The Memory Bandwidth Bottleneck and its Amelioration by a Compiler

Chen Ding Ken Kennedy
Rice University, Houston, Texas, USA
{*cding,ken*}*@cs.rice.edu*

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. 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 two questions. Do current machines provide sufficient data bandwidth? If not, can a program be restructured to consume less bandwidth? This paper answers these questions in two parts. The first part defines a new bandwidth-based performance model and demonstrates the serious performance bottleneck due to the lack of memory bandwidth. The second part describes a new set of compiler optimizations for reducing bandwidth consumption of programs.

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 simultaneously fetches two data items from memory, the actual latency per access is halved, but the memory bandwidth consumption is doubled. Since actual latency is the inverse of the consumed bandwidth, memory latency cannot be fully tolerated without infinite bandwidth. Indeed on any real machine, program performance is bounded by the limited rate at which data operands are delivered into CPU, regardless of the speed of processors or the physical latency of data access.

Because of the past focus on memory latency, the bandwidth constraint had not been carefully studied nor had the strategy of bandwidth-oriented optimization. The purpose of this paper is to address these two important issues. The first part of the paper introduces a bandwidth-based performance model and presents a performance study based on this model. By measuring and comparing the demand and supply of data bandwidth on all levels of memory hierarchy, the study reveals the serious performance bottleneck due to the lack of memory bandwidth.

The second part of the paper presents new compiler optimizations for reducing bandwidth consumption of a program. Unlike memory latency which is a local attribute of individual memory references, bandwidth consumption of a program is a global property of all memory access. Therefore, a compiler needs to transform the whole program, not just a single loop nest. In addition, since memory writebacks equally consumes bandwidth as memory reads, a compiler needs to optimize data stores, not just data loads. For these purposes, this paper introduces three new techniques. The first is *bandwidth-minimal loop fusion*, which studies how to organize global computation so that the total amount of memory transfer is minimized. The second is *storage reduction*, which reduces the data footprint of computation. The last one is *store elimination*, which removes writebacks to program data. These three techniques form a compiler strategy that can significantly alleviate the problem of memory bandwidth bottleneck.

The rest of the paper is organized as follows. Section 2 defines the bandwidth-based performance model and measures the memory bandwidth bottleneck. Section 3 describes three new compiler transformations for bandwidth reduction. Section 4 discusses related work and Section 5 summarizes.

2 Memory bandwidth bottleneck

This section first shows an example where memory latency is clearly not the primary constraint on performance. The most serious constraint is memory bandwidth, as studied in the rest of the section.

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 a R10K processor of 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 Program and machine balance

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

Both a program and a machine have balance. *Program balance* is the amount of data transfer (including both data reads and writes) that the program needs for each computation operation; *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 is the number of bytes the machine can transfer per flop in its peak flop rate. On machines with multiple levels of cache memory, the balance includes the data transfer between all adjacent levels.

Programs	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. Program and machine balance

The table in Figure 1 compares program and machine balance. The upper half of the table lists the balance of six representative scientific applications[1], including four kernels—convolution, dmxpy, matrix multiply, FFT—and two application benchmarks—SP from the NAS benchmark suite and Sweep3D from DOE. For example, the first row shows that for each flop, *convolution* requires transferring 6.4 bytes between the level-one cache (L1) and registers, 5.1 bytes between L1 and the level-two cache (L2), and 5.2 bytes between L2 and memory. The last row gives the balance of a R10K processor on SGI Origin2000[2], which shows that for each flop at its peak performance, the machine can transfer 4 bytes between registers and cache, 4 bytes between L1 and L2, but merely 0.8 bytes between cache and memory.

As the last column of the table shows, with the exception of *mm(-O3)*, all applications demand a substantially higher rate of memory transfer than that provided by Origin2000. The demands are between 2.7 to 8.4 bytes per flop, while the supply is only 0.8 byte per flop. The striking mismatch clearly confirms the fact that memory bandwidth is a serious performance bottleneck. In fact, memory bandwidth is the least sufficient resource because its mismatch is much larger than that of register and cache bandwidth, shown by the second and the third column in Figure 1. 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 at the highest optimization level of -O3, the compiler performs advanced computation block-

[1] Program balances are calculated by measuring the number of flops, register loads/stores and cache misses/writebacks through hardware counters on SGI Origin2000.

[2] The machine balance is calculated by taking the flop rate and register throughput from hardware specification and measuring memory bandwidth through STREAM[8] and cache bandwidth through CacheBench[9].

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 of demand to supply

ing, first developed by Carr and Kennedy[3]. The dramatic change of results from -O2 to -O3 is clear evidence that a compiler may significantly reduce the application's demand for memory bandwidth; nevertheless, the current compiler is not effective for all other programs. We will return to compiler issues in a moment and for the rest of this paper.

2.3 Memory Bandwidth Bottleneck

The precise ratios of the demand of data bandwidth to its supply can be calculated by dividing the program balances with the machine balance of Origin2000. The results are listed in Figure 2. They show the degree of mismatch for each application at each memory hierarchy level. The last column shows the largest gap: the programs require 3.4 to 10.5 times as much memory bandwidth as that provided by the machine, verifying that memory bandwidth is the most limited resource. The data bandwidth on the other two levels of memory hierarchy is also insufficient by factors between 1.3 to 6.0, but the problem is comparatively less serious.

The insufficient memory bandwidth compels applications into unavoidable low performance simply because data from memory cannot be delivered fast enough to keep CPU busy. For example, the Linpack kernel *dmxpy* has a ratio of 10.5, which means an average CPU utilization of no more than 1/10.5, or 9.5%. One may argue that a kernel does not contain enough computation. However, the last two rows show a grim picture even for large applications: the average CPU utilization can be no more than 16% for *NAS/SP* and 10% for *Sweep3D*. In other words, over 80% of CPU capacity is left unused because of the memory bandwidth bottleneck.

The memory bandwidth bottleneck exists on other machines as well. To fully utilize a processor of comparable speed as MIPS R10K on Origin2000, a machine would need 3.4 to 10.5 times of the 300 MB/s memory bandwidth of Origin2000. Therefore, a machine must have 1.02 GB/s to 3.15GB/s of memory bandwidth, far exceeding the capacity of current machines such as those from HP and Intel. As CPU speed rapidly increases, future systems will have even worse balance and a more serious bottleneck because of the lack of memory bandwidth.

2.4 Comparing bandwidth with latency

So far, the balance-based performance model has not considered the effect of the latency constraint and, in particular, the effect of memory latency. It is possible that memory access incurs such a high latency 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 following experiment measures the *effective bandwidth*, which is the total memory transfer (both reads and writebacks) divided by its execution time. The experiment uses 13 simple data-traversal loop kernels, which access a different number of arrays in a unit 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.

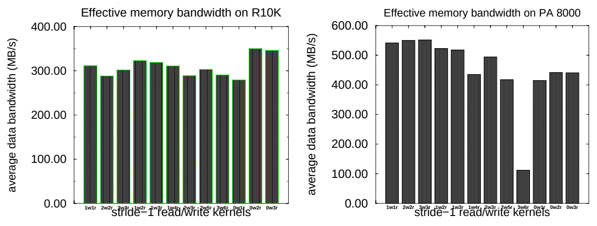

Figure 3. Effective bandwidth of kernels

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*[3]. 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

[3]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

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 a large amount of computation. Since they together resembles many programs with regular computations, their effective bandwidth suggest that regular applications saturate memory bandwidth most of the times. In these cases, memory bandwidth is a more limiting factor to performance than is memory latency.

In conclusion, the empirical study has shown that for most applications, machine memory bandwidth is between one third and one tenth of that needed. As a result, over 80% of CPU power is left un-utilized by large applications, indicating a significant performance potential that may be realized if the applications can better utilize the limited memory bandwidth. The next section introduces new compiler techniques that are aimed at exactly this goal, that is, reducing the memory bandwidth demand of applications.

3 Bandwidth reduction by a compiler

This section presents a compiler strategy for reducing the bandwidth consumption of a program. The first technique is a new formulation of loop fusion, which minimizes the overall data transfer among fused loops. One effect of global loop fusion is the localized live range of arrays. The next two techniques exploit this opportunity and further reduce the bandwidth consumption after loop fusion, including the unique opportunity of eliminating memory write-backs.

3.1 Bandwidth-minimal loop fusion

This section first formulates the problem of loop fusion for minimal memory 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 why the previous fusion model given by Gao et al.[5] and by Kennedy and McKinley[7] does not minimize bandwidth consumption.

3.1.1 Formulation

Given a sequence of loops accessing a set of data arrays, Gao et al.[5] and by Kennedy and McKinley[7] modeled both the computation and the data in a *fusion graph*. A fusion graph consists of nodes—each loop is a node— and two types of edges—directed edges for modeling data dependences and undirected edges for fusion-preventing constraints. Our formulation uses the same definition of a fusion graph. However, the objective of fusion is a different one, as stated below.

Problem 3.1 Bandwidth-minimal 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 flow only 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 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 reads 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 be freely fused with any other four loops. Loop 6 depends on loop 5. Without fusion, the total number of arrays accessed in the six loops is 20. The optimal fusion leaves loop 5 alone and fuses all other loops. The number of distinct arrays is 1 in the first partition and 6 in the second, thus the total memory transfer is reduced from 20 arrays to 7.

The optimality of bandwidth-minimal fusion is different from previous work on loop fusion of Gao et al.[5] and Kennedy and McKinley[7]. They 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 cross-partition edges is minimal. We call this formulation *edge-weighted fusion*.

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

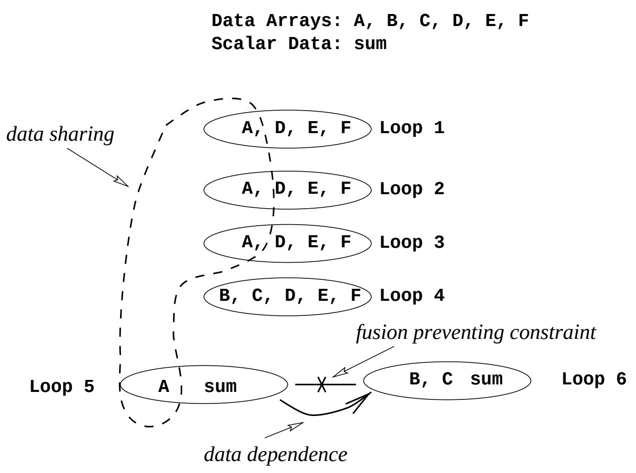

Figure 4. Example loop fusion

To show that edge-weighted fusion does not minimize bandwidth consumption, it suffices to give a counter example, which is the real purpose of Figure 4. The optimal edge-weighted fusion is to fuse the first five loops and leave loop 6 alone. The total weight of cross-partition edges is 2, which lies between loop 4 and 6. However, this fusion has to load 8 arrays (6 in the first partition and 2 in the second), while the previous bandwidth-minimal fusion needs only 7. Reversely, the total weight of inter-partition edges in the bandwidth-minimal fusion is 3, clearly not optimal based on the edge-weighted formulation. Therefore, the previous formulation by Gao et al. and Kennedy and McKinley does not minimize overall memory transfer.

To understand the effect of data sharing and the complexity of bandwidth-minimal fusion, the remaining part of this section studies a model based on a different type of graphs, hyper-graphs.

3.1.2 Solution Based On Hyper-graphs

The traditional definition of an edge is inadequate for modeling data sharing because the same data can be shared by more than two loops. Instead, the following formulation uses *hyper-graphs* because 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 of loop fusion can now be restated as follows.

Problem 3.2 Bandwidth-minimal fusion problem (II)*: Given a fusion graph as constructed by Problem 3.1, add a hyper-edge for each array in the program, which connects all loops that access the array. How can we divide all nodes into a sequence of partitions such that*

- *(Correctness) criteria are the same as Problem 3.1, but*
- *(Optimality) for each hyper-edge, let the length be the number of partitions the edge connects to after partitioning, then the goal is to minimize the total length of all hyper-edges.*

The next part 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 among non-terminal nodes. The result of fusion will produce two partitions where any non-terminal node can appear in any partition. 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* between two nodes is a sequence of hyper-edges where the first edge connects one node, the last edge connects the other node, and consecutive ones connect intersecting groups 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 size of the data plus the size of the cut (which is the 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 the 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 one of 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 algo-

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

Output A set of edges C (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 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 G'*/
construct a normal graph G'=(V',E')
   let array map[] maps from V' to E
   for each e in E, add a node v to V'
      let map[v] = e
   add edge (v1, v2) in G'
      iff map[v1] and map[v2] overlap in G

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

/* Step 2: minimal vertex cut in G' */
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 between s' and t'
convert the minimal edge cut into the
   vertex cut in G'

/* Step 3: construct the cut set in G*/
let C be the vertex cut set found in step 2
delete all edges of G corresponding to
   nodes in C
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 for hyper-graphs

rithm 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 method has assumed the absence of dependence edges. The dependence relation can be enforced by adding hyper-edges to the fusion graph. Given a fusion graph with N edges and 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.

3.1.3 The Complexity of General Loop Fusion

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

To prove, observe that the fusion problem is in NP because loops or nodes of a fusion graph can be partitioned in a non-deterministic way, and the legality and optimality can be checked in polynomial time.

The fusion problem is also NP-hard. To prove this, we reduce k-way cut problem to the fusion problem. Given a graph $G = (V, E)$ and k nodes to be designated as terminals, k-way cut is to find a set of edges of minimal total weight such that removing the edges renders all k terminals disconnected from each other. 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 new hyper-edge connecting the two end nodes of the edge. It is easy to see that a minimal k-way cut in G is an optimal fusion in G' and vice versa. Since k-way cut is NP-complete, bandwidth-minimal fusion is NP-hard when the number of partitions is greater than two. Therefore, it is NP-complete.

Aggressive fusion enables other optimizations. For example, the use of an array can become enclosed within one or a few loops. The localized use allows aggressive storage transformations that are not possible otherwise. The rest of this section describes two such storage optimizations: *storage reduction*, which replaces a large array with a small section or a scalar; and *store elimination*, which avoids writing back new values to an array. Both save a significant amount more memory bandwidth than loop fusion.

3.2 Storage Reduction

After loop fusion, if the live range of an array is shortened to stay within a single loop nest, the array can be replaced by a smaller data section or even a scalar. In particular, two opportunities exist for storage reduction. The first case is where the live range of a data element (all uses of the data element) is short, for example, within one loop iteration. The second case is where the live range spans the whole loop, but only a small section of data elements have such a live range. The first case can be optimized by *array shrinking*, where a small temporary buffer is used to carry live ranges. The second case can be optimized by *array peeling*, where only a reduced section of an array is saved in a dedicated storage. Figure 6 illustrates both transformations.

The example program in Figure 6(a) uses two large arrays $a[N, N]$ and $b[N, N]$. Loop fusion transforms the program into Figure 6(b). Not only does the fused loop contain all accesses to both arrays, the definitions and uses of many array elements are very close in computation. The live range of a b-array element is within one iteration of the inner loop. Therefore, the whole b array can be replaced by a scalar $b1$. The live range of an a-array element is longer, but it is still within every two consecutive j iterations. Therefore, array $a[N, N]$ can be reduced into a smaller buffer $a3[N]$, which carries values from one j iteration to the next. A section of $a[N, N]$ array has a live range spanning the whole loop because $a[1 \ldots N, 1]$ is defined at the beginning and used at the end. These elements can be peeled off into a smaller array $a1[N]$ and saved throughout the loop. After array shrinking and peeling, the original two arrays of size N^2 have been replaced by two arrays of size N plus two scalars, achieving a dramatic reduction in storage space.

Storage reduction directly reduces the bandwidth consumption between all levels of memory hierarchy. First, the optimized program occupies a smaller amount of memory, resulting in less memory-CPU transfer. Second, it has a smaller footprint in cache, increasing the chance of cache reuse. When an array can be reduced to a scalar, all its uses can be completed in a register, eliminating cache-register transfers as well.

3.3 Store Elimination

While storage reduction optimizes only localized arrays, the second transformation, *store elimination*, improves bandwidth utilization of arrays whose live range spans multiple loop nests. The transformation first locates the loop containing the last segment of the live range and then finishes all uses of the array so that the program no longer needs to write new values back to the array.

The program in Figure 7 illustrates this transformation.

```
For i=1, N
 res[i] = res[i]+data[i]
End for

sum = 0.0
For i=1, N
 sum += res[i]
End for

print sum
```

(a) Original program

```
sum = 0.0

For i=1, N
 sum += res[i]+data[i]
End for

print sum
```

(b) After loop fusion and store elimination

Figure 7. Store elimination

The first loop in Figure 7(a) assigns new values to the *res* array, which is used in the next loop. After the two loops are fused in (b), the writeback of the updated *res* array can be eliminated because all uses of *res* are already completed in the fused loop. The program after store elimination is shown in Figure 7(c).

The goal of store elimination differs from all previous cache optimizations because it changes only the behavior of data writebacks and it does not affect the performance of memory reads at all. Store elimination has no benefit if memory latency is the main performance constraint. However, if the bottleneck is memory bandwidth, store elimination becomes extremely useful because reducing memory writebacks is as important as reducing memory reads. The following experiment verifies the benefit of store elimination on two of today's fastest machines: HP/Convex Exemplar and SGI Origin2000 (with R10K processors).

The table in Figure 8 lists the reduction in execution time 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 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
Origin2000	0.32 sec	0.22 sec	0.16 sec
Exemplar	0.24 sec	0.21 sec	0.14 sec

Figure 8. Effect of store elimination

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.

```
// Initialization of data
For j=1, N
 For i=1, N
  read(a[i,j])
 End for
End for

// Computation
For j=2, N
 For i=1, N
   b[i,j] = f(a[i,j-1],
              a[i,j])
 End for
End for

For i=1, N
 b[i,N] = g(b[i,N],
            a[i,1])
End for

// Check results
sum = 0.0
For j=2, N
 For i=1, N
  sum += a[i,j]+b[i,j]
 End for
End for

print sum
```

(a) Original program

```
sum = 0.0

For i=1, N
  read(a[i,1])
End for

For j=2, N
 For i=1, N
    read(a[i,j])

    b[i,j] = f(a[i,j-1],
                 a[i,j])

    if (j<=N-1)
      sum += a[i,j]+b[i,j]
    else
     b[i,N] = g(b[i,N],
                  a[i,1])
     sum += b[i,N]+a[i,N]
    end if
  End for
End for

print sum
```

(b) After loop fusion

```
sum = 0.0

For i=1, N
  read(a1[i])
End for

For j=2, N
 For i=1, N
   read(a2)
   if (j=2)
     b1 = f(a1[i],a2)
    else
     b1 = f(a3[i],a2)
    end if

    if (j<=N-1)
      sum += b1+a2
      a3[i] = a2
    else
      b1 = g(b1,a1[i])
      sum += b1+a2
    end if
  End for
 End for

 print sum
```

(c) After array shrinking and peeling

Figure 6. Array shrinking and peeling

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.

Single-loop transformations such as loop blocking 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 disjoint loops. Gao et al.[5] and Kennedy and McKinley[7] 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 the bisection.

Sarkar and Gao proposed a storage reduction technique called array contraction, which replaces an array with a scalar[10]. Array peeling and shrinking, presented in this paper, is more general and powerful because they can reduce the size of arrays that cannot be substituted with a scalar or that can only be partially substituted by 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. We are not aware of any previous technique with the explicit goal of store elimination.

Loop fusion and writeback reduction are components of the compiler strategy developed in Ding's dissertation[4]. The strategy first improves global temporal reuse through loop fusion, then maximizes global spatial reuse through inter-array data regrouping, finally performs storage reduction and store elimination to further reduce the bandwidth consumption of the whole program. For dynamic applications, the strategy applies computation fusion and data grouping at run time by locality grouping and data packing. Finally, the compiler strategy supports user tuning and machine scheduling with bandwidth-based performance tuning and prediction.

Many architectural studies examined the memory bandwidth constraint. McCalpin [8] 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] measured the bandwidth constraint by simulating SPEC programs on a machine both with and without memory bandwidth constraint. Their measurement is precise but relies on machine simulation, which is experimental rather than analytical. They also used a concept called cache traffic ratio, which is similar to balance except that cache traffic ratios do not include CPU speed and load/store bandwidth as balance does. Therefore, cache traffic ratios alone cannot quantify the bound on CPU utilization and especially the performance bound due to limited register bandwidth. Our measurement on balance shows that register bandwidth is the second most critical resource after memory bandwidth for the applications tested. Burger et al. exploited the potential of better cache management by using the optimal Belady cache-replacement policy. However, the solution is not practical because it requires hardware to have beforehand the perfect knowledge of whole execution. Another related study is performed by Huang and Shen[6], who defined and measured what they called intrinsic bandwidth requirement due to the reuse of values. The intrinsic bandwidth indicates the lower bound on memory traffic. Like Burger et al. their measurement relies on program simulation. In comparison, program and machine balance are more suitable for practical use because they can be measured accurately and efficiently, and they include all bandwidth constraints (along with CPU throughput) of a system. The most important limitation shared by all architectural studies is that they assumed a fixed order of computation. None of them considered the potential of bandwidth reduction transformations. As explained in this paper, aggressive program optimizations can significantly change the balance or reduce the intrinsic bandwidth of a program. In fact, the previous studies considered only existing optimizations that were implemented in the compiler they used. For example, Burger et al.[1] relied on hardware data prefetching while most current machines such as Origin2000 uses software data prefetching.

5 Contributions

In this paper, we have presented a bandwidth-based performance model called *balance*, which measures the demand and supply of data bandwidth on all levels of memory hierarchy. Through this model, we have shown the serious performance bottleneck due to the limited memory bandwidth. To reduce the overall bandwidth consumption of a program, we have described three new compiler transformations. Bandwidth-minimal fusion used hyper-graphs to model data sharing among fused loops. It is the first formulation of loop fusion that minimizes the overall memory transfer of a program. We gave an efficient algorithm for two-partitioning cases and proved that the general fusion is NP-complete. After loop fusion, we proposed two additional transformations: array shrinking and peeling reduces a large array into a scalar or several small sections, and store elimination removes memory writebacks to the remaining arrays.

Acknowlegement

Nathaniel Dean suggested us to model bandwidth-minimal 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, Aug. 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, Dec. 1989.

[4] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Rice University, 2000.

[5] 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, Aug. 1992.

[6] S. A. Huang and J. P. 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.

[7] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).

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

[9] P. J. Mucci and K. London. The cachebench report. Technical Report ut-cs-98-394, University of Tennessee, 1998.

[10] 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.