Enhancing Performance Optimization of Multicore Chips and Multichip Nodes with Data Structure Metrics

Ashay Rane
Texas Advanced Computing Center
The University of Texas at Austin
ashay.rane@tacc.utexas.edu

James Browne
Department of Computer Science
The University of Texas at Austin
browne@cs.utexas.edu

ABSTRACT

Program performance optimization is usually based solely on measurements of execution behavior of code segments using hardware performance counters. However, memory access patterns are critical performance limiting factors for today's multicore chips where performance is highly memory bound. Therefore diagnoses and selection of optimizations based only on measurements of the execution behavior of code segments are incomplete because they do not incorporate knowledge of memory access patterns and behaviors. This paper presents a low-overhead tool (MACPO) that captures memory traces and computes metrics for the memory access behavior of source-level (C, C++, Fortran) data structures. It also presents a complete process for integrating code segment-based and memory access pattern measurements and analyses for performance optimization specifically targeting multicore chips and multichip nodes of clusters. MACPO explicitly targets the measurement and metrics important to performance optimization for multicore chips. MACPO uses more realistic cache models for computation of latency metrics than those used by previous tools. Evaluation of the effectiveness of adding memory access behavior characteristics of data structures to performance optimization was done on subsets of the ASCI, NAS and Rodina parallel benchmarks and one application program from a domain not represented in these benchmarks. Adding memory behavior characteristics enabled easier diagnoses of bottlenecks and more accurate selection of appropriate optimizations than with only code centric behavior measurements. The performance gains ranged from a few percent to 38 percent.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Performance

Keywords

performance, optimization, memory, data structures

1. INTRODUCTION

Performance optimization requires detailed characterization of the causes of and remedies for performance bottlenecks. Out of the four subtasks of performance optimization (measurement, diagnosis of bottlenecks, selection of effective optimizations, and implementation of optimizations), the measurement, diagnosis and selection of effective optimizations subtasks have, to date, largely focused on code centric measurements, which associates resource use with code segments such as functions or loops. Because memory is much slower than processors, a full understanding of the memory access patterns of important data structures is critical to accurate diagnosis of performance bottlenecks and selection of performance optimizations. A complete approach to performance optimization must therefore combine code-centric and data structure-centric measurements and analyses with code-segment local scope. Understanding memory access patterns becomes even more important for optimization of execution on multicore chips which share memory access paths such as L3 caches and chip-local memory across processing cores. Data structure-oriented measurement and analysis are also important for identifying code segments which can execute efficiently on the SIMT/SIMD accelerators, which are commonly being associated with multicore chips in computation nodes. Several excellent code segment-centric measurement and analysis tools are available. Some tools for data structure access-centric measurement and analyses are also available. However, none of the available tools for measurement of data structure access patterns and latencies provided the required data for full support of performance optimization for multicore chips. Therefore a major requirement for our goal was a data structure access behavior measurement and analysis tool which operates at the resolution of loops and functions. This paper presents the design, implementation and application of such a tool (MACPO, Memory Access Centric Performance Optimization) for measurement and analysis of data structure access patterns and latencies which provides the specific data needed for performance optimization of multicore chips. The paper also specifies and illustrates a complete process for combining code-centric and data structure-centric measurements and analyses for performance optimization for multicore chips and multichip nodes of cluster systems.

The process is implemented and illustrated by combining the measurements and analyses of the MACPO tool with the diagnostic and optimization capabilities of PerfExpert [6,13,27] which is based on code segment-based resource use measurements.

The contributions and innovations in this paper include:

1. An efficient, low-overhead and easy to use tool which resolves measurements onto data structure behavior on code segments, including extensions to incorporate the requirements for performance optimization for multicore chips and multichip nodes

2. A systematic process for combining code segment centric and data structure-centric execution behavior measurements for performance bottleneck diagnosis and identification of optimizations

3. Illustrations of the complete performance optimization process incorporating data structure memory access behaviors including both substantial application codes and examples from standard parallel benchmarks.

The case studies include illustration of the additional diagnostic and optimization capabilities afforded by the data structure behavior measurements. In many cases, adding memory behavior characteristics of data structures enabled easier diagnoses of bottlenecks and more accurate selection of appropriate optimizations than with only code centric behavior measurements.

The rest of the paper is organized as follows. Section 2 covers related research and tools but focuses mainly on previous work on data structure-centric measurement. Section 3 lists the key innovations in this work. Section 4 describes the design and implementation of the MACPO tool. Sections 5 and 6 describe and illustrate the complete analyses with the results of applying these analyses to eight codes. Section 7 recaps the content and suggests conclusions which can be drawn from this research. Section 8 describes future research directions including application to identification of code segments which will execute efficiently on SIMD accelerators such as GPGPUs.

2. RELATED WORK

We limit related work coverage to tools and systems for data structure measurement and behavior analysis and the use of data structure measurements in performance optimization. There are several previous or existing tools which focus on measurement and analysis of memory behaviors. All have some limitations with respect to comprehensive support for effective and efficient performance optimization for multicore chips and multichip nodes. In [16], the authors present an approximation to reuse distance analysis that takes $O(log\ log\ M)$ time for processing every memory reference, where M is the number of unique memory references made by the program. However, their solution only applies to single-core reuse distance calculation. We extend their reuse distance calculation to allow modeling of multicore reuse distances. A probabilistic model for determining the concurrent reuse distance based on the data locality of standalone programs is described in [19]. This approach does not take into consideration interactions between threads arising in a non-simulated environment. Schuff et al. demonstrate a technique in [25, 26] for modeling multicore reuse distances however due to a restricted cache model, their method does not take into account multiple levels of cache. Their approach suffers from over-estimation of penalties from multicore interactions and it does not appear to scale as the number of threads grows. Chameleon [29] incorporates memory trace collection using the PIN dynamic binary instrumentation tool [24], as does [25]. Chameleon is built with the objective of generating and replaying synthetic traces to evaluate cache architectures. TAXI [28] is an X86 simulation environment that collects traces using the Bochs [21] simulator but it does not include the high-level trace analyses (estimated cycles per accesses, access strides, NUMA & cache conflicts). All of the above-mentioned approaches, with the exception of [16] for which there is little information regarding the implementation, use either simulators or PIN. Simulators may not precisely model real-life scenarios such as cache thrashing and prefetching. The use of simulators often greatly increases the time to obtain the results because of the software emulation of instructions. PIN has a wide user base because of its ease of use. However, relating source-level data structures with trace analysis using PIN requires that the binary contain debug information (-g compilation flag), which may introduce some degree of perturbation of memory access patterns [2, 3] during program execution. ThreadSpotter [9] is a commercial tool that runs analyses on memory traces without requiring source-level instrumentation. However, to be able to relate measurements back to source code, the binary has to be compiled with debug information. Instead, as described later in Section 4, we instrument our code using LLVM [20] in such a way that the debug information is not required to be present in the binary. Since the instrumented code is compiled to native code, the binary can be run directly on the target platform. SLO, as described in [11, 12] uses a modified version of GCC to instrument programs to collect information about runtime reuse paths and analyze them for locality. Based on the analysis, SLO suggests source code optimizations. However, SLO restricts itself to reuse distance analysis. MACPO, in addition to reuse distance analysis, computes the other metrics needed for complete performance optimization such as average latency for source code data structures, conflicts at the cache and package levels and strides in accesses. Using Instruction Based Sampling (IBS) [15] in AMD processors, Liu and Mellor-Crummey [22] monitor latency of load and store instructions and attribute these to source-level data structures. Their approach has significantly low overhead as compared to most other approaches. However, the only metric available to the user to tune his or her code is the latency of accessing the data structure, which in reality is the symptom but not the cause of the degradation. Itzkowitz, et.al. [18] also profiled memory access characteristics using performance counters. However the value of performance counters in diagnosing the cause of the memory latency is limited; for instance level-3 cache misses could imply (without certainty) large working set size, exceeding memory bandwidth, cache thrashing, etc. Although these are useful metrics, these require that the programmer understand both the source code and the execution environment to identify the true cause of the latency (strided access, cache line invalidation, etc.).

3. INNOVATIONS

This section amplifies the three bullets in the Introduction on the contributions of this paper. It gives details of the measurement and analysis processes which differentiate

MACPO from previous data structure access behavior tools. The last subsection specifies how the code-centric and data-centric measurements and analyses are synergistically combined.

3.1 Low-overhead, code segment-local data structure access behavior measurement

The MACPO tool relates trace information collected about the program back to source-level (C, C++, Fortran) data structures, along with its location in the source code (file name and line number), without requiring the binary to be compiled with debug information (-g compilation flag). This makes it possible to incorporate the memory access behavior of each data structure separately in the diagnosis of bottlenecks and selection of optimizations. Source-level information is obtained during the instrumentation process. MACPO has been designed to keep the overhead of instrumentation low. MACPO collects data only on the data structures in those code segments which have been identified as performance bottleneck. We employ various techniques such as periodically disabling the instrumentation and spacing out the "windows" of instrumentation asymmetrically, whenever possible using atomic data structures instead of locks, etc. Also, MACPO only instruments arrays, unions and structures unlike tools that instrument all memory accesses. While a precise comparison is not possible, we believe MACPO has a measurement overhead of an order of magnitude lower compared to most other tools that use instrumentation [9,25,29], that is, about 2-6x for MACPO (as compared to the average slowdown of 30x for the fastest existing tool [25] according to our survey). We believe that this overhead can be further reduced by employing non-blocking IO operations.

3.2 Use of more accurate cache models

Previous reuse distance-based analyses in most other tools almost universally assume that caches are fully associative. MACPO uses a probabilistic model of set-associative caches based on [23] and incorporates models of each level of cache in the architecture including whether it is private to a core or shared. Incorporating latencies of the individual caches allows a more accurate analysis. As an example, using information about sharing of caches and latencies makes it possible to include the effects of cache thrashing into the memory analysis. This information also makes it possible to answer questions such as what percentage of memory accesses were likely served from the individual caches or how many memory requests were for local memory and how many for remote memory, which can be very helpful in tuning programs on NUMA configurations.

3.3 Multicore reuse distance calculation with low book-keeping overhead

Modeling the reuse distance in the MACPO tool is performed using a technique similar to that described in [16]. However, the idea is adapted for multicore cases by maintaining time-stamps for each cache. This results in $O(M)$ storage for every cache but maintains $O(N \cdot logM)$ time for reuse distance calculation, where N is the number of accesses and M is the number of unique cache lines that were accessed. Depending on the cache that is referenced, the appropriate counter is chosen for finding the reuse distance.

3.4 Stride calculation

The calculation of dominant strides for the program data structures, although simple, is very effective in quickly identifying cases of inefficient data layout. The difference in the address of the last access and the most recent access for each data structure is used for the stride calculation.

3.5 Performance optimization process

The general procedure adopted for tuning all applications using PerfExpert and MACPO is outlined below. Depending on the complexity of the problem, we often found it useful to repeat the entire process. For all codes we first ran PerfExpert on them. PerfExpert identifies the code segments which are performance bottlenecks in the code. It also aggregates various performance counter-based measurements to yield high-level metrics such as cycles spent in data accesses per executed instruction. Information about architectural parameters including memory access latencies is collected during the installation of PerfExpert and is used while calculating the high-level metrics. PerfExpert evaluates program performance on the basis of Local Cycles Per Instruction (LCPI), which is essentially a per-category metric of the cycles per instruction. For instance, the data access LCPI represents the cycles spent by the code segment in data accesses for each of its instructions. This data access LCPI is further divided into the LCPI arising from accessing each level of the data caches (L1, L2 and possibly L3). Programs exhibiting high data access LCPI may be suffering from problems like cache thrashing, capacity misses, etc. Apart from using the data access LCPI, we also consider the data translation look-aside buffer (TLB) LCPI (CPU cycles spent in accessing the data TLB for every instruction in the code segment). Such programs may include data structures that are accessed with long or irregular strides. Once functions that showed poor memory access behaviors were identified, we instrumented these functions using MACPO. Information obtained from the various analyses built in MACPO was combined to manually devise optimizations. 'NUMA hit ratios' and 'cycles per accesses' gave insight into problems arising from multi-threading. High 'stride' values generally indicated inverted loops. Although high 'reuse distances' directly indicates poor locality, correcting this problem usually required understanding the source code in detail. To verify if the change in the application source code indeed improved the total running time of the application, we ran un-instrumented versions of both pre-optimization and post-optimization codes using the Intel 11.1 compiler. The MACPO metrics and the running time for both naïve and optimized versions of the code are shown when discussing the results in Section 6.

4. DESIGN, IMPLEMENTATION AND VALIDATION

In this section, we describe the processes for measurement, generation of metrics, discovering optimizations using MACPO, the limitations of the measurements and validation of the measurements.

4.1 Instrumentation

Capturing data structure access patterns begins with intercepting memory accesses made by the program. There are various ways of solving this problem. We chose compiler-based instrumentation over simulation or binary instrumentation for the following reasons:

1. Compile-time instrumentation enables greater control over the instrumentation process (e.g. only instrumenting specific arrays) because we can operate on the intermediate representation (IR) generated by the compiler prior to the machine-code generation phase.

2. Binary instrumentation tools deal with individual machine instructions. It can be difficult to correlate multiple instructions to loop structures, boundaries and data structure-related operations such as typecasting.

3. The binary generated from compile-time instrumentation runs directly on the target architecture. On the other hand, simulators for complex multicore architectures may not be complete enough to capture advanced hardware optimization techniques (for instance, branch prediction, prefetching). Using simulators has the additional drawback of greatly increasing the time required to obtain results.

Considering these factors, we chose to use the LLVM compiler infrastructure. We added a separate compiler pass in LLVM's extensible framework to implement our instrumentation.

The instrumentation process has the following steps:

1. **Eliminate scalar variables from being instrumented:** Runs the '-mem2reg' pass which promotes local stack variables (other than structs, unions and arrays) to SSA registers, thus leaving non-scalar types, global variables and heap-allocated memory intact for further processing

2. **Run our compiler pass:** This inserts calls to a statically linked library before every access to an array, structure or union in the specified function(s)

3. **Remove debug information:** Runs the '-strip' pass on the IR to remove debug information from the program.

The call in (2) to the runtime library has the prototype:

```
void recordMemoryAccess(
        var_number, address,
        read/write access, file_number,
        line_number);
```

The `var_number`, `file_number` and `line_number` fields are used to relate the address to a location in the source code. The type of access (`read/write`) is used in estimating the latency of serving the memory request. The transformed IR is then compiled to native (x86) code to produce the executable binary.

4.2 Trace collection and analysis

Execution of the instrumented program generates a trace of memory addresses for the portions of the code that were instrumented. These logs are analyzed offline to minimize perturbations in program execution which would result from an online analysis. Most programs of interest for performance optimization are long running applications that typically have repeating patterns. We leverage this to lower overhead by capturing "snapshots" or windows of program execution instead of the entire program behavior from start to finish. Note that metrics from each of the snapshots could have been collected from different invocations to the same function and hence each of the snapshots have to be processed independently. To try and make these snapshots capture the program behavior during its various phases, the time for which the instrumentation is disabled is changed throughout the program's duration and is made to follow the Fibonacci number sequence $(1, 1, 2, 3, 5, 8, 13, 21, \cdots)$. The result is that the number of snapshots does not increase linearly with the running time. This has another advantage of reducing the size of the trace logs in comparison with a full-program trace recording.

4.3 Accuracy of measurements and validation

Instrumentation necessarily modifies the execution behavior of the program. We have been very careful to minimize this perturbation and to test the computed metrics for accuracy.

Instrumentation changes the program behavior: Through compile-time instrumentation, the program performs additional tasks like book-keeping and disk IO for writing logs to a file. Instrumentation at the level of the compiler IR makes it certain that the memory address traces recorded by the instrumentation are the same (relative to base addresses) as will have been generated by the uninstrumented binary. This ensures that the stride and reuse distance metrics should be the same for both the instrumented and un-instrumented programs. The instrumentation does impact both register use and cache use and, of course, increases the running time of the program. However this change does not influence the MACPO measurements because the instrumentation records only the memory references made by the user's source code and not those made by the instrumentation itself. We observed that the instrumented binary ran between 1.2 to 5.5 times slower (Table 1) than the un-instrumented binary. We are able to reduce the overhead and the amount of generated trace information from instrumentation by sampling, by periodically disabling the instrumentation, whenever possible using atomic data structures instead of locks, limiting the collected entries to 1 million per sampling interval and using the '-mem2reg' LLVM pass to eliminate scalar variables from being instrumented. Sampling windows are triggered using `SIGPROF` and are "active" till either the sampling buffer is full or till the sampling period (250ms) is over, whichever occurs first.

To estimate variability in output metrics, we ran the applications multiple times and with different numbers of threads. The computed metrics showed less than 10% percent variation. The sampling rates were varied as were the sizes of the data structures (and thus the running time of the program) to determine sampling rates sufficient to insure accurate measurement in nearly all cases. Additionally the reuse distances and strides computed by MACPO are consistent with those derived manually for simple code examples.

Instrumentation breaks certain compiler optimizations: Typically, optimizations like loop vectorization can no longer be applied after instrumentation because of the newly inserted function calls. As the instrumentation pass operates on the IR, it needs to be run before the code gen-

Table 1: Running time of instrumented code

Application	Running time
SRAD	2.75x
Back Propagation	1.25x
LU	3x
BT	4.66x
Sweep3D	5.52x

eration phase. To ensure that the instrumentation causes minimum possible hindrance to the applicable compiler optimizations, we run the instrumentation pass just before the machine code generation phase. This implies that only those optimizations that are applied after IR generation may not be compiled into the instrumented program.

5. ANALYSES

The analyses built into MACPO operate at the resolution of cache line sizes (usually 64-bytes chunks) instead of a single byte- or word-resolution. We term our resulting analyses as "physical" reuse distance and "physical" strides as these values give a true account of the hardware. We distinguish these physical measurements from the "logical" measurements that refer to the source code data types to determine reuse distance and strides.

MACPO records the complete 64-bit memory address for each reference to the data structures together with the `read` or `write` tag and the ID of the core making the reference. It is thus possible to compute the logical reuse distances and strides at the byte or word resolution. However, we have chosen to have MACPO compute reuse distances and strides in terms of the size of a physical fetch from memory (usually the size of the cache lines in the architecture) since this more accurately reflects the cost for satisfying the reference to a memory address.

To illustrate, consider the following code fragment:

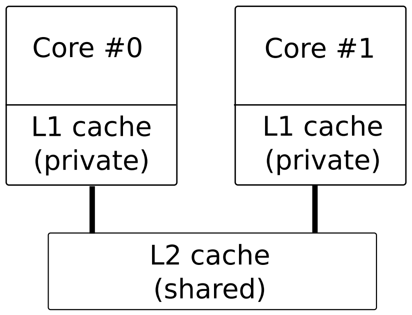
```
long i, a[16];
...
for (i=0; i<16; i++)
    a[i] = a[i]*10;
```

Looking at the source code, one can infer that the (logical) stride for the array `a` is 1. For this case, lets assume that each long integer consumes 8 bytes and that the size of the cache line is 64 bytes. Hence a single cache line can pack up to 8 long integers in one cache line.

Thus while executing the above code the hardware would see the memory references as a single one (`a[0]`) followed by 7 accesses to the same cache line (`a[1]`, `a[2]`, `a[3]`, `a[4]`, ..., `a[7]`) and one again (when the code accesses the first `long`, `a[8]`, in the next 64-byte chunk which causes another fetch from memory). Modeling at such a resolution of cache lines works closely in tandem with the functioning of the architecture and hence gives a realistic picture of the performance of the application.

The different analyses that we have built into MACPO are sketched below. We relate each of the following metrics to source-level data structures.

Figure 1: Example cache organization for memory access streams shown in Tables 2 and 3



5.1 Multicore reuse distance analysis

Reuse distance is a measure of the number of unique memory accesses made by a program between accesses to a given memory location. The definition of reuse distance changes slightly in the context of multiple cores to account for shared or private references to data items. Also, as explained earlier, MACPO computes reuse distances at a coarser resolution (cache line size) instead of byte- or word-size. As an example consider a stream of references (Table 2) made by two processing cores with a two-level cache organization shown in Figure 1. Here, since neither core references any common data, the reuse distance for data '`a`' for core #1 would be based on all references made by core #1 only. Thus the reuse distance would be based on the number of data references seen by the L1 cache of core #1. Consequently, the reuse distance for '`a`' is 2. However, if both cores performed writes to any common data, then the reuse distance calculation cannot consider the streams of each core in isolation. Consider the case in Table 3, that differs in the core that accessed '`a`' for the second time. In this case, the reference to data '`a`' made by core #2 will cause the fetch to be performed from the cache that is common to both core #1 and core #2. Consequently, MACPO, which records the core which made the reference, computes the reuse distance for '`a`' to be 4.

5.2 Non-Uniform Memory Access (NUMA) hit ratios

On machines that have memory affinity enabled, there can be a significant penalty arising from naïve patterns of allocation and reference of data structures. Threads from a different processor package may incur a high latency from remote fetches while accessing the same memory. References by a thread/process executing on one processor package to a memory location attached to a different processor package

Table 2: Memory access stream with no conflicting accesses

Data	**a**	b	c	b	d	e	**a**
Core#	**1**	2	1	2	2	1	**1**

Table 3: Memory access stream with conflicting accesses

Data	**a**	b	c	b	d	e	**a**
Core#	**1**	2	1	2	2	1	**2**

are called NUMA accesses [4]. Using the CPUID instruction [17], the memory addresses being written to the log are tagged with the ID of the core making the memory access request. During the measurement phase we record the core ID, and thus the package, that made the last request for any given cache line. If it was found that the last request to the cache line originated from a different processor package, we classify this request as a NUMA access. In summary, if a write operation is being performed on a remotely-homed memory location, then we classify that access as a NUMA access. This metric can be extended to measure the approximate conflicts among cores within a processor package.

5.3 Cycles per access

Reuse distance represents program behavior but does not take into account latencies arising from conflicting accesses to memory from different cores. The most common case is cache thrashing (two or more cores, each with a private cache, repeatedly accessing the same memory location, thus invalidating other core's cache line), where the reuse distance could be low but the penalty of accesses is high. We therefore decided to model latencies separately from reuse distances. Estimating cycles for memory accesses is derived from the reuse distance analysis, the NUMA hit ratio analysis and from architectural parameters such as cache organization, sizes and latencies. The reuse distance analysis tells us how many unique cache lines were accessed in between accesses to the same data item. Assuming a probabilistic model of set-associative caches [23], we can then estimate whether a given memory access would still be served out of L1 or would overflow the size of the cache, resulting in the memory access being served out of a higher (L2 or possibly L3) level of cache. On a NUMA miss, we associate the remote fetch latency with the memory access. Actual latencies to each cache line are them summed up and divided by the number of requests to yield the average cycles per access.

5.4 Access strides

Programs that have unit strides or small stride values generally execute faster than programs that have long or irregular access strides. This is because the hardware prefetchers can recognize patterns in data accesses and bring data into caches before it is referenced, thus reducing data access penalty. Virtual address to physical address translation can also be serviced more efficiently (using TLBs) when the code exhibits unit strides. Since we have an association between the memory address requested and the source data structure name, we can, for each data structure and thread, keep track of the differences in last requested address and the currently requested address, which is used as the stride value.

6. RESULTS

To evaluate the effectiveness of using MACPO and PerfExpert, we used programs from three benchmark suites: NAS Parallel Benchmarks (size B) [10], ASCI Sweep3D benchmark [8] and the OpenMP benchmarks from the Rodinia benchmark suite [14]. We also included a standalone application, a 2D Lattice Boltzmann solver, in the codes that were measured. Four codes (Sweep3D, NAS Block Tridiagonal, NAS LU Symmetric Gauss-Seidel and Lattice Boltzmann) are MPI codes whereas the remaining (Particle Filter, Needleman-Wunsch, Back Propagation and SRAD) are multi-threaded programs that use OpenMP. In all cases, understanding the cause of the degradation and tuning the code simply based on code segment-based measurements was difficult due to the following two main reasons:

1. Counter-based measurements were helpful in discovering the problems but not the causes

2. Code segments commonly used multiple data structures making it difficult to determine the exact bottleneck data structure

For half of the codes, we found the data from MACPO to be critical to characterizing the causes of the performance bottlenecks and devising optimizations. These optimizations included non-trivial optimizations such as changing the OpenMP loop chunk size, discovering a bad algorithm and changing it, partial loop fusion and in another case, loop fission. For the balance of the codes, information from MACPO was useful in quickly determining the cause of the problem and hence the optimization but the MACPO analysis was not critical to discovering the optimization.

The experiments were done on two architecturally different machines. Programs from the Rodinia benchmark suite were run using 8 threads on Longhorn [5], which is comprised of dual quad-core Intel Nehalem chips. Each node has 48 GB of main memory. Each of the 8 cores have 32 KB level 1 data and instruction caches. A unified level 2 cache of size 256 KB is shared between two cores while a unified level 3 cache of size 8 MB is shared by four cores. The Lattice Boltzmann solver, the ASCI Sweep3D benchmark and programs from the NAS parallel benchmarks were measured using 8 threads and tuned on Ranger [7]. Each node of Ranger consists of four quad-core AMD Barcelona processors and contains 32 GB of main memory. Each core has its private L1 data and instruction caches of size 64 KB each. Unlike the Intel Nehalem processor, AMD Barcelona processor supports a per-core unified L2 cache of size 512 KB. The L3 cache (2 MB) is per-socket and is therefore shared among the four cores [1]. The choice of selecting a particular machine for any program was purely random. For brevity, we

variable: CDF (**avg_dist**: 3.85, **count**: 524174, **avg_cpa**: 78.53, **numa_hit_ratio**: 52.38)

```
  dist  cpa  use ln#  reuse ln#    count     file:
     0  150        0        290    17010   ex_particle_OPENMP_seq.c
     1  250      290        290   129964   ex_particle_OPENMP_seq.c
     1    4      290        290   328165   ex_particle_OPENMP_seq.c
```

variable: matrix (**avg_dist**: 0.37, **count**: 50996, **avg_cpa**: 95.25, **numa_hit_ratio**: 100.00)

```
  dist  cpa  use ln#  reuse ln#    count     file:
     1    4      185        185    19122   ex_particle_OPENMP_seq.c
     0  150        0        185    31874   ex_particle_OPENMP_seq.c
```

Figure 2: Excerpt of trace analysis output for naïve version of Particle Filter code

include part of the MACPO output for one of the programs (Particle Filter) in Figure 2.

6.1 Codes for which MACPO data was critical

6.1.1 ASCI Sweep3D

The Sweep3D benchmark is part of the ASCI benchmark suite and solves the 3D Cartesian geometry discrete ordinates neutron transport problem. PerfExpert showed that most of the data access LCPI (3.7) was arising from L1 accesses (2.9). The code in question was a loop with its body of the form:

```
  do i = 1, it
    flux(i,j,k,2) = ...
    flux(i,j,k,3) = ...
    flux(i,j,k,4) = ...
  enddo
```

MACPO showed a very high dominant stride (3906 cache lines) for the `flux` data structure. The strided accesses either forced the compiler to not vectorize the loop or caused the processor to read data from memory (instead of it prefetching them). We thus split the loop into three loops, each containing a single statement. Apart from having the intended effect of accessing the array using unit strides, this also caused the compiler to vectorize the loops. The optimized code reduced the dominant stride to 1, reduced the data access LCPI to 2.6 (with 1.8 as the L1 access LCPI) and caused the total running time to decrease by 24%.

6.1.2 NAS LU Symmetric Gauss-Seidel

This code is a non-linear PDE solver. The LU code uses a symmetric successive over-relaxation solver kernel. PerfExpert showed both data accesses and data TLB to be a problem. PerfExpert also showed that the last level cache contributes the most to the data access LCPI value. Data structure analysis showed that the reuse distances for two arrays (`flux` and `u`) were high (approximately 15 and 11 cache lines respectively). We found that a few of the loops that used these arrays could be fused, thus promoting greater reuse. Measurements on the revised code showed that the reuse distance for `flux` and `u` reduced to 11 and 8 cache lines respectively. Post optimization, the running time for the code was reduced by 8%.

6.1.3 Particle Filter

The Particle Filter code tracks an object across video frames. For this code, as shown briefly in Figure 2, we noticed that two data structures (`CDF` and `matrix`) had very high estimated cycles per access (78, 95 respectively). For `matrix`, it was reported that a majority of the accesses were not being fetched from the L1 cache and that this data structure did not use unit strides. For the other data structure (`CDF`), we saw the NUMA hit ratio to be about 50% (indicating that remote fetches probably contributed to the latency). The code in question has all threads performing a linear search over a single large sorted array. To make each core operate on separate portions of the array without significant refactoring of the code, the code was changed to keep track of the successful search index from the previous iteration. The resulting analysis showed that cycles per accesses for `matrix` dropped to 13 while the number of accesses to `CDF` dropped to an insignificant number. On large inputs (40K particles), this resulted in the entire program taking about a quarter of its original running time.

6.1.4 Needleman-Wunsch

This code is a parallel implementation of a dynamic programing solution. In this code, iterations (size of the diagonal) are distributed equally among the cores by the OpenMP runtime. PerfExpert showed that the L1 data access LCPI was high but unlike most other codes, there were fewer last level cache misses. This eliminated capacity misses from the possible causes of the problem. Memory analysis showed that access to the `input_itemsets` array was becoming a problem (reuse distance as high as 13 cache lines). Combining the information that L1 cache access LCPI was high and that all threads access the same array in a loop, one can infer that the multicore reuse distance for this array is affected by simultaneous use by different cores, thus causing contention among processors. To reduce this distance we focused on increasing the affinity between each core and its data. With the objective of ensuring that shuffling of data among cores does not occur too often, we increased the OpenMP chunk size of the iteration distribution to 32. This results in at least 32 consecutive iterations being assigned to a single core, thus increasing reuse within a single cache. The optimized code shows about 4% improvement and the reuse distance for the data structure is decreased from 13 cache lines to 9 cache lines.

6.2 Codes for which MACPO data was not critically important

For the analyses discussed in this section, code centric performance counter measurements enabled identification of appropriate optimizations.

6.2.1 NAS Block Tridiagonal

The Block Tridiagonal code is another non-linear Partial Differential Equation (PDE) solver written in Fortran. This code includes three main routines [`x_solve()`, `y_solve()`, `z_solve()`] that together take about 40% of the total time. All three of them, especially `z_solve()`, have particularly bad data access LCPI values with the last level cache contributing most to the data access LCPI. Data structure analysis showed that two arrays (`fjac` and `njac`) had a reuse distance of about 9 cache lines but were accessed with a unit stride. Similarly another array (`lhs`) had a reuse distance of about 7 cache lines. Inspecting the code showed that in each iteration of the loop body, all three arrays were accessed in a way that the most frequently changing index was not the first one. Changing the data layout reduced the reuse distance for all arrays to 4 cache lines and the running time decreased by 13%.

6.2.2 Lattice Boltzmann

This Fortran code is an MPI-parallel two dimensional Lattice Boltzmann solver for the Navier-Stokes equation. Running PerfExpert on this code showed last level cache misses contributing about a third of the data access LCPI. For the same routine, through data structure measurements, we found that one of the arrays (`f`) had a relatively high reuse distance (7 cache lines). Moreover, MACPO reported that this array was often (about 2,700 times) being accessed with a stride of 2 cache lines. Inspecting the code revealed the multiple elements of this array were being accessed in the loop body in the following form:

```
... = f(i,j,0,now) + f(i,j,1,now) +
      f(i,j,2,now) + ...
```

The index that changes with each access to the array is not the first one hence we were seeing non-zero stride values. Modifying the code so that the first index changed most often caused MACPO to report unit strides with reuse distance dropping from 7 cache lines to 6 cache lines. The running time reduced by 13%.

6.2.3 Back Propagation

This machine learning code is used to determine weights in a layered neural network. Through performance counters, we learned that L1 and L2 accesses dominated the data access LCPI. The L2 miss LCPI was low indicating few capacity misses. Trace analysis on this code showed high cycles per access (64) and relatively high reuse distances (5 cache lines) for two arrays (`w`, `oldw`). Inspection of the code showed that the arrays were being accessed in a strided manner. Such an access pattern affects the processor's ability to prefetch data into the cache. The code in question is a doubly-nested loop in C, similar to the code shown below:

```
loop i = 1 to n
  loop j = 1 to n
    array[j,i] = ...
```

Assuming that program correctness is maintained, a better way to rewrite the same code is by inverting the two loops. Such an inversion for the loop in the Back Propagation code reduced the cycles per access to 5, reuse distance to 2 cache lines and the code showed 17% improvement in total time.

6.2.4 SRAD (Speckle Reducing Anisotropic Diffusion)

SRAD uses Partial Difference Equations for ultrasound imaging. PerfExpert showed that the LCPI for TLB accesses was significantly high. This suggested frequent accesses with long strides. We also noticed that the `image` array had a high reuse distance (21 cache lines) and a high access latency (20 cycles). Stride analysis showed that this array was accessed with an average stride of 32 cache lines. Code inspection showed that the loops were inverted, causing strided accesses to the data structures. Changing the loop order reduced the reuse distance to 1 cache line and access penalty to 13 cycles, ultimately reducing the total time by 38%.

7. CONCLUSIONS

We described the design, implementation and innovations of the MACPO tool which specifically targets generation of the memory access characteristics of data structures required for performance optimization for multicore chip and multichip node execution. We described a process for performance optimization which integrates code-centric measurement with measurements and analyses of data structure memory access characteristics. Illustrations of the enhanced process demonstrate the value of integrating code-centric and data structure-centric analyses and specifically demonstrate the additional capabilities for diagnosis and selection of optimizations enabled by adding data structure-centric analyses to the performance optimization process.

8. FUTURE WORK

Knowledge of the memory access patterns and resolving the cost of data accesses to data structures has a significant role to play in the future of performance optimization. We plan to incorporate this knowledge into the optimization component of PerfExpert [27]. For example, reuse and stride data can be used to automate selection of the optimal tile/block size for loop tiling. An important performance optimization may be mapping of code segments to accelerators which execute in SIMT/SIMD parallelism. The process which combines code segment centric and data structure memory behavior centric measurements can be extended to determine code segments that can readily be converted to execute in SIMT/SIMD mode. The critical properties which can be readily determined from multicore chip/multicore node execution include absence of divergent branches, scalable SPMD parallelism, small and regular access strides and high reuse factors for the data structures in the code segments. Additionally the data from MACPO can guide translation from C, C++ or Fortran to accelerator languages such as CUDA or OpenCL.

9. ACKNOWLEDGEMENTS

This work is funded in part by the National Science Foundation under OCI award #0622780.

10. REFERENCES

[1] AMD Barcelona Processor Cache Architecture. `http://developer.amd.com/documentation/articles/pages/8142007173.aspx`.

[2] GCC 4.6.2 manual. `http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/`.

[3] Intel C Compiler Manual. `http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/compiler_c/copts/common_options/option_fp_lcase.htm`.

[4] Linux support for NUMA hardware. `http://lse.sourceforge.net/numa/faq/`.

[5] Longhorn User Guide. `http://www.tacc.utexas.edu/user-services/user-guides/`.

[6] PerfExpert. `http://www.tacc.utexas.edu/perfexpert`.

[7] Ranger User Guide. `http://www.tacc.utexas.edu/user-services/user-guides/`.

[8] The ASCI Sweep3D Benchmark Code. DOE Accelerated Strategic Computing Initiative. `http://www.c3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html`.

[9] ThreadSpotter. `http://www.roguewave.com/`.

[10] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. NAS Parallel Benchmark Results 3-94. *Proceedings of the Scalable High Performance Computing Conference*, pages 386–393, 1992.

[11] K. Beyls and E. D'Hollander. Discovery of locality-improving refactorings by reuse path analysis. *High Performance Computing and Communications*, pages 220–229, 2006.

[12] K. Beyls and E. H. D'Hollander. Refactoring for Data Locality. *Computer*, 42(2):62–71, 2009.

[13] M. Burtscher, B. D. Kim, J. Diamond, J. Mccalpin, L. Koesterke, and J. Browne. PerfExpert : An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In *Computer*, pages 1–11. IEEE, 2010.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *2009 IEEE International Symposium on Workload Characterization IISWC*, 2009(c):44–54, 2009.

[15] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: hardware support for instruction-level profiling on out-of-order processors. *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 292–302, 1997.

[16] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[17] Intel. Intel Processor Identification and the CPUID Instruction. *Journal On The Theory Of Ordered Sets And Its Applications*, (August), 2009.

[18] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche. Memory profiling using hardware counters. In *In Supercomputing Conference (SC*, pages 17–30, 2003.

[19] Y. Jiang, E. Zhang, K. Tian, and X. Shen. Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors? In *Compiler Construction*, pages 264–282. Springer, 2010.

[20] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization 2004 CGO 2004*, (c):75–86, 2004.

[21] K. Lawton. Bochs IA-32 Emulator Project, 2004.

[22] X. Liu and J. Mellor-Crummey. Pinpointing Data Locality Problems Using Data-centric Analysis. *CGO*, pages 171–180, 2011.

[23] G. Marin. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *In Proceedings of the Symposium of the Las Alamos Computer Science Institute, Sante Fe*, 2005.

[24] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. *Computer Architecture*, 2004.

[25] D. L. Schuff, M. Kulkarni, V. S. Pai, and W. Lafayette. Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization. *Measurement*, pages 53–63, 2010.

[26] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. *Measurement*, page 8 pp., 2010.

[27] O. A. Sopeju, M. Burtscher, A. Rane, and J. Browne. AutoSCOPE : Automatic Suggestions for Code Optimizations using PerfExpert. *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2011.

[28] S. Vlaovic and E. S. Davidson. *TAXI: Trace Analysis for X86 Interpretation.* PhD thesis, University of Michigan, 2002.

[29] J. Weinberg and A. Snavely. Chameleon: A framework for observing, understanding, and imitating the memory behavior of applications. In *PARA08: Workshop on State-of-the-Art in Scientific and Parallel Computing, Trondheim, Norway.* Citeseer, 2008.