Memory Bandwith Based Performance Tuning Prediction

Chen Ding, Ken Kennedy

CRPC-TR98742
May 1998

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted July 1998

Memory-Bandwidth Based Performance Tuning and Prediction

Chen Ding Ken Kennedy

Department of Computer Science
Rice University

May 19, 1998

Abstract

It is the contention of this paper that memory bandwidth has become the single most important determinant of performance on modern computer systems built from commodity processors. This contention is supported by a study of several representative scientific programs executed on the Silicon Graphics Origin2000, in which memory bandwidth is far more critical to performance than CPU speed or bandwidths between different levels of cache. This result leads us to conjecture that measuring and predicting data transfer time between memory and cache could be an effective way to predict and tune program performance. We present the design of a performance tool based on this idea. This tool could use bandwidth analysis to predict program performance and locate memory hierarchy performance problems within an application. The effectiveness of such a tool was demonstrated on the NAS SP benchmark. In the tuning step, this approach identified three loop nests with low memory hierarchy performance that could be modified through simple transformations to achieve a 19% overall performance improvement. We hand-applied the static estimation method to two subroutines and found that the difference between the estimated capacity misses and the actual number of cache misses is less than 3%. Given accurate static estimation, this approach can predict SP running time with an error of less than 10%.

1 Introduction

As modern single-chip processors have increased the rate at which they execute instructions, it has become increasingly the case that the performance of applications depends on the performance of the machine memory hierarchy. For some years, there has been an intense focus in the compiler and architecture community on the impact of *latency* on performance. This work has led to extremely effective techniques for amelioration of the effects of latency, primarily through the use of program reorganization to increase the reuse of data in cache and the use of software and hardware prefetching to reduce the effective latency experienced by a program.

It is the contention of this paper that latency management techniques have been so effective that the principal determinant of performance has become memory *bandwidth*, the rate at which operands can be delivered from local or remote memories to cache. To support this contention, we have conducted experiments on the Silicon Graphics Origin2000 system, which has an aggregate memory bandwidth of 300MB/s per processor[1]. Through studies of six applications we have established that the bandwidth needed by these applications to achieve balance with computation is in every case substantively greater than the bandwidth delivered by the Origin. This result leads us to conjecture that measuring and improving data throughput between memory and cache could be an effective way to monitor and improve overall program performance.

When a program is bandwidth-limited, it should yields a constant memory throughput and its memory transfer time would determine its execution time. When assuming a constant memory throughput, we can estimate its running time by dividing the entire data size of the application (without reuse) by the bandwidth of the machine. When the execution time is available, we Given this assumption, there are two nice things

[1] measured by a set of stride-one read/write kernels

we can do. On one hand, we can locate memory hierarchy performance problems by detecting low memory throughput. On the other hand, we can predict program execution time by estimating the memory transfer time assuming a constant memory throughput.

When bandwidth-limited, a program would yield a constant memory throughput; and its memory transfer time would determine its execution time. Thus, a good estimate for its running time can be derived by dividing the entire data size of the application (without reuse) by the bandwidth of the machine. In addition, the expectation of constant memory throughput can be used in the performance tuning step to locate program fragments with relatively low data throughput as candidates for performance tuning. When memory bandwidth is the bottleneck, a low data throughput often indicates, not a lack of hardware resource, but a problem of code structure and ineffective compiler optimizations, that is, an opportunity for performance tuning.

These observations can lead us to the design of a tool for analyzing and improving performance on a modern computer system. For each loop nest in the program, the tool would measure the balance between computation and bandwidth requirements, predict computation time by calculating memory hierarchy cost and identify memory hierarchy performance problems when actual running time is available. Such a tool could be extremely useful in both uniprocessors or parallel systems.

The remainder of the paper is organized as follows. The next section describes the experiments that support the contention that bandwidth is the critical bottleneck on the Origin and argues that this conclusion extends to other systems as well. Section 3 then presents the design of a tool that would use bandwidth analysis to uncover performance problems. Finally, Section 4 describes the hand application of techniques proposed for the tool to some benchmark problems, with good results. Section 5 summarizes the contribution of this work.

2 Preliminary Bandwidth Measurement

This section presents the results showing that memory bandwidth far more limited than CPU speed and bandwidths of caches. The experiment does not consider latency as a factor, because we are trying to understand the impact of bandwidth on performance, given a certain level of latency tolerance. In fact, the results presented in Section 4 suggest that current latency tolerance techniques are effective enough to expose memory bandwidth as the major bottleneck on regular applications.

Definition of Program and Machine Balance The capacity of a machine and the resource requirement of a program can be modeled by a concept called *balance*. Callahan et. al [3] and Carr and Kennedy [5] used balance to model the relation between the number of floating-point operations and the number of load/store operations. In this section, we will extend the definition of balance to including the data transfers on all levels of memory hierarchy. This extension allow us to consider the whole CPU-memory system. Without loss of generality, we assume a memory hierarchy consists of two levels of cache.

The capacity of a machine can be characterized by sequence of numbers, each of which represents the hardware capacity of the following resources: flop rate, load/store bandwidth, cache bandwidth, and memory bandwidth. The *flop rate* is the maximal number of floating-point operations that can be finished in each second. The bandwidths are defined as the maximal number of bytes that can be transferred each second between two levels of memory hierarchy, that is, between registers and first level cache, between two caches, and between memory and the second level cache. For example, a machine capacity of 300/200/200/100 means that the machine can, at each second, finish 300 floating-point operations, load/store a total of 200 bytes, exchange 200 bytes between caches, and transfer 100 bytes between memory and cache.

The resource requirement of a program is also a sequence of numbers, each of which summarizes the program's requirement on one of the following hardware operations: the total number of the program's floating-point operations, and the amount of data transferred between adjacent levels of memory hierarchy. For example, a program balance of 300/200/200/100 indicates that the execution of the program requires executing 300 floating-point operations, transferring 200 bytes between registers and cache, 200 bytes between two caches and 100 bytes between memory and cache.

For the purpose of comparison between the capacity of a machine and the requirement of a program, we convert both machine capacity and program requirement into a sequence of ratios by normalizing the memory data transfer to be one. The ratios represent the number of operations performed at the same time of transferring one bytes between memory and cache. They can be computed by dividing each number with the last number. For example, the capacity of 300/200/200/100 is converted into a machine balance of 3/2/2/1, which means that at the same time of transferring one byte between memory and cache, the machine can finish three floating-point operations, load/store a total of two byes, and transfer two bytes between two caches. A program balance of 3/2/2/1, on the other hand, says that, for each byte transferred between memory and cache, the program needs to finish an average of three floating-point operations and transfer two bytes between registers and cache and between two caches. From now on, we call the individual ratios in a machine or a program balance as CPU ratio, load/store ratio, cache ratio and memory ratio (memory ratio is always normalized to 1).

With the definition of machine and program balance, we can directly compare machine capacity with program requirement. When a ratio of a machine balance is greater than the same ratio of a program balance, the corresponding machine resource is excessive compared to memory bandwidth. For example, assume a machine CPU ratio of 3 but a program CPU ratio of 2, then for each bytes transferred between memory and cache, the machine can finish three floating-point operations whereas the program only needs to execute two floating-point operations. Thus, the average utilization of CPU would be less than 67%.

Experimental Evaluation of Memory Bandwidth Bottleneck We conducted an experiment to measure the balance of six programs and compare them with the balance of SGI Origin2000 machine. The six programs include four kernels—convolution, dmxpy, matrix multiply, FFT—and two full applications—SP from the NAS benchmark suite and Sweep3D from DOE. The highest optimization level was used for all programs except matrix multiply, which was compiled with -O2.

The experiment uses hardware counters to measure the number of floating-point operations (*flops*) and the number of level 1 (*L1*) and level 2 (*L2*) cache misses and write backs. The total amount of memory transfer is the sum of the data transfered on L2 misses and data written back from L2 cache. Similarly, the volume of cache transfer is the total of the data transfered on L1 misses and data written back from L1 cache. The total amount of load/store between L1 cache and registers cannot be accurately measured because hardware counters do not distinguish integer loads/stores from those of floating-point numbers. We overestimate the volume of L1-Register transfers by counting all integer loads/stores as floating-point loads/stores. However, this overestimate does not affect our final conclusion.

The balance of SGI Origin is calculated based on the hardware specifications except its memory bandwidth. The sustainable memory bandwidth is measured through a set of read-write Fortran kernels, whose memory throughput were found to be within 10% variation around 300MB/s.

The following table lists the balance of programs tested and compares them with that of SGI Origin.

Applications	Program/machine Balance			
	CPU ratio	load/store ratio	cache ratio	Memory ratio
convolution	0.194	1.25	0.99	1
dmxpy	0.119	0.99	0.99	1
mmjki (O2)	0.170	4.09	1.39	1
FFT	0.366	3.05	1.08	1
SP	0.203	2.21	1.30	1
Sweep3D	0.129	1.94	1.18	1
SGI-Origin	1.30	5.20	5.20	1

The first six rows give the program balance of these programs. For example, the first row shows that *convolution* executes 0.194 floating-point operations, load/store a total of 1.25 bytes and transfers 0.99 bytes between caches for each byte transferred between memory and cache. The last row shows the machine balance of SGI Origin, which can finish 1.30 floating-point operations and transfer 5.20 bytes between registers and cache and between caches for each byte transferred between memory and cache.

Let us first compare the CPU ratios listed in the first column, which directly compares memory bandwidth with CPU speed. The column shows that CPU ratio of the Origin machine is at least three times the ratio of programs. This dramatic difference shows that memory bandwidth is much more limited than CPU power: on average, the CPU utilization is less than one third. The next two columns compares memory bandwidth with bandwidths of other levels of memory hierarchy. It shows that both load/store ratio and cache ratio of the SGI Origin are significantly higher than those of every program tested. Thus the cache bandwidths on this machine is more than adequate for these programs, while its memory bandwidth is far too low. Therefore, memory bandwidth is the critical resource for these programs on the SGI Origin.

The dramatic mismatch between program balance and machine balance is impressive. Take the CPU ratio as an example. If we assume these programs running at the peak flop rate of the SGI Origin (390 million flops per second), then the minimal bandwidth ranges from 1.1GB/s to 3.3GB/s, with all except FFT requiring greater than 1.9GB/s. These memory bandwidth requirements are far higher than the available memory bandwidth, which is 300MB/s. On the other hand, if we assume that these programs run at full memory bandwidth, then the resource utilization is less than 28% for CPU, 78% for load/store bandwidth and 27% for cache bandwidth. For two full applications, the average utilization is less than 16% for CPU, 43% for load/store bandwidth and 25% for cache bandwidth. The low utilization of these resources show that they are far more excessive than memory bandwidth.

Memory Bandwidth Bottleneck on Other Machines and Future Systems The same resource mismatch occurs on other state-of-art systems as well. For example, the HP PA8000 can issue two loads/stores and four floating-point operations per cycle. Its maximal available memory bandwidth is about 500MB/s, less than two times of the memory bandwidth of the SGI Origin. Since both its flop rate and cache-register bandwidth are doubled, its machine balance is even higher than that of the Origin. Therefore, memory bandwidth should be even more critical on an HP PA8000 processor.

The key observation from the above results is that memory bandwidth is the performance bottleneck for current machines and applications we tested. Whether this trend will continue depends on the characteristics of application, machine and the effectiveness of latency tolerance.

Bandwidth-constrained applications must have two properties: the working set does not fit in cache and the CPU ratio is greater than the ratio on the target architecture. The result presented in this section gives the CPU ratios of typical computation-intensive scientific applications. In other data-intensive programs, such as database and multimedia applications, the CPU ratio is even higher because they have much less computation.

The trend of the CPU ratio of a machine depends on the growth rate of CPU speed and memory bandwidth. In the past, the improvement of CPU power has been the fastest among the progress of all other hardware resources on microprocessors, because of the dramatic increase in silicon density. Memory bandwidth, however, has to overcome the gap from off-chip to on-chip and therefore suffers a more restrictive physical constraint. In addition, the trend of bus-based multiprocessors aggregates the memory bandwidth bottleneck by leaving each processor only a fraction of the bandwidth of a memory bus. This trend is likely to continue in computer industry driven partly by the economic factor that adding processors is cheaper than adding bandwidth.

Memory latency could be the performance bottleneck if it is not effectively hidden. The effectiveness of latency tolerance depends on the amount of parallelism in memory accesses, and the ability of compiler and hardware optimizations to fully overlap memory accesses. It is unlikely that any of these two factors can become a more restrictive performance than memory bandwidth in the near future.

Implication of Bandwidth Bottleneck: Uniform Memory Throughput Memory throughput is determined by both latency constraint (physical distance between CPU and memory) and bandwidth constraint. Although both of them translate into waiting time of a memory operation, their influence on memory throughput is quite different. The impact of latency depends on a particular program access pattern, a particular machine memory system design, and the interactions between the two. As a result, the memory throughput, in the absence of latency tolerance, varies significantly among different parts of a program and among different programs. The data throughput due to bandwidth constraint, however, is uniform and regular because the only constraint is that the program data throughput cannot exceed the maximal machine memory bandwidth. When memory bandwidth is the performance bottleneck and when data throughput is determined by the bandwidth constraint, all programs would show an identical memory throughput. Indeed, what distinguishes memory bandwidth bottleneck from previous memory system problems is the property of uniform memory throughput. As we'll see later, this property can lead us to the design of a very effective performance tuning and prediction tool.

Related Work on Machine and Program Balance Callahan, Cocke and Kennedy, and Carr and Kennedy [3, 5] defined the concept of machine and program balance as the ratio of the number of memory operations to the number flops. Our work extends their notion of balance to include bandwidths between all levels of memory hierarchy.

McCalpin [14] measured the sustainable memory bandwidth through the STREAM benchmark, which consists of four long vector operations written in Fortran. He further defined a *machine balance* to be the ratio of the peak flop rate to the sustained memory bandwidth [1]. A large class of machines (available at the time of writing, 1995) were measured, including uniprocessor workstations, vector machines and distribute memory and share-memory parallel machines. The balance for shared-memory systems were seen to be much higher than that of vector machines. On shared-memory systems, the sustainable memory bandwidth was only 3-10% of the bandwidth required to keep CPU busy. He also observed a significant correlation between SPECfp95 results and memory bandwidth on the leading uniprocessor machines at that time. McCalpin compared the trend of CPU performance and sustainable bandwidth and suggested that machines were increasingly imbalanced at an exponential rate of 115% a year [15].

Burger et al. [2] simulated the memory consumption of 14 SPEC benchmark programs (including both integer and floating-point applications) on current and future microprocessors. They concluded that "aggressive implementations of latency-tolerance techniques in future processors will expose memory bandwidth, ..., as a severe performance bottleneck". Memory bandwidth are determined by three resources, the bandwidth of DRAM, the bandwidth of a bus, and the aggregated bandwidth of all pins of a chip. They maintained that the last one is most difficult and expensive to improve. Just imagining the cost of manufacturing, assembling and maintaining a chip with 2000 pins.

3 Bandwidth-Based Performance Tuning and Prediction

We present a performance prediction and tuning tool based on measuring and predicting memory hierarchy performance, in particular, memory throughput. The key assumption is that memory-cache bandwidth is the major bottleneck for a large class of data-intensive applications. Because of this bottleneck, programs should yields a constant memory throughput and memory transfer time would determine program execution time. Given this assumption, there are two nice things we can do. On one hand, we can locate memory hierarchy performance problems by detecting low memory throughput. On the other hand, we can predict program execution time by estimating the memory transfer time assuming a constant memory throughput.

This section presents the general structure of the analysis technique based on this idea, discusses its major advantages, potential broad applications as well as limitations and extensions, and compares this approach with existing techniques.

3.1 Performance Tool Architecture

The bandwidth-based performance tool takes as input, a source program along with its inputs and parameters for the target machine. It estimates the total amount of data transfer between memory and cache. The estimated amount can be used to either predict the performance of a given program component without running the program or locate memory hierarchy performance problems given the actual running time. Figure 1 shows the structure of the tool, as well as its inputs and outputs.

The tool estimates the total amount of data transfer between memory and cache by analyzing source program, runtime program inputs and machine parameters such as cache size and cache-line size. First, the program is transformed into an interval-flow graph[11], which allows us to consider loop structures explicitly and focus on program fragments that has significant amount of data transfer. Then, the interprocedural bounded section analysis [12] is used to summarize the size and shape of data accesses. The tool factors out the memory references that are cache hits through cache reuse analysis[18, 17] because they do not incur additional memory-cache transfer. The cache conflicts due to the limited associativity of a cache can be estimated by the approximation method used in [8]. Finally, the estimation of cache misses considers the cross-loop cache reuse [7, 16].

Examining source code is not enough since the actual amount of data transfer can be reduced by data-reuse optimizations such as data blocking/tiling and loop fusion. The performance tool can get hold of these high-level transformations if it has access to compiler decisions. Most data transformations are dependent on program input size and machine parameters. Thus, a performance tool is in a perfect position to finalize

[1] McCalpin used the term "sustained memory operations per cycle"

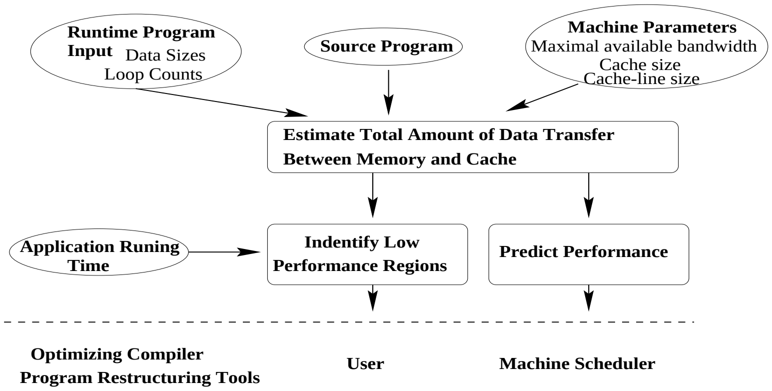

Figure 1: Structure of the Performance Tool

data optimization decisions such as tiling and loop fusion.

With the estimation of total amount memory transfer, the tool can then predict memory transfer time or detect low memory throughput.

Performance Prediction Bandwidth-based performance prediction estimates program running time with the memory transfer time, that is, the time needed to transfer all memory data given full memory bandwidth utilization. When memory bandwidth is the performance bottleneck, this strategy is not only much simpler, but also more accurate than existing techniques, and wider applicable to different machines, applications and parallelization schemes.

Performance Tuning Bandwidth-based performance tuning locates memory hierarchy performance problems by detecting low memory throughput. When memory bandwidth is the performance bottleneck, all program fragments should yield the full utilization of memory bandwidth. On the other hand, a low memory bandwidth utilization, when it happens, often indicates not lack of hardware resource, but a problem of code structure and ineffective compiler optimizations, that is, an opportunity for performance tuning.

The tuning tool first calculates the memory utilization of of each program fragments and then identifies the fragments that have low memory throughput as candidates for performance tuning. The tool calculates the potential performance gain as the difference between the current execution time and the predicted execution time assuming full memory utilization. The tuning candidates are then ordered by their potential performance gain and presented to a user or a program restructuring tool.

The detecting of tuning candidates requires no special hardware features or software simulation. The use of actual running time includes the effect of all levels of compiler and hardware optimizations. As a result, the bandwidth-based performance tuning can automatically locate memory hierarchy performance problems for a wide variety of applications on microprocessor-based systems.

Bandwidth-based performance tuning does not necessarily rely on compiler analyses because hardware counters can also measure the amount of data transfer (and with a higher precision). However, the static compiler estimation has at least two advantages. First, static analysis is applicable to all architectures while hardware counters are not available on all machines. Second, static analysis can identify the problem of excessive conflict misses but hardware counters cannot distinguish different type of misses.

3.2 Extensions to More Accurate Estimation

Performance prediction needs to estimate the computation interlock time due to loop recurrences, with the algorithm given by Callahan et al [3].

Excessive misses at other level of memory hierarchy can be more expensive than memory transfer time, e.g. L1 misses and TLB misses. Thus, the performance tool needs to measure the resource consumption of other levels of memory hierarchy. However, analysis to other levels of memory hierarchy needs not to be quantitatively accurate since these resources outweigh memory bandwidth constraint only in extreme cases.

Latency, especially remote latency need to be modeled if they cannot be effectively hidden. When a remote access is bandwidth constrained, we can calculate its latency by approximating remote bandwidth. When a local or remote access is not bandwidth-constrained, we could either improve latency tolerance techniques or train the performance estimator to recognize situations of long memory latency using the idea of training sets[1]. The bandwidth-based tuning tool can automatically collect such situations from applications.

Coherence misses need also be measured if they are a significant factor to performance. The data analysis techniques in the bandwidth-based performance tool have good knowledge to detect coherence misses, especially for compiler parallelized code [16].

3.3 Related Work

The bandwidth-based approach is different from previous techniques in its emphasis on monitoring and predicting memory data transfer time for memory constrained applications. Analyses and estimations of bandwidth requirement or constraint is far easier and simpler than those based on memory latency. As a result, bandwidth-based tuning and prediction are more efficient and accurate than previous techniques for bandwidth-limited applications.

Callahan, Kennedy and Porterfield[4] first used compiler-based approach to analyze and visualize memory hierarchy performance. They instrumented the code to obtain memory access behavior and examined it with a memory simulator.

Gallivan et al[9] documented the memory performance of programs with different load/store patterns and predicted memory performance by pattern-matching the load/store structure of a program. Their work was on a vector machine with no cache-based memory hierarchy.

The idea of inferring program performance by documenting the cost of commonly used operations are fully exploited in the use of training sets[1]. The training sets are essentially a database for the cost of commonly used (and expensive) operations. Balasundaram et al[1] used training sets to model communication performance in data-parallel programs. Although we do not model the cost of communication primitives in our work, the same idea could be used in our approach to model the pattern of computations that do not fully utilize memory bandwidth. More detailed modeling between data access pattern and achievable memory throughput will give higher accuracy in prediction and better tolerance to hardware irregularities.

Clements and Quinn[6] used static methods to give performance predictions. They predicted computation cost by counting the number of floating-point operations in a parse-tree and included memory cost by multiplying the number of cache-misses with the miss latency. The assumption of full CPU utilization is not accurate when memory bandwidth is the bottleneck because CPU utilization can be quite low due to insufficient memory bandwidth. Their model of memory cost is not valid for machines with hardware and compiler latency tolerance implementations. Their word did not cover performance tuning.

In parallel to compiler-based approaches, dynamic techniques measures program performance at runtime based on the use of trace files or machine simulators. Goldberg and Hennessy[10] measured memory stall time by comparing actual running time with simulation result of running the same program on a perfect memory. Dynamic approaches don't rely on compiler analysis and is generally more accurate in measuring hardware utilization. But because of the low-level view of programs and the use of trace files and simulators, dynamic techniques are expensive when used to measure lots of hardware events in high frequency or simulate hardware in great detail; they cannot predict performance without running the program; and they are often architecture dependent.

Micro-kernels, contrasting to whole-program-targeted approaches, directly measures hardware performance through simple computation or access patterns. McCalpin [14] quantifies sustainable memory bandwidth through a set of four long vector computation kernels, called the STREAM benchmark. Our measurement of memory bandwidth in this experiment is similar but includes kernels accessing up to seven data arrays. Hristea and Lenoski [13] measured sustainable memory bandwidth (which they call pipeline bandwidth) with a kernel of accessing only one element in each cache line, instead of stride-one access of all data. On SGI Origin, they reported a memory throughput (pipelined bandwidth) of 550MB/s. The difference in memory throughput of different-stride accesses suggests the need for a more sophisticated model of available machine memory bandwidth. Micro-kernels can be used to quantify the bandwidth-based memory performance model for a given machine.

Another large class of work we do not directly compare here is the modeling of communication performance on message-passing machines or communication/synchronization cost on share-memory systems. Our goal is different, which is to predict computation cost on a shared-memory system. Shared-memory machines have relatively low parallel overhead and communication cost. In addition, to obtain good parallel efficiency, current parallel applications are very coarse-grained, with little inter-communication or synchronization. Therefore, accurately predicting computation cost is critical in predicting overall program performance.

3.4 Result of Performance Tuning and Prediction on NAS/SP

Since we did not have an implementation of the tool, we analyzed the SP (serial version 2.3) by hand. We used the B class input size and ran *adi* for three iterations to save experiment time.

Performance Tuning The tool is extremely useful to performance tuning because it automatically locates memory hierarchy performance problems by measuring the memory bandwidth utilization of a program. Effective user tuning can then be applied to achieve higher memory hierarchy throughput, and in turn, better program performance.

We applied this idea by hand on SP in order to locate loop nests whose performance can be improved. We measured seven subroutines, which represents 95% of running time of the whole *adi* procedure. We used hardware counters to measure the achieved memory bandwidth and bandwidth utilization, which are listed below.

Subroutines	Achieved BW	BW Utilization
compute_rhs	252MB/s	84%
x_solve	266MB/s	89%
y_solve	197MB/s	66%
z_solve	262MB/s	87%
lhsx	321MB/s	107% [1]
lhsy	279MB/s	93%
lhsz	96MB/s	32%

The last column shows that all subroutines can utilize over 84% of memory bandwidth except *y_solve* and *lhsz*. The low memory bandwidth utilization indicates that these two subroutines have computation that running inefficiently. We applied our hand measurement for each loop nest of *lhsz* and found that the first loop nest has a memory bandwidth utilization of less than 11% due to excessive TLB misses. By applying array expansion and permuting the loop into memory order, we were able to eliminate excessive TLB misses and improve the running time of that loop nest by a factor of 5 and the overall execution time by over 15%.

We also analyzed the memory bandwidth utilization of loops nests in *compute_rhs* and found two of them that had low memory bandwidth utilization (65% and 44%) due to cache conflicts in L1 cache. We distributed both loops and padded one of the data array. These modifications improved the two loops by 9% and 24% individually and overall running time by 2.4%.

The effective tuning on the two loop nests in *compute_rhs* demonstrates the advantage of memory bandwidth-based tuning over those based on flop rate or FPU utilization. The flop rate of both loops are over 30 MFlop/s before tuning, corresponding to a FPU utilization of 8%. The utilization, although low, is not the lowest. For example, all loops in *lhsx* have a FPU utilization under 5% (18 MFlop/s). Thus, judging by FPU utilization alone would suggest improving *lhsx*, which is impossible because memory bandwidth is already saturated in *lhsx*.

The effective tuning we applied demonstrates the effectiveness of the bandwidth-based approach. With the measurement of memory bandwidth utilization, a tool can automatically identify loops with low memory hierarchy performance and guide user to apply effective tuning. By applying bandwidth-based performance tuning on *SP* benchmark, we (as a user) were able to improve the overall running time by 19% by just examining and modifying three loop nests among over 80 loop nests in the program. The performance of SP, which is reported in flop rates, is improved from 45.1 MFlops/s to 55.5 MFlops/s.

Performance Prediction The bandwidth-based approach approximates program performance with the amount of time needed to finish all data transfer, assuming a given memory bandwidth utilization. This

---

[1] Pure data assignment loops with little computation can achieve a memory bandwidth that is higher than 300MB/s on the SGI Origin.

section examines the accuracy of this bandwidth-based performance prediction technique on the SP benchmark. Since we do not have an implementation of the static analysis, we'll first use hardware counters and then examine additional errors that could be introduced by the static estimation of a compiler.

Given the total amount of data transfer counted by the hardware counters, we predict the running time by dividing the amount of data transfer with memory bandwidth. The following table lists the actual running time, predicted time and percent of error. The prediction is given both with and without considering the effect of TLB misses in *lhsz*. Since it is expected that full SP application may not run at full memory speed, we list two predictions, first assumes full memory bandwidth utilization, the other assumes a utilization of 90%.

Computation	Exe Time	Pred. Time I Util=100%	Err. I	Pred. Time II Util=90%	Err. II
adi w/o TLB est.	59.0s	43.8s	-26%	48.6s	-18%
adi w TLB est.	59.0s	50.9s	- 14%	55.7s	- 5.6%
adi w/o *lhsz*	47.0s	40.0s	- 15%	44.3s	- 5.7%

The first row gives estimation result on full *adi* without adding in the extra overhead of TLB misses in *lhsz*. The TLB overhead can be easily predicted by multiplying the number of TLB misses with full memory latency (338ns according to what is called restart latency in [13]), which is 7.1 seconds. The second row gives performance prediction including this TLB overhead. The third row presents result for the computation without *lhsz*, which counts for over 80% of execution time.

The third and fifth column show the error in performance predictions. When assuming full memory bandwidth, the estimation error is 26% for whole *adi* computation without considering the TLB overhead of *lhsz*, 14% when TLB overhead is included and 15% for *adi* without *lhsz*. When we assume a utilization of 90%, the error in predicted time is 18% when not considering TLB overhead, 5.6% when including the TLB cost, and 5.7% for computation without *lhsz*. These results show that, with the estimation of the TLB cost and the assumption of 90% memory bandwidth utilization, bandwidth-based prediction can achieve a good accuracy of over 94%. The close accuracy figures for the last two rows also suggests that our estimation of the cost of TLB misses is correct.

The above predictions are based on the results of hardware counters. Now the question is how accurate are static techniques. We hand-applied the compiler-based analysis to count the number of capacity misses. We did not expect to see many conflict misses because the L2 cache is two-way associative and 4MB in size.

We hand-estimated two subroutines: *compute_rhs* and *lhsx*, which counts for 40% of running time. Subroutine *compute_rhs* resembles the mixed access pattern in the whole *adi* program while *lhsx* represents computations that are accessing memory in regular, stride-one fashion. The following table lists the actual and the predicted amount of data transfer and the error of the static hand-estimation.

Subroutine	Actual Data Transfer	Predicted Data Transfer	Error
lhsx	396MB	406MB	+ 3%
compute_rhs	5308MB	5139MB	- 3%

The third column shows that our hand-estimation on capacity misses is very close to the number of all actual misses; both errors are only 3%. This result suggests that compiler-based analysis is very accurate in estimating the total amount of data transfer between memory and cache. Assuming this accuracy of estimation holds for other parts of SP, the bandwidth-based analysis tool could predict the performance of SP with an error of less than 10% assuming 90% of memory bandwidth utilization and no abnormal effect of *lhsz*.

4 Conclusion

In this paper we establish that memory bandwidth has become the major performance bottleneck for a large class of applications on modern computer systems built from commodity processors. For four kernels and two full applications we tested on the SGI Origin2000 system, the limited memory bandwidth only allows for a utilization of at most 28% of peak flop rate, 78% of load/store bandwidth and 27% of cache bandwidth. As a result, memory bandwidth has become the most restrictive hardware constraint given sufficient latency tolerance. When a program is bandwidth-limited, memory throughput is solely determined by the memory bandwidth constraint. Indeed, the average memory bandwidth utilization is equal to or over 84% for five out of seven major computation routines of NAS/SP.

Because memory bandwidth has become the single most important determinant of program performance, predicting and tuning memory throughput is the most effective way to predict and tune program performance. Based on this idea, we present the design of a tool that can provide effective performance tuning and accurate performance prediction. Performance tuning locates program fragments with low memory bandwidth utilization so that they can be restructured for better memory and program performance. Performance prediction estimates program running time by memory data transfer time, which is computed by dividing total transfer amount with sustainable memory bandwidth.

We hand-applied performance tuning and prediction on NAS/SP and found good results. Performance tuning identified three loop nests (among over 80 loops nests and 500 loops) that have low memory throughput and simple modifications at source level resulted in a 19% overall performance improvement. Compiler analysis estimated the amount of memory transfer within an error of 3% for two of the subroutines we tested. Given accurate estimation of the amount of memory transfer, performance prediction achieved an accuracy of over 90%, assuming the knowledge of TLB-miss overhead in *lhsz* (which can be easily predicted by a compiler).

References

[1] V. Balasundaram, G. Fox, K. Kenneday, and U. Kremer. A Static Performance Estimator to Guide Data Partitioning Decisions. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, pages 213–223, 1991.

[2] D. C. Burger, J. R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23th International Symposium on Computer Architecture*, 1996.

[3] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[4] David Callahan, Ken Kennedy, and Allan Porterfield. Analyzing and Visualizing Performance of Memory Hierarchies. In *Performance Instrumentation and Visualization*, pages 1–26. ACM Press, 1990.

[5] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[6] Mark J. Clement and Michael J. Quinn. Analytical Performance Prediction on Multicomputers. In *Proceedings of Supercomputing'93*, November 1993.

[7] Keith Cooper, Ken Kennedy, and Nathaniel McIntosh. Cross-loop reuse analysis and its application to cache optimizations. Springer-Verlag, August 1996.

[8] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[9] K. Gallivan, W. Jalby, A. Maloney, and H. Wijshoff. Performance Prediction for Parallel Numerical Algorithms. *International Journal of High Speed Computing*, 3(1), 1991.

[10] Aaron J. Goldberg and John L Hennessy. Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1), 1993.

[11] R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[13] Cristina Hristea and Daniel Lenoski. Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks. In *Proceedings of SC97: High Performance Networking and Computing*, 1997.

[14] John D. McCalpin. Sustainable memory bandwidth is current high performance computers. http://reality.sgi.com/mccalpin_asd/papers/bandwidth.ps, 1995.

[15] John D. McCalpin. Machine balance: Sustainable bandwidth vs. peak mflops. http://reality.sgi.com/mccalpin_asd/Balance.gif, 1998.

[16] Nathaniel McIntosh. *Compiler Support for Software Prefetching*. PhD thesis, Rice University, Houston, TX, July 1997.

[17] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[18] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.