

Future Work

○ Analysis

- formalization
- interprocedural algorithm
- implement as standalone module

○ Benchmark Suite

- more applications
- study via patching, simulation, hardware monitors

○ Transformations

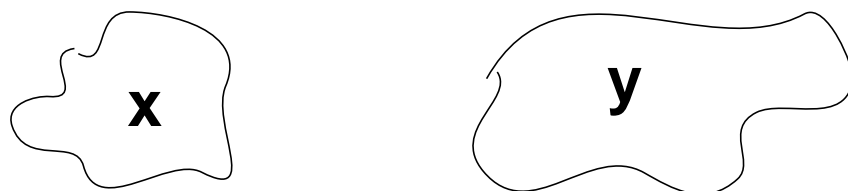
- implement in gcc or IBM xlc compiler
- measure speedups of benchmark suite

Summary: Results so Far

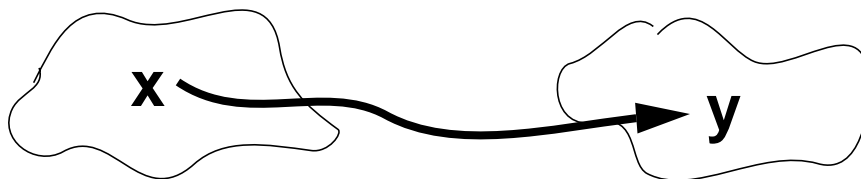
- pointer-intensive programs important
 - not sufficiently studied
- analysis provides structural information
 - works in far less restricted language
- benchmarks represent first quantitative work on complete pointer-intensive applications
- transformations identified
 - not difficult to implement
 - produce significant speedups on real programs

Connection Information

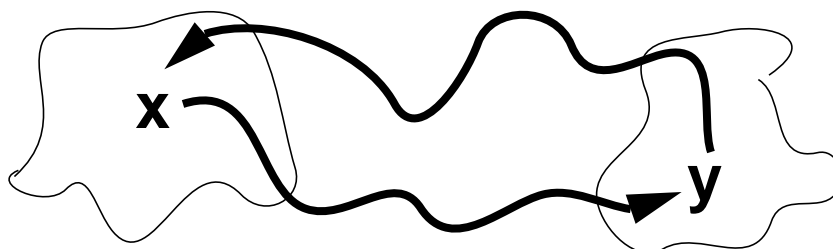
disjoint(x,y)



above(x,y)



distinct(x,y)



Procedure Call Transformations

Termination Hoisting

```
mod(tree *t, int v) {  
    if (t == NULL) return;  
    t->val = v;  
    mod(t->left);  
    mod(t->right);  
}
```

- 1/3 of sgpc is spent in such a routine

```
mod'(tree *t, int v) {  
    t->val = v;  
    if (t->left != NULL)  
        mod'(t->left);  
    if (t->right != NULL)  
        mod'(t->right);  
}
```

Transformations: Procedure Calls

- Convert linear recursion into two loops, detect exclusive control flows

```
queue *insertS1(queue *t, int time)
{
    if (t == NULL) {
        q = malloc(sizeof(queue));
        q->key = time;
        return queue;
    }
    if (time < t->key)
        t->left = insertS1(t->left, time);
    else
        t->right = insertS1(t->right, time);
    return t;
}
```

- 1/3 of swec time is spent in this routine

Sources of Procedure Overhead

- Branch Delay
- Register save/restore
- Stack frame allocation/deallocation

Transformations: Storage Allocation

- If malloc is called in a loop, it can be strip-mined

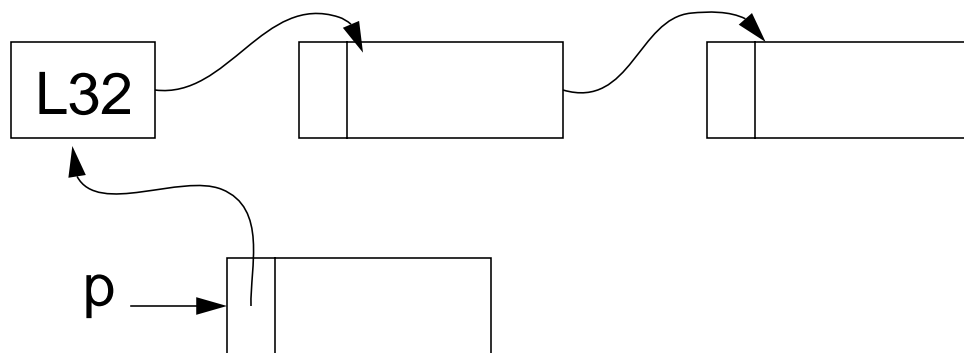
```
for (i = 0; i < n; i++)  
    a[i] = malloc(mysize);
```



```
foo *Tbase;
```

```
Tbase = multimalloc(n, mysize);  
for (i = 0; i < n; i++)  
    a[i] = Tbase+i;
```

Organization of Quickcells



Transformations: Storage Allocation

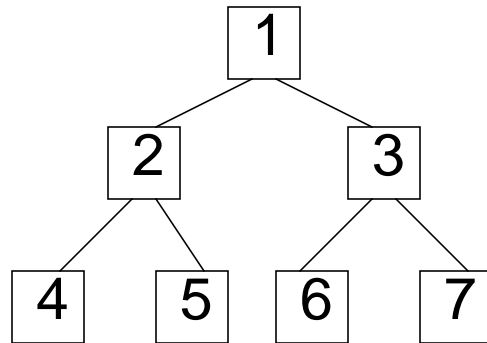
- Many malloc's use a compile-time fixed size
- Generate inline calls to fixed-size allocators
- 3-4 instructions/alloc, 4-6 instructions/free

Memory Allocation Behavior

Table 1: Percentage Time in Storage Allocation/Freeing (Sparc-2 cc-O4)

Suite	Program	malloc	custom
SPECint			
	espresso	9.9	
	gcc 1.x	0.3	6.2
	xlisp	0.0	2.1
	compress	0.0	
	eqntott	0.0	
SPECfp			
	ear	0.0	
	alvinn	0.0	
ptrBench			
	sgpc	35.2	
	swec	0.0	12.1
	voronoi	5.9	
	gcc 2.5	0.2	1.8
	pthor	1.4	

Parallel Tail Recursion Elimination



Parallel Tail-Recursion Elimination

```
mod(tree *t, int v)
{
    if (t == NULL)
        return;
    t->value += v;
    mod(t->left, v);
    mod(t->right, v);
}
```

- Recursive calls can be evaluated in parallel
- Since no ordering constraints exist, breadth-first traversal can be used to maximize parallelism

High-Level Transformations

- Use pointer analysis to remove dependence:

```
for (p = head; p != NULL; p = p->next)
    p->value += 10;
```

- Vectorize

```
node *p[N]; int i, j;
```

```
for (i = 0, p[0] = head;
     p[i] != NULL;
     p[i+1] = p[i]->next, i++);
```

```
for (j = 0; j < i; j++)
    p[j]->value += 10;
```

- Unroll and software pipeline

Opportunities for Optimization

- Memory Allocation
- Parallelism across pointer data structures
- Procedure Call Overhead

Program Optimization

Summary of Analysis

- Detects important structure properties
- Can eliminate false data dependences
- Other uses: reengineering and CASE tools
- Fundamental programming language technique

Tree Analysis Example (third pass)

```
node *tinsert(node *t, int v);
{
  < tree:left,right(t) >
  if (t == NULL) {
    item = malloc(sizeof(node));
    < null(t), malloc(item) >
    item->value = v;
    return item;
  }
  if (v < t->value)
    t->left = tinsert(t->left, v); block
    < tree:left,right(t) >
  else
    t->right = tinsert(t->right, v); block
    < tree:left,right(t) >
  return t;
} < tree:left,right(t) >

p = tinsert(p, v[i]);
< tree:left,right(p) >
```

Tree Analysis Example (second pass)

```
node *tinsert(node *t, int v);
{
  < malloc(t) >
  if (t == NULL) {
    item = malloc(sizeof(node));
    < null(t), malloc(item) >
    item->value = v;
    return item;
  }
  if (v < t->value)
    t->left = tinsert(t->left, v);
    < tree:left,right(t) >
  else
    t->right = tinsert(t->right, v);
    < tree:left,right(t) >
  return t;
} < tree:left,right(t) >

p = tinsert(p, v[i]);
< tree:left,right(p) >
```

Tree Analysis Example (first pass)

```
node *tinsert(node *t, int v);
{
  < null(t) >
  if (t == NULL) {
    item = malloc(sizeof(node));
    < malloc(t) >
    item->value = v;
    return item;
  }
  if (v < t->value)
    t->left = tinsert(t->left, v);

  else
    t->right = tinsert(t->right, v);

  return t;
} < malloc(t) >

p = tinsert(p, v[i]);
< malloc(p) >
```

Tree Analysis Example (cont)

```
node *tinsert(node *t, int v);
{

    if (t == NULL) {
        item = malloc(sizeof(node));

        item->value = v;
        return item;
    }
    if (v < t->value)
        t->left = tinsert(t->left, v);
    else
        t->right = tinsert(t->right, v);
    return t;
}
```

Tree Analysis Example

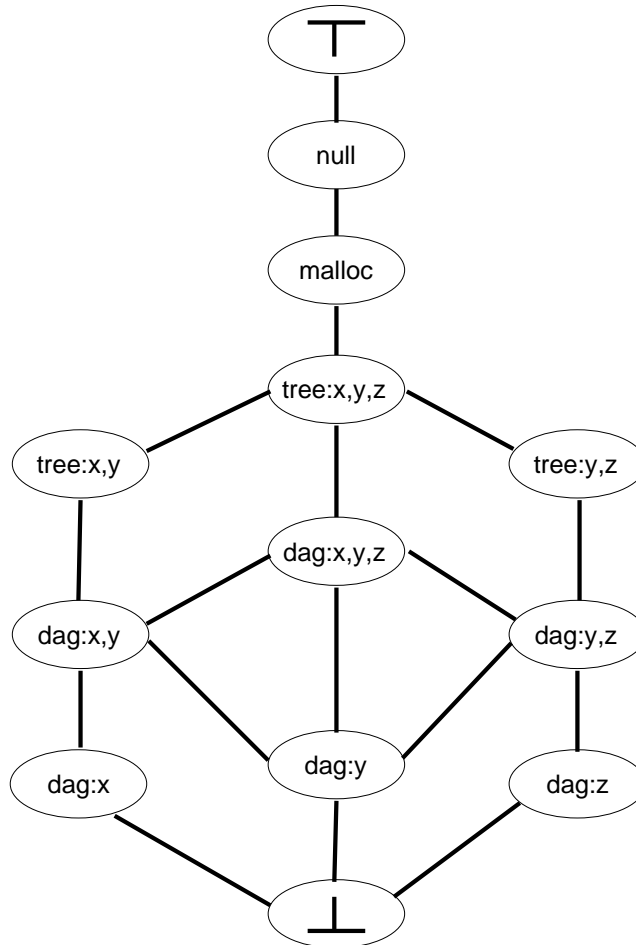
```
typedef struct node_s {
    struct node_s *left, *right;
    int value;
} node;

node *p;

p = NULL;

for (i = 0; i < N; i++)
    p = tinsert(p, v[i]);
```

Complex Structure Lattice



```
typedef struct node_s {  
    struct node_s *x, *y, *z;  
    int data;  
} node;
```

```
node *p;
```

Example (third pass)

```
< malloc(p), malloc(x) > meet < dag:next(p), dag:next(x) >
for (i = 0; i < n; i++) {
    < dag:next(p), dag:next(x) >
    x = malloc(sizeof(node));
    < dag:next(p), malloc(x), disjoint(p,x) >
    x->value = v[i];
    x->next = p;
    < dag:next(p), dag:next(x), above(x,p) >
    p = x;
    < dag:next(p), dag:next(x) >
}
```

Example (second pass)

```
< malloc(p), malloc(x) > meet < null(p) >
for (i = 0; i < n; i++) {
    < malloc(p), malloc(x) >
    x = malloc(sizeof(node));
    < malloc(p), malloc(x), disjoint(p,x) >
    x->value = v[i];
    x->next = p;
    < malloc(p), dag:next(x), above(x,p) >
    p = x;
    < dag:next(p), dag:next(x) >
}
```

Example (first pass)

```
p = NULL;
< null(p) >
for (i = 0; i < n; i++) {
  < null(p) >
  x = malloc(sizeof(node));
  < null(p), malloc(x), disjoint(p,x) >
  x->value = v[i];
  x->next = p;
  < null(p), malloc(x) >
  p = x;
  < malloc(p), malloc(x) >
}
```

Basic Analysis Example

```
node *p, *x;

p = NULL;

for (i = 0; i < n; i++) {

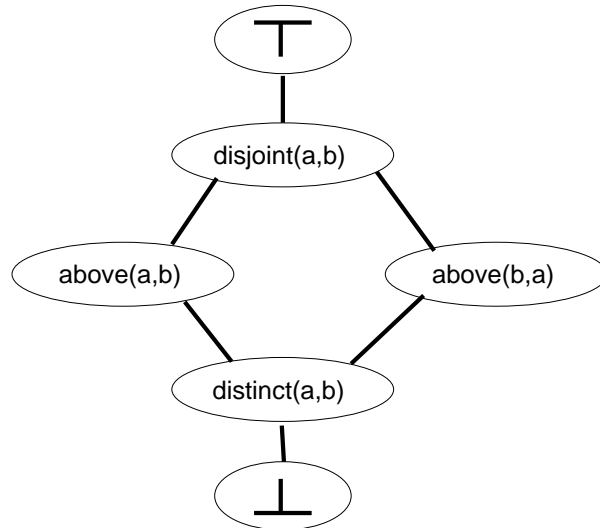
    x = malloc(sizeof(node));
    x->value = v[i];

    x->next = p;

    p = x;

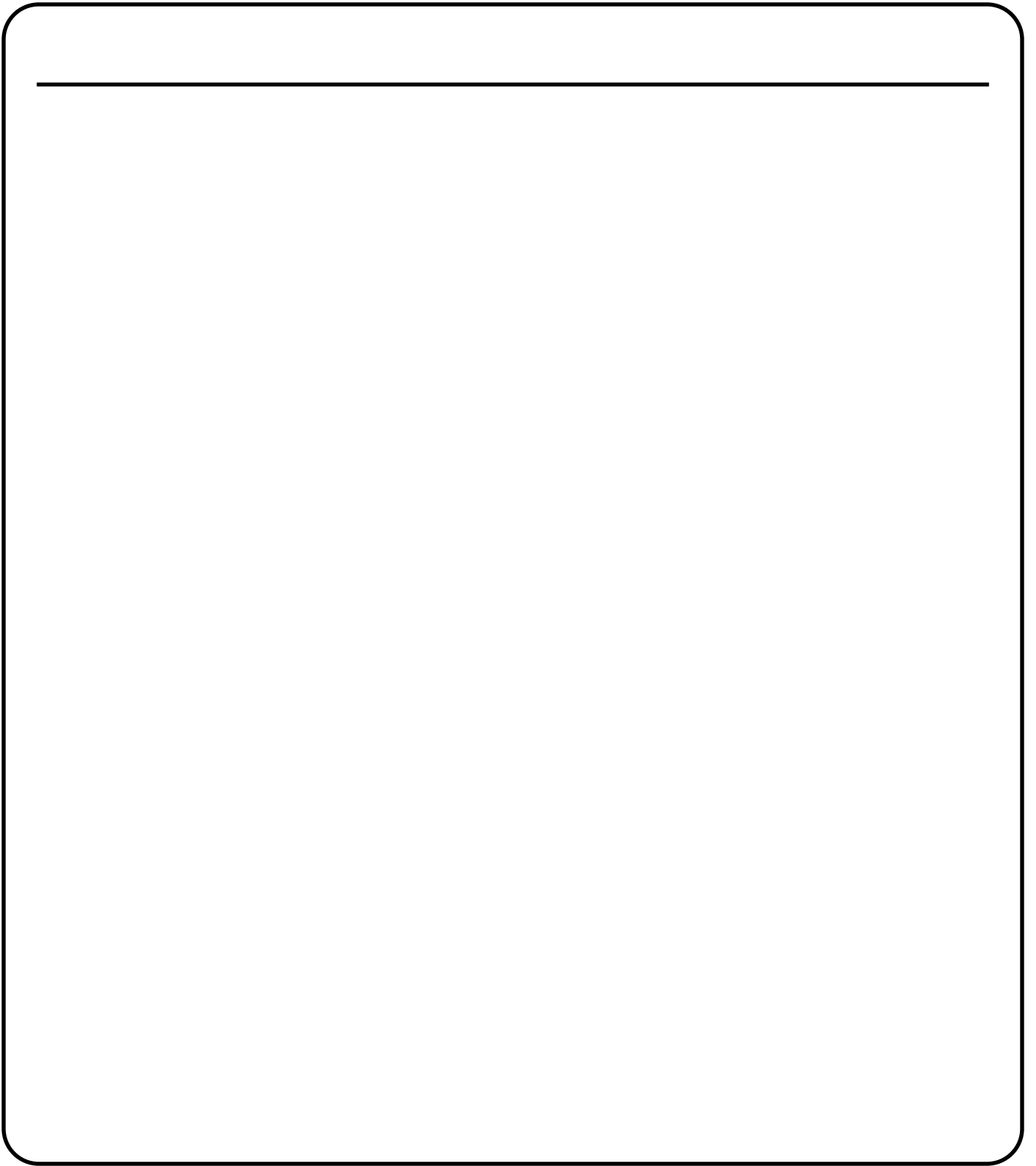
}
```

Connection Lattice

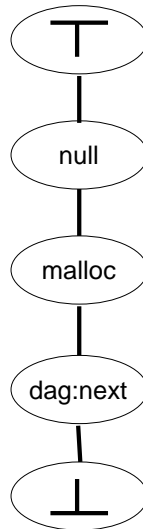


`node *a, *b;`

- Performs similar function to path matrix analysis



Structure Lattice



```
typedef struct s {  
    struct s *next;  
    int value;  
} node;
```

```
node *x;
```

Structure Creation: Basic Analysis

○ Sequential code: transfer functions

<x is a list, y is a list; x and y are disjoint>

x->next = y;

<x is a list, y is a list, a path exists from x to y>

○ Control flow: meet operator

```
if ( ... ) {
```

```
    ...
```

```
    <x is a list>
```

```
} else {
```

```
    ...
```

```
    <x is null>
```

```
}
```

```
<x is a list>
```

Analysis of Structure Use

```
for (p = head; p != NULL; p = p->next)
    p->value += 10;
```

- Loop contains no modification of `next` fields
- Loop performs no input/output
- Single exit

Data Structure Analysis

- Use static analysis to find
 - acyclic lists
 - acyclic graphs
 - trees
- Analysis can make use of information from
 - creation of structures
 - use of structures

Program Analysis

Benchmark Suite

- SWEC (Circuit Timing Simulator)
- PTHOR (Logic Level Circuit Simulator)
- GCC 2.5 (C Compiler)
- Voronoi (Computational Geometry)
- SGPC (Genetic Programming)
- xlisp (Lisp Interpreter)

A Pointer Benchmark Suite

- Study/compare characteristics quantitatively
 - Instructions per Cycle
 - Cache Behavior (miss rates, spatial locality)
 - Procedure Call Overhead (Recursion)
 - Storage Management
- Tools: program patching, Power2 perfmon
- Compare to SPEC and Malloc benchmarks

Research Plan

- Study characteristics of pointer-intensive code
 - assemble a benchmark suite
- Develop or adapt appropriate transformations
- Develop program analysis techniques
 - handle loops and recursion
- Implement
 - analysis as stand-alone module
 - low-level transformations (gcc or IBM xlc)
- Evaluate speedups using benchmark suite

Research Plan

Evaluation Frameworks: Benchmarks

- SPEC
 - 7 C programs
 - only 1 (xlisp) is pointer-intensive
 - gcc 1.x has no optimizer
 - espresso's pointer usage is not CPU intensive
- Malloc evaluation suite [Grunwald,Zorn]
- Compiler vs. Hardware benchmarks
 - SPEC is “pre-optimized”

Hendren et al

○ Path Matrix Analysis

- Determines path between variables

a->left = b->right->left->left

- Requires restriction on data shapes

○ Abstract Description of Data Structures

- Properties are specified by user

```
type tree [Down] {
  int data;
  tree *left, *right is uniquely
  forward along Down;
  tree *parent is backward along Down
};
```

Parallelizing Lisp

○ Curare [Larus 88]

- restructures Scheme programs to expose fork/join parallelism on a multiprocessor
- source of parallelism is spawning recursive calls, parallel argument evaluation
- dependence analysis uses alias graph, suffers from same problems as alias analysis
- techniques unlikely to speed up C programs on superscalar processors

○ PARCEL/Miprac [Harrison 86]

- uses an alternate representation of S-expressions to allow parallel evaluation of list operations

Problems with Alias Analysis

- Slow

- 30% to 1500% increase in standard compile time [Landi,Ryder 93]

- k-limiting

- Provides no information for loops

```
for (p = head; p != NULL; p = p->next)
    p->value += 4;
```

- can only represent finite structures

- No studies on speedups due to alias analysis

Software Approaches: Alias Analysis

```
p->value = 7;  
q->value = 8;
```

- Do p and q refer to the same location?
- Good solutions exist for pointerless languages [Cooper, Kennedy 89]
 - Only aliasing from reference parameters
- Alias computation with pointers is NP-hard
- Approximate algorithms exist [Landi, Ryder 93], [Choi, Burke, Carini 93]

Hardware Approaches

- Branch prediction and prefetch
- Speculative execution beyond branches:
 - Sentinel scheduling [Mahlke et al 1993]
 - Boosting [Smith, Horowitz, Lam 1990, 1992]
- Solves control-dependence problem

```
for (p = head; p != NULL; p = p->next)
    if (p->value == target)
        return p->index;
```

- But not the data dependence problem:

```
for (p = head; p != NULL; p = p->next)
    p->value += 4;
```

Related Work

Outline

- Related Work

- hardware
- software

- Research Plan

- Benchmark Suite
- Analysis Techniques
- Transformations

- Measurement and Evaluation

- Current Status

- Future Work

Motivation

- Software trend: more use of pointers
 - increasingly complex methods use increasingly sophisticated data structures
- Hardware trend: more instruction parallelism
 - Power2 chipset can execute 6 instructions/cycle
- Hardware/Software Gap
 - pointers inhibit compiler analysis
 - high-level transformations only apply to arrays
 - potential ILP can not be exploited

Research Goal

Speed Up Pointer-Intensive Programs

- use compiler analysis and transformation
- target current and future superscalar machines

What is a Pointer-Intensive Program?

- Program spends most of its time manipulating
 - linked lists
 - trees
 - graphs
- Recursive data structures (not arrays)
- Examples
 - Circuit Simulation
 - Optimizing Compiler

Optimization of Pointer-Intensive C Programs

David F. Bacon
University of California, Berkeley