and the NUMA domain of the thread performing the access. Our profiler uses the `move_pages` API from `libnuma` [14] to query the NUMA domain for the effective address. To identify the NUMA domain for a thread, we have two mechanisms. With PMU support for address sampling, a sample includes the CPU ID performing the access. For Soft-IBS, we bind each thread to a core and maintain a static mapping between thread and CPU ID that we query at runtime. Our tool uses `libnuma`'s `numa_node_of_cpu` to map the CPU ID to its NUMA domain. For each address sample, if the effective address and thread are in the same NUMA domain, we increment M_l; otherwise, we increment M_r.

To evaluate memory request balance, our tool counts the number of sampled memory accesses to each NUMA domain by each thread. As before, we identify the NUMA domain for an access using the `libnuma` `move_pages` API. If the aggregate number of accesses to some NUMA domains is much larger than others, the access pattern may cause memory bandwidth congestion.

By themselves, the aforementioned metrics can be misleading. For example, if a thread loads a variable allocated into its private cache and touches it frequently, though no further remote accesses occur, the M_r caused by this thread is high, because the variable is deemed in the remote NUMA domain by `move_pages`. Therefore, one needs to use other metrics to eliminate this bias. In the next section, we describe a latency-per-instruction metric, which can be computed on some architectures, that addresses this shortcoming.

4.2 NUMA latency per instruction

IBS and PEBS-LL support measuring the latency of remote accesses. When this information is available, our tool computes the NUMA latency per instruction to provide additional insight into NUMA bottlenecks. We compute the NUMA latency per instruction, lpi_{NUMA}, to quantify a NUMA bottleneck's impact on overall program performance. If a NUMA bottleneck's lpi_{NUMA} is small, even if M_r is large, NUMA optimization of the program can't improve performance much. Equation 1 defines lpi_{NUMA}.

$$\begin{aligned} lpi_{NUMA} &= \frac{l_{NUMA}}{I} \\ &= \frac{l_{NUMA}}{I_{NUMA}} \times \frac{I_{NUMA}}{I_{MEM}} \times \frac{I_{MEM}}{I} \end{aligned} \quad (1)$$

In the equation, l_{NUMA} is the total latency of all remote accesses (including remote caches and memory); I_{NUMA}, I_{MEM}, and I represent the number of remote accesses, memory accesses, and instructions executed, respectively. This metric can be computed for the whole program or any code region. The equation can be computed as the product of three terms: the average latency per remote access, the fraction of memory accesses that are remote, and the ratio of memory accesses per instruction executed. The resulting quantity indicates whether NUMA performance losses are significant for a code region.

Equation 2 shows how we approximate lpi_{NUMA} using address sampling with IBS.

$$lpi_{NUMA} \approx \frac{l^s_{NUMA}}{I^s} \quad (2)$$

l^s_{NUMA} is the accumulated latency for sampled remote accesses; I^s is the number of sampled instructions. Calculating lpi_{NUMA} this way yields an approximate value because l^s_{NUMA} and I^s are representative subsets of l_{NUMA} and I. Equation 3 shows how we approximate lpi_{NUMA} using address sampling with PEBS-LL.

$$lpi_{NUMA} \approx \frac{l^s_{NUMA}}{E^s_{NUMA}} \times \frac{E_{NUMA}}{I} \quad (3)$$

As PEBS-LL samples memory access events, we can obtain an absolute event number E_{NUMA} and an average latency per sampled remote access event $\frac{l^s_{NUMA}}{E^s_{NUMA}}$ for the whole program or any code region. With a conventional PMU counter, we can collect the absolute number of instructions I executed by the monitored program or any code region. Experimentally, we have found that if lpi_{NUMA} is larger than 0.1 cycle per instruction, the NUMA losses for a program or important code region are significant enough to warrant optimization.

5. Metric attribution

To help a user understand NUMA performance losses, our tool attributes metrics using three different approaches.

- *Code-centric* attribution correlates performance metrics to instructions, loops, and functions that have high access latency.
- *Data-centric* attribution associates performance metrics with variables that have high access latency.
- *Address-centric* attribution summarizes a thread's memory accesses, which is useful for understanding data access patterns.

Each attribution technique highlights different aspects of NUMA performance losses. Together, they provide deep insight into NUMA bottlenecks. We describe these attribution methods in the next two sections.

5.1 Code- and data-centric attribution

Using hardware or software support for address sampling, our tool can accurately attribute costs to both code and data. Our approach for code- and data-centric attribution leverages existing support in HPCToolkit [21].

For code-centric attribution, we determine the call path for each address sample by unwinding the call stack. We then associate NUMA metrics with the call path.

For data-centric attribution, we directly associate metrics with variables, including static variables and dynamically allocated heap data. Our tool identifies address ranges associated with static variables by reading symbols in the executable and dynamically loaded libraries. We identify address ranges associated with heap-allocated variables by tracking memory allocations and frees. Our tool attributes each sampled heap variable accesse to the full calling context where the heap variable was allocated.

5.2 Address-centric attribution

Address-centric attribution provides insight into memory access patterns of each thread. Such information is needed by application developers to help them adjust data distributions to minimize NUMA overhead. Prior data-centric tools, e.g., [15, 21, 24], identify problematic code regions and data objects, but don't provide insight into data access patterns to guide optimization. Below, we first describe a naive address-centric attribution strategy and then introduce refinements to make this approach useful.

> For each memory access *m* to a variable `x`, e.g., an array, we compute the addresses that form the lower and upper bounds of the range accessed by *m* and update the lower and upper bounds of `x` accessed for each procedure along the call path to *m*. At analysis time, we plot the upper and lower bounds of the data range accessed by each thread for any variable in any calling context to gain insight into the data access patterns across threads for code executed in that context.

This approach is too simplistic to be useful. Often data ranges are accessed non-uniformly because of loops, conditionals, and indirect references. For some program regions, a hot variable segment may account for 90% of a thread's accesses, while other segments of this variable only account for 10%. Therefore, instead of computing the minimum and maximum effective addresses for the whole memory range allocated for the variable, we represent a variable's address range with a sequence of bins, each bin representing a subrange. We treat each bin as a separate synthetic variable that gets its own data- and address-centric attributions. As performance measurements are associated with individual bins, one can easily identify the hot bins of a variable. We only use the access patterns of the hot bins to represent the access patterns of the whole variable. It is worth noting that selecting the number of bins for variables is important. A large number of bins for a variable can show fine-grained hot ranges but may ignore some important patterns. Currently, our tool divides a variable with an address range larger than five pages into five bins by default; one can change this number via an environment variable.

The aforementioned approach for maintaining address ranges merges them online to keep profiles compact. However, different memory accesses experience different latencies. For that reason, when analyzing access ranges for variables at different levels of abstraction (statement, loop, procedure, and various levels in the call path), one should not give equal weight to access ranges in all contexts. One can

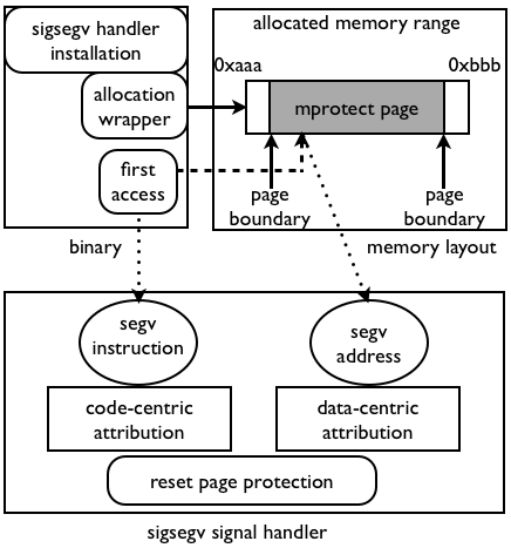


Figure 2. Identifying the first touch context for each heap variable. Our NUMA profiler applies both code-centric and data-centric attribution for first touches.

use aggregate latency measurements attributed to a context as a guide to identify what program contexts are important to consider for NUMA locality optimization, and then use address range information for those contexts as a guide to understand what changes to data and/or code mappings will be needed to improve NUMA performance.

6. Pinpointing first touches

As discussed in Section 2, identifying the source code *first touching* variables associated with NUMA performance losses is essential for optimization. However, manually identifying code performing first touches to variables is often difficult for complex programs. To automatically identify first touches, our tool uses a novel approach that employs page protection in Linux. Our strategy does not require any instrumentation of memory accesses, so it has low runtime overhead.

Figure 2 shows how we trap a first touch access to a large variable, e.g., an array. Our tool first installs a SIGSEGV handler before a monitored program begins execution. Then, our tool monitors memory allocations in the program using wrappers for allocation functions. Inside the wrapper, our tool masks off the read and write permission of the allocated memory range between the first and last page boundaries within the variable's extent. When the monitored executable accesses the protected pages, the OS delivers a SIGSEGV signal. Our tool's SIGSEGV handler catches the signal and does three things. First, it uses the signal context to perform code-centric attribution. Second, it uses the data address that caused the fault (available in the signal info structure) to perform data-centric attribution. With both code- and data-centric attribution of the SIGSEGV signal, one knows the location of first touches to every monitored variable. Finally, it restores read and write permissions for the variable's monitored pages. It is worth noting that multiple threads may initialize a variable concurrently in a parallel loop, so more than one thread may enter the SIGSEGV handler. Thus, multiple threads may concurrently identify first touches and record code- and data-centric attributions. Call paths of first touches to the same variable from different threads are merged postmortemly.

7. Tool implementation

Our tool is implemented as extensions to HPCToolkit [1]—an open-source performance tool for call path profiling of parallel programs. HPCToolkit accepts a compiled binary executable as its input. The binary can be compiled by any compiler with any level of optimization. HPCToolkit launches an executable and then collects per-thread call path profiles, which it attributes to both code and data addresses. HPCToolkit uses an offline analyzer to merge profiles from multiple threads and attribute performance metrics to source code, static variables, and call paths of heap allocated data. Finally, HPCToolkit provides a graphical user interface for exploring performance data. In the rest of this section, we describe how we extended HPCToolkit's measurement, analysis, and presentation tools to support analysis of NUMA performance problems. We refer to our modified version of HPCToolkit for pinpointing and analyzing NUMA bottlenecks as HPCToolkit-NUMA.

7.1 Online profiler

Our extensions to HPCToolkit for NUMA performance analysis perform three tasks. First, they configure PMU hardware for address sampling. We extended HPCToolkit to leverage Perfmon [5] to control PMUs that employ IBS, PEBS and DEAR. Our extensions use Linux `Perf_events` [31] to configure PMUs for architectures that support MRK and PEBS-LL. For software-based sampling, we extended HPCToolkit to override callbacks for instrumentation hooks inserted for loads and stores by LLVM; these instrumentation callbacks record information each time a predefined threshold of accesses occurs.

Second, HPCToolkit's `hpcrun` utility captures these address samples and attributes them to code and data, recording them in augmented calling context trees (CCTs) [21]. The augmented CCT our NUMA extensions record is a mixture of variable allocation paths, memory access call paths, and first touch call paths. Dummy nodes in the augmented CCT separate segments of calling context sequences recorded for different purposes.

Third, the profiler collects NUMA metrics including M_l, M_r, metrics that show the number of sampled accesses touching each NUMA domain, latency metrics, and address-centric metrics that summarize each thread's variable accesses in each subtree of the CCT.

Sampling mechanisms	Processors	Threads	Events	Sampling periods
Instruction-based sampling (IBS)	AMD Magny-Cours	48	IBS op	64K instructions
Marked event sampling (MRK)	IBM POWER 7	128	PM_MRK_FROM_L3MISS	1
Precise event-based sampling (PEBS)	Intel Xeon Harpertown	8	INST_RETIRED:ANY_P	1000000
Data event address registers (DEAR)	Intel Itanium 2	8	DATA_EAR_CACHE_LAT4	20000
PEBS with load latency (PEBS-LL)	Intel Ivy Bridge	8	LATENCY_ABOVE_THRESHOLD	500000
Software-supported IBS (Soft-IBS)	AMD Magny-Cours[1]	48	memory accesses	10000000

Table 1. Configurations of different sampling mechanism on different architectures.

7.2 Offline analyzer and viewer

Adapting HPCToolkit's `hpcprof` offline profile analyzer for NUMA measurement was trivial. The only enhancement needed was the ability to perform [min, max] range computations when merging different thread profiles. Instead of accumulating metric values associated with the same context, [min, max] merging requires a customized reduction function.

We added an address-centric view to HPCToolkit's `hpcviewer` interface to display address-centric measurements for all threads. The view plots the minimum and maximum address accessed for a variable vs. the thread index. The address range for a variable is normalized to the interval [0, 1]. The upper right pane in Figure 3 shows an example of this view for a heap-allocated variable. In the next section, we show how this novel view provides insight that helps guide optimization for NUMA platforms.

8. Experiments

We tested HPCToolkit-NUMA on five different architectures to evaluate its functionality using different hardware and software support for address sampling. Table 1 shows our choices for events and sampling periods for evaluating each of the address sampling mechanisms. The criteria for choosing events is based on (1) sampling every memory access (not only NUMA-related events or instructions) to avoid biased results for access patterns and (2) sampling all instructions (if possible) to support computing NUMA latency per instruction. For the tests we ran, the sampling period we chose for each event except MRK gives 100–1000 samples per second per thread.[2] To evaluate the tool, we used four multi-threaded benchmarks:

- LULESH [16] is a shock hydrodynamics application benchmark from Lawrence Livermore National Laboratory (LLNL) written in C++ and parallelized with OpenMP.
- AMG2006 is an algebraic multi grid benchmark from LLNL's Sequoia benchmark suite [18]. AMG2006 is written in C and parallelized with MPI and/or OpenMP. In this study, we used OpenMP but not MPI.
- Blacksholes is a benchmark from PARSEC benchmark suite [3]. It performs option pricing with Black-Scholes Partial Differential Equation (PDE). It is coded in C and parallelized using OpenMP.
- UMT2013 is a benchmark from LLNL Coral benchmark suite [17]. It performs three-dimensional, non-linear, radiation transport calculations using deterministic methods. UMT2013 is coded in hybrid C, C++, Fortran and parallelized with MPI and OpenMP. Like AMG2006, we only used OpenMP but not MPI in this study.

[1] Soft-IBS works on all of listed platforms; we choose AMD Magny-Cours for testing.

[2] Marked event sampling on POWER7 with the fastest sampling rate under user control generates less than 100 samples/second per thread.

Methods	LULESH	AMG2006	Blacksholes
IBS	295s (+24%)	89s (+37%)	192s (+6%)
MRK	93s (+5%)	27s (+7%)	132s (+4%)
PEBS	65s (+45%)	96s (+52%)	82s(+25%)
DEAR	90s (+7%)	120s (+12%)	73s (+4%)
PEBS-LL	35(+6%)	57s (+8%)	67s (+3%)
Soft-IBS	604s (+200%)	220s (+180%)	270s (+30%)

Table 2. Runtime overhead measurement of HPCToolkit-NUMA. The number outside the parenthesis is the execution time without monitoring and the percentage in the parenthesis is the monitoring overhead. The absolute execution time on different architectures is incomparable because we adjust the benchmark inputs according to the number of cores in the system.

Some NUMA optimization of LULESH and AMG2006 has previously been described in the literature [21]. Guided by insights from HPCToolkit-NUMA, we were able to develop significantly better NUMA-aware optimizations for both LULESH and AMG2006, advancing the state of the art.

Table 2 shows the measurement overhead when running HPCToolkit-NUMA with different architectures with different sampling methods. From the table, we can see that the overhead of HPCToolkit-NUMA differs using different sampling methods. Soft-IBS incurs the highest runtime overhead because it is based on software instrumentation; PEBS incurs the second highest overhead because we compensate for for off-by-1 attribution by the PMU using online binary analysis to identify the previous instruction, which is diffi-

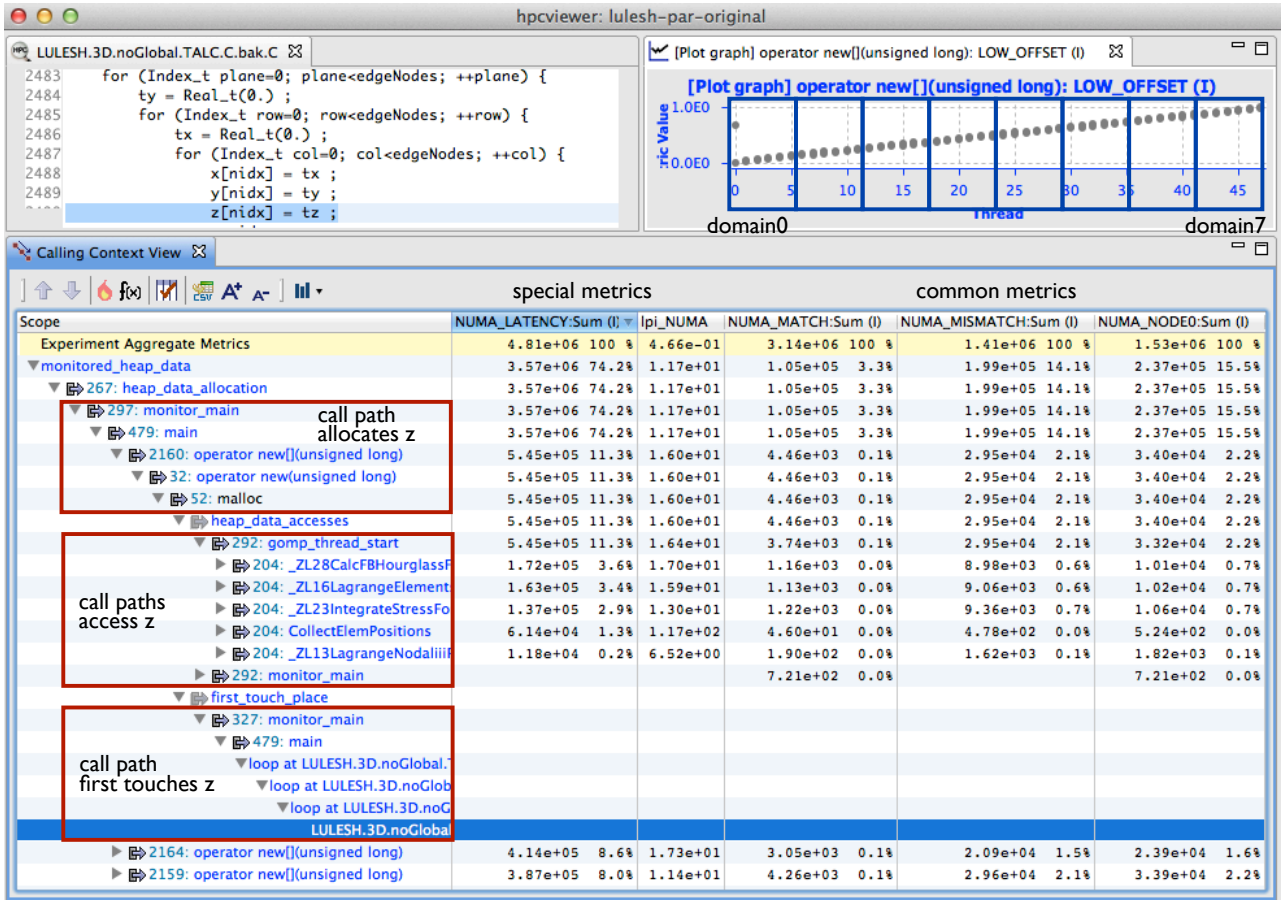


Figure 3. Identifying NUMA bottlenecks in LULESH using code-centric, data-centric and address-centric attributions.

cult for x86 code;[3] IBS has the third highest overhead because it samples all kinds of instructions and its sampling rate is high; and other sampling methods incur very low runtime overhead. At runtime, the aggregate runtime footprint of HPCToolkit-NUMA's data structures was less than 40MB for any sampling method on any of the architectures.

In the case studies that follow, we only show measurements obtained using IBS and MRK because HPCToolkit-NUMA can provide similar analysis results using any sampling method. Moreover, IBS is one of the two PMU hardware types that supports the lpi_{NUMA} metric, while MRK can show how we analyze NUMA bottlenecks using NUMA metrics we derived without the help of latency information. For experiments using IBS, we use a system with four 12-core AMD Magny-Cours processors. Overall, the system has 48 cores and 128GB memory, which is evenly divided into eight NUMA domains. For experiments using MRK, we use a system with four eight-core POWER7 processors. Overall, the system has 128 SMT hardware threads and 64GB memory. In this study, we consider each socket a NUMA domain. We run LULESH, AMG2006 and Blacksholes on all hardware threads; we run UMT2013 with 32 threads because its standard input size is limited to 32 threads.

[3] It would be better to perform this correction during post-mortem analysis.

8.1 LULESH

We first measure LULESH with IBS on the AMD machine. Figure 3 shows the results of our NUMA performance analysis result displayed in a modified version of HPCToolkit's `hpcviewer`. The top left pane shows the source code of the monitored program. The top right pane shows the address-centric view, which represents memory ranges accessed by individual threads. The bottom left pane shows the program structure of synthetic CCTs. Annotations in this pane show a mixture of call paths in the CCT. The bottom right pane shows our NUMA metrics.

The overall program's NUMA latency per instruction (lpi_{NUMA}) is 0.466. This is significantly larger than our 0.1 rule of thumb, which means the NUMA problems in LULESH are significant enough to warrant optimization. We first investigate the heap-allocated variables and then other kinds of variables. The heap-allocated variables have a lpi_{NUMA} of 11.7 and 74.2% of the total latency is caused by remote NUMA domain accesses. We drill down the call path in the bottom left pane for the call sites (`operator`

new[] in the figure) of variable allocations, discovering three variables with more than 8% of total latency caused by NUMA accesses. One can identify the variable names from the source code pane by clicking their allocation sites (marked as 2160, 2164 and 2159 respectively to the left of operator new[]). Here, we study the variable z, which causes the most significant NUMA losses.

Overall, z accounts for 11.3% of the total latency caused by remote accesses. We observe two facts (1) M_r (labeled as NUMA_MISMATCH in the metric pane) is roughly seven times of M_l (labeled as NUMA_MATCH in the metric pane); and (2) all accesses to z come from NUMA domain 0 ($NUMA\_NODE0$ equals to the sum of M_l and M_r). Therefore, we infer that pages for z are all allocated in NUMA domain 0 but accessed by threads in other domains. The top right pane in Figure 3 plots the min and max addresses accessed in z by each thread. From the figure, we can see that other than thread 0, each thread touches a subset of z. Threads with higher ranks touch higher address intervals in z. Visualizing the results of address sampling provides the key insight about how to adjust the layout of heap data to improve performance, namely by co-locating data and computation upon it in the same NUMA domain.

Based on this address-centric view, it is clear that one could improve the NUMA performance of LULESH by dividing the memory range allocated for z into eight continuous regions, segmented by rectangles shown in the top right pane of Figure 3. One should allocate each block to the appropriate NUMA domain so it will be co-located with the threads that access it. We implemented this strategy by adjusting the code that first touches z, identified by our tool, and shown in the top left pane of Figure 3. This optimization exploits the higher bandwidth and lower latency of local memory. It also reduces the bandwidth consumed between NUMA domains. We similarly optimize other heap-allocated variables, including x, y, xd, yd and zd.

LULESH also makes heavy use of a stack-allocated variable nodelist. Since our tool does not currently provide detailed NUMA analysis of stack data, we modified the source code for the program to declare nodelist as a static variable. HPCToolkit-NUMA's data-centric analysis shows that nodelist accounts for 20.3% of total latency caused by remote accesses and 13.3% mismatching of memory accesses (M_r). There is high lpi_{NUMA} associated with nodelist, meaning that it has high NUMA latency that warrants optimization. The M_r metric associated with nodelist is about seven times as large as M_l and all accesses come from NUMA domain 0, which means that nodelist is initialized by the master thread but accessed by worker threads of other NUMA domains in parallel. Address-centric analysis identifies that nodelist has the same access pattern per thread as z does shown in Figure 3. Like optimization for z, such an access pattern reveals that a block-wise distribution of pages allocated for nodelist would be appropriate, as before for z.

Guided by our tool, the block-wise data distribution we implemented for both heap-allocated and stack variables, we were able to speedup LULESH by 25% on our AMD system. Using with the page interleaving strategy suggested by our prior work [21] gave only a 13% improvement over the original code not tuned for NUMA architectures.

Measuring LULESH with MRK on POWER7 showed similar NUMA problems. 66% of L3 cache misses access remote memory. Heap allocated arrays, such as x, y, z, xd, yd and zd, account for 65% of remote accesses, while the stack variable nodelist accounts for 31%. On our POWER7 system, HPCToolkit-NUMA's address-centric view showed that these variables have access patterns similar to those we observed on our AMD system. Using a block-wise page distribution for these variables improved execution time for LULESH by 7.5% on our POWER7 system. In contrast, using an interleaved page allocation (as suggested by prior work [21]) degraded performance on POWER7 by 16.4%.

8.2 AMG2006

We ran AMG2006 with 48 threads on our AMD system, measuring it using IBS. HPCToolkit-NUMA showed that AMG2006 has a lpi_{NUMA} of more than 0.92, which means it has significant NUMA problems (more severe than LULESH) and worthy of investigation. The heap-allocated variables of AMG2006 account for 61.8% of the total memory latency caused by remote accesses. By looking at heap variable allocation call paths, we identified one problematic variable RAP_diag_data. RAP_diag_data accounts for 18.6% of total latency, with a lpi_{NUMA} of 15.9 and 8.1% of total M_r. By examining the sampled accesses and the first-touch access to RAP_diag_data, we found that RAP_diag_data was allocated and initialized by the master thread but accessed by all worker threads in other NUMA domains.

The address-centric view in Figure 4 shows the access patterns of RAP_diag_data across all 48 threads aggregated over the whole program. However, these threads do not show an obvious access pattern that can guide page distribution for this variable. We further investigate threads' access patterns for RAP_diag_data in individual OpenMP parallel regions rather than the whole program. The most interesting parallel region shown in the call path is hypre_boomerAMGRelax._omp, which accounts for 74.2% (13.8/18.6) of NUMA access latency caused by RAP_diag_data. Figure 5 shows the access patterns of RAP_diag_data in this parallel region. Obviously, threads have a regular access pattern of RAP_diag_data in this parallel region. Because accesses in hypre_boomerAMGRelax._omp dominate the costs of accessing RAP_diag_data, we can use this access pattern to direct the data distribution. Like optimization for LULESH, we apply block-wise distribution at the first touch place iden-

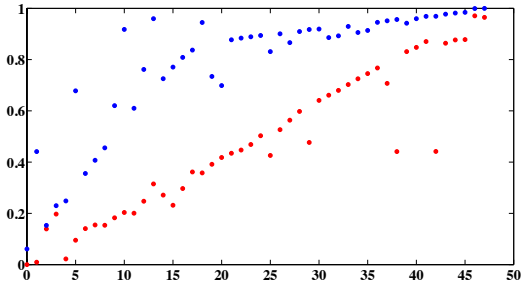


Figure 4. Address-centric view showing the overall access patterns of `RAP_diag_data` in AMG2006 across all threads.

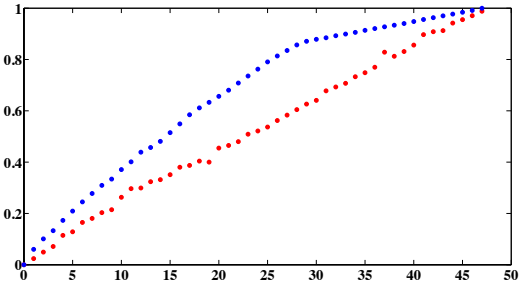


Figure 5. Address-centric view for the accesses to `RAP_diag_data` in the most significant loop in AMG2006.

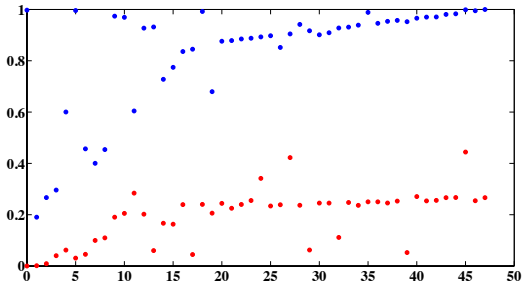


Figure 6. Address-centric view showing the overall access patterns of `RAP_diag_j` in AMG2006 across all threads

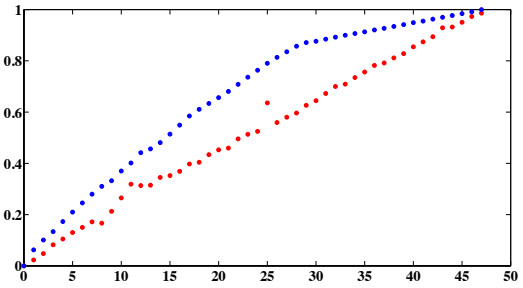


Figure 7. Address-centric view for the accesses to `RAP_diag_j` in the most significant loop in AMG2006.

tified by our tool to evenly allocate `RAP_diag_data` across the NUMA domains.

If we analyze the source code through the code-centric attribution, we find that accesses to `RAP_diag_data` in `hypre_boomerAMGRelax._omp` use the values of another array as indices (i.e., `RAP_diag_data[A_diag_i[i]]`), leading to indirect memory accesses. Without our address-centric analysis analysis, one cannot determine where data layout changes are needed and how to refine them to improve performance.

We examined other hot variables and found another NUMA bottleneck: `RAP_diag_j` accounts for 10.6% of total latency caused by NUMA accesses. Figure 6 and Figure 7 show its address-centric analysis results by considering all accesses and accesses in the most significant parallel region respectively. Obviously, the access patterns on the bottom are much more regular than the one on the top. As memory accesses in this parallel region account for 73.6% of total latency for `RAP_diag_j` in the whole program, we use its regular access pattern to allocate pages of `RAP_diag_j` in a block-wise fashion at its first touch location.

Besides these two variables, there are other three heap-allocated variables suffering from high remote access latency. According to access patterns from address-centric analysis, one of them can be optimized using block-wise distribution as for `RAP_diag_data` and `RAP_diag_j`. The other two show that each thread accesses the whole range of the variable, leading to an optimization of using interleaved page allocation. Our optimizations achieve a 51% reduction in the running time of the solver phase of AMG2006. In production codes that employ this software, the running time of the solver is most important. Without guidance from our address-centric analysis, prior NUMA optimization of AMG2006 used interleaved allocation for every problematic variable [21] , which only improved the solver phase performance of AMG2006 by 36%.

8.3 Blackscholes

We measured Blackscholes on our AMD system using IBS. HPCToolkit-NUMA shows a much smaller lpi_{NUMA} value (0.035 cycle per instruction) than the threshold (0.1) over the entire program, indicating that Blackscholes would not benefit from NUMA optimization. To validate this assessment, we eliminated NUMA bottlenecks in Blackscholes and showed that this optimization barely improved the program's performance.

HPCToolkit-NUMA identified that heap-allocated variables account for 66.8% of total latency caused by NUMA accesses and 51.6% of the latency associated with the variable `buffer`. With the values of M_l and M_r, together with the data source metrics, we identified that `buffer` is allocated in only one NUMA domain by the master thread and evenly accessed by all threads in the system.

HPCToolkit-NUMA's address-centric analysis in Figure 8 shows a regular access pattern across all threads. Each thread touches a sub-range of `buffer` in an ascending order, with large overlaps. To understand why the program reveals such pattern, we analyzed the source code. The top of Fig-

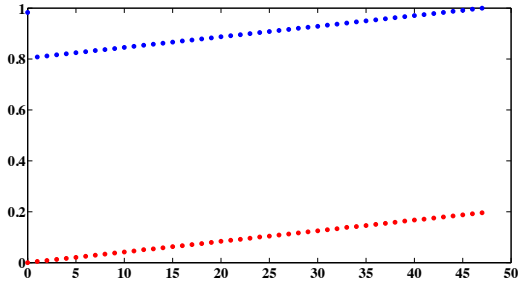
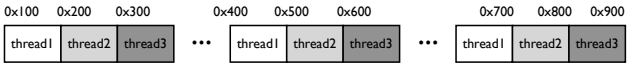
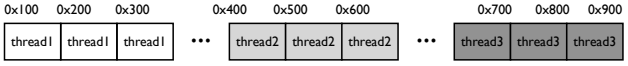


Figure 8. Address-centric view showing the access patterns of `buffer` across all threads.



(a) Original access pattern.



(b) Improved access pattern.

Figure 9. Memory access patterns across threads in BlackScholes.

ure 9 shows the memory layout for `buffer`. The program sets five pointers to point different sections of `buffer`. All threads access to each section in parallel, leading to the access pattern shown in Figure 9a. According to our address-centric analysis, the three example threads in the figure touch the address ranges of (0x100, 0x700), (0x200, 0x800) and (0x300, 0x900) respectively, matching the pattern revealed by address-centric results shown in Figure 8.

As threads show non-local accesses to `buffer`, we regroup these sections into an array of structures as shown in Figure 9b. With this optimization, the three example threads touch (0x100, 0x300), (0x400, 0x600), (0x700, 0x900) respectively, with no overlap.

To allocate data block-wise, we changed the `buffer` initialization loop that our tool identified as the first touch location. Originally, only the master thread initializes `buffer`. We parallelized the initialization loop using OpenMP to make sure that each thread first touches its own data. With this optimization, there is no longer any latency related to `buffer` caused by remote accesses.

Although we largely eliminated the NUMA latency in the program by co-locating data and computation, the execution time of Blackscholes improves less than 0.1%. The trivial improvement proves that our derived metric lpi_{NUMA} reflects the severity of NUMA problems. One can estimate potential gains from NUMA optimization by examining lpi_{NUMA}.

8.4 UMT2013

We ran UMT2013 on our POWER7 system with 32 threads, sampling instructions that cause L3 data cache misses. We

```
do c=1,nCorner
  do ig=1,Groups
    source=Z%STotal(ig,c)+Z%STime(ig,c,Angle)
  enddo
enddo
```

Figure 10. A loop kernel in UMT2013 that has many remote accesses to `STime`.

bounded each thread to each hardware core in each of four NUMA domains. According to M_l and M_r, HPCToolkit-NUMA showed that 86% L3 cache misses lead to remote memory accesses and 47% of remote accesses due to the references of heap allocated variables.

Using HPCToolkit-NUMA, we identified the allocation call path of a hot variable – `STime`, which is a three-dimensional array that accounts for 18.2% of total remote accesses. Code-centric analysis in HPCToolkit-NUMA associates all remote accesses to `STime` with the reference shown in Figure 10. The reason for the high remote access ratio is that `STime` is allocated in one NUMA domain but accessed by threads in all NUMA domains. With address-centric analysis, HPCToolkit-NUMA identifies that `STime` has a staggered access pattern across threads, similar to the variable `buffer` in BlackScholes, shown in Figure 8. A deep analysis of the source code showed that the nested loop iterating over `STime` shown in Figure 10 is in an OpenMP parallel region. Two-dimensional planes of `STime` indexed by `Angle` are assigned to threads in a round-robin fashion. To ensure the data is co-located with its computation, we parallelized the initialization loop of `STime` identified by first touch analysis. This strategy has each thread initialize the part of `STime` that it accesses in the computation stage. This optimization eliminates most remote accesses to `STime` and yielded a 7% speedup for the program as a whole.

9. Related work

Several performance tools provide support for analyzing NUMA performance issues with multi-threaded programs. These tools mainly use two kinds of methods: simulation and measurement. The simulation tools such as MACPO [25] and NUMAgrind [32] collect memory traces and feed into a cache simulator. The simulator simulates an architecture with NUMA memory hierarchies to analyze the memory traces. However, such simulation-based tools incur high runtime overhead to the monitored program. For example, MACPO slows the program by 2x–5x and NUMAgrind has more than 100x runtime overhead.

On the other hand, measurement-based tools can provide insights with low overhead. For example, Memphis [24] uses AMD instruction-based sampling (IBS) to capture remote accesses and associates them with static variables. It compares IBS with the traditional hardware counters, showing

that address sampling gives deeper insights into a program's NUMA bottlenecks.

MemProf [15], another measurement-based tool also uses AMD IBS to measure a program's NUMA bottlenecks. MemProf associates NUMA metrics with heap-allocated variables and partially supports attribution to static variables by coarsely aggregating metrics incurred by static variables in the same load module (executable or shared libraries).

Prior work on data-centric analysis in HPCToolkit used both IBS and MRK hardware support to attribute NUMA latency to both code and data [21]. Here, we extended that work with support for both address-centric and first-touch analysis. While our prior work enabled us to identify variables with the most remote accesses, it didn't provide insight into access patterns, which a code developer needs to reorganize data layout to co-locate it with computation.

Besides NUMA profilers, some previous work improves NUMA locality and performance using support from libraries [4], compilers [23] and operating systems [6, 7]. Unlike our approach, which is designed to help programmers fix NUMA problems in their code, these approaches aim to ameliorate NUMA problems to the greatest extent possible without source code changes. While these approaches require developers to use specific libraries, compilers, or OS kernels, our tool guides offline optimization of the source code which yields better code that can be run anywhere without any restrictions.

10. Conclusions and future work

In this paper, we present a profiling tool that identifies and analyzes NUMA bottlenecks in multi-threaded programs and helps guide program performance tuning by providing new metrics and insights into data access patterns. Using PMU support in modern microprocessors, our measurement-based tool can gather the information it needs with low runtime overhead. We demonstrate the utility of our tool and the information it provides to optimize four well-known benchmarks. While one code didn't warrant NUMA optimization (according to our metrics) or benefit significantly when it was applied anyway, our tool delivered insight and guidance that enabled us to significantly improve the performance of the other three codes.

In our experiments using different hardware for address sampling, we observed that not all mechanisms are equally-suited for our analysis. Although both IBS and MRK sample instructions, IBS samples all kinds of instructions, so one needs to filter out samples not of interest in software, which adds extra overhead. With IBS it is trivial to compute the load/store fraction in the whole instruction stream to assess the performance impact of memory instructions. In contrast, MRK can only sample instructions causing specific events (such as L3 cache misses or remote accesses). Consequently, MRK can highlight problematic memory instruction with low overhead. DEAR, PEBS and PEBS-LL sample events. PEBS and PEBS-LL can directly sample NUMA events, while DEAR does not support NUMA events. Both instruction sampling and event sampling can effectively identify problematic memory accesses. Finally, IBS and PEBS-LL can measure latency for sampled load instructions. This information can be used to derive the metrics described in Section 4.

Our future work is five-fold. First, we plan to add full support for monitoring stack variables instead of requiring them to be changed to static or heap allocated ones for detailed measurements. Second, we plan to extend our tool to analyze more complex access patterns. Third, we plan to collect trace-based measurements to study time-varying NUMA patterns in addition to profiles. Fourth, we plan to augment `hpcviewer` with a new view to better present code-and data-centric measurements. Finally, our strategy for pinpointing first touches is only implemented at present for heap-allocated variables. We plan to extend it for static variables by protecting their pages when the executable or libraries are loaded before execution begins.

Acknowledgments

We thank Laksono Adhianto, Mike Fagan, and Mark Krentel for their contributions to HPCToolkit. Without their efforts, this work would not have been possible. This research was supported in part by Lawrence Livermore National Laboratories by subcontract B602160 of prime contract DE-AC52-07NA27344.

References

[1] L. Adhianto et al. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22:685–701, 2010.

[2] Advanced Micro Devices. AMD CodeAnalyst performance analyzer. `http://developer.amd.com/tools-and-sdks/heterogeneous-computing/archived-tools/amd-codeanalyst-performance-analyzer/`. Last accessed: Jan. 6, 2013.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *Intl. Journal of Parallel Programming*, 38(5-6):418–439, 2010.

[5] H.-P. Corporation. Perfmon kernel interface. `http://perfmon2.sourceforge.net/`. Last accessed: Dec. 12, 2013.

[6] A. Cox and R. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with PLATINUM. In *Proc. of the* 12^{th} *ACM Symp. on Operating Systems Principles*, SOSP '89, pages 32–44, New York, NY, USA, 1989.

[7] M. Dashti et al. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proc. of the*

18^{th} *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013.

[8] L. DeRose, B. Homer, D. Johnson, S. Kaufmann, and H. Poxon. Cray performance analysis tools. In *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, 2008.

[9] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. `http://developer.amd.com/Assets/AMD_IBS_paper_EN.pdf`, November 2007. Last accessed: Dec. 13, 2013.

[10] IBM Corporation. IBM Visual Performance Analyzer User Guide, version 6.2. `http://bit.ly/ibm-vpa-62`. Last accessed: Dec. 12, 2013.

[11] Intel VTune Amplifier XE 2013. `http://software.intel.com/en-us/intel-vtune-amplifier-xe`, April 2013. Last accessed: Dec. 12, 2013.

[12] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, Volume 3B: System programming guide, Part 2, Number 253669-032, June 2010.

[13] Intel Corporation. Intel Itanium Processor 9300 series reference manual for software development and optimization, Number 323602-001, March 2010.

[14] A. Kleen. A NUMA API for Linux. `http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf`, 2005. Last accessed: Dec. 12, 2013.

[15] R. Lachaize, B. Lepers, and V. Quéma. MemProf: a memory profiler for NUMA multicore systems. In *Proc. of the 2012 USENIX Annual Technical Conf.*, USENIX ATC'12, Berkeley, CA, USA, 2012.

[16] Lawrence Livermore National Laboratory. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). `https://codesign.llnl.gov/lulesh.php`. Last accessed: Dec. 12, 2013.

[17] Lawrence Livermore National Laboratory. LLNL Coral Benchmarks. `https://asc.llnl.gov/CORAL-benchmarks`. Last accessed: Dec. 12, 2013.

[18] Lawrence Livermore National Laboratory. LLNL Sequoia Benchmarks. `https://asc.llnl.gov/sequoia/benchmarks`. Last accessed: Dec. 12, 2013.

[19] X. Liu and J. Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proc. of the 9th IEEE/ACM Intl. Symp. on Code Generation and Optimization*, pages 171–180, Washington, DC, 2011.

[20] X. Liu and J. Mellor-Crummey. Pinpointing data locality bottlenecks with low overheads. In *Proc. of the 2013 IEEE Intl. Symp. on Performance Analysis of Systems and Software*, Austin, TX, USA, April 21-23, 2013.

[21] X. Liu and J. M. Mellor-Crummey. A data-centric profiler for parallel programs. In *Proc. of the 2013 ACM/IEEE Conference on Supercomputing*, Denver, CO, USA, 2013.

[22] LLVM Compiler Infrastructure. `http://www.llvm.org`. Last accessed: Jan. 7, 2013.

[23] Z. Majo and T. R. Gross. Matching memory access patterns and data placement for NUMA systems. In *Proc. of the* 10^{th} *IEEE/ACM Intl. Symp. on Code Generation and Optimization*, pages 230–241, New York, NY, USA, 2012.

[24] C. McCurdy and J. S. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proc. of 2010 IEEE Intl. Symp. on Performance Analysis of Systems Software*, pages 87–96, Mar. 2010.

[25] A. Rane and J. Browne. Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In *Proc. of the* 12^{th} *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, USA, 2012.

[26] Rogue Wave Software. ThreadSpotter manual, version 2012.1. `http://www.roguewave.com/documents.aspx?Command=Core_Download&EntryId=1492`, August 2012. Last accessed: Dec. 12, 2013.

[27] S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.

[28] B.-W. Silas et al. Corey: an operating system for many cores. In *Proc. of the* 8^{th} *USENIX conference on Operating Systems Design and Implementation*, pages 43–57, Berkeley, CA, USA, 2008.

[29] B. Sinharoy et al. IBM POWER7 multicore server processor. *IBM JRD*, 55(3):1:1–29, May 2011.

[30] M. Srinivas et al. IBM POWER7 performance modeling, verification, and evaluation. *IBM JRD*, 55(3):4:1–19, May/June 2011.

[31] V. Weaver. The unofficial Linux Perf Events web-page. `http://web.eece.maine.edu/~vweaver/projects/perf_events`. Last accessed: Dec. 12, 2013.

[32] R. Yang et al. Profiling directed NUMA optimization on Linux systems: A case study of the Gaussian computational chemistry code. In *Proc. of the 2011 IEEE Intl. Parallel & Distributed Processing Symposium*, pages 1046–1057, 2011.