Enhancing Performance Optimization of Multicore/Multichip Nodes with Data Structure Metrics

ASHAY RANE and JAMES BROWNE, University of Texas at Austin

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 article 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. MACPO explicitly targets the measurement and metrics important to performance optimization for multicore chips. The article also presents a complete process for integrating measurement and analyses of code execution with measurements and analyses of memory access patterns and behaviors for performance optimization, specifically targeting multicore chips and multichip nodes of clusters. 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 Rodinia parallel benchmarks and two versions of one application program from a domain not represented in these benchmarks. Adding characteristics of the behavior of data structures 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

Additional Key Words and Phrases: Performance, optimization, memory, data structures

ACM Reference Format:
Rane, A. and Browne, J. 2014. Enhancing performance optimization of multicore/multichip nodes with data structure metrics. *ACM Trans. Parallel Comput.* 1, 1, Article 3 (May 2014), 20 pages.
DOI: http://dx.doi.org/10.1145/2588788

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 based on performance counters. These analyses associate 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

This work is funded in part by the National Science Foundation under OCI award #0622780.
Authors' address: A. Rane (corresponding author) and J. Browne, Department of Computer Science, University of Texas at Austin, 1616 Guadalupe, Suite 2.408, Austin, TX 78701; email: ashay.rane@utexas.edu.

DOI: http://dx.doi.org/10.1145/2588788

optimizations. A complete approach to performance optimization must therefore combine code-centric and memory-centric measurements and analyses with a scope limited to specific code segments. Understanding memory access patterns becomes even more important for optimization of execution on multicore chips that share memory access paths such as L3 caches and chip-local memory across processing cores. SIMT/SIMD accelerators are commonly being associated with multicore chips in computation nodes. Data structure-oriented measurement and analysis are also important for identifying code segments that can execute efficiently on the SIMT/SIMD accelerators. Several excellent code segment-centric measurement and analysis tools are available. Some tools centered around measurement and analyses of memory access patterns are also available. However, none of the available tools for measurement of memory access patterns and latencies provided all the required data for full support of performance optimization for multicore chips. Therefore a major requirement for our goal was a measurement and analysis tool for memory access patterns that operates at the resolution of loops and functions. This article presents the design, implementation and application of such a tool (MACPO, Memory Access Centric Performance Optimization) for measurement and analysis of memory access patterns and latencies. MACPO provides the specific data needed for performance optimization of multicore chips. The article also specifies and illustrates a complete process for combining code-centric and memory-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[1] [Burtscher et al. 2010; Sopeju et al. 2011]. PerfExpert is based on resource use measurement of code segments.

The contributions and innovations in this article include the following.

(1) We define the role of and requirements for data structure access metrics in performance optimization for multicore chips and multichip nodes. We include reuse factors and number of streams in a loop nest into our metrics, neither of which, to the best of our knowledge, have been previously used for systematic performance optimization. A reuse factor is the number of times a cache line is accessed after it has been loaded and before it is evicted.
(2) We provide an efficient, low-overhead and easy to use tool that resolves measurements of data structure behavior on code segments, including extensions to incorporate the requirements for performance optimization for multicore chips and multichip nodes.
(3) We present a systematic process for combining performance information of code segments and data structures for bottleneck diagnosis and identification of optimizations and case studies illustrating this process.

The case studies include illustration of the additional diagnostic and optimization capabilities afforded by the measurements of data structure behavior. 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 article is organized as follows. Section 2 covers related research and tools but focuses mainly on previous work on measurement of memory access patterns. Section 3 details the contributions and innovations in this article. Section 4 describes the design and implementation of the MACPO tool. Sections 5 and 6 describe and

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

illustrate the complete analyses with the results of applying these analyses to eight codes. Section 7 recaps the content and suggests conclusions that can be drawn from this research. Section 8 describes future research directions including application to identification of code segments that will execute efficiently on SIMT/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 that 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. Ding and Zhong [2003] 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. Jiang et al. [2010] describe a probabilistic model for determining the concurrent reuse distance based on the data locality of standalone programs. This approach does not take into consideration interactions between threads arising in a non-simulated environment. Schuff et al. [2010a, 2010b] demonstrate a technique 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 [Weinberg and Snavely 2008] incorporates memory trace collection using the PIN dynamic binary instrumentation tool [Reddi et al. 2004], as do Schuff et al. [2010a]. Chameleon is built with the objective of generating and replaying synthetic traces to evaluate cache architectures. TAXI [Vlaovic and Davidson 2002] is an X86 simulation environment that collects traces using the Bochs [Lawton 2004] 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 Ding and Zhong [2003] 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). using the -g compilation flag may introduce some degree of perturbation of memory access patterns[2] during program execution. ThreadSpotter[3] 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 [Lattner and Adve 2004] 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 [Beyls and D'Hollander 2006, 2009] uses a modified version of GCC to instrument

[2]GNU `gcc` manual: http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/, Intel `icc` manual: http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/copts/common_options/option_fomit_frame_pointer.htm.

[3]ThreadSpotter: http://www.roguewave.com/.

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 including reuse factors, average latency for source code data structures, conflicts at the cache and package levels, strides of accesses, NUMA hit ratios and the number of streams in loop nests. Using Instruction Based Sampling (IBS) [Dean et al. 1997] in AMD processors, Liu and Mellor-Crummey [2011] 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. The latency of memory accesses is, in reality, the symptom but not the cause of the degradation. Itzkowitz et al. [2003] 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, they 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 provides details of the three bullets in Section 1 on the contributions of this article. It gives details of the measurement and analysis processes which differentiate MACPO from previous tools that measure data structure access behavior. The last subsection specifies how the code-centric and data-centric measurements and analyses are synergistically combined.

3.1. Data Structure Metrics Useful for Performance Optimization

The performance gap between DRAM chips and the CPU forces close attention to memory-related optimizations to achieve good scalability. Modern processors include multiple sophisticated means (caches, Translation Lookaside Buffers (TLBs), and prefetchers) to try to overcome as much of this gap as possible. As a result, applications can be made to scale well if they can effectively leverage these hardware provisions. However, a prerequisite to the optimal use of these hardware resources is knowing how well they are currently being used and thus how much potential for more efficient use remains. Processor performance metrics can give the relative efficiency of memory access in a code segment but determining the potential for more efficient memory access in that code segment requires knowledge of the access patterns of the data structures in the code segment. Specifically, patterns of memory access by data structure have the potential to provide insight into the software's use of the memory access hardware while, at the same time, aggregating the information to a level that can be easily understood by a human and/or a knowledge based analysis system.

The metrics characterizing memory access that MACPO computes characterize the current access patterns and enable comparison to the optimal access patterns leading to more efficient use of caches, TLBs and prefetchers. Reuse distance (a measure of the number of unique memory accesses made by a program between accesses to a given memory location) and reuse factor (count of the number of times a cache line is reused in a specific level of the cache before eviction) provides an understanding of the locality of data structures. The number of streams (distinct array references) used within a loop allows estimation of the prefetcher effectiveness. Non-UniformMemory Access (NUMA) hit ratios specify the chip-level locality of memory accesses. Cache conflicts specify the occurrence of multiple threads processing overlapping spans of

one or more data structures. Access strides determine the span of memory accesses by data structure which impacts the frequency of cache and TLB hits and misses. MACPO records memory traces, computes these metrics and analyzes them to understand how to improve memory access behaviors in the user's application.

3.2. 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 data structures in those code segments that have been identified as performance bottleneck. We employ various techniques such as periodically disabling the instrumentation and spacing out the "windows" of instrumentation asymmetrically and using atomic data structures instead of locks when possible to minimize overhead in trace collection. 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 custom instrumentation [Schuff et al. 2010a; Weinberg and Snavely 2008]. MACPO has an overhead of about 3x-6x as compared to the average slowdown of 30x for the fastest existing tool [Schuff et al. 2010a] according to our survey. We believe that this overhead can be further reduced by employing non-blocking IO operations or by using optimized tracing frameworks[4].

3.3. Performance Optimization Process

The general procedure adopted for tuning all applications using PerfExpert and MACPO is outlined in the following. 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 that 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). LCPI 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 such as cache thrashing and capacity misses. Apart from using the data access LCPI, we also considered 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

[4]Linux Trace Toolkit - next generation (LTTng): http://lttng.org/.

'cycles per accesses' gave insight into problems arising from multi-threading. High 'stride' values generally indicated inverted loops. Although high 'reuse distances' directly indicate 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 uninstrumented versions of both preoptimization and postoptimization 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 compile time instrumentation using the LLVM compiler infrastructure. We added a separate compiler pass in LLVM's extensible framework to implement our instrumentation to the LLVM IR.

The instrumentation process has the following steps.

(1) *Eliminate scalar variables from being instrumented.* Runs the '`-mem2reg`' pass. This pass 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 calls functions in 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 following form.

```
void recordMemoryAccess(
  var_id, address,
  read/write access,
  file_id, line_number
);
```

The `var_id`, `file_id` 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 that 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 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 uninstrumented programs. The instrumentation does impact both register use and cache use and, of course, increases the running time of the program. However these changes do 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 13% to 450% slower (Table I) than the uninstrumented 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% 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

Table I. Running Time and Space Overhead for Instrumented Codes

Application	Execution time (sec)	Instrumented execution time (sec)	Lines of code	Inst. lines of code	Log size (MB)
2D (MPI) Lattice Boltzmann	104	217	2292	113	25
3D (OpenMP) Lattice Boltzmann	109	144	2131	243	0.9
ASCI Sweep3d	135	713	1973	658	50
NAS BT	248	1371	5326	1214	109
NAS LU	175	531	5403	455	144
Rodinia Back Propagation	31	35	676	22	6.4
Rodinia SRAD	82	341	691	352	45
Needleman Wunsch	58	68	253	162	1.6

strides computed by MACPO are consistent with those derived manually for simple code examples.

Instrumentation breaks certain compiler optimizations. Typically, optimizat0ions 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 generation 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]`,

Table II. Memory Access Stream with No Conflicting Accesses

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

Table III. Memory Access Stream with Conflicting Accesses

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

`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 in the following. We relate each of the following metrics to source-level data structures.

5.1. Multicore Reuse Distance and Reuse Factor

Data locality has a significant impact on the performance of applications on current architectures. In this section, we present two metrics, reuse distance and reuse factor, that help to understand the locality of accesses to data structures.

5.1.1. Multicore Reuse Distance. 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 II) made by two processing cores, each having a private L1 cache and a shared L2 cache. 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 III, 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 that made the reference, computes the reuse distance for 'a' to be 4.

5.1.2. Multicore Reuse Factor. Reuse distance analysis tells us how many unique cache lines were accessed in between accesses to the same data element. Assuming a probabilistic model of set-associative caches [Marin 2005], 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. This analysis permits us to then estimate the multicore reuse factor. The multicore reuse factor is a count of the number of times a given cache line is reused in a specific level of the cache.

We use the cache size, cache line count and associativity of all caches on the system under test to calculate the multicore reuse factor. These parameters are obtained by probing the OS via the `/sys/devices/system/cpu/` file system. An easy (although inefficient) way to calculate cache performance is to keep track of all lines in all caches and match the cache lines with a specific address from the memory. Some trace-based cache analyzers use this approach. Previous research [Uhlig and Mudge 1997] has found the overhead of trace-based memory analyses to be prohibitively high. Instead we employ the technique used by Marin and Mellor-Crummey [Marin 2005] to identify the level of the cache that most likely served the request. Using this technique, we are able

to determine the required performance metrics without increasing storage requirements or processing time significantly.

5.1.3. Using Reuse Metrics for Performance Optimization. Reuse distance tries to provide an architecture-agnostic viewpoint of the locality of data structure accesses. A higher value of the reuse distance indicates poorer locality. Multicore reuse distance enhances this metric by including limited information about the cache organization. Unlike reuse distance, which is an unbounded metric, reuse factor provides an intuitive sense of the performance of the application or it's data structures. For instance, we noticed that our optimization to the ASCI Sweep3D application caused the L1 reuse factor for a data structure to grow significantly. We were thus able to easily reason about the performance of the data structure.

5.2. Stream Count

We define a stream to be a regular sequence of small increments in either positive or negative direction in a small memory region. Streams are generated by references to data structures such as arrays in loop nests. Stream counts in loop nests have not, as far as we are able to determine, been computed or used as a metric by any other performance optimization system previous to MACPO.

5.2.1. Streams and Performance. The importance of the number of streams in a loop nest for performance of a memory system is due primarily to hardware prefetchers for caches. A hardware prefetcher analyses a sequence of memory references. When addresses follow a regular sequence of increments in either positive or negative direction, the prefetcher identifies the existence of a stream and begins prefetching consecutive blocks in the same direction. Prefetchers are most effective when accessing contiguous memory locations. A multicore chip can recognize a fixed number of simultaneously accessed streams. Therefore each core can effectively prefetch only a limited number of streams. The memory access pattern, the thread count and the number of arrays that are accessed in a loop nest (the number of streams), all affect the prefetcher's performance. Knowing the number of active streams in a loop nest can assist in formulating code structures that maximize effective memory bandwidth and minimize memory latency.

```
1 PARALLEL DO i=1 to n
2   W(i) = a1(i) + a2(i) + a3(i) +
       a4(i) + a5(i) + a6(i) + a7(i)
        + a8(i) + a9(i) + a10(i)
3 ENDDO
```

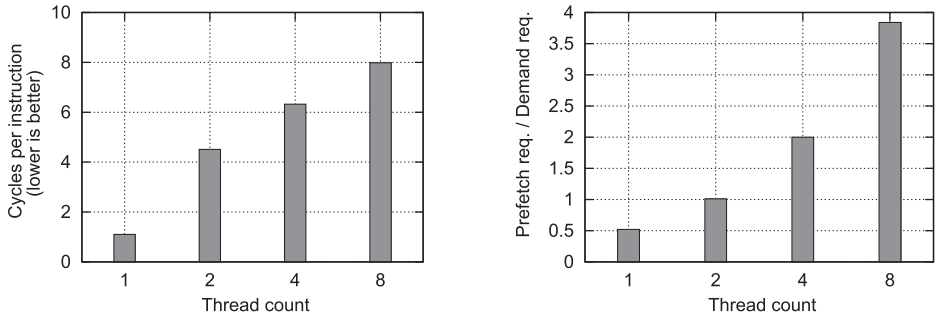
Listing 1. Loop with multiple streams

```
1 PARALLEL DO i=1 to n
2   W(i) = a1(i) + a3(i) + a5(i) +
       a7(i) + a9(i)
3 ENDDO
4
5 PARALLEL DO i=1 to n
6   W(i)+= a2(i) + a4(i) + a6(i) +
       a8(i) + a10(i)
7 ENDDO
```

Listing 2. Split loops

Consider the loop shown in Listing 1, where the largest value of n in an[i] may exceed the number of recognizable streams by the processor. Even a simple kernel as that shown in Listing 1 may scale very poorly with increasing threads due to memory bandwidth limitations. Figure 1(a) shows the performance of this loop kernel on an Intel Sandy Bridge processor in terms of cycles per instruction. Note that increasing the thread count provides no benefit in performance.

Figure 1(b) plots the ratio of prefetch request count to the demand read request count as a function of number of threads. Prefetch request count is measured using the performance event L2_RQSTS:ALL_PF and demand read request count is measured



(a) Performance of loop kernel for increasing thread count

(b) Contribution of prefetch requests to consumed memory bandwidth

Fig. 1. Performance metrics for loop kernel shown in Listing 1.

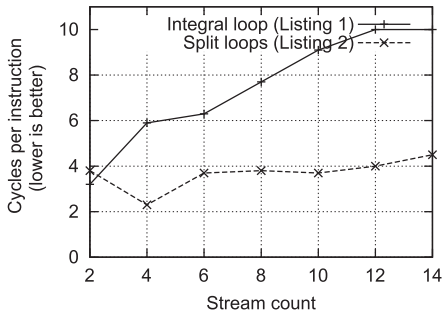


Fig. 2. Effect of loop fission on cycles per instruction. Kernels under test model code shown in Listings 1 and 2.

using `L2_RQSTS:ALL_DEMAND_DATA_RD`. The former counts the number of requests from the L2 hardware prefetcher while the latter counts demand load requests and L1 hardware prefetch requests. A high value of this ratio implies greater contribution from the prefetcher towards the memory bandwidth consumed by the application. The plot demonstrates that a growing amount of the memory bandwidth consumption results from prefetching requests. When this inference is combined with the data from Figure 1(a), we can see that although prefetching is generally indispensable for optimizing memory accesses, it is degrading performance for the given loop kernel. The optimized version of this kernel (shown in Listing 2), splits the single loop kernel into two loop kernels, each iterating over one half of the streams.

The plot in Figure 2 shows the performance of each of these versions of the loop as the number of memory streams is varied by increasing the number of arrays referenced in the loops on an Intel Sandy Bridge processor. The code that operates on the same number of memory streams over two loops sustains much greater performance for a high stream count. The plot in Figure 2 also provides a useful indication of the optimal number of streams in a loop. Note that the fissioned code contains two separate loops, hence the number of instructions executed in the fissioned loop is greater than those in the non-fissioned loop but the performance of the fissioned loops is still much better than the single loop.

The 3D Lattice Boltzmann code (described later in Section 6.2.2) is an application that illustrates the need for stream counts. The Lattice Boltzmann code references two different segments of 23 different one dimensional arrays in a single loop nest. Each of the arrays has one index variable with a constant stride leading to more than

46 streams. While there is good L1 cache reuse on the dimension with unit increment, the number of streams is far greater than the number of streams that can be efficiently recognized and prefetched by the processor.

5.2.2. Computation of Stream Metric. When compiling the application, MACPO can recognize loop boundaries and count the number of variables being referenced within the loop to determine the stream count. Since this analysis is flow- and context-insensitive, MACPO overestimates the number of streams that may actually be referenced in the code at runtime. Note that the use of the '`-mem2reg`' pass before instrumentation by MACPO leaves only heap- and global-variables in the LLVM IR. We prune this list of remaining variables based on debug metadata to exclude scalar variables and analyze only arrays, structures and unions. When compiling the application, MACPO can recognize loop boundaries and count the number of variables being referenced within the loop to determine the stream count.

5.3. Non-Uniform Memory Access (NUMA) Hit Ratios

The memory subsystem of current generation processors is spread over multiple levels, forming a hierarchy. Each level of the memory hierarchy varies in the speed of access and the manufacturing cost, thus influencing the available memory size. A typical memory hierarchy starts with the registers, includes L1, L2 and L3 caches then the locally-managed memory (i.e., memory managed by the processor's memory controller) followed by the node-wide memory and distributed memory (i.e., memory on other nodes). Note that performance optimization is usually focused on the use of caches, node-wide memory and on distributed memory, mostly neglecting intranode but remote memory accesses. In this section, we motivate an understanding of intranode remote memory accesses and the need to consider them while optimizing performance.

5.3.1. NUMA Hit Ratios and Performance. A process is said to "own" a portion of the memory if any thread running on the processor is the first one to access the given portion of memory. Every processor chip includes a memory controller to manage the processor's owned memory. While memory owned by another processor can be accessed by any processor on the same node, the request for such nonlocal (or remote) memory imposes a cost penalty. This cost model leads to a Non-Uniform Memory Access (NUMA) cost. Note that accesses to local memory are also termed as a NUMA hits while accesses to remote memory are termed as NUMA misses.

NUMA hits and misses typically manifest themselves in high-level source code as shown in Listing 5.3.2. Here, thread #0 initializes memory (`data`). As it is the first among all threads to access '`data`', the processor that runs thread #0 owns the memory associated with the '`data`' array. However, when each thread processes it's local chunk (`data[i]`), it accesses the memory that is owned by the processor running thread #0. If a given thread '`t`' does not run on the same chip that runs thread #0, '`t`' will refer to '`data[threadID]`' as a remote memory location, incurring NUMA access penalty.

5.3.2. Computation of NUMA Hit Ratios. Measurement of the number of requests to remote memory from the total set of memory accesses is non-obvious due to difficulties involved in programming the required performance counters. We overcome this problem by tagging memory addresses being written to the log with the ID of the core making the memory access request. During the measurement phase we record the core ID, and thus the chip, 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 chip, we classify this request as a NUMA miss. In other words, if a write operation is being performed on a remotely-homed memory location, then we classify that access as a NUMA miss.

```
1  int* data;
2
3  tid = get_thread_id();
4  if (tid == 0)
5  {
6    // initialize memory here
7    init(data);
8  }
9
10 barrier;
11 ...
12
13 // each thread processes local
       memory
14 process(data[tid]);
```

Listing 3. Source code pattern causing NUMA misses

5.4. Cache Conflicts

Cache conflicts arise when multiple processors, each with at least one private cache, repeatedly claim exclusive access to a portion of memory. Such a situation may arise due to false sharing (illustrated in Listing 4). Here, each thread writes to (logically) separate portions of the `data` array. However multiple array elements may map to the (physically) same cache line. To maintain coherency in the presence of such sharing, processors exchange messages on the bus to request exclusive copies of cache lines for writing and to notify other processors about the state of cache lines. Frequent exchange of such messages causes a significant performance penalty.

```
1
2  char data[1024];
3
4  #pragma omp parallel
5  {
6      int tid = omp_get_thread_num();
7      int count = omp_get_num_threads();
8      for (i=tid; i<1024; i+= count)
9          data[i]++;
10 }
```

Listing 4. False sharing causing performance penalty from cache conflicts

5.4.1. Computation of Cache Conflicts. Cache conflicts can be difficult to detect using either static analysis or using performance events. The difficulty in static analysis arises because data access patterns that cause cache conflicts may not be recognizable at the time of compilation. Hardware performance events such as cache misses may indicate that the code suffers from cache conflicts. However, as cache misses can occur due to other reasons as well (e.g., capacity misses and low reuse), hardware performance events cannot be used to accurately diagnose cache conflicts. As MACPO knows the memory addresses accessed by each thread, it can easily identify memory accesses from different cores that write to the same cache line, thus identifying cache conflicts.

5.5. 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 false sharing, where the reuse distance could be low but the penalty of accesses

is high. We therefore decided to model latencies separately from reuse distances. This metric allows an easy understanding of the poor-performing data structures in the application.

5.5.1. Computing Cycles per Access. Cycles for memory accesses are derived from the reuse distance analysis, the NUMA hit ratio analysis and from architectural parameters such as cache organization, sizes and latencies. As explained in Section 5.1, given the reuse distance, we can 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. Thus, the appropriate penalty can be added to the calculated cost of the memory access. 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 (CPA).

The calculation of the CPA metric makes certain simplifying assumptions about the processor. In particular, we assume that the processor does not support out-of-order instruction execution and that it does not prefetch data into the caches (i.e., it does not fetch data into caches before the data is requested by the instruction). This implies that MACPO presents a conservative value for the CPA metric. Based on our knowledge, no public-domain processor simulator exists that can model the prefetching logic and out-of-order instruction execution of an Intel Sandy Bridge processor or an Intel Westmere processor (our test platforms for MACPO). Hence we are not able to verify the deviation in performance numbers between the actual cost of a memory accesses versus that reported by MACPO. However, we have verified that among micro-kernels that use just one dominant data structure, the CPAs reported by MACPO vary in proportion with the CPAs reported by PerfExpert, which are derived from performance counter measures.

5.6. Access Strides

A stride is the distance in address space between two consecutive references to the same data structure. MACPO computes the access strides in units of cache line size.

5.6.1. Access Strides and Performance. Programs that have unit strides or small regular stride values generally execute faster than programs that have long or irregular access strides. There are several factors giving better memory access performance. Since data is typically fetched from memory as cache lines, unit strides increase reuse. Hardware prefetchers can recognize small regular 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. Tabulation of stride values for each data structure enables both recognition of access patterns that yield poor memory access but also provide information to define restructurings of the loop structure. In Section 6.1 we illustrate the code transformations motivated by long and/or irregular access strides for Sweep3D and for the NAS LU Symmetric Gauss-Seidel solver.

5.6.2. Computation of Access Strides. Since we have an association between the memory address requested and the source code data structure name, we can, for each data structure and thread, compute the difference between the last requested address and the currently requested address, which is the stride value. MACPO outputs each stride for each data structure and can, if needed compute other metrics including the more commonly used average stride.

6. RESULTS

To evaluate the effectiveness of using MACPO and PerfExpert, we used programs from three benchmark suites: NAS Parallel Benchmarks (size `B`) [Bailey et al. 1992], ASCI Sweep3D benchmark[5] and the OpenMP benchmarks from the Rodinia benchmark suite [Che et al. 2009]. We also included two versions of a standalone application, a 2D Lattice Boltzmann solver parallelized using MPI and a 3D Lattice Boltzman solver parallelized using OpenMP, in the codes that were measured. Four codes (Sweep3D, NAS Block Tridiagonal, NAS LU Symmetric Gauss-Seidel and 2D Lattice Boltzmann) are MPI codes whereas the remaining (Particle Filter, Needleman-Wunsch, Back Propagation, SRAD and 3D Lattice Boltzmann) are multi-threaded programs that use OpenMP. In all cases, understanding the cause of the degradation and tuning the code simply based on measurement of code segments 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 specific data structures with poor memory performance.

For about 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 run on three different generations of processors. Programs from the Rodinia benchmark suite were run using 8 threads on Longhorn[6]. Each node on Longhorn is comprised of dual quad-core Intel Nehalem chips. Each node has 48GB of main memory. Each of the 8 cores have 32KB level 1 data and instruction caches. A unified level 2 cache of size 256KB is shared between two cores while a unified level 3 cache of size 8MB is shared by four cores. The MPI 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 32GB of main memory. Each core has its private L1 data and instruction caches of size 64KB each. Unlike the Intel Nehalem processor, the AMD Barcelona processor supports a per-core unified L2 cache of size 512KB. The L3 cache (2MB) is per-socket and is therefore shared among the four cores[8]. The OpenMP version of the Lattice Boltzmann solver code was measured and run on Stampede. Each node on Stampede is comprised of two eight-core Sandy Bridge processors. Each of the eight cores on a chip has a private 32KB instruction cache, a private 32KB data cache and a private 256KB L2 cache. The L3 cache (20MB) is shared by all eight cores on the chip. The choice of selecting a particular machine for any program was purely random. For brevity, we include part of the MACPO output for one of the programs (Particle Filter) in Figure 3.

[5]ASCI Sweep3D: `http://wwwc3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html`.
[6]TACC Longhorn user guide: http://www.tacc.utexas.edu/user-services/user-guides/longhorn-user-guide.
[7]TACC Ranger user guide: http://www.tacc.utexas.edu/user-services/user-guides/ranger-user-guide.
[8]http://blogs.amd.com/developer/2007/09/10/barcelona-processor-feature-shared-l3-cache/

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

Fig. 3. Excerpt of trace analysis output for the naïve version of the Particle Filter code.

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 of the form shown in Listing 5.

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 as shown in Listing 6. 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%.

```
1 DO k = ...
2   DO i = ...
3     DO i = ...
4       flux(i,j,k,2) = ...
5       flux(i,j,k,3) = ...
6       flux(i,j,k,4) = ...
7     ENDDO
8   ENDDO
9 ENDDO
```

Listing 5. Fused loop producing strided memory accesses

```
1  DO k = ...
2    DO i = ...
3      DO i = ...
4        flux(i,j,k,2) = ...
5      ENDDO
6      DO i = ...
7        flux(i,j,k,3) = ...
8      ENDDO
9      DO i = ...
10       flux(i,j,k,4) = ...
11     ENDDO
12   ENDDO
13 ENDDO
```

Listing 6. Loop fission caused unit-stride memory accesses

6.1.2. NAS LU Symmetric Gauss-Seidel. This code is a nonlinear 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 (LLC) 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 3, 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. Three-Dimensional (OpenMP) Lattice Boltzmann. This code implements the Lattice Boltzmann solver for the Navier-Stokes equations. For the OpenMP code, we observed a high (5.3) data access LCPI with a significant contribution (2.9) from L3 miss LCPI. Unlike most stencil codes, this code refers to multiple data arrays with almost zero reuse within an iteration. The low data reuse eliminates cache blocking from the list of applicable optimizations. MACPO reported that each iteration of the computation loop references 23 streams. As shown in Section 5.2, loops that contained more than 6 streams in the body suffered a severe execution penalty. The penalty arose from the prefetcher not being able to load data into the cache in time. To improve the effectiveness of the prefetcher, we applied loop fission so each loop body did not reference more than 6 streams. In other words, given a code as shown in Listing 1 (in Section 5.2.1), we converted it to the form shown in Listing 2 (in Section 5.2.1). Using this optimization, we saw a 19% improvement in the running time of the application.

6.1.5. 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 nonlinear 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. Two-Dimensional (MPI) Lattice Boltzmann. Running PerfExpert on the MPI 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 a form as shown in Listing 7.

```
1 DO I = xl, xu
2   DO J = yl, yu
3     ! Velocity field at each node
4     u(i,j,1) = ( f(i,j,1,now) - f(
          i,j,3,now) + f(i,j,5,now) -
           f(i,j,6,now) - f(i,j,7,now
          ) + f(i,j,8,now) )*invRho
5   ENDDO
6 ENDDO
```

Listing 7. Strided accesses in loop body

```
1 DO J = yl, yu
2   DO I = xl, xu
3     ! Velocity field at each node
4     u(i,j,1) = ( f(i,j,1,now) - f(
          i,j,3,now) + f(i,j,5,now) -
           f(i,j,6,now) - f(i,j,7,now
          ) + f(i,j,8,now) )*invRho
5   ENDDO
6 ENDDO
```

Listing 8. Interchanged loops

The index that changes with each access to the array is not the first one hence we were seeing non-zero stride values. Interchanging the loops (as shown in Listing 8) 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 was 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. 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

Table IV. Summary of Benchmark Optimizations

Application	Optimization	Improvement
2D (MPI) Lattice Boltzmann	Loop interchange	13%
3D (OpenMP) Lattice Boltzmann	Loop fission	19%
ASCI Sweep3d	Loop fission	24%
NAS BT	Change layout of data structures	13%
NAS LU	Loop fusion	8%
Rodinia Back Propagation	Loop interchange	17%
Rodinia SRAD	Loop interchange	38%
Needleman Wunsch	Increase OpenMP loop chunk size	4%

latency (20 cycles). Stride analysis showed that this array was accessed with an average stride of 32 cache lines. The reuse factor indicated that only 67% of the cache lines in L1 were being reused. Code inspection showed that the loops were inverted, causing strided accesses to the data structures. Changing the loop order reduced the access strides to 1 cache line. The unit strides also improved reuse in the L1 cache line and the L1 reuse factor for the revised code was 87%. The overall access penalty was reduced to 13 cycles, ultimately reducing the total time by 38%.

7. CONCLUSIONS

We described the design, implementation and innovations of the MACPO tool. MACPO 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 that integrates measurements and analyses of code segments and memory access characteristics. Illustrations of the enhanced process demonstrate the value of integrating analyses of code segments & data structures and specifically demonstrate the additional capabilities for diagnosis and selection of optimizations enabled by adding analyses centered around data structures 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 [Sopeju et al. 2011]. 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 that execute in SIMT/SIMD parallelism. The process of combining measurements of code segments and data structure memory behavior can be extended to determine code segments that can readily be converted to execute in SIMT/SIMD mode. The critical properties that 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.

REFERENCES

D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. 1992. NAS parallel benchmark results 3-94. In *Proceedings of the Scalable High Performance Computing Conference*. 386–393.

Kristof Beyls and E. D'Hollander. 2006. Discovery of locality-improving refactorings by reuse path analysis. In *High Performance Computing and Communications*. 220–229.

Kristof Beyls and Erik H D'Hollander. 2009. Refactoring for data locality. *Computer 42*, 2, 62–71. DOI:http://dx.doi.org/10.1109/MC.2009.57.

Martin Burtscher, Byoung Do Kim, Jeff Diamond, John Mccalpin, Lars Koesterke, and James Browne. 2010. PerfExpert: An easy-to-use performance diagnosis tool for HPC applications. *IEEE Computer*, 1–11. DOI:http://dx.doi.org/10.1109/SC.2010.41.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 44–54. DOI:http://dx.doi.org/10.1109/IISWC.2009.5306797.

J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. 1997. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 292–302. DOI:http://dx.doi.org/10.1109/MICRO.1997.645821.

Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Marty Itzkowitz, Brian J. N. Wylie, Christopher Aoki, and Nicolai Kosche. 2003. Memory profiling using hardware counters. In *Proceedings of the Supercomputing Conference*. 17–30.

Yunlian Jiang, E. Zhang, K. Tian, and X. Shen. 2010. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Compiler Construction*, Springer, 264–282.

Chris Lattner and V Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 75–86. DOI:http://dx.doi.org/10.1109/CGO.2004.1281665.

K. Lawton. 2004. Bochs IA-32 Emulator Project. bochs.sourceforge.net.

Xu Liu and John Mellor-Crummey. 2011. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*. 171–180. DOI:http://dx.doi.org/10.1109/CGO.2011.5764685.

Gabriel Marin. 2005. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Los Alamos Computer Science Institute*.

Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. 2004. PIN: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the Workshop on Computer Architecture Education* Held in Conjunction with *the 31st International Symposium on Computer Architecture*.

Derek L. Schuff, Milind Kulkarni, Vijay S. Pai, and West Lafayette. 2010a. Accelerating multicore reuse distance analysis with sampling and parallelization. *Measurement*, 53–63.

D. L. Schuff, B. S. Parsons, and V. S. Pai. 2010b. Multicore-aware reuse distance analysis. *Measurement*.

Olalekan A. Sopeju, Martin Burtscher, Ashay Rane, and James Browne. 2011. AutoSCOPE: Automatic suggestions for code optimizations using PerfExpert. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*.

Richard A. Uhlig and Trevor N. Mudge. 1997. Trace-driven memory simulation: A survey. *ACM Comput. Surv. 29*, 2, 128–170. DOI:http://dx.doi.org/10.1145/254180.254184.

S. Vlaovic and E. S. Davidson. 2002. TAXI: Trace analysis for X86 interpretation. Ph.D. dissertation, University of Michigan.

Jonathan Weinberg and Allan Snavely. 2008. Chameleon: A framework for observing, understanding, and imitating the memory behavior of applications. In *Proceedings of the Workshop on State-of-the-Art in Scientific and Parallel Computing*.

Received February 2013; revised August 2013, November 2013; accepted November 2013