Memory Bandwidth Bottleneck and Its Amelioration by a Compiler

Chen Ding and Ken Kennedy

CRPC-TR99808-S
September 1999

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

Submitted November 1999

The Memory Bandwidth Bottleneck and its Amelioration by a Compiler

Chen Ding* Ken Kennedy
Rice University
Houston, TX

September 24, 1999

Abstract

As the speed gap between CPU and memory widens, memory hierarchy has become the primary factor limiting program performance. Until now, the principal focus of hardware and software innovations has been overcoming latency. However, the advent of latency tolerance techniques such as non-blocking cache and software prefetching begins the process of trading bandwidth for latency by overlapping and pipelining memory transfers. A direct consequence of such parallel memory transfers is the increased consumption of memory bandwidth. Since actual latency is the inverse of the consumed bandwidth, memory latency cannot be fully tolerated without infinite bandwidth. This perspective has led us to several intriguing questions. How much data bandwidth a program actually needs? Do current machines provide sufficient data bandwidth? If not, can a program be restructured to consume less bandwidth? How different is bandwidth reduction from traditionally studied problem of latency reduction? This paper answers these questions in two parts. The first part measures the demand and supply of data bandwidth through a new performance model and demonstrates the serious performance constraint due to the lack of memory bandwidth. The second part studies the problem of bandwidth reduction including the need for writeback elimination. A new set of compiler techniques are then proposed to minimize the overall memory transfer of a program.

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 *memory latency* on performance. This work has led to extremely effective techniques for reducing and tolerating memory latency, primarily through cache reuse and data prefetching.

As exposed memory latency is reduced, memory bandwidth consumption is increased. For example, when CPU prefetches two data items from memory, the actual latency per access is halved, but the memory bandwidth consumption is doubled. Cache or register reuse reduces the need for memory transfer, however, at the expense of higher bandwidth consumption at cache and register level. Since actual latency is the inverse of the consumed bandwidth, memory latency cannot be fully tolerated without infinite bandwidth. It is very likely that machine bandwidth may not be sufficient, and if so, program performance will be limited by its effective bandwidth regardless of the speed of processors or the physical latency of data access.

Because of the previous focus on memory latency, the potential bandwidth bottleneck has not been carefully studied nor has the strategy of bandwidth-oriented optimization. The purpose of this paper is to make a first step toward a better understanding of the bandwidth constraint on current machines and the key difficulties of reducing this bandwidth constraint. To this end, the paper first introduces a bandwidth-based performance model to measure and compare the demand and supply of data bandwidth and then develops new techniques that address the unique aspects of the bandwidth problem.

The new performance model measures the fundamental balance between the rate of computation and the rate of data transfer. The comparison between the balance of a machine and the balance of a program

[1]Email: cding@rice.edu, Phone: 713-527-8101 ext 2267, Fax: 713-285-5930.

can show whether the supply of data bandwidth meets its demand at all levels of memory hierarchy. Such a comparison can give quantitative answers to a number of intriguing questions. For example, whether machine bandwidth is adequate, if machine bandwidth is insufficient, which memory level is the most restrictive, and how program performance is limited because of insufficient bandwidth.

Once the bandwidth bottleneck is identified, this paper demonstrates the need for new techniques to address the bandwidth problem. Indeed, reducing bandwidth consumption or the amount of data transfer is not the same problem as reducing access latency. While latency is a local and isolated behavior of each data access, bandwidth consumption is a global and collective property of the whole program. Therefore, the most urgent need is global optimizations, which have not been fully studied previously. Another salient example is data writebacks. While writebacks do not incur long latency as data reads, they consume the valuable bandwidth. New techniques are needed to minimize memory writebacks because one of the previous techniques has been targeted specifically for this purpose.

The rest of the paper flashes out the performance model and the compiler strategy for amelioration the bandwidth problem. Section 2 defines the bandwidth-based performance model and uses it to capture the existence and the effect of the memory bandwidth bottleneck on modern machines. Section 3 discusses the implications of the bandwidth bottleneck and describes new compiler transformations for bandwidth reduction. Finally, Section 4 discusses related work and Section 5 summarizes the contributions of this work.

2 Memory bandwidth bottleneck

This section starts with a convincing example which demonstrates that memory latency is no longer the primary factor in determining program performance. The reason, which is the memory bandwidth bottleneck, is then sought through the measurement and comparison of the balance of computation and data transfer. Finally, the section returns to the latency issue and demonstrates that bandwidth constraints are indeed dominant in determining actual memory latency in real-world programs.

2.1 A simple example

The example program has two loops performing stride-one access to a large data array. The only difference is that the first one also writes the array back to memory. Since both loops perform the same reads, they should have the same latency and the same performance if latency is the determining factor. Both loops also have the same number of floating-point operations.

```
double precision A[2000000]

for i=1 to N
   A[i] = A[i]+0.4
end for

for i=1 to N
   sum = sum+A[i]
end for
```

Surprisingly, the first loop takes 0.104 second on SGI Origin2000, which is almost twice the execution time of the second loop, 0.054 second. On HP/Convex Exemplar, the first loop takes 0.055 second and the second 0.036. The reason, as shown next, is that the performance is determined by memory bandwidth, not memory latency. The first loop takes twice as long because it writes the array to memory and consequently consumes twice as much memory bandwidth.

2.2 Mismatch between program and machine balance

To understand the supply and demand of memory bandwidth as well as other computing resources it is necessary to go back to the basis of a computing system: the balance between computation and data transfer. This section formulates a performance model based on the concept of balance and uses the model to examine the performance bottleneck on current machines.

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

Applications	Program/machine Balance		
	L1-Reg	L2-L1	Mem-L2
convolution	6.4	5.1	5.2
dmxpy	8.3	8.3	8.4
mm (-O2)	24.0	8.2	5.9
mm (-O3)	8.08	0.97	0.04
FFT	8.3	3.0	2.7
NAS/SP	10.8	6.4	4.9
Sweep3D	15.0	9.1	7.8
Origin2000	4	4	0.8

Figure 1: Comparison between program and machine balance

computation operation; the balance of a machine (*machine balance*) is the amount of data transfer that the machine provides for each machine operation. Specifically, for a scientific program, the program balance is the average number of bytes that must be transferred per floating-point operation (*flop*) in the program; the machine balance the number of bytes the machine can transfer per flop in its CPU peak rate. On machines with multiple levels of cache memory, the balance includes the data transfer between all adjacent levels.

We measured the balance of six representative scientific applications, including four kernels—convolution, dmxpy, matrix multiply, FFT—and two full applications—SP from the NAS benchmark suite and Sweep3D from DOE. We compiled all programs with the highest optimization level, and during their executed on Origin2000, we measured the total number of flops, register loads and stores, and cache misses and writebacks through the hardware counters. We then obtained the program balances by calculating the average bytes of data transfer per flop, which are shown in the first seven rows of Figure 1. For example, for each flop, *convolution* requires transferring 6.4 bytes between level-one cache (L1) and registers, 5.1 between two caches, and 5.2 between level-two cache (L2) and memory. In addition to fully optimized matrix multiply, (*mm (-O3)*, we also tested the program with a lower optimization level as (*mm (-O2)*.

We calculated the machine balance of Origin2000 by dividing its peak flop rate with its peak data bandwidth at all levels of memory hierarchy. The peak flop rate and load/store bandwidth are given in hardware specification. The cache bandwidth is measured by CacheBench[14], and the memory bandwidth by STREAM[12]. The last row of Figure 1 gives the machine balance, which shows that for each flop on its peak flop rate, Origin2000 can only transfer 4 bytes between registers and L1 cache, 4 bytes between two caches, and merely 0.8 byte between L2 cache and memory.

As the table in Figure 1 shows, with the exception of *mm(-O3)*, all applications demand substantially higher amount of data transfer between L2 and memory (in the last column) than that provided by Origin2000. The demands are 2.7 to 8.4 bytes per flop while the supply is only 0.8 byte per flop. The striking mismatch clearly confirms that memory bandwidth is a serious performance bottleneck. In fact, memory bandwidth is the most insufficient resource because the mismatch at L2-Memory is much larger than all other levels of memory hierarchy, shown by the second and the third column. The next section will take a closer look at this memory bandwidth bottleneck.

The reason matrix multiply *mm (-O3)* requires very little memory transfer is that the compiler exploits sufficient register and cache reuse with advanced computation blocking, first developed by Carr and Kennedy[3]. The result of *mm (-O3)* is a clear evidence showing that it is possible for a compiler to significantly reduce the application's demand for memory bandwidth; however, the current compiler is not effective for all other programs. We will return to compiler issues in the second part of the paper.

2.3 Memory Bandwidth Bottleneck

The precise ratios of between the demand of data bandwidth and its supply can be calculated by normalizing program balances with the machine balance of Origin2000. The results are listed in Figure 2, which shows that for each application at each memory hierarchy level, the application demands how many times as much data bandwidth as provided by the machine. The last column shows that programs require memory

Applications	Ratios of demand over 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
Sweep3D	3.8	2.3	9.8

Figure 2: Ratios between bandwidth demand and supply

bandwidth 3.4 to 10.5 times as much as that machine memory bandwidth, making memory bandwidth the most critical resource. The data bandwidth on other levels of memory hierarchy is also insufficient by factors between 1.3 to 6.0, but the problem is less serious.

The insufficient memory bandwidth compels applications into unavoidable low performance simply because memory data cannot be delivered fast enough to keep CPU busy. A ratio of 10.5 means a CPU utilization of no more than 1/10.5, or 9.5%. The last two rows show a grim picture for even full applications: their average CPU utilization can be no more than 16% for *NAS/SP* and 10% for *Sweep3D*. In other words, over 80% of CPU power is left unused because of the memory bandwidth bottleneck.

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. Therefore, a machine needs to have 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 CPU speed rapidly increases, future systems will have even worse balance and more serious bottleneck because of the lack of memory bandwidth.

2.4 Bandwidth constraint vs. latency

The balance-based performance model does not include the effect of the latency constraint, in particular, the effect of memory latency. It is possible that memory access incurs a latency that is so high that even the limited memory bandwidth is scarcely used. So the question remains that whether the insufficient memory bandwidth is directly limiting program performance. In other words, is current memory bandwidth saturated. This section uses a set of program kernels and a large application to show that memory bandwidth is indeed saturated in most cases. Later sections will also confirm the dominating effect of the bandwidth bottleneck on latency by showing the significant performance gain through bandwidth-reduction optimizations.

The *effective bandwidth* of a program is measured by dividing the total memory transfer (both reads and writebacks) with its execution time. We used 13 kernels, which access a different number of arrays in a single stride. The kernels are named by the number of arrays they read and write. For example, kernel $1w1r$ reads and writes a single array, and kernel $1w2r$ reads two arrays and writes to one of them. Figure 3 shows both the effective memory bandwidth of these kernels on both Origin2000 and Exemplar.

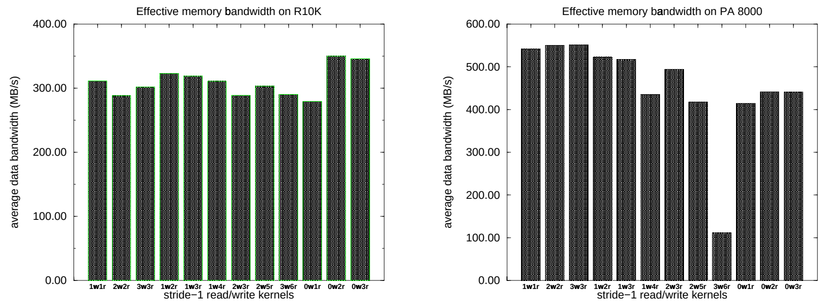

Figure 3: Effective memory bandwidth of read/write kernels

The results in Figure 3 show that all kernels have similar effective bandwidth. On Origin2000, the difference is within 20% among all kernels. On Exemplar, the effective bandwidth ranges from 417 MB/s to 551 MB/s with the exception of *3w6r*[1]. Given the typical variations of measurements on these parallel machines, the results strongly suggest that all kernels are reaching the bandwidth limit of the machine.

In addition to these kernels, we measured a 3000-line application, the *NAS/SP* benchmark from NASA. We found that 5 out of its 7 major computation subroutines utilized 84% or higher of the memory bandwidth of Origin2000. The high bandwidth utilization shows that memory bandwidth is the major performance bottleneck for *SP*, and the bandwidth saturation we see on those kernels is indeed happening on full applications as well. We cannot measure the bandwidth utilization of *SP* on Exemplar because of the absence of hardware counters.

The kernels represent difference patterns of stride-one memory access. *NAS/SP* is a real application with regular data access patterns and heavy computations. Since they resemble most programs with regular computations, the above results indicate that these programs are saturating memory bandwidth on current machines. Thus, memory bandwidth is indeed the performance bottleneck for these applications.

In conclusion, the empirical study has shown that for most applications, memory bandwidth is up to 10 times insufficient. As a result, over 80% of CPU power is unutilized by large applications, indicating a potential performance gain of large factors if only the applications can reduce their demand on memory transfer. The next section introduces new compiler techniques that are aimed at exactly this goal, that is, reducing memory bandwidth demand of applications.

3 Bandwidth reduction by a compiler

Memory bandwidth bottleneck poses a problem very different from that of memory latency, and consequently bandwidth-oriented optimizations are not the same as latency-oriented ones. For many years, a large body of research has been devoted to ameliorating the effect of long memory latency. The techniques fall into two categories: latency hiding through data prefetching and latency reduction through cache reuse. Data prefetching does not help the bandwidth problem at all because it does not reduce a single byte of memory transfer. In fact, prefetching techniques often cause additional memory transfer due to the prefetching overhead.

Latency reduction techniques reduce memory bandwidth consumption. However, previous techniques are not adequate mainly for two reasons. For latency, it is sufficient to consider only memory reads, and to deal with each loop nest individually. To fully address the bandwidth problem, however, a program needs to minimize the total amount of memory transfer, not just memory reads or inside single loops. For example, as we have seen in the measurement of program balance, the amount of computation in a single loop is too little to sustain full CPU utilization during the time of memory transfer. The natural way to change the balance is to bring in other computations on the same data from other parts of the program, that is, to exploit global data reuse. Without the bandwidth bottleneck, however, global data reuse is not necessary since latency hiding in a single loop can alway shorten the actual latency low enough to match the CPU throughput.

This section explores solutions to global cache reuse and memory writebacks reduction. It first formulates the global loop fusion problem, studies its complexity and solutions. While many early researches have studied loop fusion, they did not formulate it with the goal of maximal data reuse and minimal memory bandwidth consumption. The rest of the section then describes two techniques for reducing memory writebacks, first by avoiding writing back read-only data and then by aggressively eliminating even the writebacks of read-write data.

3.1 Reuse-based loop fusion

The section first formulates the problem of loop fusion for minimizing program data transfer, then gives a polynomial solution to a restricted form of this problem, and finally proves that the complexity of the unrestricted form is NP-complete. In the process, it also points out the inadequacy of the fusion model used by the previous existing work by Gao et al.[9] and Kennedy and McKinley[11]. In this section, a program is

[1]We suspect that *3w6r* kernel causes excessive cache conflicts because it accesses 6 large arrays on a direct-mapped cache. Cache conflicts result in a much higher amount of data transfer, which we cannot measure because of the absence of hardware counters on Exemplar

assumed to be structured in loops and arrays, but the formulation and solution apply to programs in other language structures such as recursive functions and object-based data.

Loop fusion for minimal memory transfer

Given a sequence of loops accessing a set of data arrays, we can model both the computation and the data in a *fusion graph*. A fusion graph consists of nodes—each loop is a node— and three types of edges—directed edges for modeling data dependences and undirected edges for fusion-preventing constraints. Although this definition of a fusion graph looks similar to that of previous work, the objective of fusion is radically different as stated in the following.

Problem 3.1 Reuse-based fusion problem: *Given a fusion graph, how can we divide the nodes into a sequence of partitions such that*

- *(Correctness) each node appears in one and only one partition; the nodes in each partition have no fusion preventing constraint among them; and dependence edges only flow from an earlier partition to a later partition in the sequence,*
- *(Optimality) the sum of the number of distinct arrays in all partitions is minimal.*

The correctness constraint ensures that loop fusion obeys data dependences and fusion preventing constraints. Assuming arrays are large enough to prohibit any cache reuse among disjoint loops, the second requirement ensures optimality because for each loop, the number of distinct arrays is the number of arrays the loop needs to transfer from memory during execution. Therefore, the minimal number of arrays in all partitions means the minimal memory transfer and minimal bandwidth consumption for the whole program.

For example, Figure 4 shows the fusion graph of six loops. Assuming that loop 5 and loop 6 cannot be fused, but either of them can freely fuse with any other four loops, and loop 6 depends on loop 5. Without any fusion, the total number of arrays in the six loops are 20, so the program needs a total memory transfer of 20 arrays.

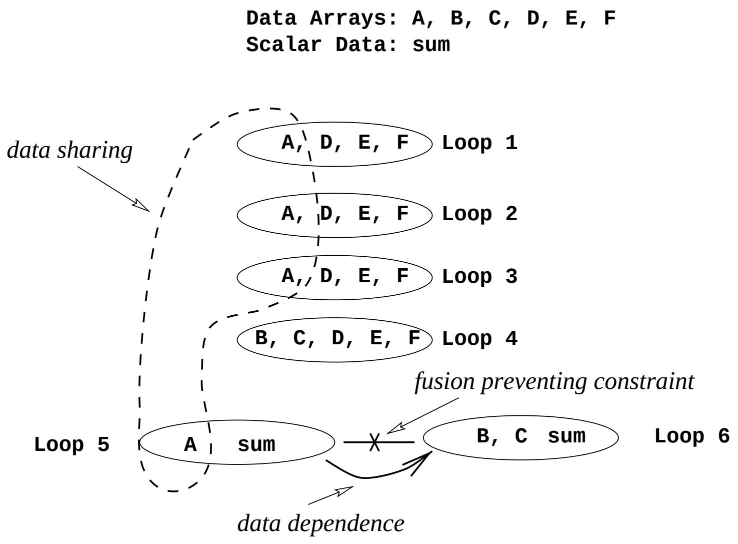

Figure 4: Example formulation of bandwidth-minimal loop fusion

The optimal fusion would leave loop 5 alone and fuse all other loops. The total number of arrays is one in the first partition and six in the second, thus the total amount of memory transfer is reduced from 20 arrays to 7.

The optimality definition of reuse-based fusion is different from previous work on loop fusion. Both Gao et al.[9] and Kennedy and McKinley[11] studied loop fusion for register reuse. They constructed a fusion graph in a similar way but modeled data reuse as weighted edges between graph nodes. For example, the edge weight between loop 1 and 2 would be 4 because they share four arrays. Their goal is to partition the nodes so that the total weight of the cross-partition edges is minimal.

The simple addition of edge weights does not model data reuse precisely. For example, in Figure 4, each of loop 1 to 3 has a single-weight edge to loop 5. But the summarized reuse between the first three loops and loop 5 should not be the addition of the edge weights, which adds to 3; on the contrary, the summarized reuse is only one because they share access to only array A.

To show that weighted-edge formulation is not optimal, it is suffice to give a counter example, which is the real purpose of Figure 4. The optimal weighted-edge fusion is to fuse the first five loops and leave loop 6 alone. Only the edge between loop 4 and 6 lies between the two partitions after fusion, which has a total weight of 2. No other fusion can result in a smaller inter-partition edge weight. However, this fusion has to load 8 arrays (6 in the first partition and 2 in the second), while the previous reuse-based fusion needs only 7. In fact, the total inter-partition edge weight of the previous fusion is 3, clearly not optimal based on weighted-edge formulation. There, fore the previous formulations of weighted-edge fusion does not minimize overall memory transfer.

Not only is the formulation of reuse-based fusion different from previous work, but also the complexity of the problem is different. While two-partitioning on an weighted-edge graph can be efficiently solved by the well-known max-flow/min-cut algorithm, the same algorithm would not find the optimal cut for the fusion problem in Figure 4. To understand the complexity of reuse-based fusion, the next section studies a model based on a different type of graphs.

Solution based on hyper-graphs

The traditional definition of an edge is inadequate in modeling data use because the same data can be shared by more than two loops. To model arbitrary data sharing, reuse-based fusion uses hyper-edges. A *hyper-edge* can connect any number of nodes in a graph. A graph with hyper-edges is called a *hyper-graph.* The optimality requirement can now be restated—the goal of fusion is to minimize the sum of the number of hyper-edges connecting each partition.

This section first solves the problem of optimal two-partitioning on hyper-graphs and then proves the NP-completeness of multi-partitioning.

Two-partitioning is a special class of the fusion problem where the fusion graph has only one fusion-preventing edge and no data dependence edge. The result of fusion will produce two partitions. The example in Figure 4 is a two-partitioning problem.

Two-partitioning can be solved as a connectivity problem between two nodes. Two nodes are connected if there is a path between them. A *path* exists between two nodes in a hyper-graph if there is a sequence of hyper-edges where the first one connects one node, the last one connects the other, and consecutive ones connect a same group of nodes.

Given a hyper-graph with two end nodes, a *cut* is a set of hyper-edges such that taking out these edges would disconnect the end nodes. In a two-partitioning problem, any cut is a legal partitioning. The size of the cut determines the total amount of data loading, which is the total amount of data plus the size of the cut (which is the total amount of data reloading). Therefore, to obtain the optimal fusion is to find a minimal cut.

The algorithm given is Figure 5 finds a minimal cut for a hyper-graph. At its first step, the algorithm transforms the hyper-graph into a normal graph by converting each hyper-edge into a node, and connecting two nodes in the new graph when the respective hyper-edges overlap. The conversion also constructs two new end nodes for the transformed graph. The problem now becomes finding minimal vertex cut on a normal graph. The second step applies standard algorithm for minimal vertex cut, which converts the graph into a directed graph, splits each node into two and connects them with a directed edge, and finally finds the edge cut set by the standard Ford-Fulkerson method. The last step transforms the vertex-cut to the hyper-edge cut in the fusion graph and constructs the two partitions.

Although algorithm in Figure 5 can find minimal cut for hyper-edges with non-negative weights, we are only concerned with fusion graphs where edges have unit-weight. In this case, the first step of the minimal-cut algorithm in Figure 5 takes $O(E + V)$; the second step takes $O(V'(E' + V'))$ if breadth-first search is used to find augmenting paths; finally, the last step takes $O(E + V)$. Since $V' = E$ in the second step, the overall cost is $O(E(E' + E) + V)$, where E is the number of arrays, V is the number of loops and E' is the number of the pair of arrays that are accessed by the same loop. In the worst case, $E' = E^2$, and the

Input A hyper-graph $G = (V, E)$.
Two nodes s and $t \in V$.

Output A set of edges C, which is a minimal cut between s and t.
Two partitions V_1 and V_2, where $s \in V_1$, $t \in V_2$, $V_1 = V - V_2$, and a edge e connects V_1 and V_2 iff $e \in C$.

Algorithm

```
/* Initialization */
let C, V1 and V2 be empty sets

/* Step 1: convert G to a normal graph */
construct a normal graph G'=(V',E')
   let array map be the one-to-one map between V' and E
   add a node v to V' for each hyper-edge e in E; let map[v] = e
   add edge (v1, v2) in G' iff map[v1] and map[v2] overlap in G

   /* add in two end nodes */
   add two new nodes s' and t' to V'
   add edge (s', v) if map[v] contains s
   add edge (t', v) if map[v] contains t

/* Step 2: find the minimal vertex cut in G' between s' and t' */
convert G' into a directed graph
split each node in V' and add in a directed edge in between
use For-Fulkerson method to find the minimal edge cut
convert the minimal edge cut into the vertex cut in G'

/* Step 3: construct the cut set and the partitions in G*/
let C be the cut set found in the previous step
delete all edges in C from G
let V1 be the set of nodes connected to s in G; let V2 be V-V1
return C, V1 and V2
```

Figure 5: Minimal-cut algorithm for a hyper-graph

algorithm takes $O(E^3 + V)$. What is surprising is that although the time is cubic to the number of arrays, it is linear to the number of loops in a program.

By far the solution methods have assumed no dependence edges. The dependence relation can be enforced by adding hyper-edges to the fusion graph. Given a fusion graph with N edges with two end nodes s and t, assume the dependence relations form an acyclic graph. Then if node a depends on b, we can add three sets of N edges connecting s and a, a and b, and b and t. Minimal-cut will still find the minimal cut although each dependence adds a weight of N to the total weight of minimal cut. Any dependence violation would add an extra N to the weight of a cut, which makes it impossible to be minimal. In other words, any minimal cut will not place a before b, and the dependence is observed. However, adding such edges would increase the time complexity because the number of hyper-edges will be in the same order as the number of dependence edges.

Complexity of reuse-based fusion

Although the two-partitioning problem can be solved in polynomial time, the multi-partitioning form of reuse-based fusion is NP-complete.

First, the fusion problem is in NP because loops or nodes of a fusion graph can be partitioned in a nondeterministic way, and the legality and optimality can be checked in polynomial time.

The fusion problem is also NP-hard. To prove this, we reduce Multi-way cut problem to the fusion problem. Given a graph $G = (V, E)$ and k nodes to be designated as terminals, k-way cut problem is to find a set of edges of minimal total weight such that removing the edges renders all k terminals disconnected from each other. Dahlhaus et al.[4] proved that k-way cut is NP-hard when $k \geq 3$, even for edges of equal weight. To convert a k-way cut problem to a fusion problem, we construct a hyper-graph $G' = (V', E')$ where $V' = V$. We add in a fusion preventing edge between each pair of terminals, and for each edge in E, we add a hyper-edge connecting only two nodes. It is easy to see that a minimal k-way cut in G is an optimal fusion in G' and vice versa. Therefore, reuse-based fusion is NP-complete when the number of partitions after fusion is greater than or equal to three.

3.2 Writeback reduction

This section addresses another unique aspect of the bandwidth problem, that is, the need to reduce memory writebacks. We present and evaluate two bandwidth-reduction methods: the first, *data regrouping*, avoids writing back read-only data by separating read-only data from read/write data, the second, *store elimination*, reduces the writeback of read/write data by pro-actively completing all the uses of the data.

Data regrouping

Data regrouping avoids writing back read-only data by separating read-only data from read/write data, as illustrated by the following example loop. Assuming a cache line holds two real numbers, the loop accesses only a half of each cache line because it uses only one number in the first dimension of data arrays.

```
real*8 state1(2,N),step1(2,N), state2(2,N),step2(2,N)

do i=1,N
   state1(t,i) = state1(t,i) + step1(t,i)
   state2(t,i) = state2(t,i) + step2(t,i)
enddo
```

To improve cache-block utilization, the program can regroup the simultaneously used data into the same cache block. Since the ith element of four arrays are always accessed at the same time, they should be placed together in memory. The choice of regrouping, however, is different depending on whether memory writebacks are considered. If memory latency is the only concern, four arrays should be interleaved into a single array, as shown in the left-hand side of Figure 6. However, regrouping all four arrays mixes read-write data with read-only data and causes redundant data writebacks. Another regrouping transforms four arrays into two arrays: one contains the read-only arrays, and the other holds the read-write ones. The result of regrouping is shown in the right-hand side of Figure 6. Although the second regrouping has the same memory read latency, it writes back only half as much as data as the first regrouping. We call the first regrouping *latency reduction* and the second one *bandwidth reduction*.

```
real*8 s(4,2,N)

do i=1,N
   s(1,t,i) = s(1,t,i) + s(2,t,i)
   s(3,t,i) = s(3,t,i) + s(4,t,i)
enddo
```

```
real*8 s1(2,2,N), s2(2,2,N)

do i=1,N
   s1(1,t,i) = s1(1,t,i) + s2(1,t,i)
   s1(2,t,i) = s1(2,t,i) + s2(2,t,i)
enddo
```

Figure 6: Difference between latency reduction and bandwidth reduction

The performance of the two regrouping were measured on both Origin2000 and Exemplar; the execution time and the amount of memory traffic are shown in Table 1. We slightly changed the example because the L2 cache line is 128 bytes on Origin2000 and 64 bytes on Exemplar. The results show that the regrouping for minimal memory writebacks achieved much better performance than no regrouping and regrouping for latency only. On both machines, writeback reduction was about 25% faster than latency reduction. The hardware counters also verified that bandwidth reduction eliminates a half of the memory writebacks on Origin2000.

machine	measurement	no regrouping	latency reduction	writeback reduction
Origin2000	execution time	1.66 sec	1.30 sec	1.01 sec
	memory reads	512 MB	256 MB	256 MB
	memory writebacks	256 MB	256 MB	128 MB
Exemplar	execution time	0.68 sec	0.25 sec	0.19 sec

Table 1: Effect of data regrouping on writeback reduction

The general formulation and solution of data regrouping is given in[8]. As illustrated by the paper, the regrouping algorithm first divides a program into computation phases and partitions its arrays into compatible sets. Regrouping is applied within each compatible set by grouping only those arrays that are always accessed together. The above example uses the same algorithm to separate read-only data from read-write data. In other words, two arrays are merged if they are either both read-only or both read-write whenever they are accessed.

A hardware solution for this problem would be putting a modify-bit for each word in a cache line and, at the time of writeback, transferring only the data that has been changed. But this requires modifications to the hardware design of memory system and causes extra overhead in communication protocols. The tradeoff is similar to that of using small cache blocks, where the benefit may be negated by the higher hardware cost and lower performance for well-behaved programs. The compiler-directed regrouping, on the other hand, does not require hardware changes nor sacrifice peak performance. Since a compiler has a global view of computation and data, it may well eliminate most of the unnecessary writebacks of read-only data. Furthermore, a compiler can capture the opportunities that hardware cannot, as shown in the next subsection.

Store elimination

Since the purpose of writebacks is to use the data later in the program, we can eliminate the writebacks entirely by pro-actively finishing all the uses of the data. Figure 7 illustrates this store-elimination transformation.

The first loop in Figure 7 assigns new values to the *results* array, which is used in the next loop. We can fuse the two loops and use the results right after they are computed. Then the writeback of the updated *results* array is no longer necessary. The transformed code after fusion and store elimination is shown as the second program in Figure 7.

Table 2 lists the reduction in execution times by loop fusion and store elimination. Fusion without store elimination reduces running time by 31% on Origin and 13% on Exemplar; store elimination further reduces

```
real*8 sum, results(N),data(N)

do i=1,N
   results(i) = results(i) + data(i)
enddo

sum = 0.0
do i=1,N
   sum = sum + results(i)
enddo

print sum
```

```
real*8 sum, results(N),data(N)

sum = 0.0
do i=1,N
   sum = sum + results(i) + data(i)
enddo

print sum
```

Figure 7: Example of store elimination

execution time by 27% on Origin and 33% on Exemplar. The combined effect is a speedup of almost 2 on both machines, clearly demonstrating the benefit of store elimination.

machines	original	fusion only	store elimination
Origin	0.32 sec	0.22 sec	0.16 sec
Exemplar	0.24 sec	0.21 sec	0.14 sec

Table 2: Effect of store elimination

Even better, if we can further fuse not only all the uses of an array, but also its definition, we can eliminate the use of the whole array. In the previous example, if *results* array carries no live value upon entering the first loop, we can eliminate of the use of the *results* array entirely.

In real programs, loop alignment needs to be used to allow the fusion of loops that have certain dependences. As a result, the distance between the definitions and uses of an array element may span multiple iterations. Indeed, the use of an array element may be in the next iteration of the outer loop. In such cases, the use of a whole array cannot be eliminated. However, store elimination can use two other techniques: array peeling and array shrinking. Array peeling saves the fraction of the array whose uses cannot be completed until some later part of the program. Array shrinking uses a smaller buffer to hold array elements that live across multiple iterations. In both cases, the majority of the writebacks can be eliminated. We are currently designing and evaluating techniques of array peeling and array shrinking.

4 Related work

Callahan et al. first used the concept *balance* to study whether register throughput can keep up with the CPU demand for scientific applications[2]. They also studied compiler transformations that can restore program balance by reducing the number of loads/stores through register reuse. Our work is close in spirit to their work, but we extend the balance to include all levels of memory hierarchy, and our compiler transformations focus on reducing memory transfer.

Many loop-level transformations have been used to reduce memory latency through register and cache reuse. These transformations also reduce the amount of memory transfer, but they do not exploit global data reuse because they do not bring together data access of multiple loops. Gao et al.[9] and Kennedy and McKinley[11] pioneered loop fusion for the purpose of achieving register reuse across loop boundary. They used weighted edges to represent data reuse between a pair of loops. Normal edges, however, cannot model data reuse accurately because the same data can be shared by more than two loops. Consequently, their formulation does not maximize data reuse and minimize total amount of data transfer into registers or cache. The loop fusion formulation presented in this paper uses hyper-edges to model data reuse precisely and therefore minimizes the total amount of data transfer through maximal data reuse among all loop nests. Kennedy and McKinley proved that k-way fusion is NP-hard, and both Gao et al. and Kennedy and McKinley gave a heuristic which recursively bisect the fusion graph through minimal cut. The minimal-cut algorithm presented in this paper can be used in their heuristic to perform k-way fusion.

We are not aware of any previous study with the explicit goal of writeback reduction. One technique called array contraction, designed by Sarkar and Gao[15] for latency reduction, can also eliminate writebacks by replacing an array with a scalar. The store elimination presented in this paper is more general and powerful because it can eliminate the writebacks of arrays that cannot be substituted with a scalar. In addition, the method in this paper relies on loop fusion to provide opportunities for store elimination while the previous work avoided this problem by requiring programs to be written in a single-assignment functional language.

The reuse-based fusion and writeback reduction are components of a compiler strategy currently being developed in Ding's dissertation[5]. The strategy seeks to minimize memory transfer through a two-step principle: to first fuse the computation on the same data and then group the data used by the same computation. The strategy first employs the techniques presented in this paper to perform computation fusion and data grouping at the global level. The fusion and grouping transformation can also be applied at run time for dynamic applications with locality grouping and data packing[7]. Finally, the compiler strategy supports user tuning and machine scheduling with bandwidth-based performance tuning and prediction, which is much simpler than latency-based performance tools and is very effective for tuning and predicting performance for large applications[6].

Many architectural studies examined the memory bandwidth constraint. McCalpin [13] used the STREAM benchmark to demonstrate that machines had became increasingly imbalanced because of the limited memory bandwidth. He did not provide a model nor measurement to examine the overall balance of programs and machines. Burger et al.[1] simulated SPEC programs and concluded that memory bandwidth would be the major architectural bottleneck for future processors. Huang and Shen [10] studied what they called intrinsic bandwidth requirement by directly measuring the reuse of values. Both Burger et al and Huang and Shen used machine simulation so they could examine the effect of different cache parameters. They measured programs and machines in detail at the expense of slow and architecture-dependent simulation. The balance model used in this work focuses directly and solely on the bandwidth of all levels of memory hierarchy. The program and machine balance can be measured directly, and the comparison between machine bandwidth and program effective bandwidth can examine the effect of latency constraint, both without relying on expansive program and machine simulation. Thus the bandwidth-based performance model presented in this paper is much better for practical use because it is much faster and more accurate than previous simulation-based performance models. The new technique is also portable because it can be easily adapted to different machines and compilers. Another important limitation of the previous simulation studies is that they considered only existing program optimizations that were implemented in the compiler they used. For example, Burger et al. relied on hardware data prefetching while most current machines such as Origin2000 are equipped with software data prefetching. None of the simulation work considered the potential of bandwidth reduction transformations.

5 Contributions

In this paper, we have shown the serious performance bottleneck due to the limited memory bandwidth, and we have presented new compiler techniques for reducing memory traffic through reuse-based computation fusion and writeback reduction. Specifically, the contributions are

- A bandwidth-based performance model called *balance*, which measures the demand and supply of data bandwidth on all levels of memory hierarchy.
- A new model of loop fusion which, unlike previous existing work, is the first formulation that minimizes the overall bandwidth consumption of a program. An optimal two-partitioning solution that takes $O(E^3 + V)$, where E is the number of arrays and V is the number of loops. An NP-completeness proof for cases of more than two partitions.
- Compiler transformations for minimizing memory writebacks, which employs loop fusion, data grouping and store elimination with array peeling and shrinking.

Acknowlegement

Special thanks to Nathaniel Dean, who suggested modeling reuse-based loop fusion as a connectivity problem on hyper-graphs.

References

[1] 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.

[2] 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.

[3] S. Carr and K. Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.

[4] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The complexity of multiway cuts. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, May 1992.

[5] C. Ding. Improving effective bandwidth through compiler enhancement of global and dynamic cache reuse. Dissertation in preparation, Rice University.

[6] C. Ding and K. Kennedy. Bandwidth-based performance tuning and prediction. In *Proceedings of IASTED International Conference on Parallel Computing and Distributed Systems*, November 1999.

[7] 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.

[8] 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.

[9] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[10] S. Andrew Huang and John Paul Shen. The intrinsic bandwidth requirements of ordinary programs. In *Proceedings of the 7th International Conferences on Architectural Support for Programming Languages and Operating Systems*, 1996.

[11] K. Kennedy and K. S. M^{c}Kinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, August 1993. (also available as CRPC-TR94370).

[12] John D. McCalpin. The stream benchmark site. http://www.cs.virginia.edu/stream/.

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

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

[15] V. Sarkar and G. Gao. Optimization of array accesses by collective loop transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.