# Code and Data Partitioning for the Local Stores on the Cell/BE processor

Georgia Tech STI/Cell/BE Workshop

IBM TJ Watson Static Compiler Team

© 2007 IBM Corporation

# Shared Memory Processor

- **CBE can be explicitly programmed as a shared-memory multiprocessor using two different instruction sets**
- **The SPEs and the PPE can be programmed to fully inter-operate in a cache-coherent Shared-Memory Multiprocessor Model**
  - Cache-coherent DMA operations for SPEs
  - DMA operations use effective address common to all PPE and SPEs
  - SPE shared-memory store instructions are replaced
    - A store from the register file to the LS
    - DMA operation from LS to shared memory
  - SPE shared-memory load instructions are replaced
    - DMA operation from shared memory to LS
    - A load from LS to register file
- **Of course … a compiler could provide much of this functionality.**

# Compiling a single source file for the cell

Single source

```
foo1 ();

#pragma omp parallel for
for (i=0; i < N; i++)
    A[i]  =  x * B[i];

foo2 ();
```

outline

foo3(LB,UB)

```
for (i=LB; i < UB; i++)
    A[i]  =  x * B[i];
Runtime barrier
```

clone

foo3_SPU (LB,UB)

```
for (i=LB; i < UB; i++)
    A[i]  =  x * B[i];
Runtime barrier
```

```
foo1 ();
Runtime distribution of wo
    invoke foo3, for i=[0,N)
Runtime barrier
foo2 ();
```

In SPE code:
A, B, and x are shared


© 2007 IBM Corporation

Single source

```
foo1 ();

#pragma omp parallel for
for (i=0; i < N; i++)
    A[i]  =  x * B[i];

foo2 ();
```

outline

foo3(LB,UB)

```
for (i=LB; i < UB; i++)
    A[i]  =  x * B[i];
Runtime barrier
```

clone

foo3_SPU (LB,UB)

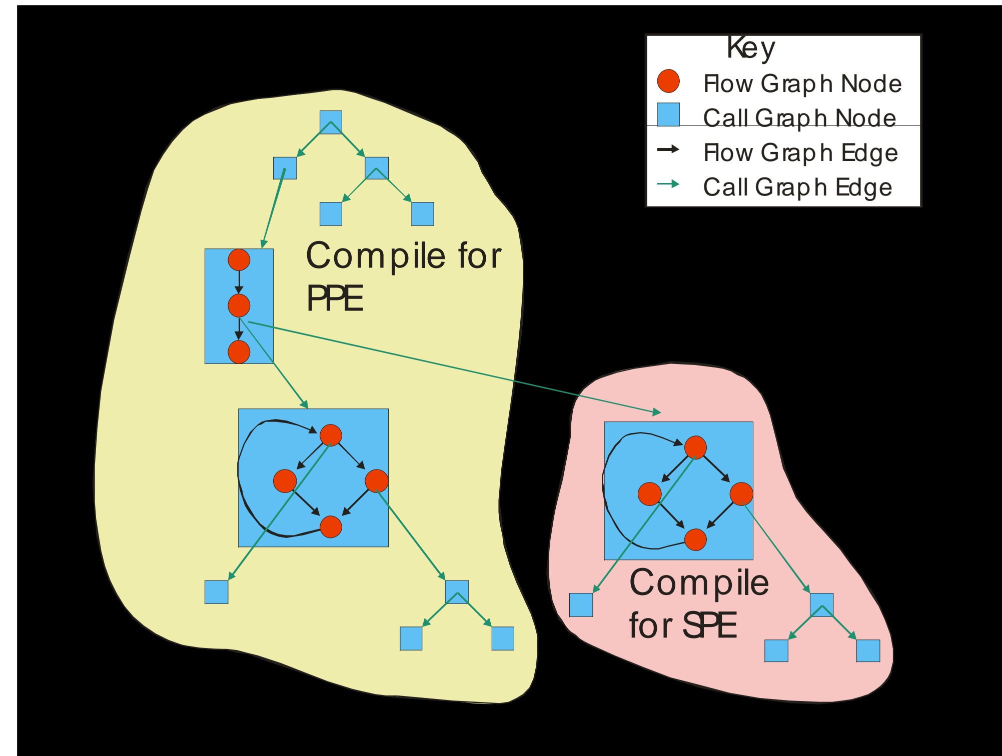
```
/** buffers A´[M], B´[M] **/

for ( k=LB;  k < UB;  k+=M) {
    DMA M elements of B into B´
    for (j=0;  j<M;  j++) {
        A´[j] = cache_lookup(x) * B´[j];
    }
    DMA M elements of A out of A´
}

Runtime barrier
```

```
foo1 ();
Runtime distribution of wo
    invoke foo3 and foo3_SP
    for i=[0,N)
Runtime barrier
foo2 ();
```


© 2007 IBM Corporation

# Cloning for heterogeneous processors

- **Outlined function becomes new node in call graph**
- **In pass 2 of TPO, using whole program call graph, outlined function is cloned, then specialized to create a ppe and an spe version**
- **All called functions must also be cloned**
- **SPE call sites modified to call SPE versions of cloned subroutines**
- **Partitioning pass creates SPE and PPE partitions and invokes lower level optimizer for machine specific optimization**



# Data Partitioning

- **Single Source assumption: all data lives in System Memory**
- **Naïve implementation, every load and store requires a dma operation**
  - Too costly (~700 cycles per load or store)
  - MP will require locking on every reference
- **What can be done to make this acceptable?**

# Prefetching

- **Example:**

Original Code

```
for(i=0;i<100000;i++)
    a[i]=b[i]+c[i];
```

Blocked, with prefetc

```
for(i=0;i<100000;i+=
    dma_get(b',b[i],4(
    dma_get(c',c[i],4(
    for(ii=0;ii<100;ii++
        a'[ii]=b'[ii]+c'[ii]
     dma_put(a[i],a',4
    }
```

Software Pipelined Prefetch

```
dma_get(b',b[0],400);
dma_get(c',c[0],400);
for(i=0;i<99900;i+=100) {
    dma_get(b",b[i+100],400);
    dma_get(c",c[i+100],400);
    for(ii=0;ii<100;ii++)
        a'[ii]=b'[ii]+c'[ii];
     dma_put(a[i],a',400);
     swap(a',a");
     swap(b',b");
     swap(c',c");
     }
for(ii=0;ii<100;ii++)
    a"[ii]=b"[ii]+c"[ii];
dma_put(a[i],a",400);
```


© 2007 IBM Corporation

# Irregular Accesses

***What do we do about this?***

```
for(i=0;i<100000;i++)
    a[i]=b[i]+c[i]*d[f(i)];
```

- **b and c can be prefetched, but d has an irregular access pattern, thus we cannot predict what elements of d are required**
- **we seem to be thrown back on the naïve implementation, d[f(i)] must be fetched on each iteration with a consequent large slowdown of the loop**
- **observation: it's as if every access to d incurred a cache miss**


© 2007 IBM Corporation

# Software Caching

Original Code

```
for(i=0;i<100000;i++)
    = … d[f(i)];
```

Code with explicit Cache Lookup

```
for(i=0;i<100000;i++)
    t=cache_lookup(d[f(i)];
    = … t;
```

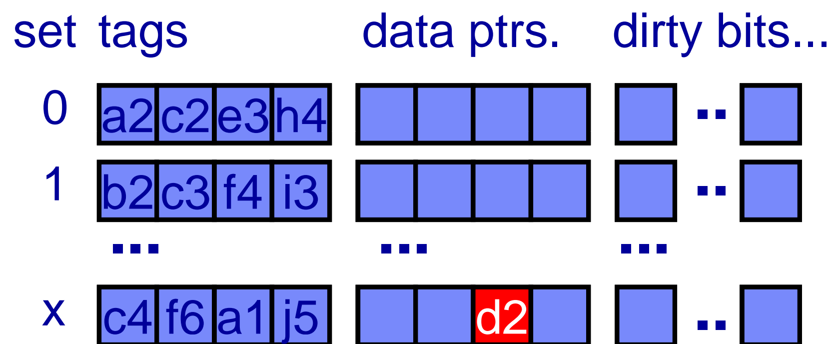
```
inline vector cache_lookup(addr) {
    if (cache_directory[addr&key_mask] != (addr&tag_mask))
       miss_handler(addr);
    return cache_data[addr&key_mask][addr&offset_mask];
    }
```

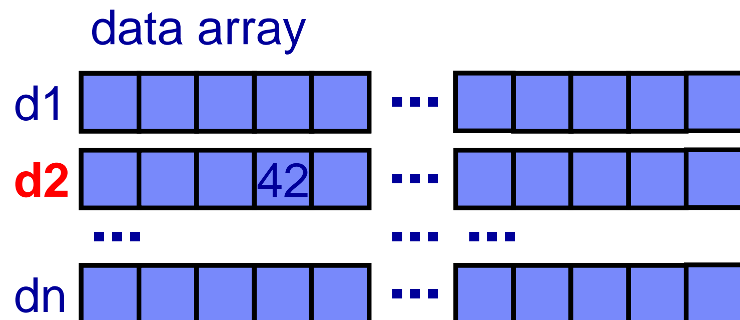
the miss handler will dma the required data, and some suitable quantity of surrounding data

higher degrees of associativity can be supported, possibly for little extra cost on a SIMD processor

 © 2007 IBM Corporation

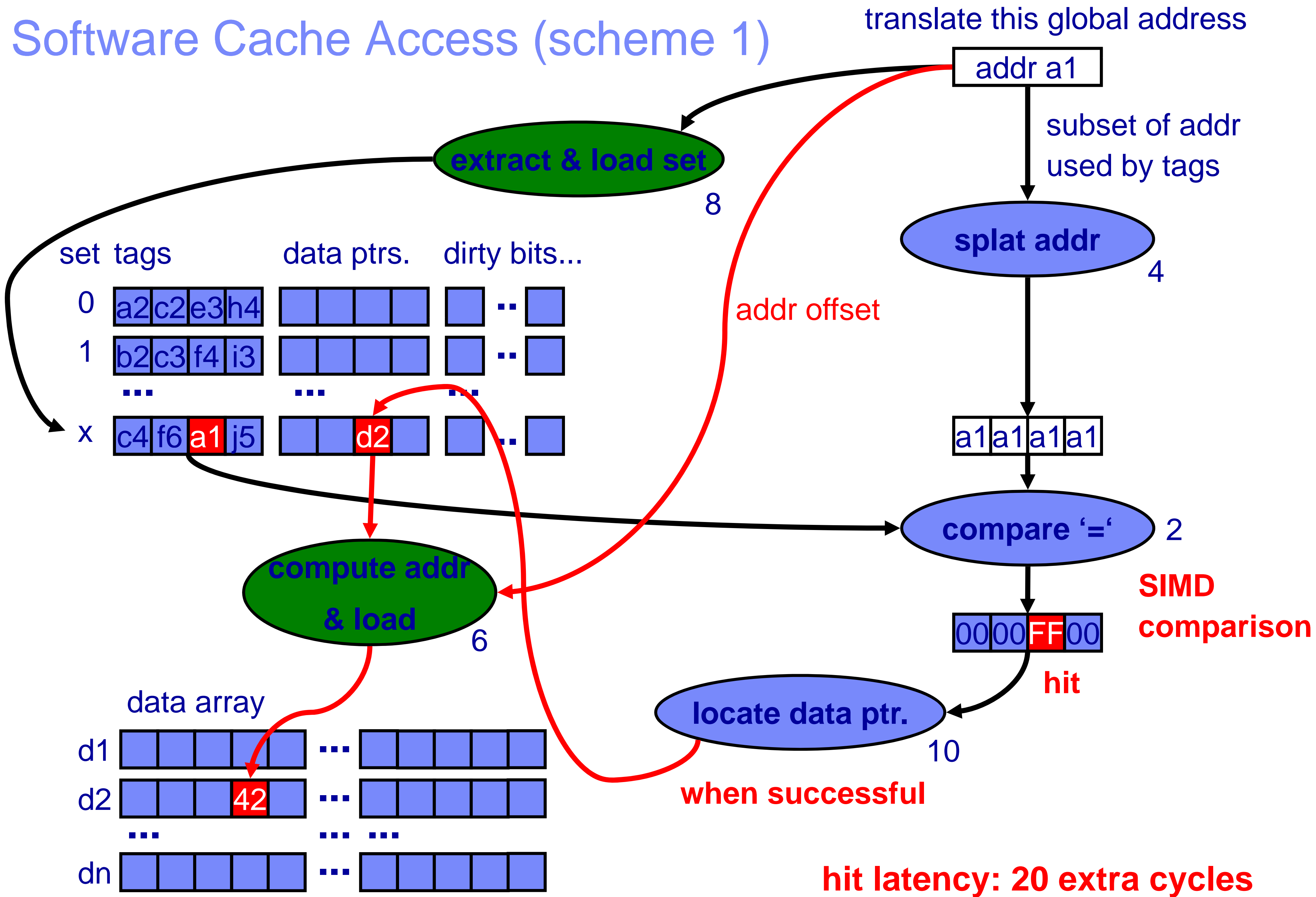
# Software Cache Architecture (scheme 1)




- **Cache directory**
  - 128-set, 4-way set associative
  - pointers to data lines
  - use 16KByte of data
  - Address: (a1 | a2), where a1 is used to compute the set to look into, and to compare with tags; a2 where for a given data array line, it select which value to return. For example here, a1 maps to set x, and a2 select the 4th element in the data array line d2
- **Data in a separate structure**
  - 512 x 128B lines
  - use 64KByte of data

# Software Cache Access (scheme 1)



# first scheme

```
 0 addr0 = addr & mask-keep-set        addr1 = splat(addr)
 1                                     addr3 = addr & mask-keep-offset
 2 tag = load addr0, base of tag
 3 ptrs = load addr0, base of tag+16
 4                                     addr2 = addr1 & mask-remove-offs
 5
 6
 7
 8 hit = (tag ==addr2)
 9
10 hitbyte = gbb hit
11
12
13
14 zeros = count-zero hitbyte
15
16 hit-ptr = rotate-quad-byte ptrs, zeros
17
18
19
20 data = load data-addr, addr3
21
22
23
24
25
26 data available
```

# Combining Prefetch with Software Cache

Original Code

```
for(i=0;i<100000;i++)
    a[i]=b[i]+c[i]*b[f(i)];
```

Prefetching and Caching

```
for(i=0;i<100000;i+=100) {
    dma_get(b',b[i],400);
    dma_get(c',c[i],400);
    for(ii=0;ii<100;ii++) {
        t=cache_look_up(b[f(i)]);
        a'[ii]=b'[ii]+c'[ii]*t;
    }
    dma_put(a[i],a',400);
}
```

- **We may already have b[f(i)] in local store as a result of prefetching**
- **in this example, the only effect is to cause unneccesary misses**
- **but if we substitute a[f(i)] for the last term …**

**Prefetching must also update the cache directory**

**and**

**Miss handling must not evict prefetched data**


© 2007 IBM Corporation

# Coherence Problem

- **SPE accesses data in global memory through two mechanisms:**
  - Software controlled cache
  - Static buffers
- **Incorrect value may be used or generated if coherence is not maintained. Examples:**
  - Two copies of data in software controlled cache and static buffer. One changes the value and the other one may read a stale value
  - Multiple copies of data in different static buffers
- **Approaches:**
  - Compiler: no runtime overhead,
  - Runtime: more powerful but complicated


© 2007 IBM Corporation

# Solution Overview

- **Combine two approaches for optimal solution**
  - Try to apply compiler solution as much as possible
  - Resort to runtime solution if necessary
- **Components**
  - Local coherence simplification
  - Global coherence avoidance analysis
  - Dynamic coherence maintenance


© 2007 IBM Corporation

# Local Coherence Simplification

- **There is no coherence problem for this static buffer in the loop**
- **Runtime coherence maintenance is needed only**
  - At the entry of loop: DMA read and check whether the software controlled cache has updated data
  - At the exit of loop:
    - Write-through: update the hit cache line and DMA write
    - Write-back: put the static buffer content into cache
- **Pros/Cons**
  - Requires local data dependence info, which may be more likely to be available
  - The structure of software controlled cache remains unchanged
- **References are put into static buffer in a loop only when there is no data dependence between the reference and any other reference accessed by software controlled cache or another static buffer in the loop.**
  - The coherence maintenance can be overlapped with DMA operations
  - Candidates for static buffer may be lost if the data dependence information is too conservative


© 2007 IBM Corporation

# Global Coherence Avoidance Analysis

- **Runtime coherence maintenance can be avoided by compiler analysis**
  - At entry: if there is no updated cache line for this static buffer
  - At exit: if there is no cache line for this static buffer already in cache that will be referenced later
- **How the compiler predicts cache contents**
  - No lines in cache after flush
  - If data is carefully aligned or padded, compiler can assume different variables will never be in the same cache line
  - Can not predict the replacement. A line will be assumed to stay in cache until flush


© 2007 IBM Corporation

# Details About Compiler Analysis

- **The coherence avoidance analysis is a special inter-procedural data flow analysis for flush and data accesses**
- **For DMA read: similar to Def-Use analysis**
  - Control flow: forward
  - GEN: a store to this variable through software controlled cache
  - KILL: explicit/implicit flushes in OMP
  - Merge function: OR
  - Initial value: empty
  - Result: if data flow status set at the exit of the loop is empty,  no runtime coherence maintenance for DMA read is need


© 2007 IBM Corporation

# Details About Compiler Analysis

- **Coherence maintenance is needed for DMA write only when the following two conditions are TRUE :**
  - C1: There is a line before the loop, and
  - C2: There is a read after the loop
- **Analysis for condition C1 is similar to the previous analysis except**
  - Gen: read/write variables ( which may share cache line with this DMA write) through software controlled cache
- **Analysis for condition C2 is only needed for write through policy:**
  - Control flow: backward
  - Gen: read this variable through software cache
  - Kill: flush
  - Merge: OR
  - Initial value: empty
- **Result: if data flow status set of C1 or C2 at the entry of the loop is empty, no runtime coherence maintenance for DMA write is need**
- **More precise result can be obtained if array subscript is processed**
  - Arrays are treated as scalars now.
  - Different parts of arrays may be referenced
  - More kill in the C2 analysis


© 2007 IBM Corporation

# Optimization with Flushes

- **When runtime coherence maintenance is needed by the previous analysis, it may be profitable to insert extra cache flushes to avoid the coherence maintenance**
- **Flush can be a flush for one variable or combine them as flush all**
- **The previous analysis can provide information about the possible insertion points for flush**
  - Move in the control flow graph to reduce the overhead
  - Similar to the algorithm of partial redundant elimination.
  - Branch profiling may help


© 2007 IBM Corporation

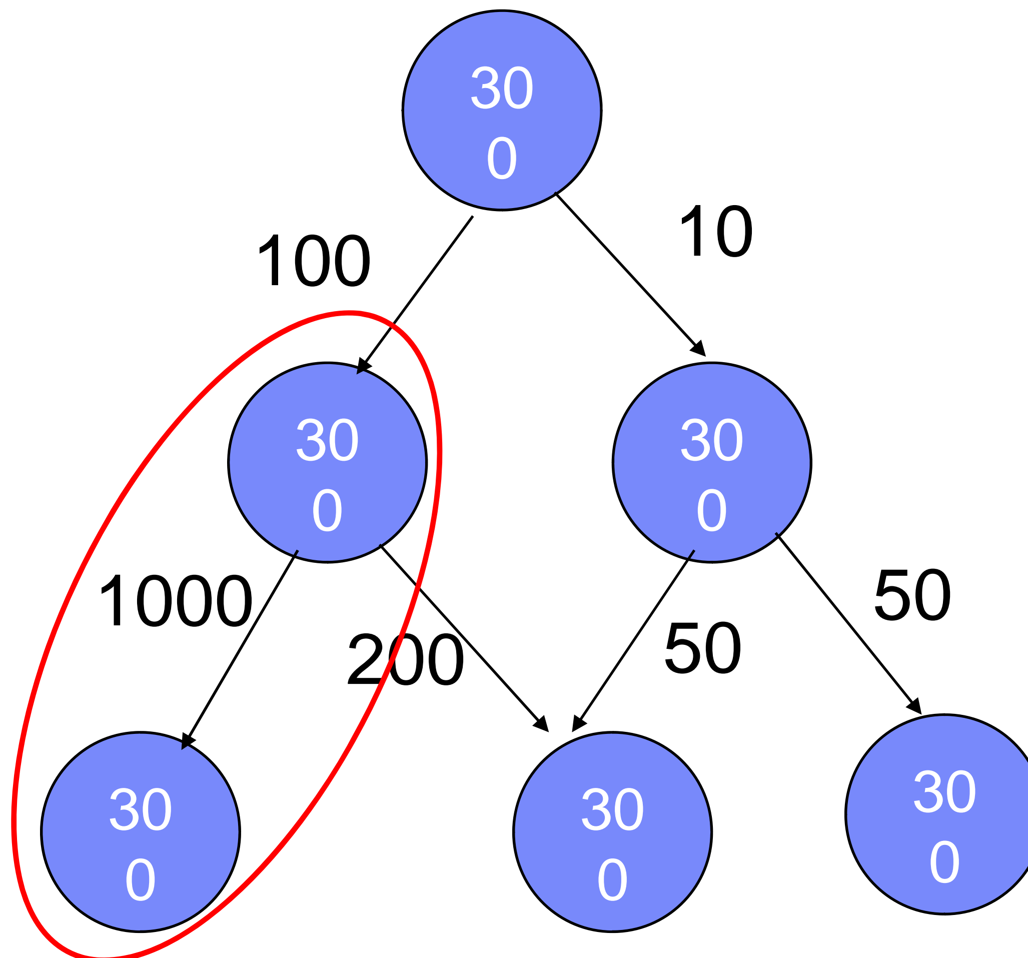
# Call Graph Partitioning Algorithm

- **Build an affinity graph based on the global call graph.**
  - Each global call graph node becomes a node in the affinity graph and costs some memory
  - Each call graph edge becomes an edge in the affinity graph
- **Each call graph edge is weighted.**
  - Estimated through static program analysis
  - Profiling
- **Apply maximum spanning tree algorithm to the affinity graph.**
  - Process edges by the order of the weight
  - If merging the two nodes of the edge does not exceed the memory limitation, then merge, and so on.
- **Each (merged) node left is a program partition.**


© 2007 IBM Corporation

# An Example of Call Graph Partitioning

Assume Memory Limitation is 1000

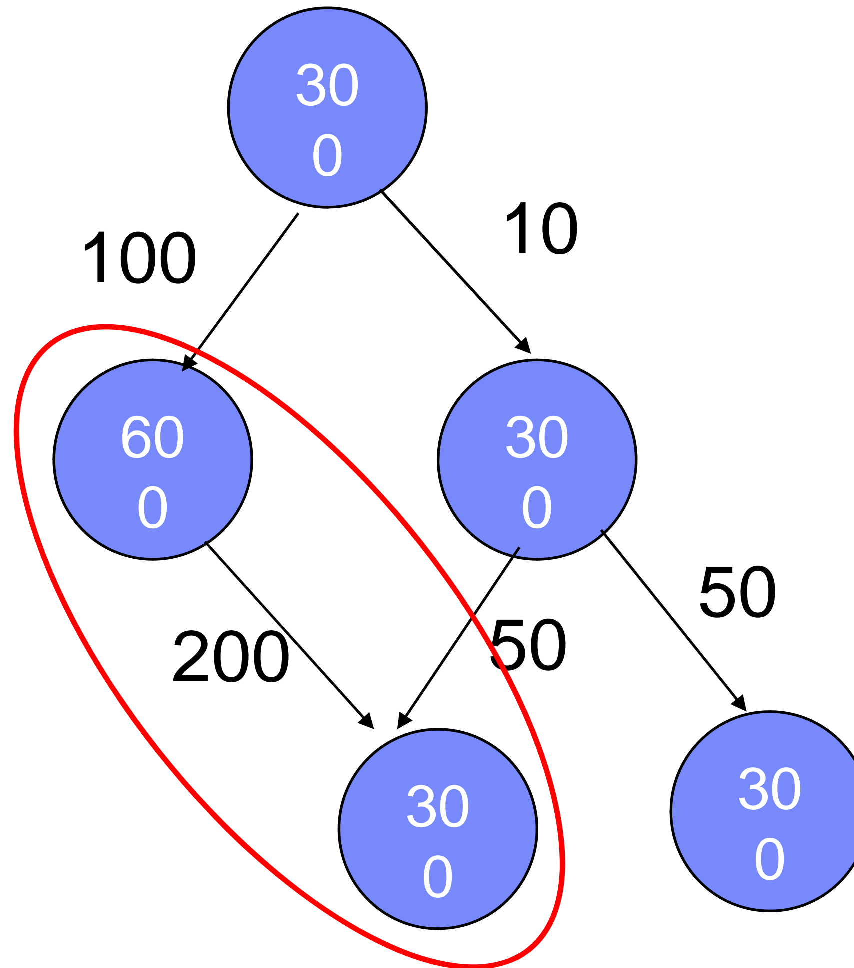




© 2007 IBM Corporation

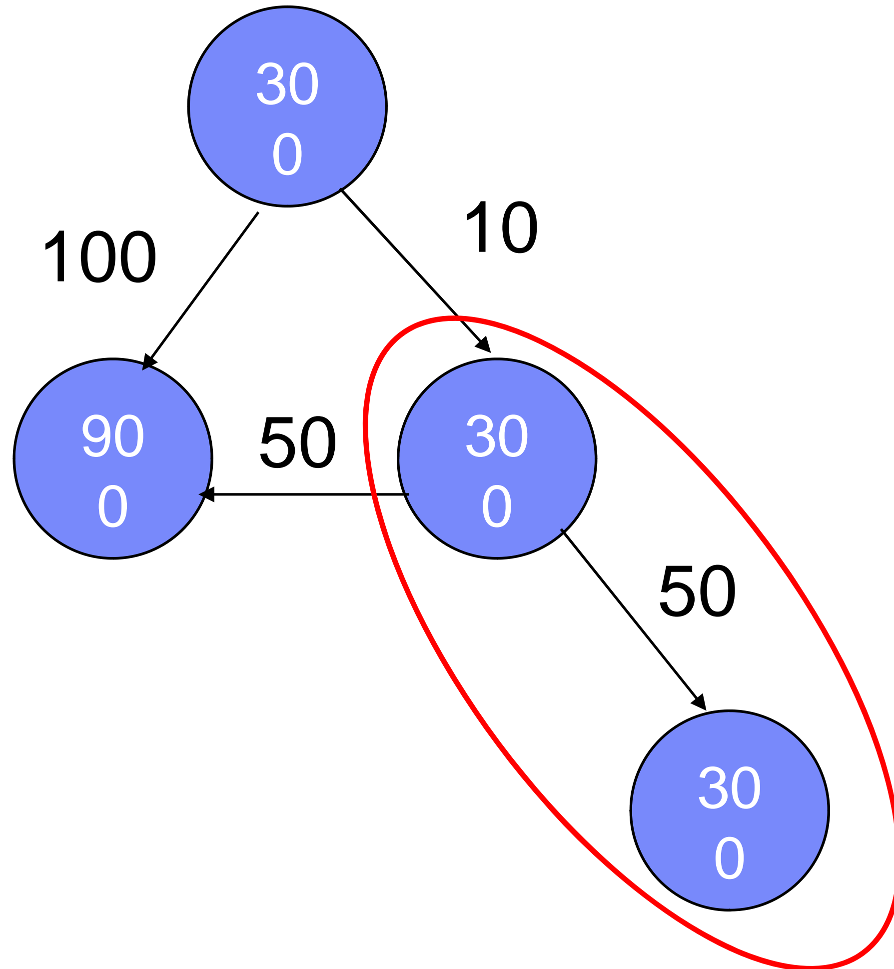
# An Example of Call Graph Partitioning

Assume Memory Limitation is 1000




 © 2007 IBM Corporation

# An Example of Call Graph Partitioning

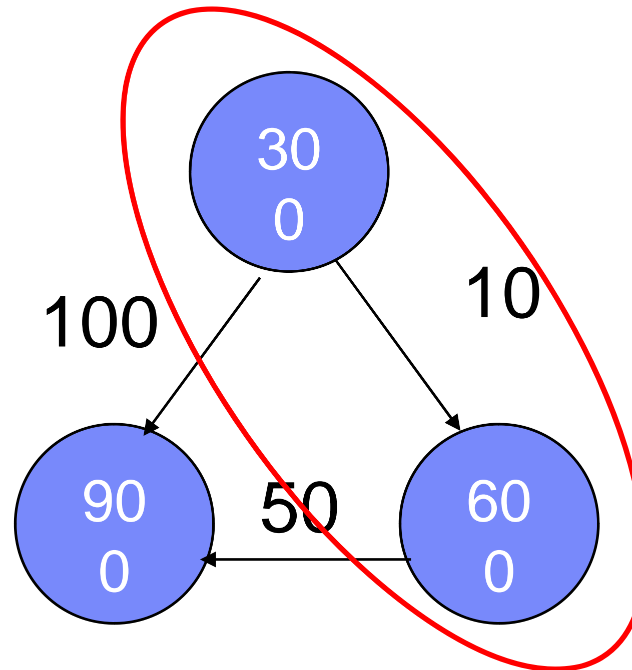




© 2007 IBM Corporation

# An Example of Call Graph Partitioning

Assume Memory Limitation is 1000

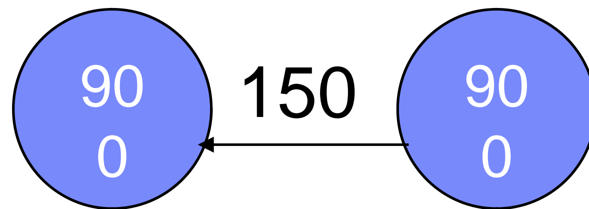




© 2007 IBM Corporation

# An Example of Call Graph Partitioning

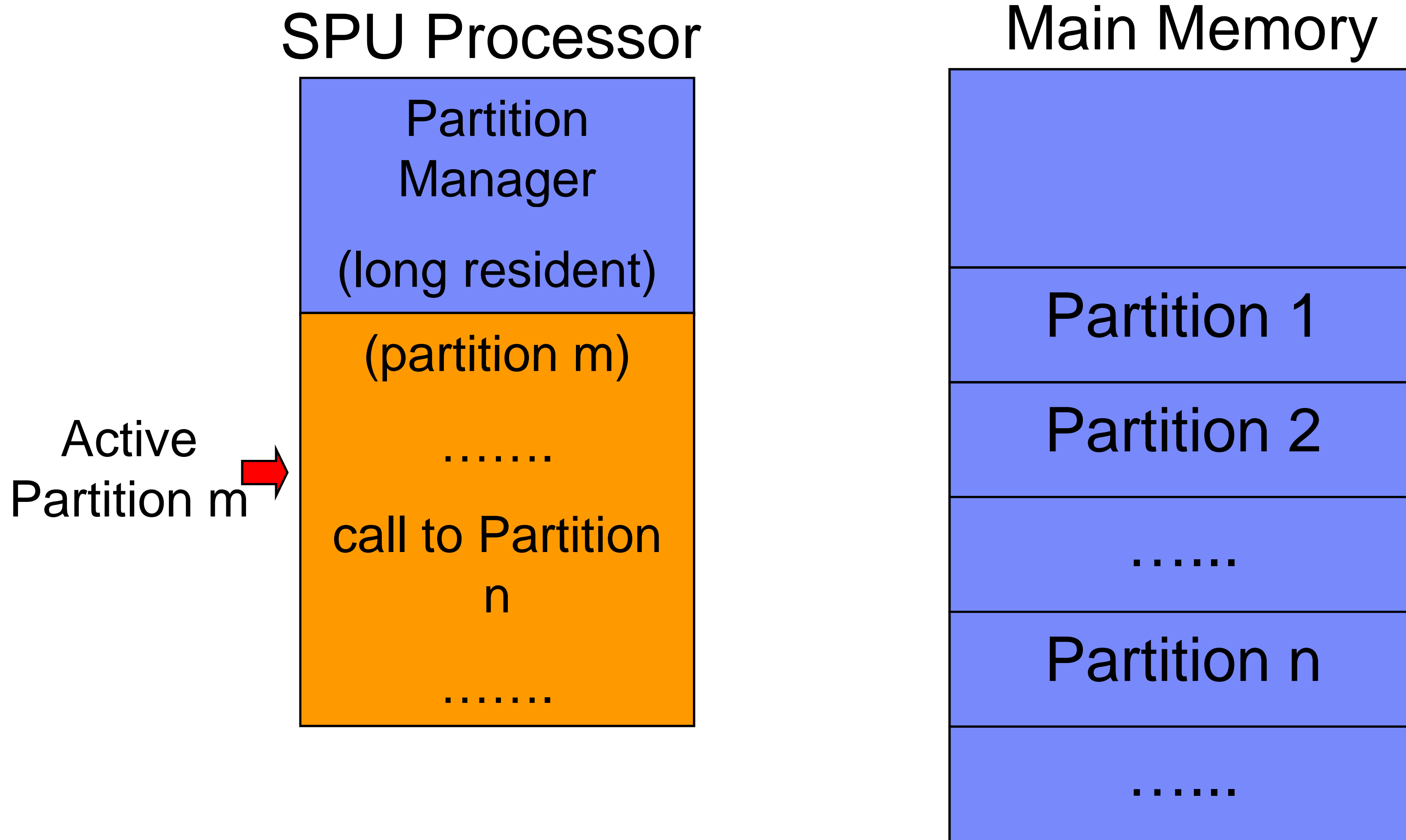
Assume Memory Limitation is 1000





© 2007 IBM Corporation

# Partition Manager Overview



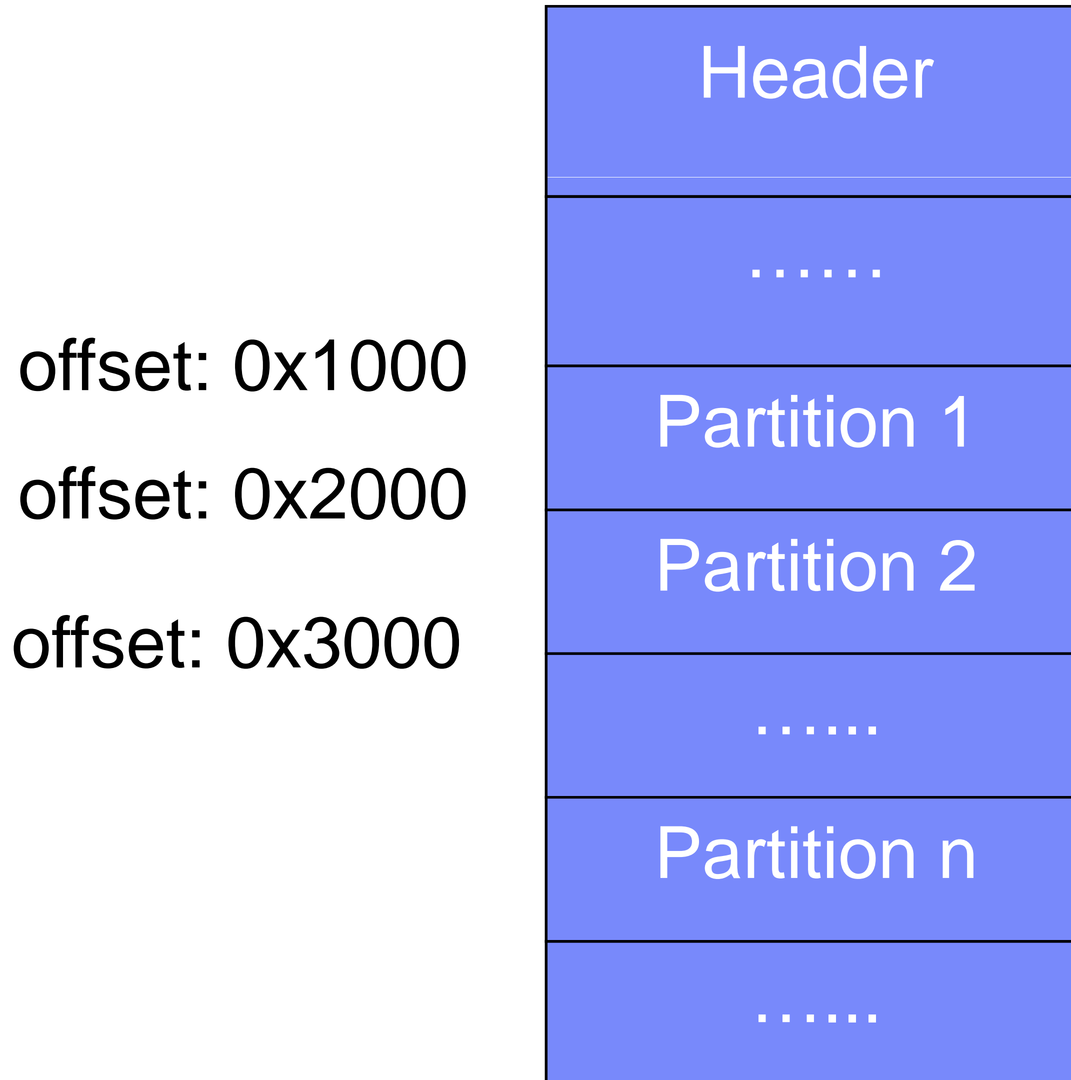


© 2007 IBM Corporation

# Partition Manager Overview

© 2007 IBM Corporation

# OVERLAY command effect: Binary View

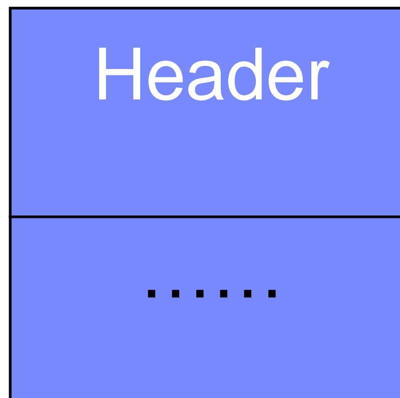




© 2007 IBM Corporation

# OVERLAY command effect: Execution View

Program Memory Image




virtual address: 0x1000

Partition 1

Partition 2

……

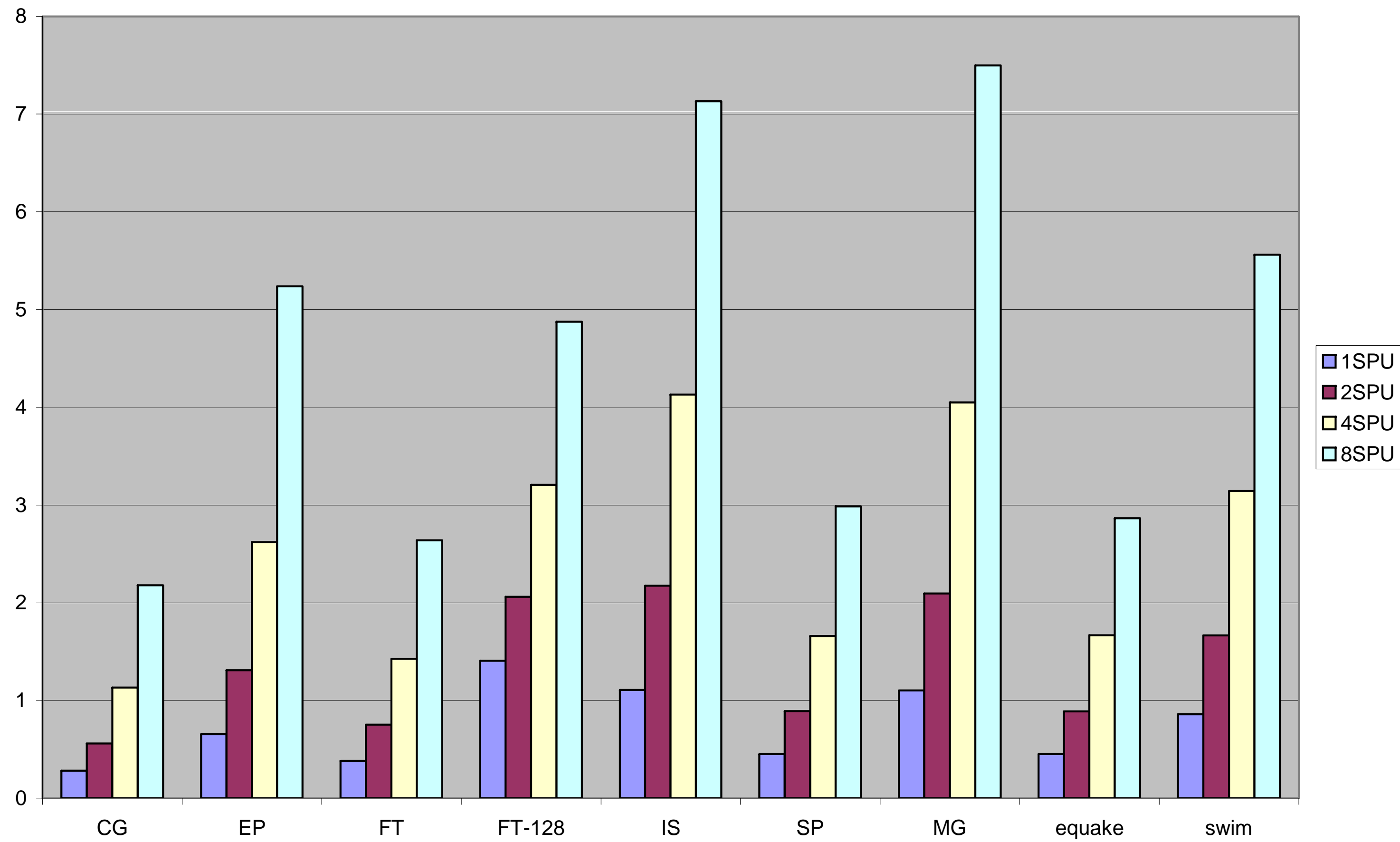
Partition n

 © 2007 IBM Corporation

# Optimizations

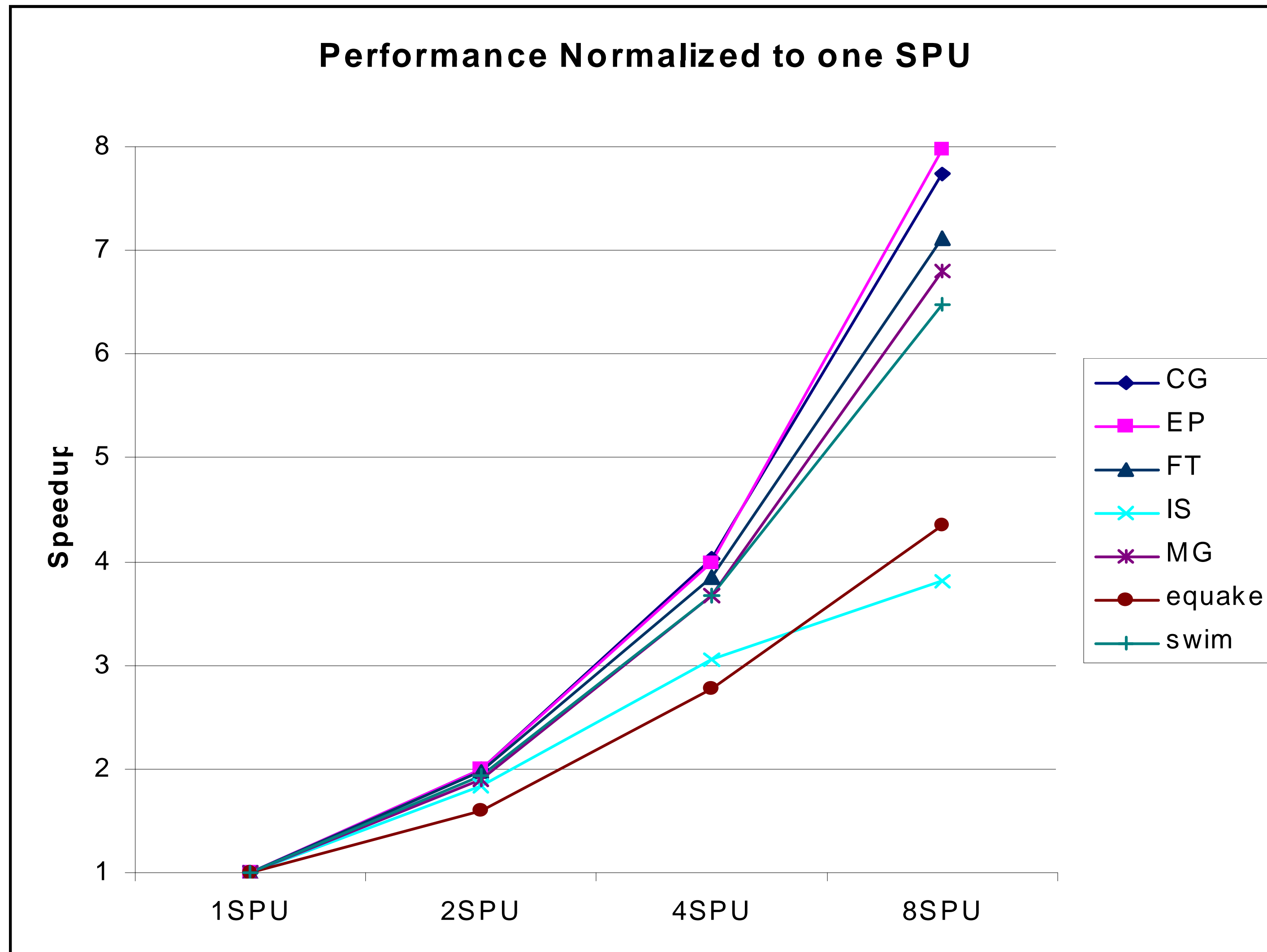
- Profiling to get accurate call edge frequencies
  - Especially with the presence of a lot of indirect calls through function pointers
- Get the accurate function code size
  - Currently estimated
  - Conservative, very rough
- Leaf function duplication
  - Some leaf functions are referenced by two non-coalescable partitions
  - May be beneficial to duplicate the function
- Double Buffering
  - Rely on good prefetching to be beneficial
  - Prefetching is a difficult problem


© 2007 IBM Corporation





© 2007 IBM Corporation




- **NAS and SPEC OMP benchmarks, speedups against 1 SPE**
- **Scalability generally very good**
  - IS and equake not good due to non-parallelized loops


© 2007 IBM Corporation