Optimization of Pointer-Intensive Programs

David F. Bacon

IBM T.J. Watson Research Center

Outline

- Research Goal and the Current Situation
- Pointers vs. Arrays
- Problems with Existing Techniques
- New Optimizations
- Performance Analysis
- Conclusions and Further Reading

Goal

- Develop high-performance optimizations for pointer-intensive programs
- Improve speed of existing programs
- Make more natural use of data structures possible

What is "Pointer-Intensive"

- Program that spends significant time manipulating pointers
- Written in standard languages (C, C++, Pascal, etc)
- Primary data structures:
  - list
  - graph
  - tree

Today's Situation

- Most optimizations are for scalars or arrays
- Superscalar parallelism is widening the performance gap for non-array code
- Increasing use of complex data structures

How Did We Get Here?

- Pointer optimizations considered too hard
- Limited machine parallelism reduced payoffs
- Array optimizations are much easier

Unique Property of Arrays

- Distinctness can be shown mathematically:

```
do i = 2, 10
     do j = 1, 9
          a[i,j] = a[i-1,j+1]
     end do
end do
```

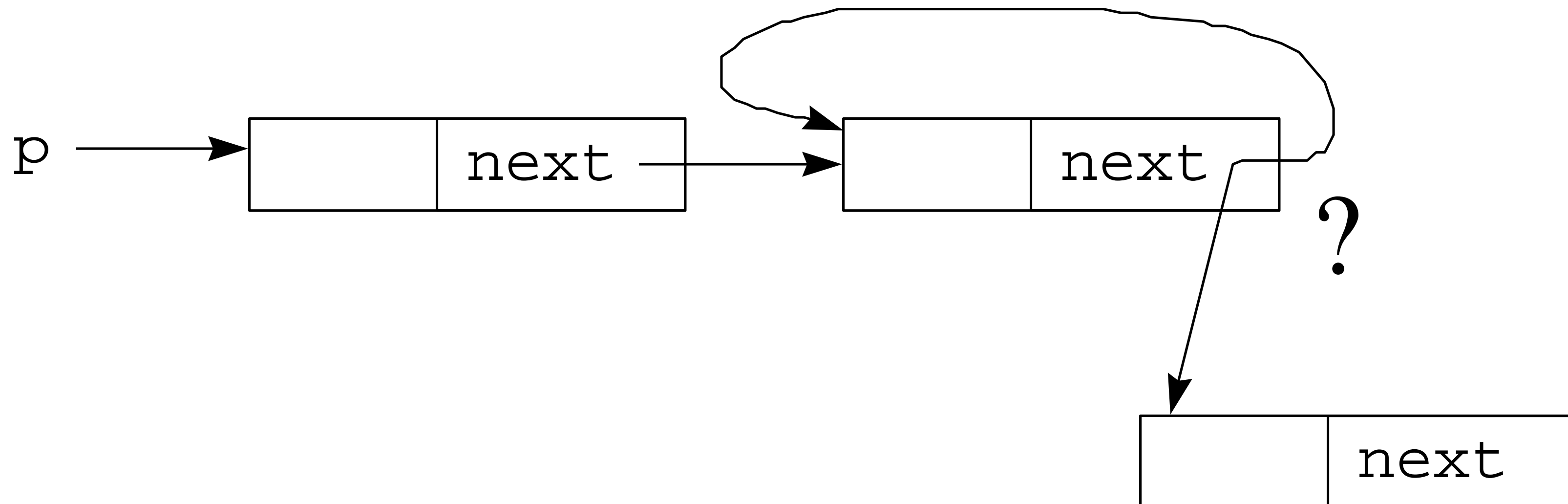
Distance vector = *(1,-1)*

Inner loop is parallelizable

Complexity of Pointers

- No pointer expressions guaranteed unique:

`p->next = p->next->next` *?*



Alias Analysis Insufficient

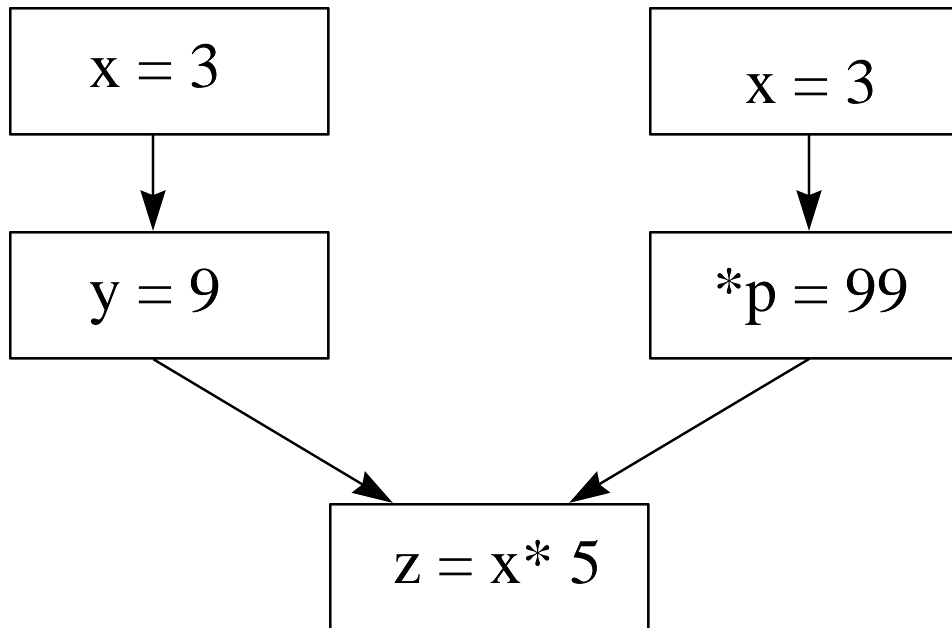
- Alias analysis answers:
  - is **p** aliased to **q**?
- We want to know:
  - is **p** in iteration **i** aliased to **p** in iteration **j** ?

Example

```
for (p=head; p != NULL; p = p->next)
        p->value = p->value + delta;
```

- Alias analysis always assumes the alias **<p,p>**
- But loop actually is parallelizable

Alias Analysis is Designed for Scalar Problems



Alias analysis tells us: does ***z = 15*** ?

My Approach

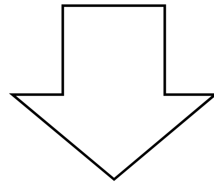
- Devise new transformations
- Assess potential for speedup
- Implement analysis or programmer support
  - static analysis
  - pragmas
  - tuned class libraries

New Transformations

- Pointer expansion
- Common Linearization Elimination
- Malloc strip-mining
- Multiple tail-recursion elimination

Pointer Expansion

```
for (p=head; p != NULL; p = p->next)
     p->value = p->value + delta;
```



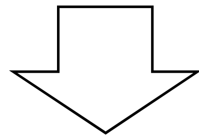
```
node *Temp[#];  int TX;
for (TX=0, Temp[TX]=head;
       Temp[TX]!=NULL;
       Temp[TX+1]=Temp[TX]->next, TX++)
    Temp[TX]->value += delta;
```

Common Linearization Elimination

- Re-use linearized pointers to nodes
- Requires no change to structure
- Eliminates fundamentally serial part of operations

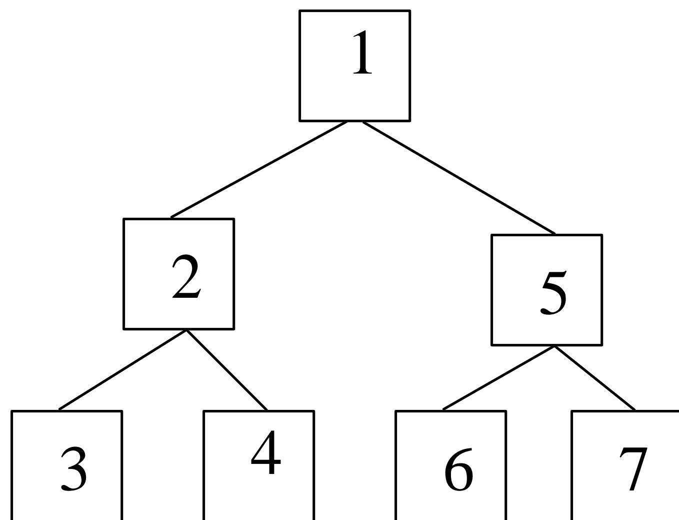
Malloc Strip-Mining

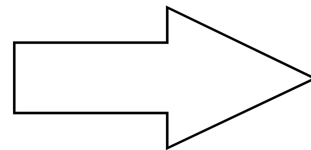
```
for (i = 0; i < n; i++)
    T[i] = malloc(SIZE);
```

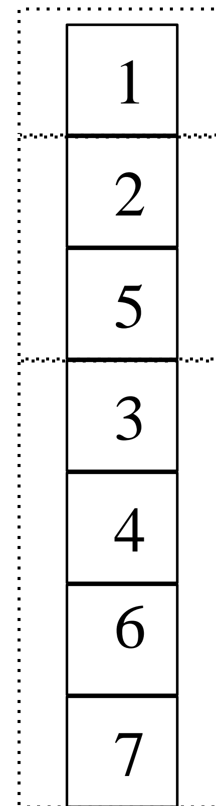


```
if (n > 0) {
    B = multimalloc(n, SIZE);
    for (i = 0; i < n; i++)
      T[i] = B+i;
  }
```

Multi-tail Recursion Elimination







Multi-tail Recursion Elimination

```
void treeplus(tree *t, float delta)
{
    if (! t)
      return;
    t->value += delta;
    treeplus(t->left, delta);
    treeplus(t->right, delta);
}
```

Multi-tail Recursion Elimination

```
void treeplus(tree *t, float delta)
{
    tree *Tt[#];  int Ti, Te;

    for (Tt[Ti=0] = t, Te=1;  Ti < Te; Ti++) {
      Tt[Ti]->value += delta;
      if (Tt[Ti]->left)
        Tt[Te++] = Tt[Ti]->left;
      if (Tt[Ti]->right)
        Tt[Te++] = Tt[Ti]->right;
    }
}
```

Performance: Linked-List Update

Recursive Singly-Linked List Update

Machine	Compiler	-g	-O4	Linearized	Prelinearized
RS/6000 530	xlc	6.92	1.07	2.27	0.96
RS/6000 590	xlc	1.72	0.35	0.64	0.29
Sparc-10	gcc	31.67	30.78	1.75	1.31
Sparc-10	cc	32.46	1.13	1.54	1.21
Cray C90	scc	4.15	1.47	0.63	0.0023

Performance: Tree Update

Recursive Tree Update

Machine	Compiler	-g	-O4	Linearized	Prelinearized	Unw
RS/6000 530	xlc	14.72	9.97	8.27	6.21	8.44
RS/6000 590	xlc	3.46	2.32	1.91	1.35	1.84
Sparc-10	gcc	9.24	9.18	5.83	4.67	5.33
Sparc-10	cc	9.69	6.03	4.95	3.84	4.68
Cray C90	scc	6.66	3.86	2.70	0.02	2.61

Performance: SPEC Benchmarks

- Of all SPECint92 benchmarks, only xlisp is pointer-intensive
- Two possibilities:
  - SPECint is not a representative sample
  - Pointer manipulations unimportant to performance

Pointer-Intensive Benchmarks

- A suite of 5 pointer-intensive C applications
- Performance improvements up to 50%
- But analysis is intractable in some cases

Available Parallelism Study

- Compare pointer-intensive/other programs
- Use simulated infinite VLIW machine
- Only honor data dependences
  - eliminate dependences due to induction variables

Conclusion

- Pointer-intensive code not well studied
- Optimizations exist; show promise
- Analysis is complex
  - class library approach may be more practical
- Performance benefits difficult to evaluate
- More study required

Further Reading

- Alias analysis: Choi et al, Landi & Ryder
- Pointer analysis: Hendren et al
- Array-based optimizations: Bacon et al
- Malloc optimizations: Calder & Grunwald