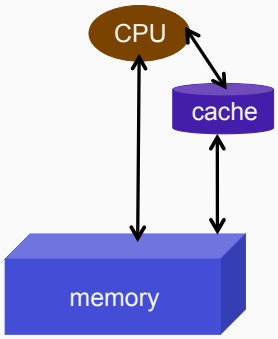
Compiler Techniques for Improving Global and Dynamic Cache Reuse

Memory Performance



- Problem
  - high memory latency
- Improvement 1
  - fast cache
- Improvement 2
  - Data prefetching

Is there enough bandwidth?

Bandwidth Bottleneck

- Hardware trends
  - CPU speed improved 6400 times in 20 years
  - Memory bandwidth improved 139 times
- Software trends
  - large data sets
  - dynamic content and computation
  - modularized programming

Performance Model

- Balance
  - Callahan, Cocke, and Kennedy. JPDC 1988.
  - Ding and Kennedy. JPDC 2004.
- Machine balance
  - max words per cycle divided by max flops per cycle
  - # load/store units divided by # floating-point units
- Program balance
  - # words accessed divided by # flops executed
  - total loads/stores divided by total floating-point ops
- Consequences
  - MB = PB → full utilization
  - MB > PB → memory idle
  - MB < PB → CPU idle

Program and Machine Balance

[Callahan, Cocke, and Kennedy, JPDC 1988]
[Ding and Kennedy, IPDPS 2000 and JPDC 2004]

program/ machine	Program/machine balance		
	L1-Reg	L2-L1	Mem-L2
Convolution	6.4	5.1	5.2
Dmxpy	8.3	8.3	8.4
Mmjki (-O2)	24.0	8.2	5.9
Mmjki (-O3)	8.1	1.0	0.04
FFT	8.3	3.0	2.7
SP	10.8	6.4	4.9
Sweep3D	15.0	9.1	7.8
Origin2000	4.0	4.0	0.8

Memory-Bandwidth Bottleneck

- Ratios of demand to supply

Applications	Ratio: demand/supply		
	Reg BW	Cache BW	Mem BW
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
SP	2.7	1.6	6.1
Sweep3D	3.8	2.3	9.8

- Memory bandwidth is least sufficient
- Maximal CPU utilization: 10% to 33%
- **The imbalance is getting worse**
- **Software solution: Better caching**

Vector Architectures: Past, Present and Future

Roger Espasa, Mateo Valero* James E. Smith†

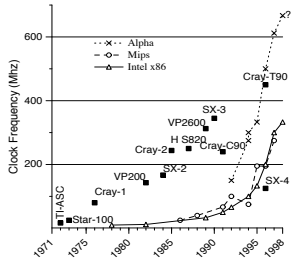


Figure 2: Evolution of clock frequency over the years for vector supercomputers and for several microprocessors (source: vendor information).

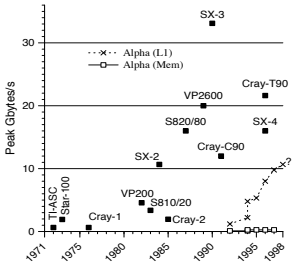
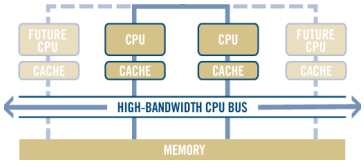


Figure 4: Evolution of peak main memory bandwidth for vector supercomputers and for an Alpha-based Workstation. The values for the Alpha main memory bandwidths are derived using the 33Mhz 21072 and 21172 chipsets [9].

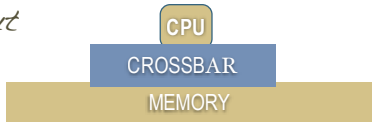
Steve Wallach's Fall 1999 Seminar

Instead of

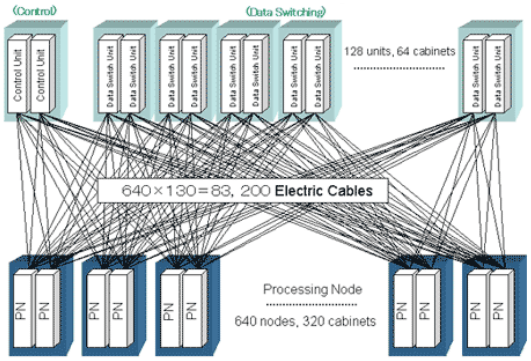


http://gdn.qnx.com/images/articles/

How about



Earth Simulator 2002



http://www.es.jamstec.go.jp/esc/jp/hardware/image/IN_02_b.gif

IBM BlueGene/L 2005

Single-Chip Bottleneck

- Intel Core 2 Quad
  - four 2.40GHz cores
  - 8 64-bit flops per cycle per core
  - sustained memory bandwidth
    measured by STREAM benchmark
    - 5.3GB/s for four threads
    - 0.55 byte per cycle per core, 0.009 dword per flop
  - 8MB L2 cache
- Locality
  - How (in)frequent a program accesses main memory?
  - How much data does it actively use?
  - Must model long-range program behavior
- Program and machine balance
  - [Callahan, Cocke, Kennedy, JPDC 88] [Ding, Kennedy, JPDC 04]

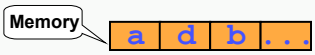
Problem of Caching

- An example
  - 7 accesses
  - single-element cache
- Cache management
  - LRU: 1 reuse
  - Belady: 2 reuses
- Our approach
  - fuse computation on the same data: 4 reuses
  - group data used by the same computation

adbaabd

adbaabd

aaddbb



Computation Fusion

```
Function Initialize:
 read input[1:N].data1
 read input[1:N].data2
End Initialize

Function Process:
 //Step_A
 A_tmp[1:N].data1
   ← input[1:N].data1

 //Step_B
 B_tmp[1:N].data1
   ← input[1:N].data2
 ...
End Process
```

```
//Fused_step_1
for each i in [1:N]
   read input[i].data1
   A_tmp[i].data1
      ← input[i].data1
end for

//Fused_step_2
for each i in [1:N]
   read input[i].data2
   B_tmp[i].data1
      ← input[i].data2
end for
  ...
```


Data Grouping

```
//Fused_step_1
for each i in [1:N]
   read input[i].data1
   A_tmp[i].data1
      ← input[i].data1
end for

//Fused_step_2
for each i in [1:N]
   read input[i].data2
   B_tmp[i].data1
      ← input[i].data2
end for
  ...
```

```
//Fused_step_1
for each i in [1:N]
   read group1[i].data1
   group1[i].data2
      ← group1[i].data1
end for

//Fused_step_2
for each i in [1:N]
   read group2[i].data1
   group2[i].data2
      ← group2[i].data1
end for
  ...
```


Original

```
Function Initialize:
 read input[1:N].data1
 read input[1:N].data2
End Initialize

Function Process:
 A_tmp[1:N].data1
   ← input[1:N].data1

 B_tmp[1:N].data1
   ← input[1:N].data2
 ...
End Process
```

Transformed

```
for each i in [1:N]
   read group1[i].data1
   group1[i].data2
      ← group1[i].data1
end for

for each i in [1:N]
   read group2[i].data1
   group2[i].data2
      ← group2[i].data1
end for
  ...
```

- Computation fusion recombines all functions
- Data grouping reshuffles all data

But How?

- Programmers
  - loss of modularity
  - data layout depends on function
- Hardware/operating system
  - limited scope
  - run-time overhead
- Compilers
  - global scope
  - off-line analysis/transformation
  - imprecise information

Overall Fusion Process

for each statement in program

  find its data sharing predecessor

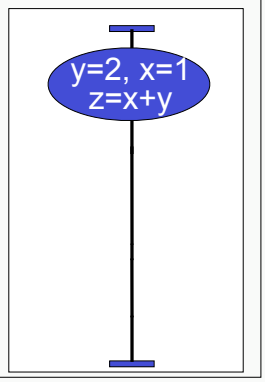
  try clustering them (fusion)

  if succeed

    apply fusion recursively

  end if

end for



Example Fusion

```
for i=2, N
  a[i] = f(a[i-1])
end for

a[1] = a[N]
a[2] = 0.0

for i=3, N
  b[i] = g(a[i-2])
end for
```

Difficulties

- incompatible shapes
- data dependence

Three cases of fusion

- between iteration & loop
  - embedding
- between iterations
  - interleaving + alignment
- otherwise
  - iteration reordering,
  - e.g. loop splitting

Example Fusion

```
for i=2, N
  a[i] = f(a[i-1])
end for

a[1] = a[N]
a[2] = 0.0

for i=3, N
  b[i] = g(a[i-2])
end for
```

```
for i=2, N
  a[i]=f(a[i-1])
  if (i==3)
    a[2]=0.0
  else if (i==N)
    a[1]= a[N]
  end if
end for

for i=3, N
  b[i] = g(a[i-2])
end for
```

- loop embedding

Example Fusion

```
for i=2, N
  a[i] = f(a[i-1])
end for

a[1] = a[N]
a[2] = 0.0

for i=3, N
  b[i] = g(a[i-2])
end for
```

```
for i=2, N
  a[i]=f(a[i-1])
  if (i==3)
    a[2]=0.0
  else if (i==N)
    a[1]= a[N]
  end if
end for

for i=4, N
  b[i] = g(a[i-2])
end for

b[3] = g(a[1])
```

- loop embedding, loop splitting,

Example Fusion

```
for i=2, N
  a[i] = f(a[i-1])
end for

a[1] = a[N]
a[2] = 0.0

for i=3, N
  b[i] = g(a[i-2])
end for
```

```
for i=2, N
  a[i]=f(a[i-1])
  if (i==3)
    a[2]=0.0
  else if (i==N)
    a[1]= a[N]
  end if

  if (i>2 && i<N)
    b[i+1] = g(a[i-1])
  end if
end for

b[3] = g(a[1])
```

- loop embedding, loop splitting, interleaving+alignment

More on Fusion

- Multi-level fusion
  - gives priority to fusion at outer levels
- Optimal fusion
  - minimal data sharing
  - an NP-hard problem
  - hyper-graph formulation of data sharing
  - a polynomial-time case

Is Fusion Enough?

```
//Fused_step_1
for each i in [1:N]
   read input[i].data1
   A_tmp[i].data1
      ← input[i].data1
end for

//Fused_step_2
for each i in [1:N]
   read input[i].data2
   B_tmp[i].data1
     ← input[i].data2
end for
  ...
```

```
//Fused_step_1
for each i in [1:N]
   read group1[i].data1
   group1[i].data2
      ← group1[i].data1
end for

//Fused_step_2
for each i in [1:N]
   read group2[i].data1
   group2[i].data2
     ← group2[i].data1
end for
  ...
```

Data Regrouping

Program Analysis

- Computation phases
  - amount of data accessed
- Compatible arrays
  - array size
  - order of access
- Arrays are split into the smallest unit possible
  - A(3,N) -> A1(N), A2(N), A3(N)
- Regrouping is applied to individual set of compatible arrays

Example

Computation phases	Arrays accessed
Constructing neighbor list	position
Smoothing attributes	position, velocity, heat, derivative, viscosity
Hydrodynamics 1	density, momentum
Hydrodynamics 2	momentum, volume, energy, cumulative totals
Stress 1	volume, energy, strength, cumulative totals
Stress 2	density, strength

Regrouping Algorithm

- Requirements
  - 1. regroup as many arrays as possible
  - 2. do not introduce useless data
- Solution
  - group two arrays if and only if they are always accessed together
- Properties
  - time complexity is $O(N_{arrays} * N_{phases})$
  - compile-time optimal
  - minimal page-table working set
  - eliminate all useless data in cache

Example

Computation phases	Arrays accessed
Constructing neighbor list	position
Smoothing attributes	position, velocity, heat, derivative, viscosity
Hydrodynamics 1	density, momentum
Hydrodynamics 2	momentum, volume, energy, cumulative totals
Stress 1	volume, energy, strength, cumulative totals
Stress 2	density, strength

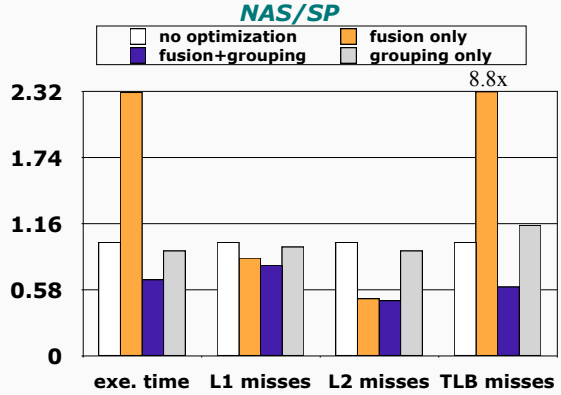
More on Regrouping

- Multi-level regrouping
  - grouping array segments as well as array elements
  - consistent regrouping
- Extensions
  - allowing useless data
  - allowing dynamic remapping
  - NP-complete proofs
  - architecture-dependent data layout

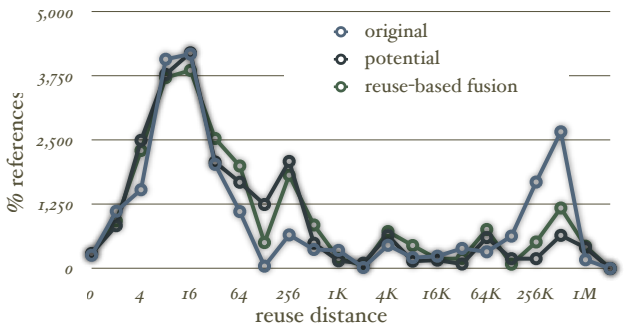
NAS/SP

- Benchmark application from NASA
  - computational fluid dynamics (CFD)
  - class B input, 102x102x102
  - 218 loops in 67 loop nests, distributed into 482 loops
  - 15 global arrays, split into 42 arrays
- Optimizations
  - fused into 8 loop nests
  - grouped into 17 new arrays, e.g.
    - {ainv[n,n,n], us[n,n,n], qs[n,n,n], u[n,n,n,1-5]}
    - {lhs[n,n,n,6-8], lhs[n,n,n,11-13]}

NAS/SP



Limit of Computation Fusion



- Ding and Kennedy, IPDPS 2001 (best paper), JPDC 2004

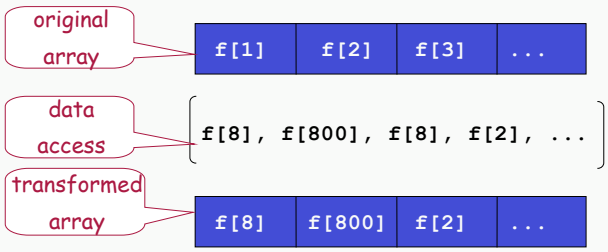
Dynamic Optimizations

Unknown Access

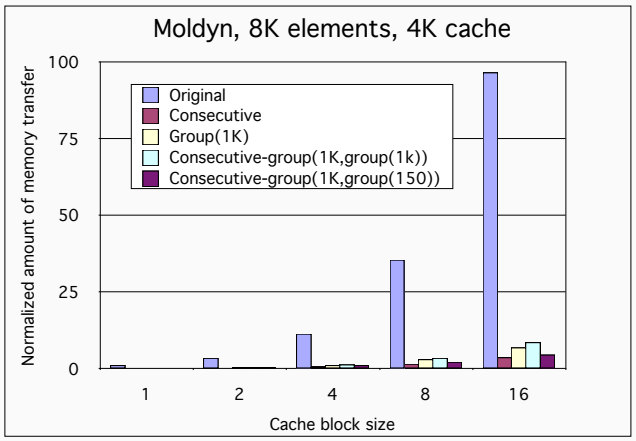
"Every problem can be solved by adding one more level of indirection."

- Irregular and dynamic applications
  - Irregular data structures are unknown until run time
  - Data and their uses may change during the computation
- For example
  - Molecular dynamics
  - Sparse matrix
- Problems
  - How to optimize at run time?
  - How to automate?

Example packing



Software remapping:
f[t[i]] → f[remap[t[i]]] → f[t'[i]]
f[i] → f[remap[i]] → f[i]



Dynamic Optimizations

- Locality grouping & Dynamic packing
  - run-time versions of computation fusion & data grouping
  - linear time and space cost
- Compiler support
  - analyze data indirections
  - find all optimization candidates
  - use run-time maps to guarantee correctness
  - remove unnecessary remappings
    - pointer update
    - array alignment
- The first set of compiler-generated run-time transformations


```
packing Directive: apply packing using interactions

for each pair (i,j) in interactions
   compute_force( force[i], force[j] )
end for

for each object i
   update_location( location[i], force[i] )
end for
```



```
apply_packing(interactions[*],force[*],inter_map[*])
for each pair (i,j) in interactions
   compute_force( force[inter_map[i]],
                  force[inter_map[j]] )
end for

for each object i
   update_location(location[i],force[inter_map[i]])
end for
```

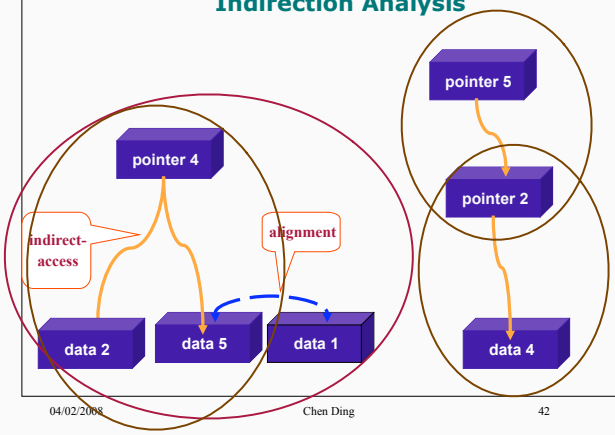
indirections


```
apply_packing(interactions[*],force[*],
              inter_map[*], update_map[*])
update_indirection_array(interactions[*],
                         update_map[*])
transform_data_array(location[*],update_map[*])

for each pair (i,j) in interactions
   compute_force( force[i], force[j] )
end for

for each object i
   update_location( location[i], force[i] )
end for
```

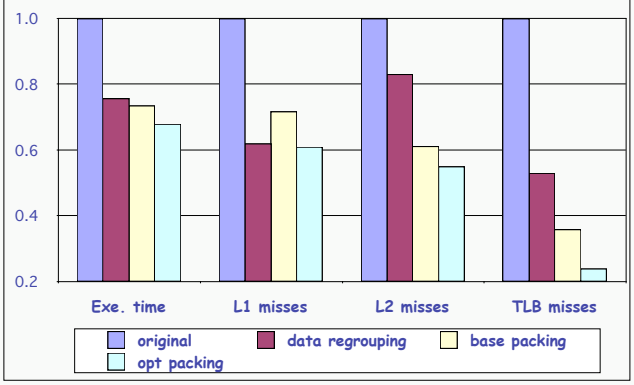

Indirection Analysis



DoD/Magi

- A real application from DoD Philips Lab
  - particle hydrodynamics
  - almost 10,000 lines of code
  - user supplied input of 28K particles
  - 22 arrays in major phases, split into 26
- Optimizations
  - grouped into 6 arrays
  - inserted 1114 indirections to guarantee correctness
  - optimization reorganized 19 more arrays
  - removed 379 indirections in loops
  - reorganized 45 arrays 4 times during execution

Magi



Overall Comparison

programs	L2 misses			TLB misses			Speedup
	NoOpt	SGI	New	NoOpt	SGI	New	over SGI
Swim	1.00	1.10	0.94	1.00	1.60	1.05	1.14
Tomcatv	1.00	0.49	0.39	1.00	0.010	0.010	1.17
ADI	1.00	0.94	0.53	1.00	0.011	0.005	2.33
NAS/SP	1.00	1.00	0.49	1.00	1.09	0.67	1.49
Average	1.00	0.88	0.59	1.00	0.68	0.43	1.52
Moldyn	1.00	0.99	0.19	1.00	0.77	0.10	3.02
Mesh	1.00	1.34	0.39	1.00	0.57	0.57	1.20
Magi	1.00	1.25	0.76	1.00	1.00	0.36	1.47
NAS/CG	1.00	0.95	0.15	1.00	0.97	0.03	4.36
Average	1.00	1.13	0.37	1.00	0.83	0.27	2.51

Software Techniques Summary

main steps	sub-steps	example techniques (* studied in my work)
temporal reuse	global (multi-loop)	*loop fusion
	local (single loop)	blocking, register allocation
	dynamic	*dynamic partitioning
spatial reuse	global (inter-array)	*inter-array data regrouping
	local (intra-array)	loop permutation, array reshaping, combined schemes
	dynamic	*dynamic data packing
	cache interference	padding
latency tolerance	local (single loop)	data prefetching, instruction scheduling
program tuning & scheduling	global (whole program)	*balance model *bandwidth-based perf. tuning & prediction

Summary

- Global transformations
  - Combining both computation and data reordering at a large scale
- Dynamic transformations
  - Combining compile-time and run-time analysis and transformation
- Compiling for locality
  - splits and regroups global computation and data
  - for the whole program and at all times