Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3-6, 1999 in Cambridge Massachusetts, USA

Bandwidth-Based Performance Tuning and Prediction

Chen Ding Ken Kennedy

{*cding, ken*}*@cs.rice.edu*
Computer Science Department
Rice University
Houston, TX

Abstract

As the speed gap widens between CPU and memory, memory hierarchy performance has become the bottleneck for most applications. This is due in part to the difficulty of fully utilizing the deep and complex memory hierarchies found on most modern machines. In the past, various tools on performance tuning and prediction have been developed to improve machine utilization. However, these tools are not effective in practice because they either do not consider memory hierarchy or do so with expensive and machine-specific program simulations. In this paper, we first demonstrate that application performance is now primarily limited by memory bandwidth. With this observation, we describe a new approach based on estimating and monitoring memory bandwidth consumption, which can achieve accurate and efficient performance tuning and prediction. When evaluated on a 3000-line benchmark program, *NAS/SP*, the bandwidth-based method has enabled a user to obtain a speedup of 1.19 by inspecting and tuning only 5% of the source code. Furthermore its compile-time prediction of overall execution time was within 10% of the actual running time.

Key Words: memory bandwidth, performance tuning prediction, compiler

1 Introduction

As modern single-chip processors improve 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 ameliorating the impact of *latency* on performance. This work has led to extremely effective techniques for reducing and tolerating memory latency, primarily through loop-level cache reuse and data prefetching.

As exposed memory latency is reduced, memory bandwidth becomes the dominant constraint because the limited memory bandwidth restricts the rate of data transfer between memory and CPU regardless of the speed of processors or the latency of memory access. Indeed, we found in an earlier study that the bandwidth needed to achieve peak performance levels on intensive scientific applications is up to 10 times greater than that provided by the memory system, resulting in a serious memory bandwidth bottleneck[1].

At Rice University, we are developing a compiler strategy for ameliorating the effect of memory-bandwidth bottleneck. The first part is to minimize the overall amount of memory transfer through automatic compiler enhancement of global and dynamic cache reuse[1, 2, 3]. Although effective, automatic optimizations are not perfect both because they may fail in some cases and because they do not estimate program execution time, which is important for subsequent task scheduling. To overcome these problems, this paper presents the second part of this compiler strategy, a bandwidth-based performance tool that assists user tuning and provides performance prediction.

Since memory bandwidth has become the critical resource, program performance is largely determined by how well memory bandwidth is utilized. This dependence suggests that an accurate modeling of program performance can be obtained by measuring memory bandwidth consumption. On the one hand, when low performance is resulted from low memory bandwidth utilization, a compiler can locate ill-behaved program regions based on measuring memory bandwidth consumption. Such automatic assistance can find for a user performance problems that are not only hidden behind immense program sources but are also often sensitive to specific machines or compilers. On the other hand, when bandwidth is fully or mostly utilized, a compiler can approximate program running time by dividing the entire data size of the application by the bandwidth of the machine. Such static performance prediction enables balanced program parallelization and efficient run-time scheduling, without relying on extra hardware support or expensive simulations.

In this paper we present the design of a bandwidth-based performance tool, along with an evaluation of its usefulness. The core support is the compiler analysis that estimates the amount of memory transfer of a program. Based on the same design, the tool can model the bandwidth of all other levels of memory hierarchy such as cache and register bandwidth. Such a tool could be extremely useful in both uniprocessors and parallel machines.

The rest of the paper is organized as follows. The next section demonstrates memory bandwidth bottleneck on current machines. The design of the bandwidth-based performance tool is described in Section 3 and evaluated in Section 4. Finally, Section 5 discusses related work and Section 6 concludes.

2 Memory Bandwidth Bottleneck

In an earlier study[1], we measured the fundamental balance between computation and data transfer and observed a striking mismatch between the balance of applications and those of modern machines. As a result, applications' demand on memory bandwidth far exceeds that provided by the machines, leading to a serious performance bottleneck.

Both a program and a machine have a balance. The balance of a program (*program balance*) is the amount of data transfer (including both memory reads and writebacks) that the program needs for each computation operation; the balance of a machine (*machine balance*) is the amount of data transfer that the machine can provide for each machine operation. The comparison between a program balance and a machine balance shows the relation between the resource demand of the program and resource supply of the machine. In the study reported in [1], we measured the balance of six representative applications and compared them with the balance of SGI Origin2000. Table 1 shows the ratios of demand over supply for data bandwidth of all levels of memory hierarchy including registers, two caches and main memory.

Applications	Ratio: demand/supply		
	L1-Reg	L2-L1	Mem-L2
convolution	1.6	1.3	6.5
dmxpy	2.1	2.1	10.5
mmjki (-O2)	6.0	2.1	7.4
FFT	2.1	0.8	3.4
NAS/SP	2.7	1.6	6.1
DOE/Sweep3D	3.8	2.3	9.8

Table 1: Ratios between bandwidth demand and supply

As the last column of Table 1 shows, programs require memory bandwidth 3.4 to 10.5 times as much as that provided by Origin2000, making memory bandwidth the most critical resource; The cache and register bandwidth is also insufficient by factors up to 6.0, but the gap is not as large as that of memory bandwidth.

Because of the limited memory bandwidth, other hardware resources are under utilized on average. Take *NAS/SP* as an example, the ratios imply that the CPU utilization can be no more than 16% on average; furthermore, the utilization can be no more than 44% for register bandwidth and 26% for cache bandwidth.

In addition to measuring program and machine balance, the study also compared the bandwidth constraint with the constraint of memory latency. It found that the physical memory latency is not a significant factor because current latency-hiding techniques are effective enough for programs to saturate the available memory bandwidth.

Memory bandwidth bottleneck happens on other machines as well. To fully utilize a processor of comparable speed of Origin2000, a machine would require 3.4 to 10.5 times of the 300 MB/s memory bandwidth of Origin2000, or to be exact, a machine needs 1.02 GB/s to 3.15GB/s of memory bandwidth, far exceeding the memory bandwidth on any current machines such as those from HP and Intel. As the CPU speed rapidly increases, future systems will suffer from a even worse balance and consequently a more serious bottleneck on memory bandwidth.

3 Bandwidth-Based Tuning and Prediction

Since memory bandwidth has become the primary factor limiting program performance, we can monitor and estimate program performance based on its memory bandwidth consumption. This section presents the design of a bandwidth-based performance tool. It first describes the structure of the tool and the analysis technique it employs. It then shows how the tool is used to tune or predict program performance.

3.1 Performance Tool Structure

The bandwidth-based performance tool takes as input, a source program along with its inputs and parameters for the target machine. It first estimates the total amount of data transfer between memory and cache. This figure can then 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.

Data Analysis The core support of the tool is the data analysis that estimates the total amount of data transfer between memory and cache. First, a compiler partitions the program into a hierarchy of computation units. A computation unit is defined as a segment of the program that accesses data larger than cache. Given a loop structure, a compiler can automatically measure the bounds and stride of array access through, for example, interprocedural bounded-section analysis developed by Havlak and Kennedy[4]. The bounded array sections is then used to calculate the total amount of data access and to determine whether the amount is greater than the size of the cache. The additional amount of memory transfer due to cache interferences can be approximated by the technique given by Ferrante et al[5]. Once a program is partitioned into a hierarchy of computation units, a bottom-up pass is needed to summarize the total amount of memory access for each computation unit in the program hierarchy until the root node—the whole program.

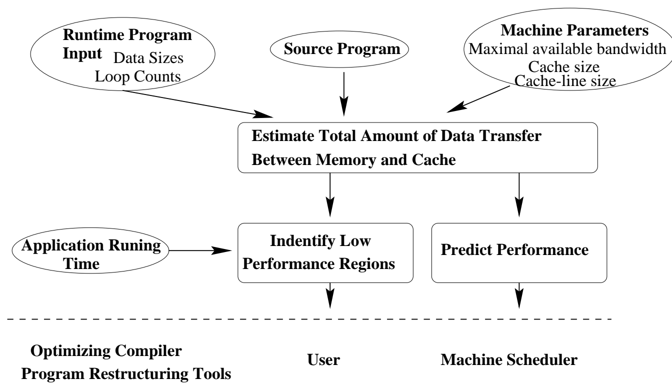

Figure 1: Structure of the Performance Tool

Since exact data analysis requires precise information on the bounds of loops and coefficients of array access, the analysis step needs to have run-time program input to make the correct estimation, especially for programs with varying input sizes. In certain cases, however, the number of iterations is still unknown until the end of execution. An example is an iterative refinement computation, whose termination point is determined by a convergence test at run time. In these cases, the analysis can represent the total amount of memory access as a symbolic formula with the number of iterations as an unknown term. A compiler can still successfully identify the amount of data access within each iteration and provide performance tuning and prediction at the granularity of one computation iteration.

Performance Tuning In bandwidth-based performance tuning, a compiler searches for computation units that have abnormally low memory bandwidth utilization for the user. Because of memory bandwidth bottleneck, a low bandwidth utilization implies a low utilization of all other hardware resources, therefore signaling an opportunity for tuning. A compiler can automatically identify all such tuning opportunities in the following two steps.

1. The first step executes the program and collects the running time of all its computation units. The achieved memory bandwidth is calculated by dividing the data transfer of each computation unit with its execution time. The achieved memory bandwidth is then compared with machine memory bandwidth to obtain the bandwidth utilization.

2. Second, the tool singles out the computation units that have low memory bandwidth utilization as candidates for performance tuning. For each candidate, the tool calculates the potential performance gain as the difference between the current execution time and the predicted execution time assuming full bandwidth utilization. The tuning candidates are ordered by their potential performance gain and then presented to a user.

Bandwidth-based performance tuning requires no special hardware support or software simulation. It is well-suited for different machines and compilers because the use of actual running time includes the effect of all levels of compiler and hardware optimizations. Therefore, it is not only automatic, but also accurate and widely applicable.

Bandwidth-based performance tuning does not necessarily rely on compiler-directed data analysis when applied on machines with hardware counters such as MIPS R10K and Intel Pentium III. The hardware counters can accurately measure the number and the size of memory transfers. With these counters, bandwidth-based tuning can be applied to programs that are not amenable to static compiler analysis. However, compiler analysis should be used whenever possible for three reasons. First, source-level analysis is necessary to partition a program into computation units and help a user to understand the performance at the source level. Second, static analysis is more accurate for tuning because it can identify the problem of excessive conflict misses, while hardware counters cannot distinguish different types of misses. Third, the compiler-directed analysis is portable because it can be applied to all machine architectures including those with no hardware counters.

Performance Prediction When a program uses all or most of machine memory bandwidth, its execution time can be predicted by its estimated memory-transfer time, that is, by dividing the total amount of memory transfer with the available memory bandwidth. This bandwidth-based prediction is simple, accurate and widely applicable to different machines, applications and parallelization schemes.

The assumption that a program utilizes all available bandwidth is not always true—some parts of the program may have a low memory throughput even after performance tuning. However, low memory throughput should happen very infrequently and it should not seriously distort the overall memory bandwidth utilization. The variations in the overall utilization should be small and should not introduce large errors into performance prediction. Otherwise, the program must have a performance bottleneck other than memory bandwidth. Section **??** discusses techniques for detecting other resource bottlenecks such as loop recurrence or bandwidth saturation between caches.

Integration with Compiler Since all data-analysis steps are performed statically, the performance tool can be integrated into the program compiler. In fact, an optimizing compiler may already have these analyses built in. So including this tool into the compiler is not only feasible but also straight-forward. Although the tool requires additional information about the run-time program inputs, the data analysis can proceed at compile time with symbolic program inputs and then re-evaluate the symbolic results before execution.

The integration of the tool into the compiler is not only

feasible but also profitable for both the tool and the compiler. First, the tool should be aware of certain compiler transformations such as data-reuse optimizations because they may change the actual amount of memory transfer. The most notable is the global transformations such as reuse-based loop fusion[1] and global data regrouping[2], which can radically change the structure of both the computation and data of a program and can reduce the overall amount of memory transfer by integral factors. The performance tool must know these high-level transformations for it to obtain an accurate estimate of memory transfer.

In addition to helping data analysis, the integration of the tool helps the compiler to make better optimization decisions. Since the tool has the additional knowledge on the program inputs, it can supply this information into the compiler. The precise knowledge about run-time data and machine parameters is often necessary to certain compiler optimizations such as cache blocking and array padding. Therefore, the integration of the compiler and the performance tool improves not only the accuracy of the performance tool but also the effectiveness of the compiler.

3.2 Extensions to More Accurate Estimation

Although the latency of arithmetic operations on modern machines is very small compared to the time of memory transfer, it is still possible that computations in a loop recurrence may involve so many operations that they become the performance bottleneck. So the tool should identify such cases with the computation-interlock analysis developed by Callahan et al[6].

Excessive misses at other levels of memory hierarchy can be more expensive than memory transfer. The examples are excessive register loads/stores, higher-level cache misses, and TLB misses. To correctly detect these cases, the performance tool needs to measure the resource consumption on other levels of memory hierarchy. In fact, the tool can extend its data analysis to measure the number of higher-level cache misses and TLB misses, which are in fact special cases of the existing data analysis.

On a machine with distributed memory modules, memory references may incur remote data access. When a remote access is bandwidth limited, the tool can estimate its access time with the same bandwidth-based method except that it needs to consider the communication bandwidth in addition to memory bandwidth. The bandwidth-based method also needs to model bandwidth contention either at a memory module or in the network. When a remote access is not bandwidth-constrained, we can train the performance estimator to recognize cases of long memory latency using the idea of training sets[7]. The bandwidth-based tuning tool can automatically collect such cases from applications because they do not fully utilize bandwidth.

Coherence misses in parallel programs should also be measured if they carry a significant cost. A compiler can detect coherence misses, especially for compiler parallelized code[8].

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%

Table 2: Memory bandwidth utilization of *SP*

3.3 Implementation Status

We are in the process of implementing the bandwidth-based performance tool into a whole-program compiler. In addition to performance tuning and prediction, the compiler performs global and dynamic computation and data transformations described in [1, 2, 3]. In fact, the partitioning of computation units and the characterization of data access are needed for both the compiler optimizations and the performance tool. However, at the time of writing, the implementation of data analysis is not complete. Therefore, we have to use hardware counters and manual analysis in the following evaluation. Since the hardware counters can provide an accurate estimation of data transfer, we use them whenever possible. Furthermore, we manually simulate data analysis in some representative parts of a program to verify the accuracy of compiler analysis.

4 Performance Tuning and Prediction on NAS/SP

This section evaluates bandwidth-based performance tuning and prediction on a well-known benchmark application, *SP* from NASA. It is a complete application with over 20 subroutines and 3000 lines of Fortran77 code. Since we did not have an implementation of the tool, we analyzed the program by hand and verified it with hardware counters on SGI Origin2000. Since the program consists of sequences of regular loop nests, we partitioned it into mainly two levels of computation units—loop nests and then subroutines. We used the class-B input size and ran only three iterations to save the experiment time.

4.1 Performance Tuning

To identify the tuning opportunities, we measured the bandwidth utilization of each subroutine and each loop nest. Instead of using compiler analysis, we used hardware counters to measure the total amount of memory transfer. Table 2 lists the effective memory bandwidth of seven major subroutines, which represents 95% of overall running time.

The last column of Table 2 shows that all subroutines utilized 84% or higher memory bandwidth except *y_solve*

and *lhsz*. The low memory bandwidth utilization prompted the need for user tuning. Subroutine *lhsz* had the largest potential gain for performance tuning. The subroutine has three loop nests, all had normal bandwidth utilization except the first one, which had an extremely low bandwidth utilization of less than 11%. By simply looking at the loop, we found that the problem was due to excessive TLB misses. By applying array expansion and loop permutation, we were able to eliminate a large portion of the TLB misses and improve the running time of the loop nest by a factor of 5 and the overall execution time by over 15%.

We then applied tuning to *compute_rhs*. By examining loop-level bandwidth utilization, we found two loops that utilized 65% and 44% of memory bandwidth because of the cache conflicts in L1. We distributed both loops and padded one of the data arrays. The hand modifications improved the two loops by 9% and 24% individually and overall running time by another 2.4%. With the tuning in *lhsz* and *compute_rhs*, the performance of *SP* was improved from 45.1 MFlops/s to 55.5 MFlops/s, a speedup of 1.19.

Bandwidth-based tuning is more accurate in locating performance problems than other tuning techniques because it monitors the most critical resource—memory bandwidth. For example, flop rates are not as effective. The flop rates of the previously mentioned two loops in *compute_rhs* are over 30 MFlop/s before tuning, which are not much lower than other parts of the program. For example, all loops in *lhsx* have a flop rate of under 18 MFlop/s. By comparing the flop rates, a user may draw the wrong conclusion that the loops in *lhsx* are better candidates for tuning. However, the loops in *lhsx* cannot be improved because they already saturate the available memory bandwidth. Their flop rates are low because they are data-copying loops with little computation.

The successful tuning of *SP* shows that the automatic tuning support is extremely effective for a user to correct performance problems in large applications. Although there were over 80 loop nests in *SP*, bandwidth-based tuning automatically located three loop nests for performance tuning. As a result, we as programmers only needed to inspect these three loops, and simple source-level changes improved overall performance by 19%. In other words, bandwidth-based tuning tool allowed us to obtain 19% of overall improvement by looking at and modifying less than 5% of the code.

4.2 Performance Prediction

Bandwidth-based performance prediction estimates program performance with its memory-transfer time, that is, the total amount of memory transfer divided by the memory bandwidth of the machine. This section examines the accuracy of this prediction technique on the *SP* benchmark. Since the prediction requires accurate estimation of the amount of memory transfer, we will first measure it with hardware counters and then apply compiler analysis by hand to verify the accuracy of the compiler-based estimation.

With the total amount of data transfer measured by the hardware counters, we calculated the memory-transfer time by dividing the amount of memory transfer by memory bandwidth. Table 3 lists the actual running time, the predicted time and the percent of error. The prediction is given both with and without considering the effect of TLB misses in the first loop of *lhsz*, discussed in the previous section. We list two predictions, the first assumes full memory bandwidth utilization for the whole program, and the other assumes an average utilization of 90%.

The first row of Table 3 gives the estimation results for one iteration of the computation without giving special consideration to 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 [9]), which adds to a total of 7.1 seconds. The second row gives the performance prediction including this TLB overhead. The third row predicts performance for the program without *lhsz* (the rest represents over 80% of the overall execution time).

The third and fifth column of Table 3 show the error of prediction. When assuming full bandwidth utilization, the prediction error is 26% for the whole computation without considering the abnormal TLB overhead, 14% when the TLB overhead is included, and 15% for the program without *lhsz*. When we assume a utilization of 90%, the prediction error 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 is very accurate, with an error of less that 6%. The similar errors in the last two rows also suggest that our static estimation of the TLB overhead is accurate.

In the above predictions, we measured the amount of memory transfer through hardware counters. This was undesirable because we should predict program performance without running the program. So the next question was how accurate is the static estimation of a compiler. We hand applied the data analysis described in Section 3.1 to estimate the amount of memory transfer. In fact, we only used the bounded-section analysis, which counted only the number of capacity misses in each loop nest. We did not expect to see many conflict misses because the L2 cache on SGI Origin2000 is two-way set associative and 4MB in size.

We manually analyzed two subroutines: *compute_rhs* and *lhsx*, which together consist of 40% of the total running time. Subroutine *compute_rhs* had the largest code source and the longest running time among all subroutines. It also resembled the mixed access patterns in the whole program because it iterated the cubic data structures through three directions. The subroutine *lhsx*, however, accessed memory contiguously in a single stride. The following table lists the actual memory transfer measured by hardware counters, the predicted memory transfer by the static analysis, and the er-

[1] Pure data-copying loops with little computation can achieve a memory bandwidth that is slightly higher than 300MB/s on SGI Origin2000.

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%

Table 3: Actual and predicted execution time

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

Table 4: Actual and predicted data transfer

ror of the static estimation.

The errors shown in the third column of Table 4 are within 3%, indicating that the static estimation is indeed very accurate. In other words, compiler analysis is very accurate in estimating the total amount of data transfer between memory and cache. Assuming this accuracy holds for other parts of *SP*, the bandwidth-based analysis tool could predict the overall performance within an error of less than 10%, assuming 90% of average memory bandwidth utilization.

5 Related Work

Bandwidth-based performance tuning and prediction is different from previous techniques because of its focus on memory bandwidth consumption at the source level. Previous techniques either do not consider the effect of memory hierarchy or they use low-level performance models such as program traces or machine simulators. Compared to these techniques, the bandwidth-based method is simpler, more accurate, and more widely applicable to different machines and applications.

In the past, the modeling of memory hierarchy performance relied on measuring memory latency through machine simulation. Callahan et al.[10] first used compiler-based approach to analyze and visualize memory hierarchy performance with a memory simulator. Another approach is taken by Goldberg and Hennessy[11], who simulated program execution and measured memory stall time by comparing actual running time with simulation result of running the same program on a perfect memory. Machine simulation is not always convenient because it is expensive when measuring hardware events at high frequency or in great detail. So it has to sacrifice either simulation accuracy or use small program inputs. Simulation programs are architectural dependent and therefore not easily portable across different machines. Finally, simulation can not be used for predicting memory hierarchy performance because they have to run the program before collecting any performance data. Since most simulation methods impose the machine-level view of programs, it is not clear how to relate the results back to the program source and to the programmer.

Static or semi-static methods can be used to approximate run-time program behavior and thus predict program performance. Gallivan et al.[12] 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. To model communication performance in data-parallel programs, Bala et al. used training sets, which are a database for the cost of various communication operations[7]. They did not consider cache performance, although the idea of training can be applied to cache, for example, to predict effective bandwidth of memory access of different strides. Clements and Quinn predicted cache performance by multiplying the number of cache-misses with memory latency[13]. Their method is no longer accurate on modern machines, where memory accesses proceed in parallel with each other as well as with CPU computations. Moreover, they did not extend their work to support performance tuning.

Recently, researchers began to use bandwidth to measure machine memory performance. Examples are the STREAM benchmark programs by McCalpin[14] and the CacheBench programs by Mucci and London [15]. Both used simple program kernels to measure the available memory or cache bandwidth on machines. None of the previous work has explored the direction of tuning and predicting performance of full applications based on their bandwidth consumption.

Another large class of work we do not directly compare 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 uniprocessor. Since efficient parallel programs have coarse-grained parallelism and infrequent communication and synchronization, accurately predicting their computation cost is critical in predicting their overall performance. Furthermore, the bandwidth-based method presented in this paper can be extended to estimate communication performance on shared-memory machines, where communication bandwidth is frequently saturated as in the case of memory bandwidth.

6 Conclusions

This paper has presented the design of a compiler-directed tool for performance tuning and prediction. The tool sup-

ports user tuning by automatically locating program fragments with low memory hierarchy performance; it predicts performance statically by approximating it with the amount of memory-transfer time. Both methods are simple to implement but are very accurate and widely applicable to different machines and applications. When evaluated on the 3000-line NAS *SP* benchmark, the tool enabled a user to obtain an overall speedup of 1.19 by inspecting and tuning merely 5% of program source. It predicted the whole-program execution time at compile time to within 10% of the measured execution time.

Bandwidth-based performance tuning and prediction should be an integral part of any compiler strategy that seeks to maximize memory hierarchy performance. It complements the automatic compiler optimizations by enabling efficient manual tuning. It assists task or data parallelization by providing accurate performance estimation. Since the tool is part of an optimizing compiler, its estimation is accurate because the tool considers the effect of compiler transformations, and furthermore, it helps the compiler to make better optimization decisions.

References

[1] C. Ding and K. Kennedy. Memory bandwidth bottleneck and its amelioration by a compiler. Technical report, Rice University, May 1999. Submitted for publication.

[2] C. Ding and K. Kennedy. Inter-array data regrouping. In *Proceedings of The 12th International Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[3] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[4] 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.

[5] 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.

[6] 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.

[7] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

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

[9] 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.

[10] D. Callahan, K. Kennedy, and A. Porterfield. Analyzing and visualizing performance of memory hierarchies. In *Performance Instrumentation and Visualization*, pages 1–26. ACM Press, 1990.

[11] 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.

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

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

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

[15] Philips J. Mucci and Kevin London. The cachebench report. Technical Report ut-cs-98-394, University of Tennessee, 1998.