Billion Transistor Chips
Multicore Low Power Architectures

Krishna Kavi

Department of Computer Science and Engineering

The University of North Texas

kavi@cse.unt.edu

http://csrl.csci.unt.edu/~kavi

Billion Transistor Chips

How to garner the silicon real-estate for improved performance?

More CPU's per chip -- Multi-core systems

More threads per core -- hyper-threading

More cache and cache levels (L1, L2, L3)

System on a chip and Network on chip

Hybrid system including reconfigurable logic

But, embedded system require careful management of energy

Billion Transistor Chips

How to garner the silicon real-estate for improved performance?

We propose innovative architecture that “scales” in performance as needed, but disables hardware elements when not needed.

We address several processor elements for performance and energy savings

Multithreaded CPUs
Cache Memories
Redundant function elimination
Offload administrative functions

Computer Architecture Research

A new multithreaded architecture called Scheduled Dataflow(SDF)

Uses Non-Blocking Multithreaded Model
Decouples Memory access from execution pipelines
Uses in-order execution model (less hardware complexity)

The simpler hardware of SDF may lend itself better for embedded applications with stringent power requirements

Computer Architecture Research

Intelligent Memory Devices (IRAM)

- Delegate all memory management functions to a separate processing unit embedded inside DRAM chips
- More efficient hardware implementations of memory management are possible
- Less cache conflicts between application processing and memory management

- More innovations are possible

Computer Architecture Research

Array and Scalar Cache memories

Most processing systems have a data cache and instruction cache. WHY?

Can we split data cache into a cache for scalar data and one for arrays?

We show significant performance gains
with 4K scalar cache and 1k array cache we get the same performance as a 16K cache

Computer Architecture Research

Function Reuse

Consider a simple example of a recursive function like Fib

```
int fib (int);
int main()
{ printf ("The value is %d .\n ", fib (num) )}
int fib (int num)
{ if (num == 1) return 1;
     if (num == 2) return 1;
     else  {return fib (num-1) + fib (num-2);}
```

For Fib (n), we call Fib(n-1) and **Fib(n-2);**

For Fib(n-1) we call **Fib(n-2)** and Fib (n-3)

So we are calling Fib(n-2) twice

Can we somehow eliminate such redundant calls?

Computer Architecture Research

What we propose is to build a table in hardware and save function Calls.

Keep the "name", and the input values and results of functions

When a function is called, check this table if the same function is called with the same inputs

If so, skip the function call, and use the result from a previous call

This slide is deliberately left blank

Overview of our multithreaded SDF

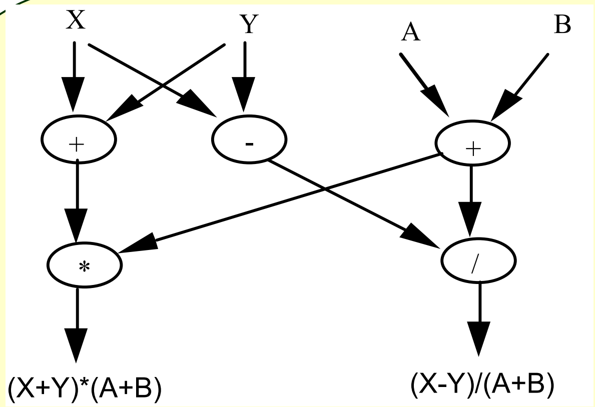
- Based on our past work with Dataflow and Functional Architectures
- Non-Blocking Multithreaded Architecture
  - Contains multiple functional units like superscalar and other multithreaded systems
  - Contains multiple register contexts like other multithreaded systems
- Decoupled Access - Execute Architecture
  - Completely separates memory accesses from execution pipeline

Background

How does a program run on a computer?

- A program is translated into machine (or assembly) language
- The program instructions and data are stored in memory (DRAM)
- The program is then executed by ‘fetching’ one instruction at a time
- The instruction to be fetched is controlled by a special pointer called program counter
- If an instruction is a branch or jump, the program counter is changed to the address of the target of the branch

Dataflow Model



MIPS like instructions

```
1. LOAD    R2, A            / load A into R2
2. LOAD    R3, B            / load B into R3
3. ADD     R11, R2, R3      / R11 = A+B
4. LOAD    R4, X            / load X into R4
5. LOAD    R5, Y            / load Y into R5
6. ADD     R10, R4, R5      / R10 = X+Y
7. SUB     R12, R4, R5      / R12 = X-Y
8. MULT    R14, R10, R11    / R14 = (X+Y)*(A+B)
9. DIV     R15, R12, R11    / R15 = (X-Y)/(A+B)
10. STORE  ....,  R14       / store first result
11. STORE  .....,  R15      / store second result
```

Pure Dataflow Instructions

```
1:  LOAD   3L        / load  A, send to Instruction 3
2:  LOAD   3R        / load B, send to Instruction 3
3:  ADD    8R, 9R    / A+B, send to Instructions 8 and 9
4:  LOAD   6L, 7L    / load X, send to Instructions 6 and 7
5:  LOAD   6R, 7R    / load Y, send to Instructions 6 and 7
6:  ADD    8L        / X+Y, send to Instructions 8
7:  SUB    9L        / X-Y,  send to Instruction 9
8:  MULT   10L       / (X+Y)*(A+B), send to Instruction 10
9:  DIV    11L       / (X-Y)/(A+B), send to Instruction 11
10: STORE            / store first result
11: STORE            / store second result
```

SDF Dataflow Model

We use dataflow model at thread level
Instructions within a thread are executed sequentially

We also call this non-blocking thread model

Blocking vs Non-Blocking Thread Models

Traditional multithreaded systems use blocking models

- A thread is blocked (or preempted)
- A blocked thread is switched out
  and execution resumes in future

- In some cases, the resources of a blocked thread
- (including register context) may be assigned to other
- awaiting threads.
- Blocking models require more context switches

In a non-blocking model, once a thread begins execution, it will not be stopped (or preempted) before it completes execution

Non-Blocking Threads

Most functional and dataflow systems use non-blocking threads

A thread/code block is enabled when all its inputs are available.

A scheduled thread will run to completion.

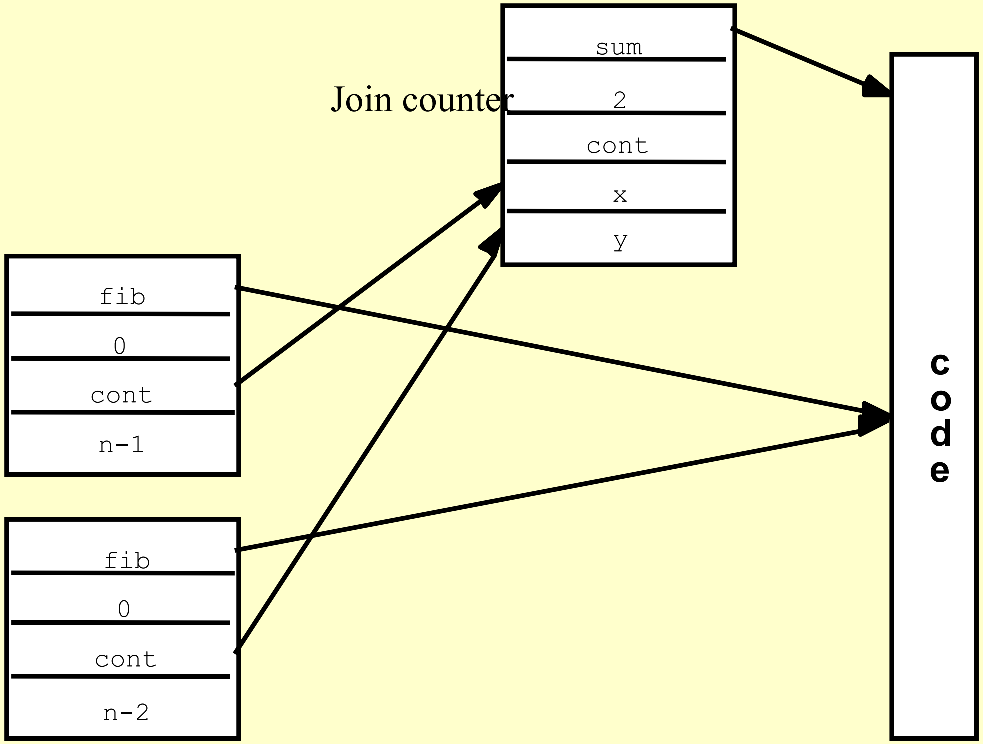
Similar to Cilk Programming model

Note that recent versions of Cilk (Clik-5) permits thread blocking and preemptions

Cilk Programming Example

```
thread fib (cont int k, int n)
    {   if (n<2)
            send_argument (k, n)
        else{
            cont int x, y;
            spawn_next sum (k, ?x, ?y);    /* create a successor thread
            spawn fib (x, n-1);            /* fork a child thread
            spawn fib (y, n-2);           /* fork a child thread
                        }}
  thread sum (cont int k, int x, int y)
       {send_argument (k, x+y);}        /* return results to parent's
                                        /*successor
```

Cilk Programming Example

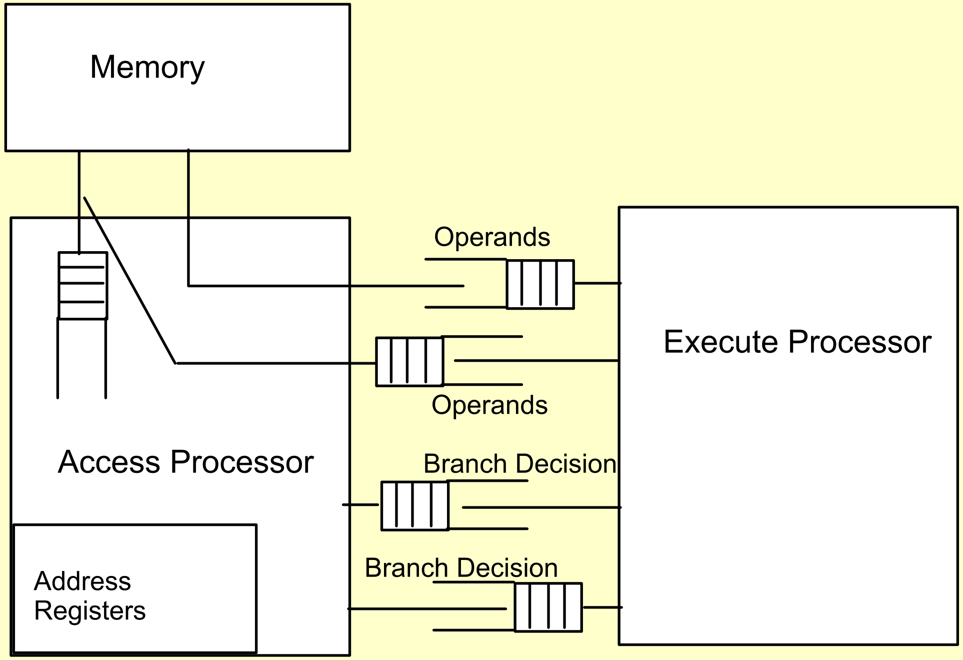


Decoupled Architectures

Separate memory accesses from execution

Separate Processor to handle all memory accesses

The earliest suggestion by J.E. Smith -- DAE architecture



Limitations of DAE Architecture

- Designed for STRETCH system with no pipelines
- Single instruction stream
- Instructions for Execute processor must be coordinated with the data accesses performed by Access processor
- Very tight synchronization needed
- Coordinating conditional branches complicates the design
- Generation of coordinated instruction streams for Execute and Access my prevent traditional compiler optimizations

Our Decoupled Architecture

We use multithreading along with decoupling ideas

Group all LOAD instructions together at the head of a thread

Pre-load thread's data into registers before scheduling for execution

- During execution the thread does not access memory

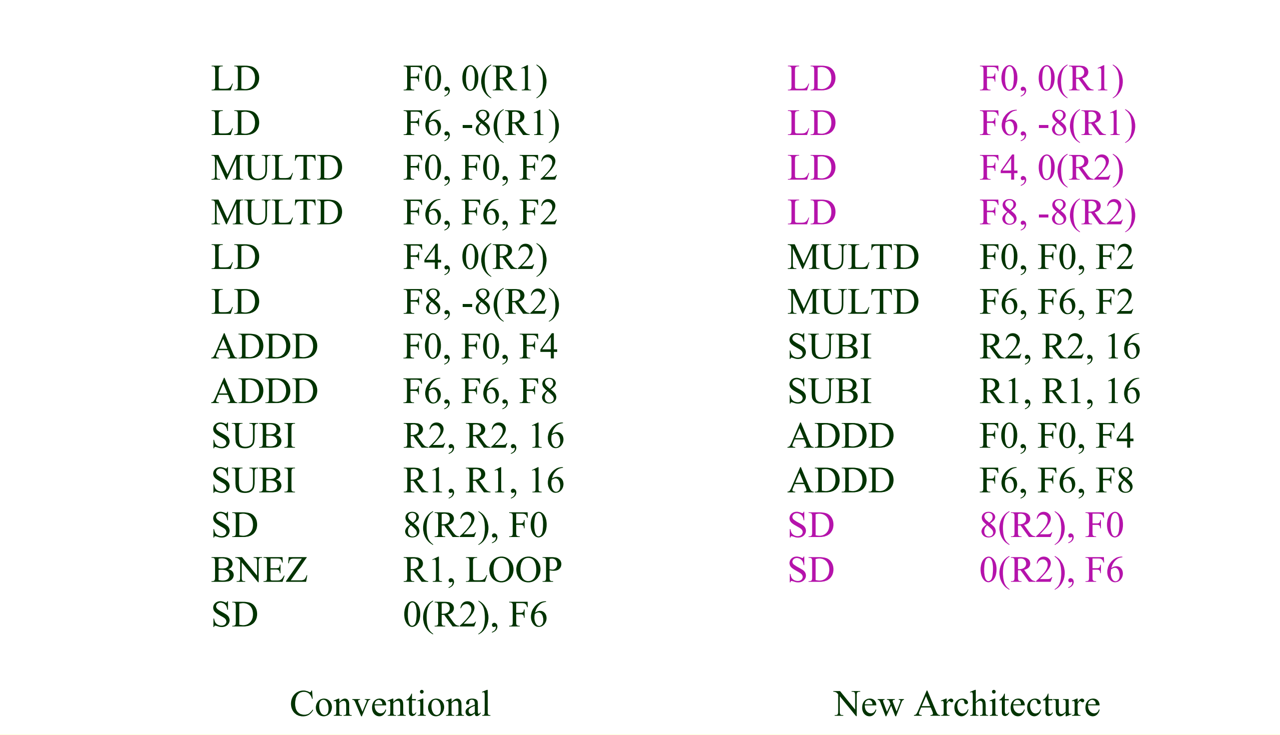
Group all STORE instructions together at the tail of the thread

Post-store thread results into memory after thread completes execution

- Data may be stored in awaiting Frames

Our non-blocking and fine grained threads facilitates a clean separation of memory accesses into Pre-load and Post-store

Pre-Load and Post-Store



```
LD       F0, 0(R1)
LD       F6, -8(R1)
MULTD    F0, F0, F2
MULTD    F6, F6, F2
LD       F4, 0(R2)
LD       F8, -8(R2)
ADDD     F0, F0, F4
ADDD     F6, F6, F8
SUBI     R2, R2, 16
SUBI     R1, R1, 16
SD       8(R2), F0
BNEZ     R1, LOOP
SD       0(R2), F6
```

Conventional

```
LD       F0, 0(R1)
LD       F6, -8(R1)
LD       F4, 0(R2)
LD       F8, -8(R2)
MULTD    F0, F0, F2
MULTD    F6, F6, F2
SUBI     R2, R2, 16
SUBI     R1, R1, 16
ADDD     F0, F0, F4
ADDD     F6, F6, F8
SD       8(R2), F0
SD       0(R2), F6
```

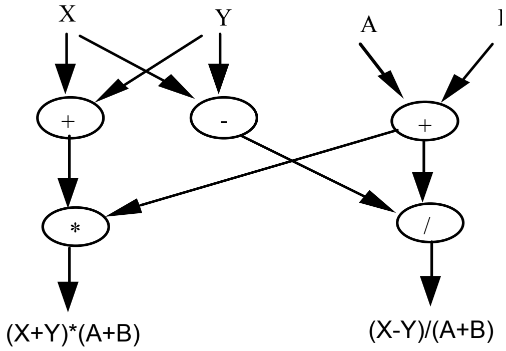
New Architecture

Features Of Our Decoupled System

- No pipeline bubbles due to cache misses
- Overlapped execution of threads
- Opportunities for better data placement and prefetching
- Fine-grained threads -- A limitation?
- Multiple hardware contexts add to hardware complexity

If 36% of instructions are memory access instructions, PL/PS can achieve 36% increase in performance with sufficient thread parallelism and completely mask memory access delays!

A Programming Example



Pre-Load

```
LOAD  RFP| 2,   R2          / load A into R2
LOAD  RFP| 3,   R3          / load B into R3
LOAD  RFP| 4,   R4          / load X into R4
LOAD  RFP| 5,   R5          / load Y into R5
LOAD RFP| 6,    R6          / frame pointer for returning first result
LOAD RFP| 7,    R7          / frame offset for returning first result
LOAD RFP| 8,    R8          / frame pointer for returning second result
LOAD RFP| 9,   R9           / frame offset for returning second result
```

Execute

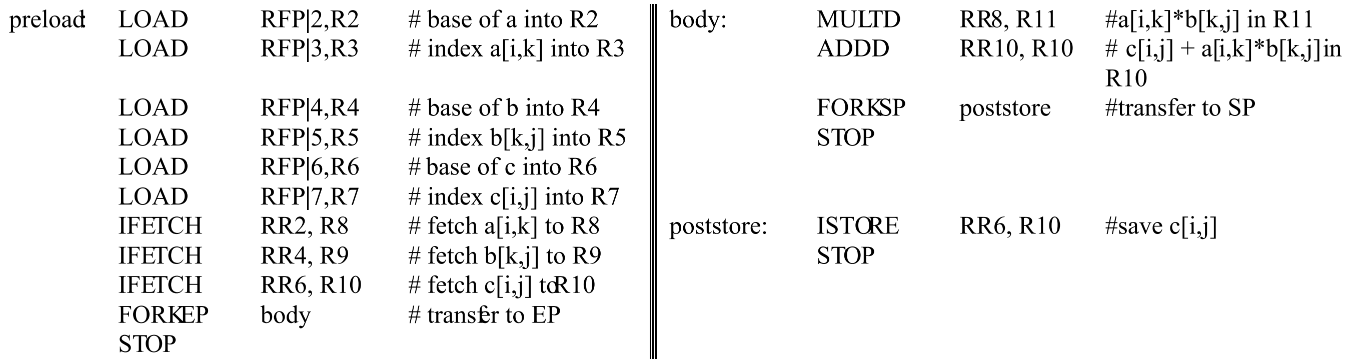
```
ADD     RR2, R11, R13       / compute A+B,  Result in R11 and R13
ADD     RR4, R10            / compute X+Y,  Result in R10
SUB     RR4,  R12           / compute X – Y, Result in R12
MULT    RR10, R14           / compute (X+Y)*(A+B), Result in R14
DIV     RR12, R15           / compute (X-Y)/(A+B), Result in R15
```

Post-Store

```
STORE   R14,   R6|R7        / store first result

STORE   R15,  R8|R9         / store second result
```

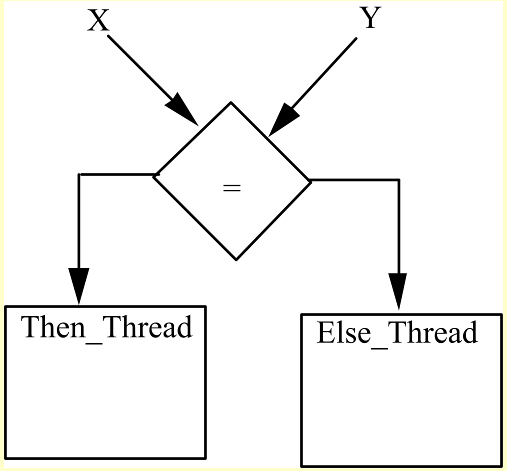
A Programming Example



```
preload:  LOAD     RFP|2,R2    # base of a into R2       ||  body:       MULTD    RR8, R11     #a[i,k]*b[k,j] in R11
          LOAD     RFP|3,R3    # index a[i,k] into R3    ||              ADDD     RR10, R10    # c[i,j] + a[i,k]*b[k,j]in
                                                         ||                                    R10
          LOAD     RFP|4,R4    # base of b into R4       ||              FORKSP   poststore    #transfer to SP
          LOAD     RFP|5,R5    # index b[k,j] into R5    ||              STOP
          LOAD     RFP|6,R6    # base of c into R6       ||
          LOAD     RFP|7,R7    # index c[i,j] into R7    ||
          IFETCH   RR2, R8     # fetch a[i,k] to R8      ||  poststore:  ISTORE   RR6, R10     #save c[i,j]
          IFETCH   RR4, R9     # fetch b[k,j] to R9      ||              STOP
          IFETCH   RR6, R10    # fetch c[i,j] to R10     ||
          FORKEP   body        # transfer to EP          ||
          STOP                                           ||
```

Figure 4: A SDF Code Example

Conditional Statements in SDF



Pre-Load

```
LOAD  RFP| 2,    R2        / load X into R2
LOAD  RFP| 3,    R3        / load Y into R3
                        / frame pointers for returning  results
                          / frame offsets for returning results
```

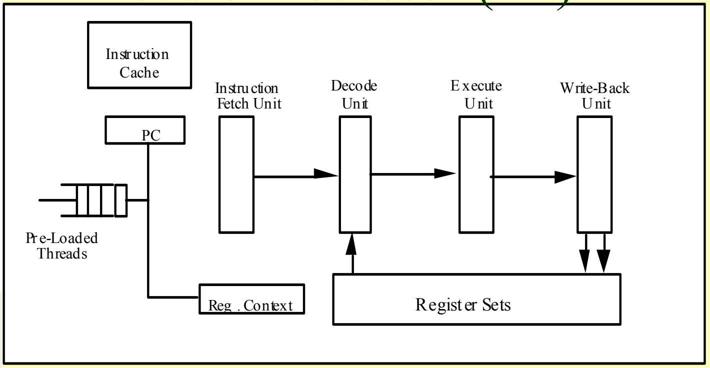
Execute

```
EQ              RR2, R4                  / compare R2 and R3, Result in R4
NOT             R4, R5                   / Complement of R4 in R5
FALLOC          "Then_Thread"            / Create Then Thread (Allocate Frame memory, Set Synch-Count,
FALLOC          "Else_Thread"            / Create Else Thread (Allocate Frame memory, Set Synch-Count,
FORKSP          R4, "Then_Store"         /If X=Y, get ready post-store "Then_Thread"
FORKSP          R5, "Else_Store"         /Else, get ready pre-store "Else_Thread"
STOP
```

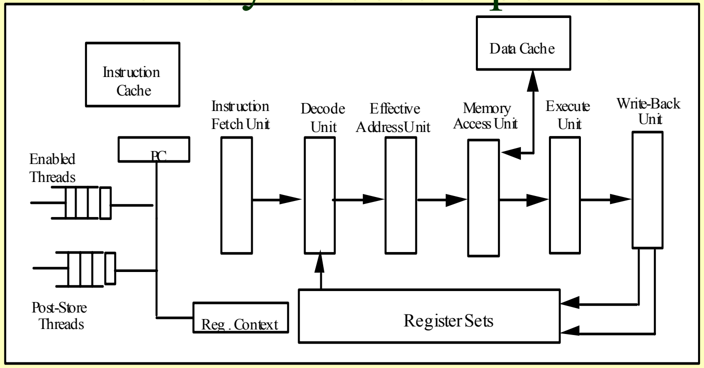
In Then_Thread, We de-allocate (FFREE) the Else_Thread
and vice-versa

SDF Architecture

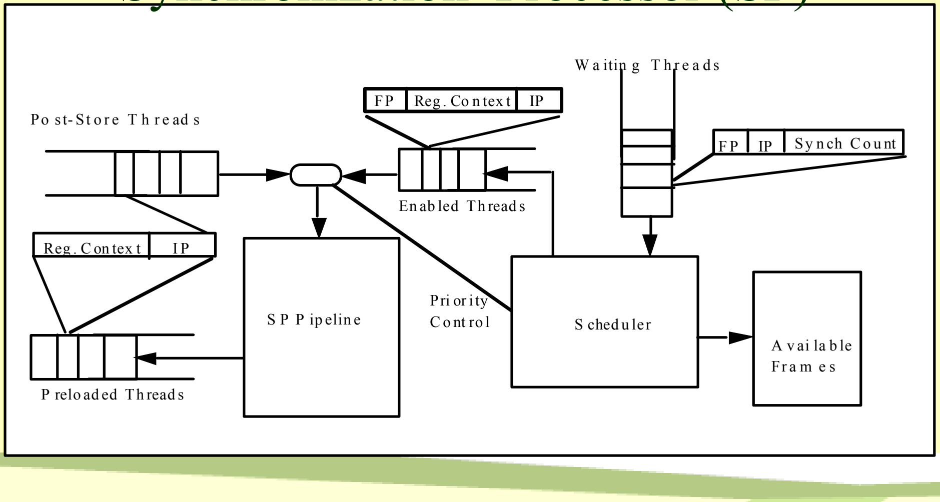
Execute Processor (EP)



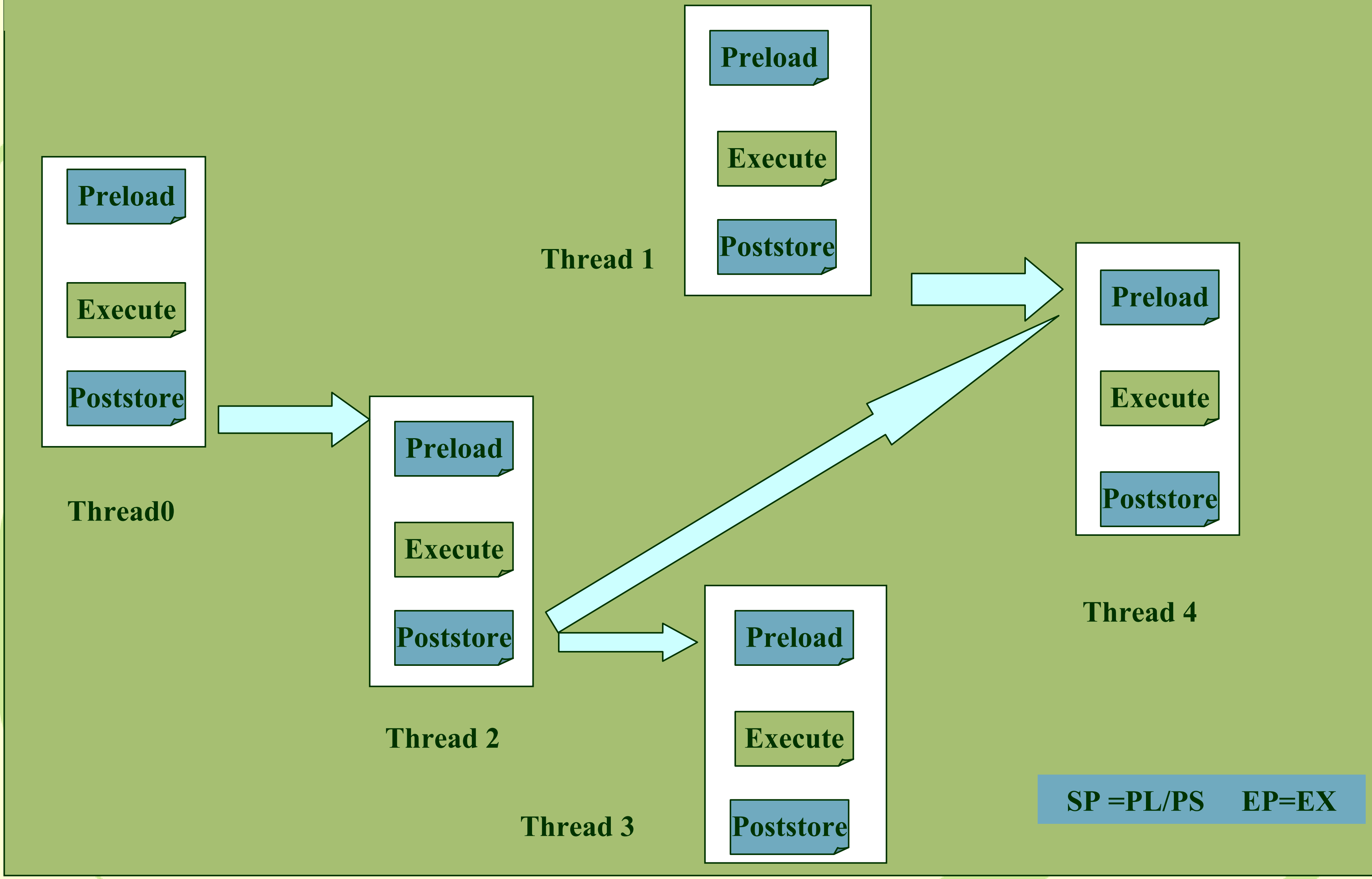
Memory Access Pipeline



Synchronization Processor (SP)



Execution of SDF Programs



Thread0

Preload

Execute

Poststore

Thread 1

Preload

Execute

Poststore

Thread 2

Preload

Execute

Poststore

Thread 3

Preload

Execute

Poststore

Thread 4

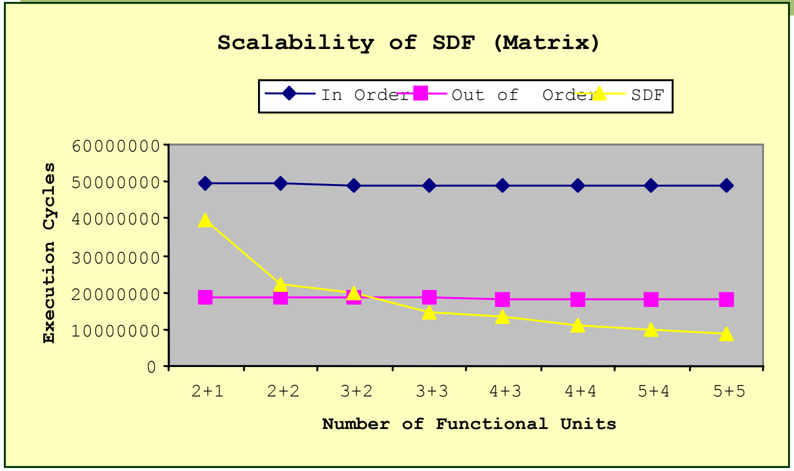
Preload

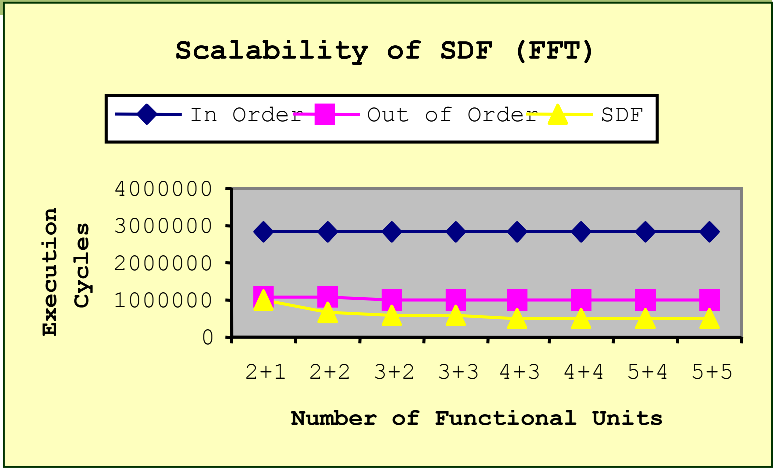
Execute

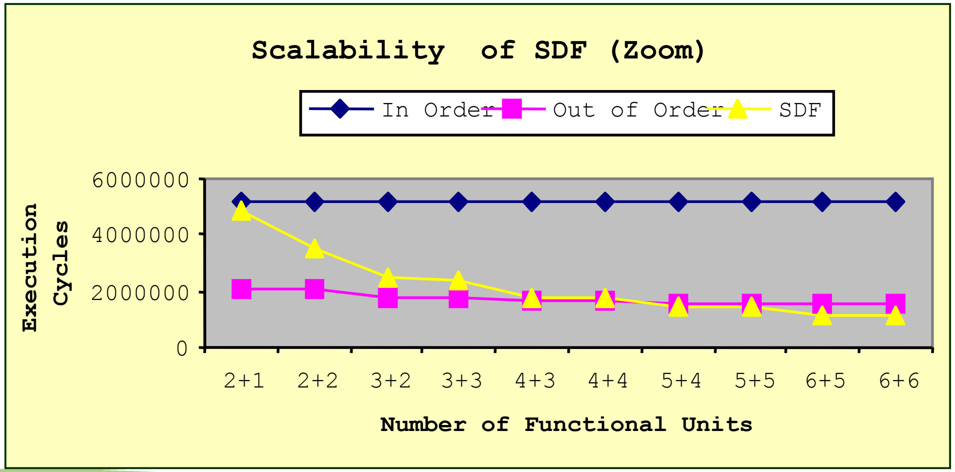
Poststore

SP =PL/PS EP=EX

Some Performance Results

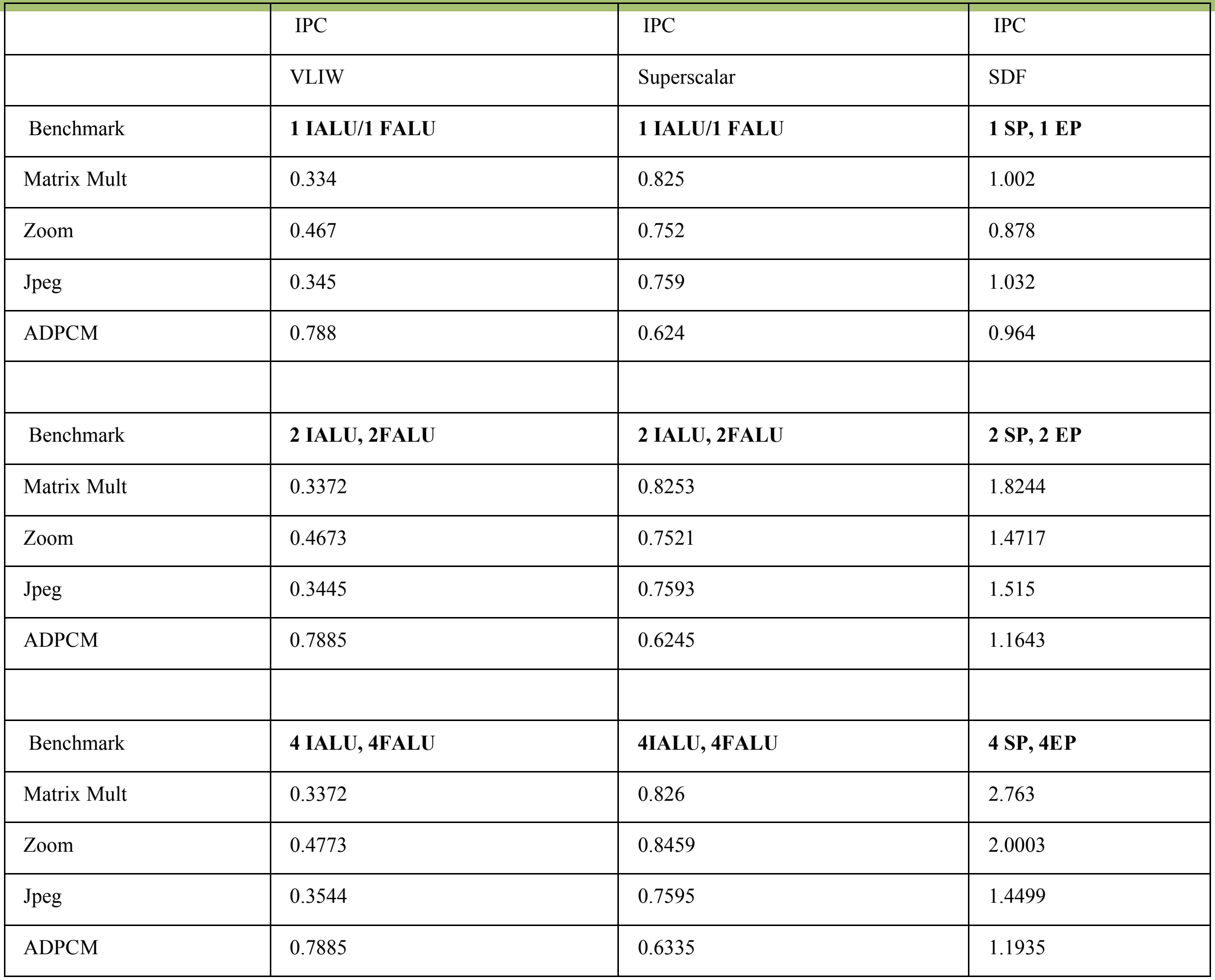






Some Performance Results

SDF vs Supersclar and VLIW



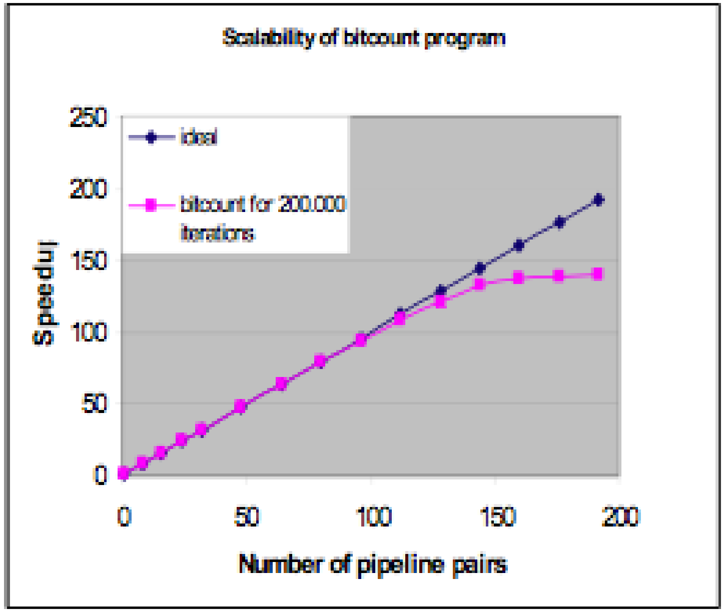
	IPC	IPC	IPC
	VLIW	Superscalar	SDF
Benchmark	**1 IALU/1 FALU**	**1 IALU/1 FALU**	**1 SP, 1 EP**
Matrix Mult	0.334	0.825	1.002
Zoom	0.467	0.752	0.878
Jpeg	0.345	0.759	1.032
ADPCM	0.788	0.624	0.964
Benchmark	**2 IALU, 2FALU**	**2 IALU, 2FALU**	**2 SP, 2 EP**
Matrix Mult	0.3372	0.8253	1.8244
Zoom	0.4673	0.7521	1.4717
Jpeg	0.3445	0.7593	1.515
ADPCM	0.7885	0.6245	1.1643
Benchmark	**4 IALU, 4FALU**	**4IALU, 4FALU**	**4 SP, 4EP**
Matrix Mult	0.3372	0.826	2.763
Zoom	0.4773	0.8459	2.0003
Jpeg	0.3544	0.7595	1.4499
ADPCM	0.7885	0.6335	1.1935

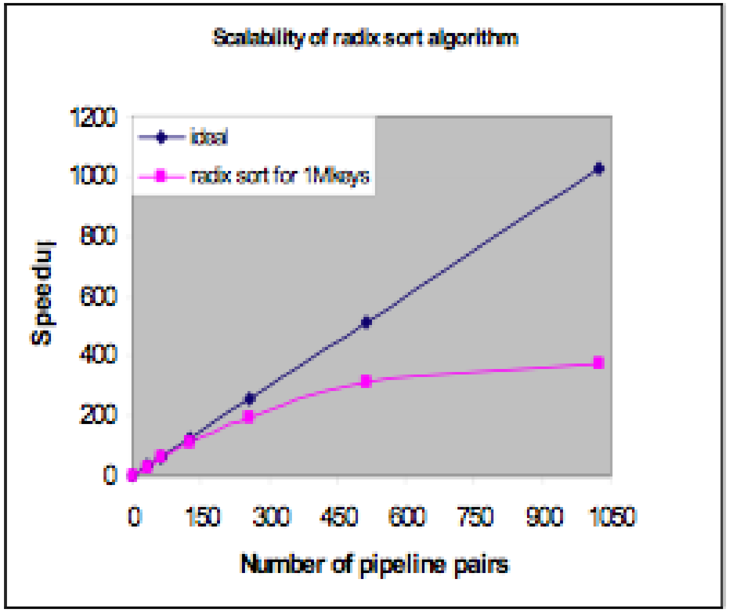
Some Performance Results

SDF vs SMT

	IPC	IPC
	SMT	SDF
Benchmark	**2 threads**	**2 threads**
Matrix Mult	1.9885	1.8586
Zoom	1.8067	1.7689
Jpeg	1.9803	2.1063
ADPCM	1.316	1.9792
Benchmark	**4 threads**	**4 threads**
Matrix Mult	3.6153	3.6711
Zoom	2.513	2.9585
Jpeg	3.6219	3.8641
ADPCM	1.982	2.5065
Benchmark		**6 threads**
Matrix Mult		5.1445
Zoom		4.223
Jpeg		4.7495
ADPCM		3.7397

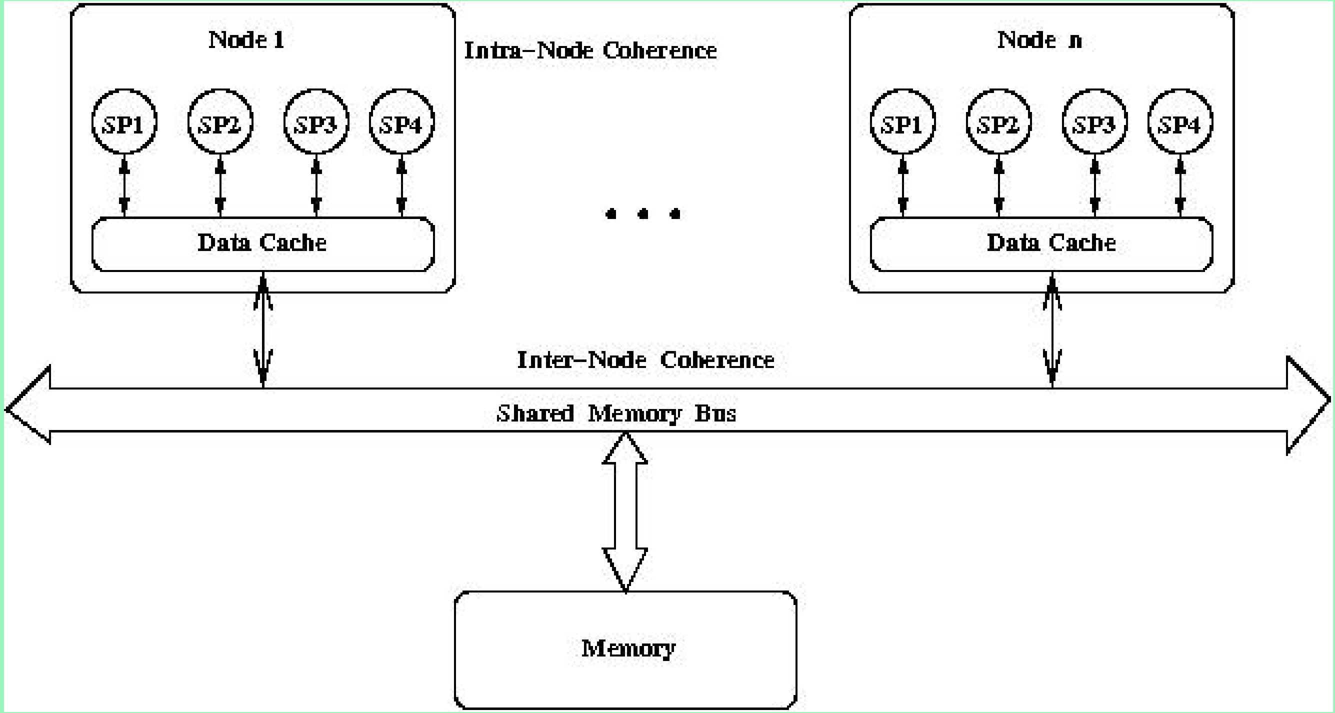
Some Scalability Data





Speculative Thread Execution

Architecture Model



Speculative Thread Execution

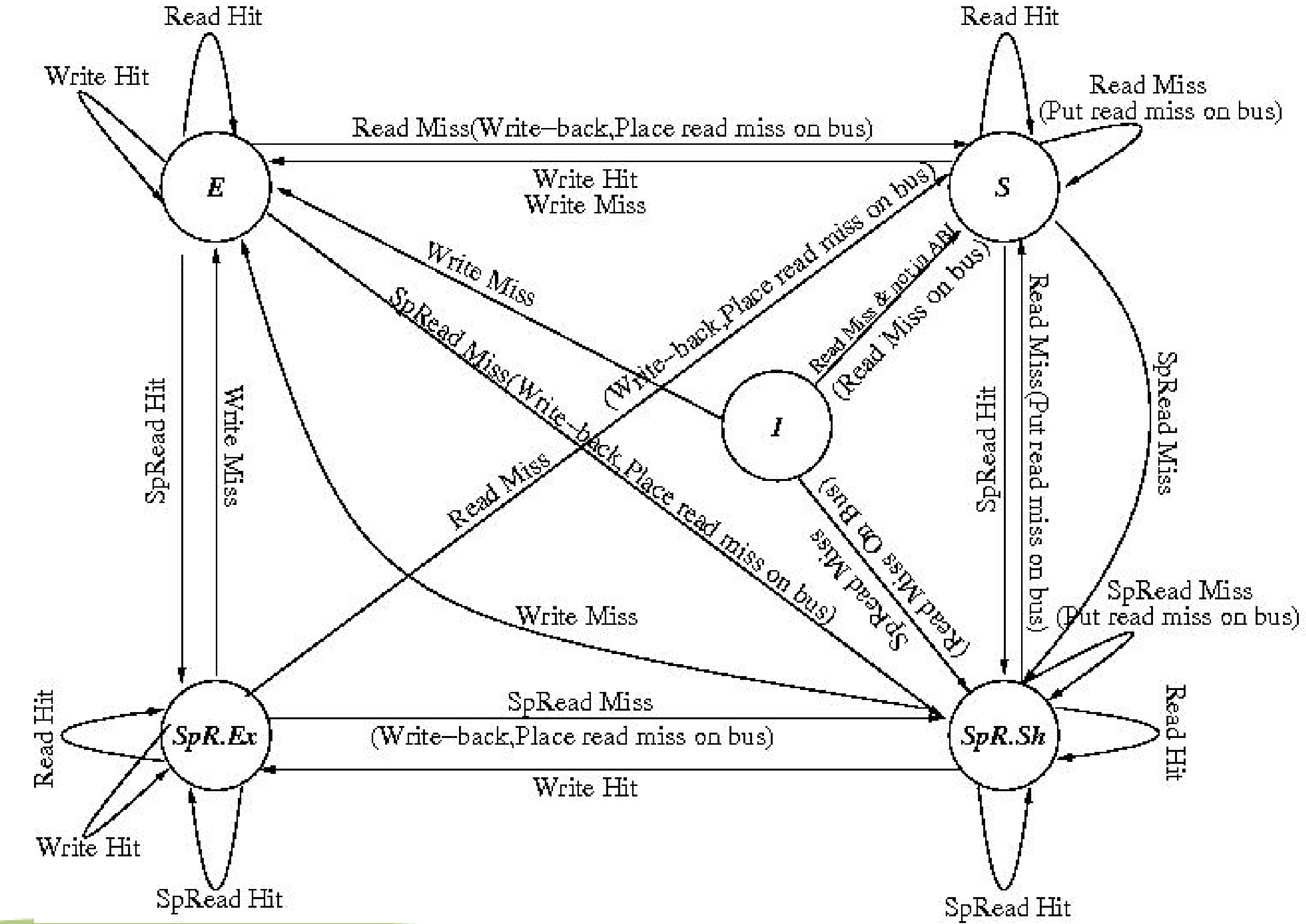
We extend MESI cache coherency protocol

Our states are:

	SpRead	**Valid**	**Dirty(Exclusive)**
I	X	0	X
E/M	0	1	1
S	0	1	0
SpR.Ex	1	1	1
SpR.Sh	1	1	0

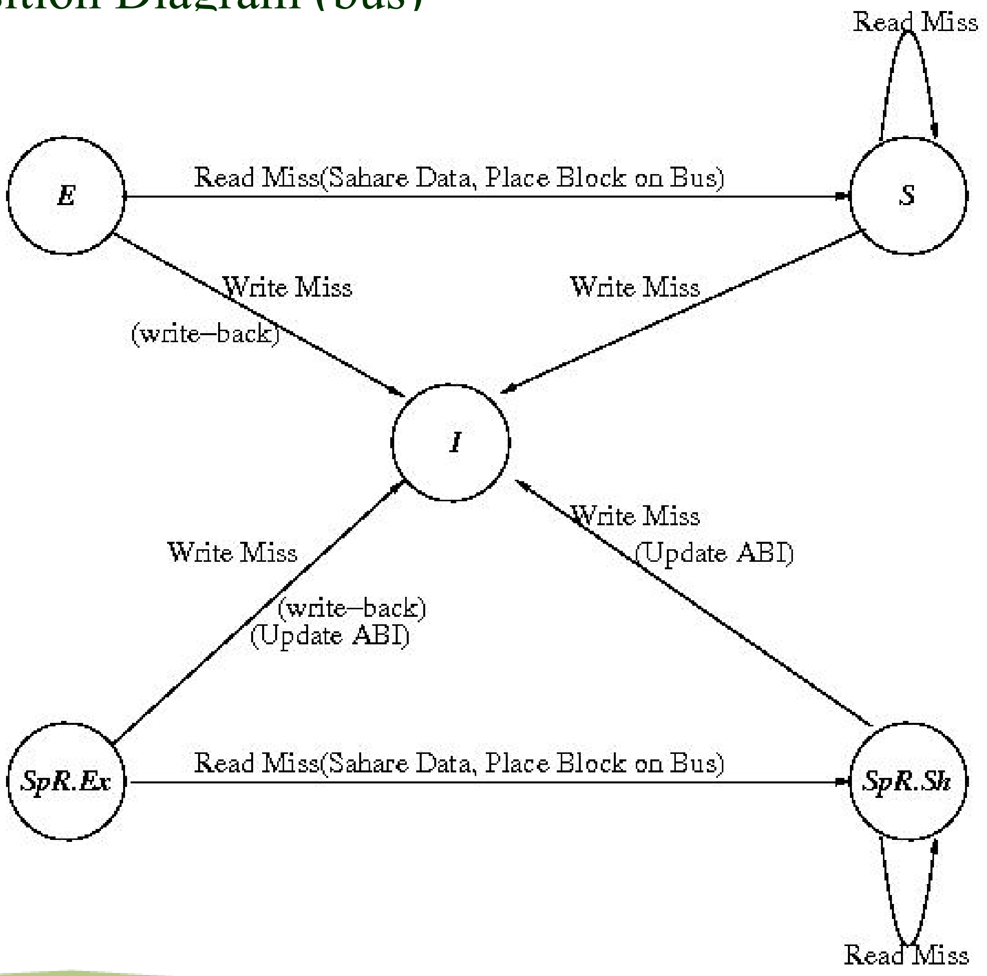
Speculative Thread Execution

- Transition Diagram (processor)



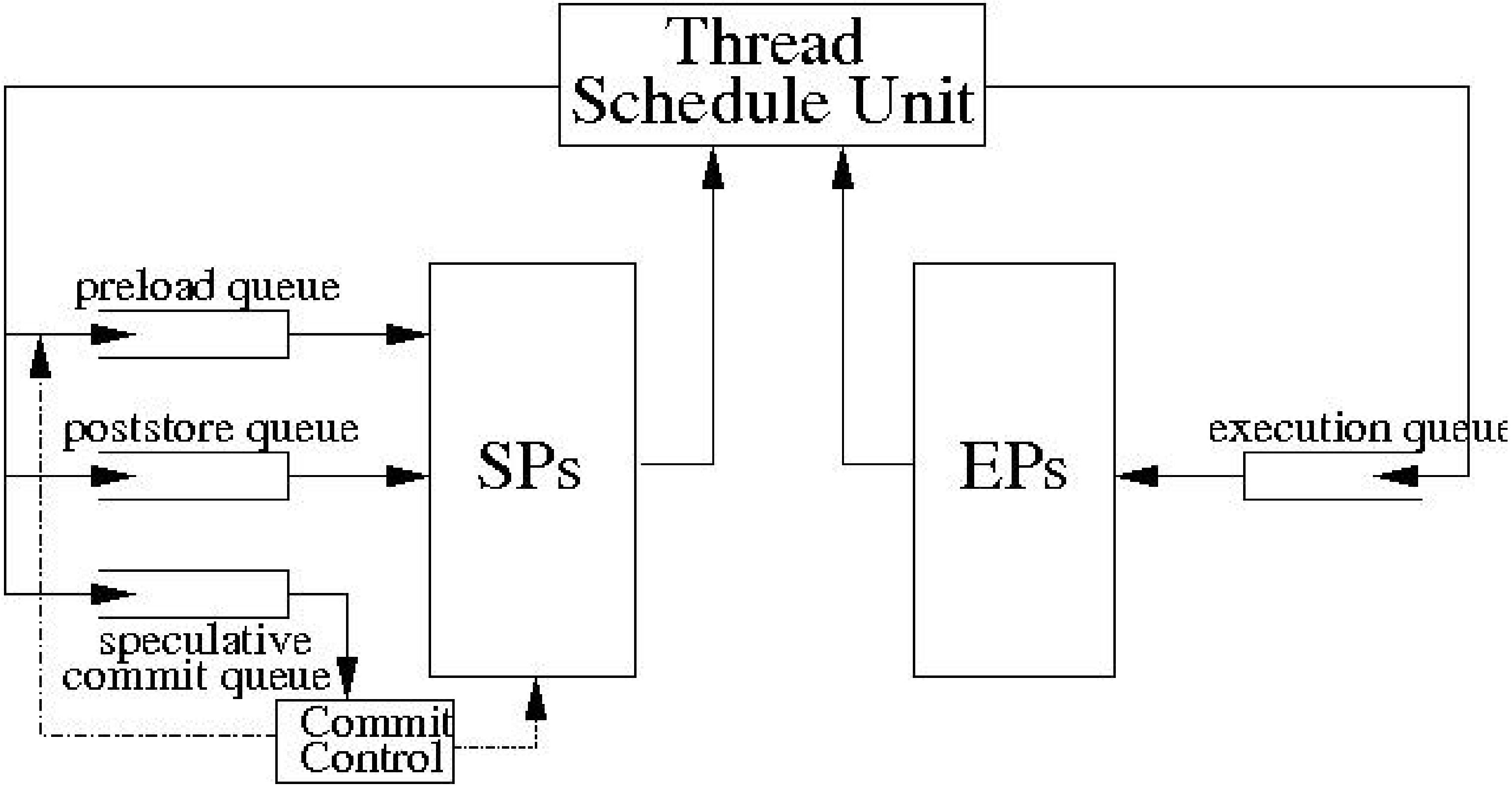
Speculative Thread Execution

- Transition Diagram (bus)



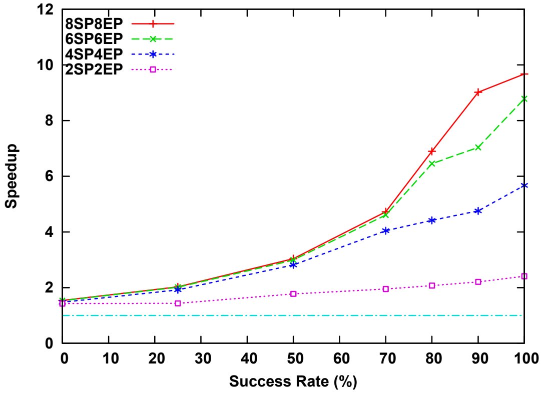
Speculative Thread Execution

- Node structure

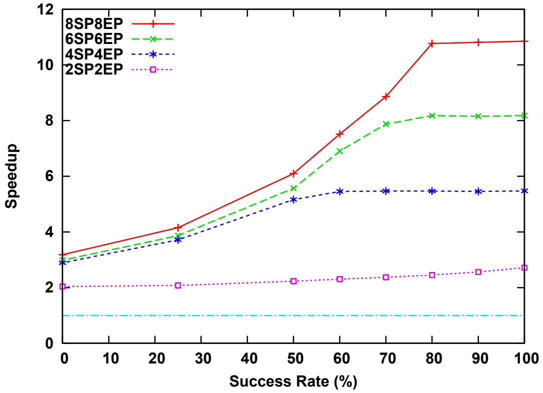


Speculative Thread Execution

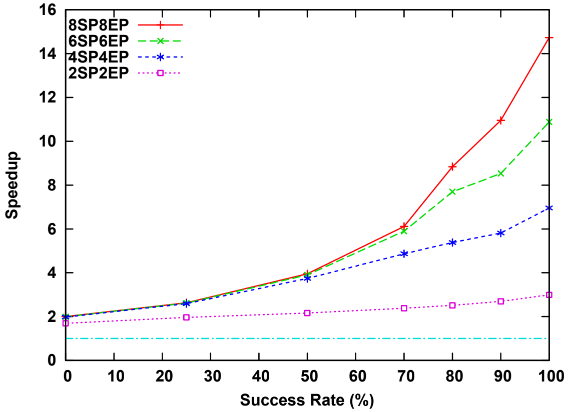
- Synthetic Benchmark Result



a. SP:EP 33%:66%



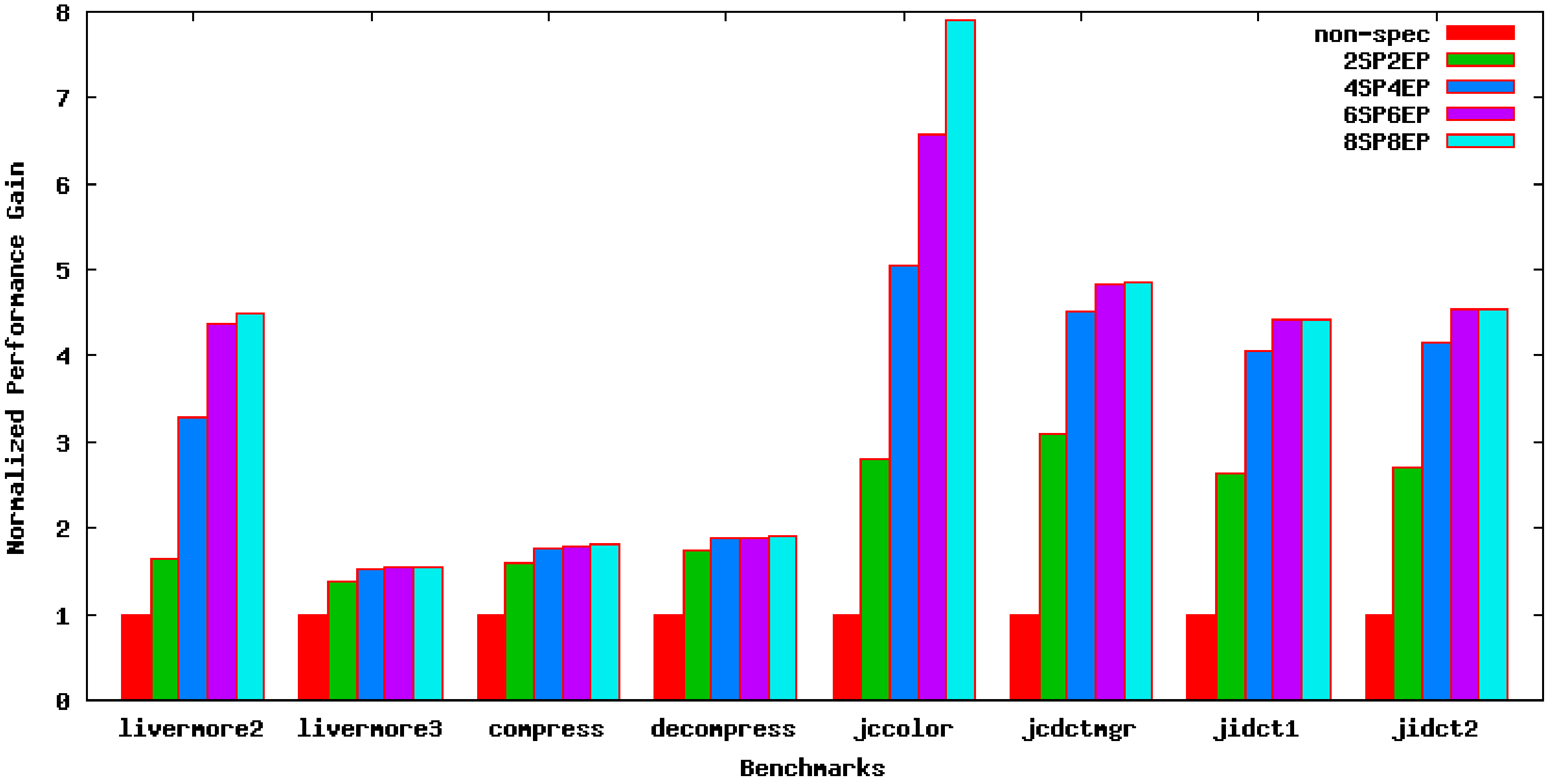
b. SP-EP 66%:33%



c. SP:EP 50%:50%

Speculative Thread Execution

- Real Benchmarks



This slide is deliberately left blank

Array and Scalar Caches

Two types of localities exhibited by programs

Temporal: an item accessed now may be accessed in the near future

Spatial: If an item is accessed now, nearby items are likely to be accessed in the near future

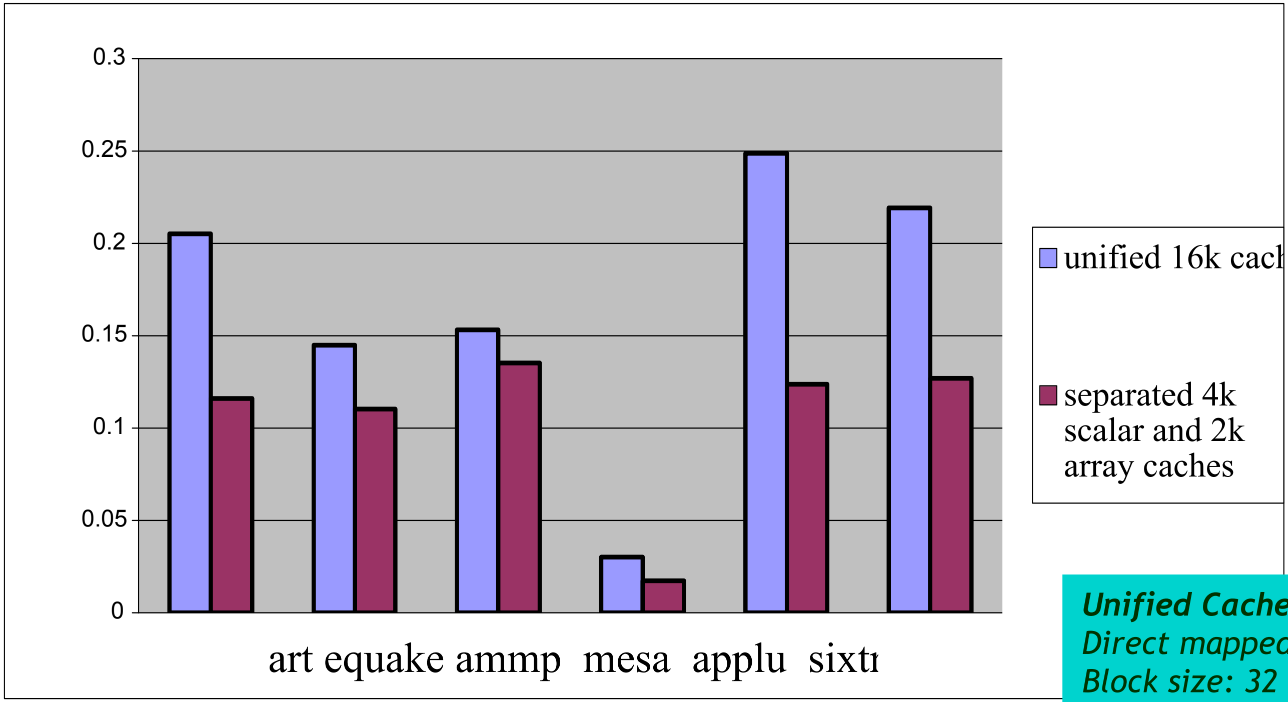
Instructions and Array data exhibit spatial

Scalar data items (such as loop index variable) exhibit temporal

So, we should try to design different types of caches for arrays and scalar data

Array and Scalar Caches

Comparing Split and Unified Cache



Unified Cache
Direct mapped
Block size: 32 bytes
Split Cache
scalar cache: 2-way set associative
32 bytes blocks
array cache: Direct mapped
128 byte blocks

Array and Scalar Caches

Summary of Results with array and scalar caches
using SPEC 2000 Benchmarks

- 43% reduction in Miss rate for benchmark art and mesa
- 24% reduction in Miss rate for benchmark equake
- 12% reduction in Miss rate for benchmark ammp

Array and Scalar Caches

Augmenting scalar cache with victim cache and array cache with prefetch buffers

What is a Victim Cache?

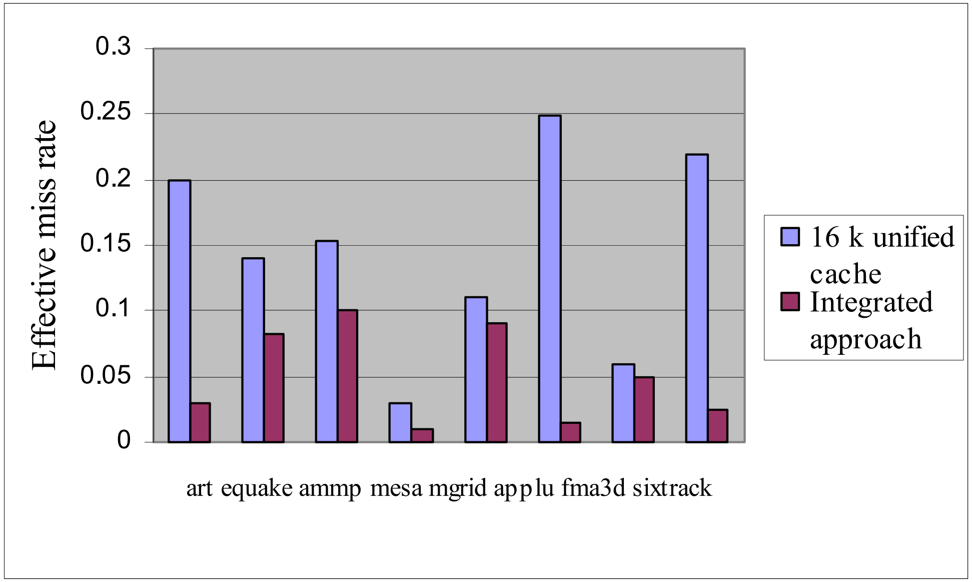
A small fully associative cache to augment L1 direct mapped cache

On a cache miss, the displaced cache entry is moved to victim cache

Minimizes the number of conflict misses

Array and Scalar Caches

Results



Conventional cache configuration: 16k, 32 bytes block, Direct mapped
Scalar cache configuration: 4k, 64 bytes block, Direct mapped
with 8 lined Victim cache
Array cache configuration: 4k, 64 bytes block, Direct mapped with multiple (4)
10 lined stream buffers

Array and Scalar Caches

Embedded applications

Tighter constraints on both functionality and implementation.

Must meet strict timing constraints

Must be designed to function within limited resources such as memory size, available power, and allowable weight

Split caches can address these challenges

Array and Scalar Caches

Reconfigurability

- The performance of a given cache architecture is largely determined by the behavior of the applications
- Manufacturer typically sets the cache architecture as a compromise across several applications
- This leads to conflicts in deciding on total cache size, line size and associativity
- For embedded systems where everything needs to be cost effective, this “one-size-fits-all” design philosophy is not adequate

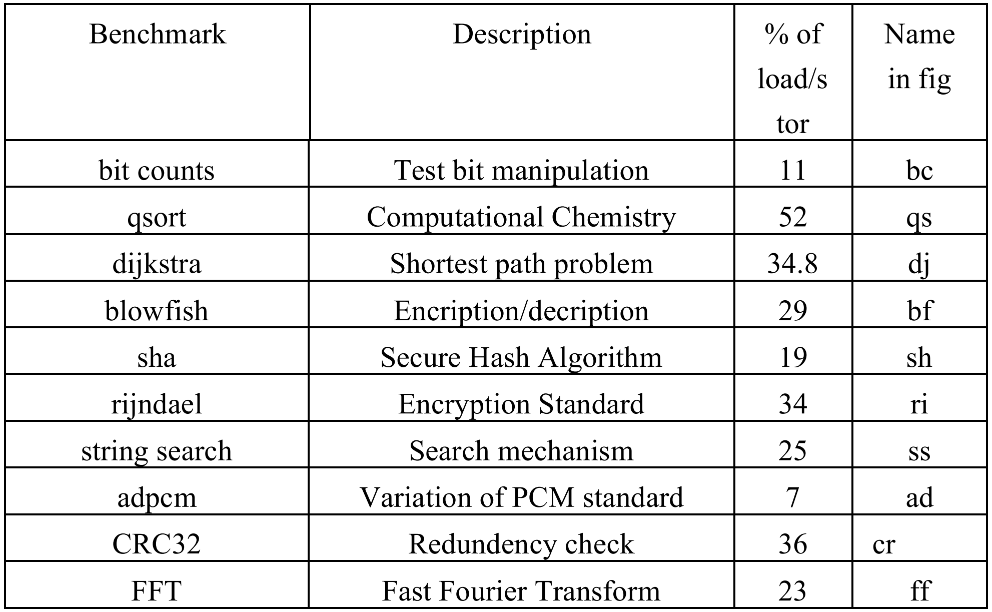
Array and Scalar Caches

Reconfigurability

- **Our goal is to design caches that achieve high performance for embedded applications while remaining both energy and area efficient**
- **We apply reconfigurability to the design of caches to address these conflicting requirements**
- **Emphasize only on cache size**
- **We did not implement reconfigurability for associativity as cache splitting and victim caching solves that problem**

Array and Scalar Caches

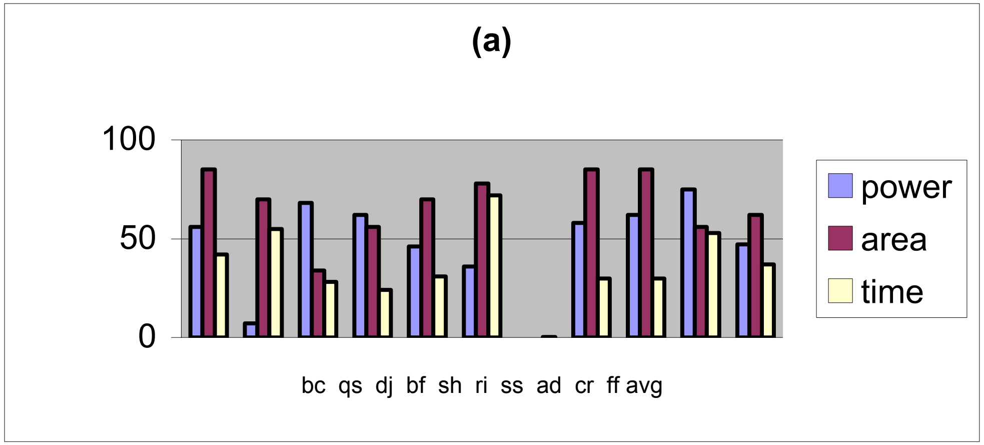
Benchmarks



Benchmark	Description	% of load/stor	Name in fig
bit counts	Test bit manipulation	11	bc
qsort	Computational Chemistry	52	qs
dijkstra	Shortest path problem	34.8	dj
blowfish	Encription/decription	29	bf
sha	Secure Hash Algorithm	19	sh
rijndael	Encryption Standard	34	ri
string search	Search mechanism	25	ss
adpcm	Variation of PCM standard	7	ad
CRC32	Redundency check	36	cr
FFT	Fast Fourier Transform	23	ff

Array and Scalar Caches

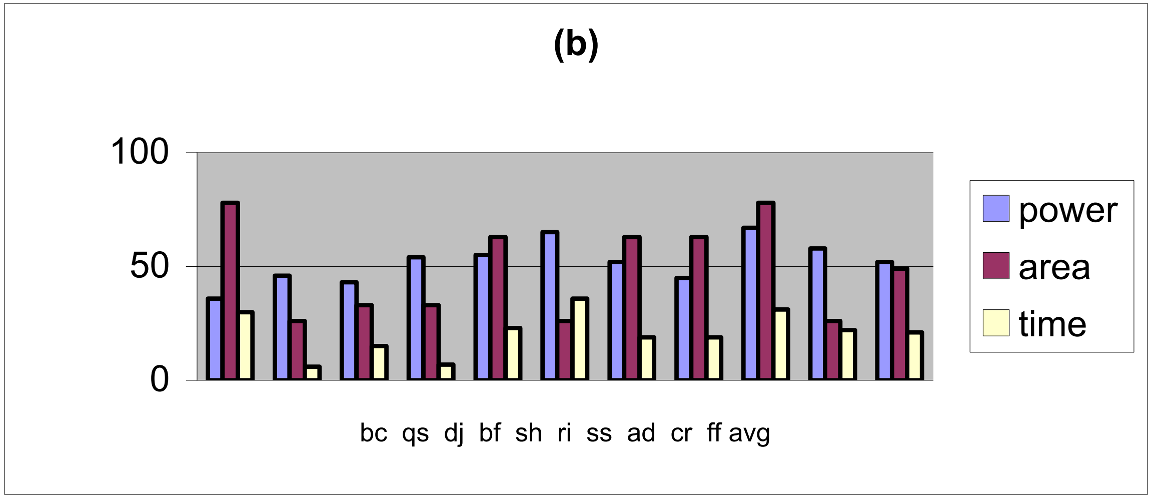
Percentage reduction of power, area and cycle for instruction cache



Conventional cache configuration: 8k, Direct mapped instruction cache,
32k 4-way Unified level 2 cache
Our Instruction cache configuration: Size variable, Direct mapped
with variable sized prefetch buffer

Array and Scalar Caches

Percentage reduction of power, area and cycle for data cache



Conventional cache configuration: 8k, Direct mapped data cache, 32k 4-way Unified level 2 cache

Scalar cache configuration: Size variable, Direct mapped with 2 lined Victim cache

Array cache configuration: Size variable, Direct mapped

Array and Scalar Caches

Cache configurations yielding lowest power, area and cache access time

Benchmark	Instruction cache	Prefetch buffer	Array cache	Scalar cache
bit counts	256 bytes	256 bytes	512 bytes	512 bytes
qsort	256 bytes	512 bytes	1k	4k
dijkstra	1k	2k	512 bytes	4k
blowfish	1k	1k	512 bytes	4k
sha	256 bytes	512 bytes	512 bytes	1k
rijndael	512 bytes	512 bytes	1k	4k
string search	256 bytes	No prefetching	512 bytes	1k
adpcm	256 bytes	256 bytes	1k	512 bytes
CRC32	256 bytes	256 bytes	512 bytes	512 bytes
FFT	1k	1k	1k	4k

Array and Scalar Caches

Summarizing

For instruction cache
85% (average 62%) reduction in cache size
72% (average 37%) reduction in cache access time
75% (average 47%) reduction in energy consumption

For data cache
78% (average 49%) reduction in cache size
36% (average 21%) reduction in cache access time
67% (average 52%) reduction in energy consumption

when compared with an 8KB L-1 instruction cache and an 8KB L-1 unified data cache with a 32KB level-2 cache

This slide is deliberately left blank

Function Reuse

Eliminate redundant function execution

If there are no “side-effects” then a function with the same

Inputs, will generate the same output.

Compiler can help in making sure that if a function has

Side-effects or not

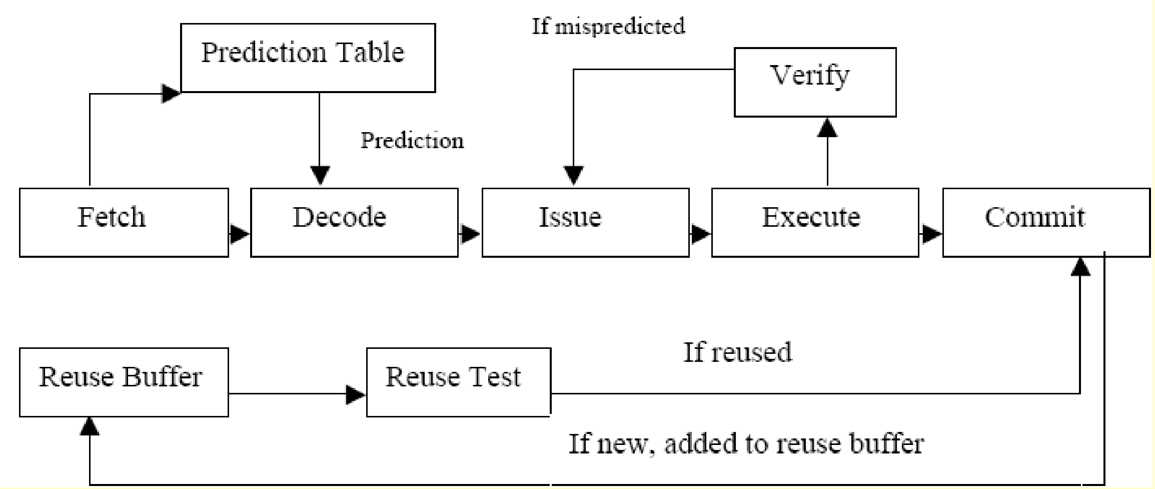
At runtime, when we decode “JAL” instruction we know

that we are calling a function

At that time, look up a table to see if the function is

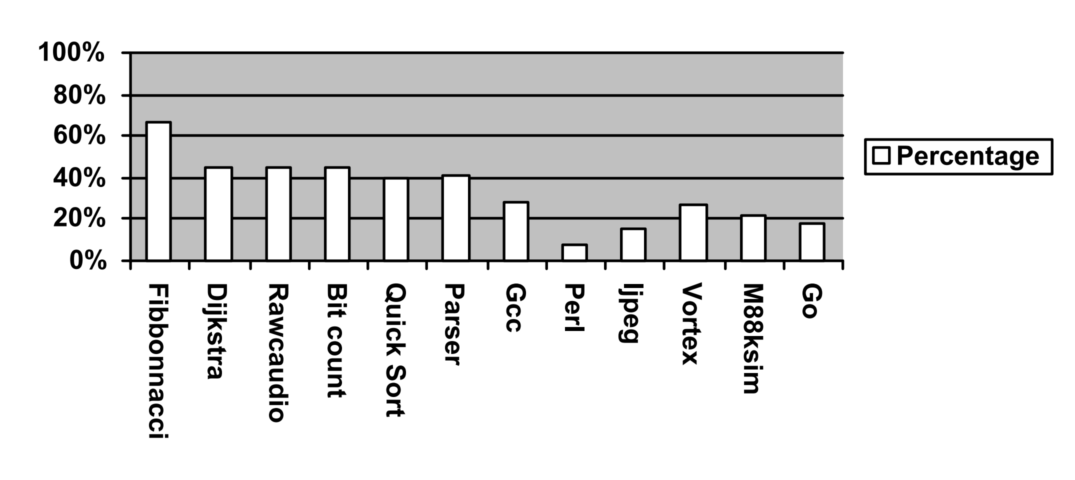
called before with the same arguments

Function Reuse



Function Reuse

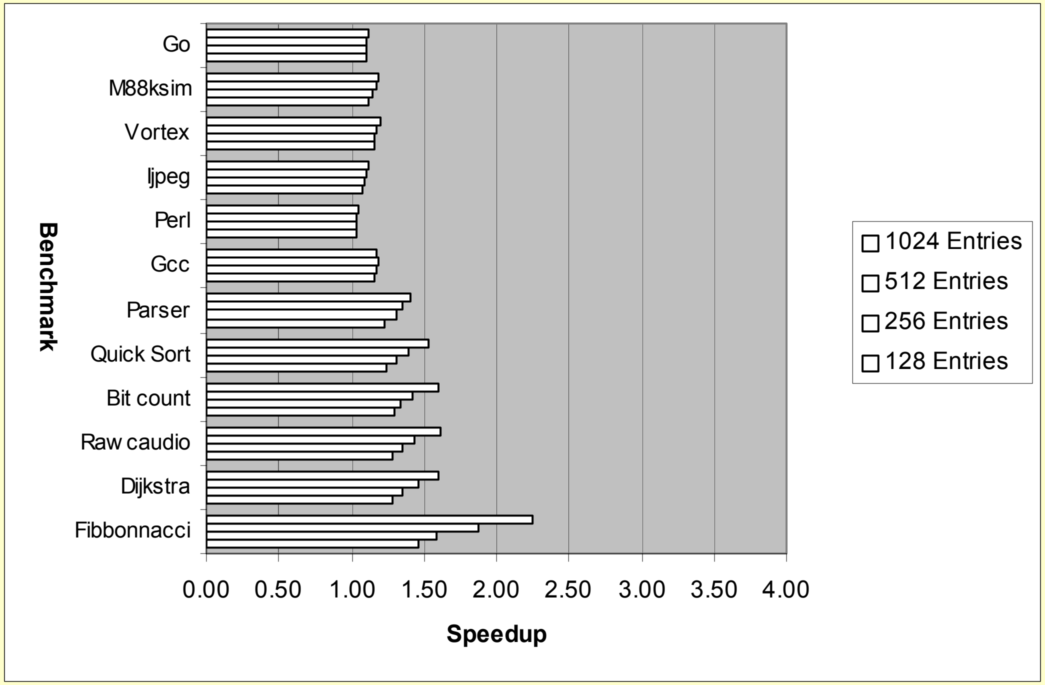
Here we show what percentage of functions are "redundant" and can be be "reused"



Function Reuse

Benchmark	Speedup
Fib	3.23
Dijkstra	1.83
Rawcaudio	1.81
Bit Count	1.81
QuickSort	1.67
Parser	1.71
Gcc	1.40
Perl	1.22
Ijpeg	1.27
Vortex	1.42
M88ksim	1.38
Go	1.37

Function Reuse



For More Information

Visit our website

http://csrl.csci.unt.edu/

You will find our papers and tools

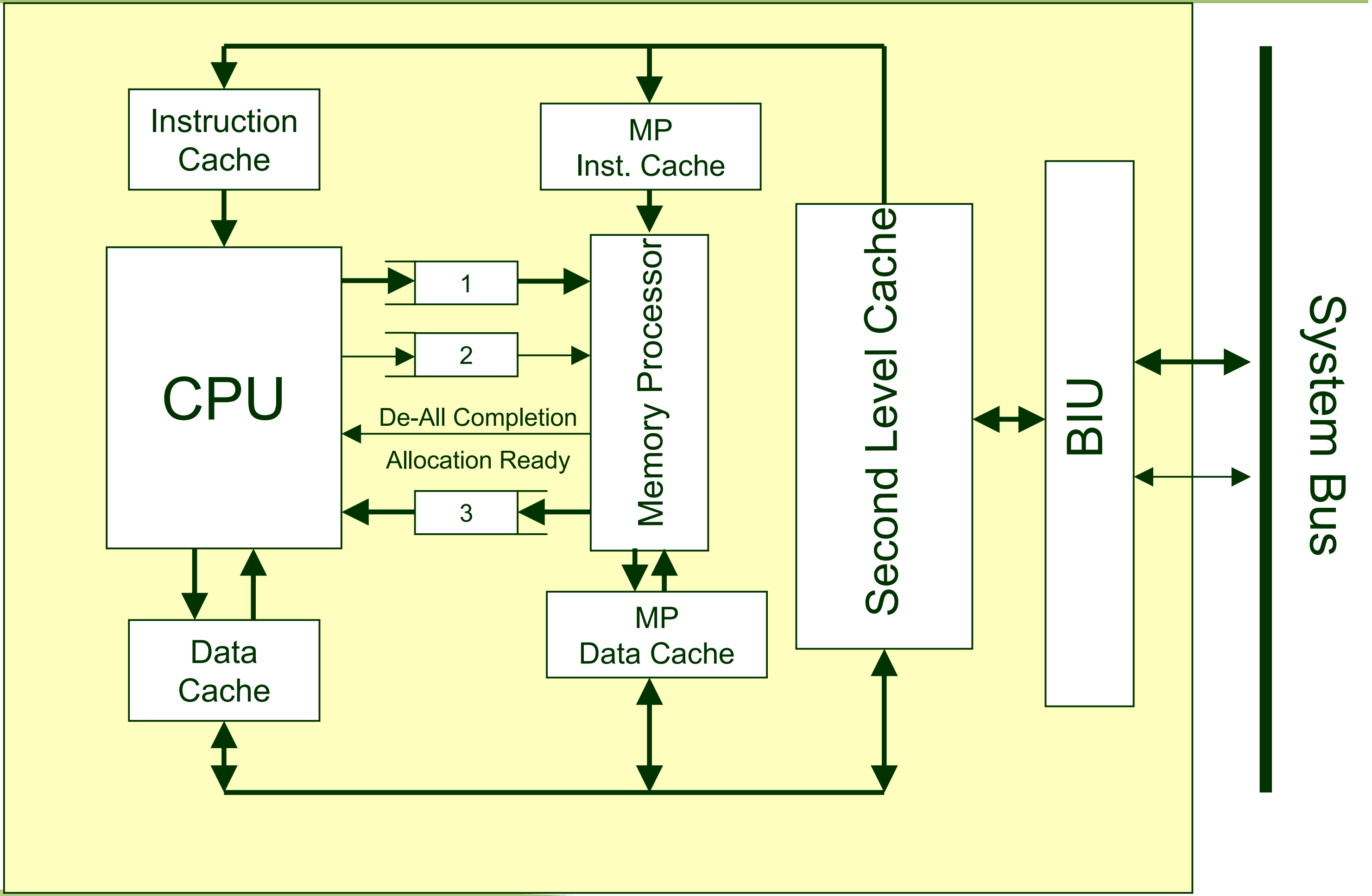
This slide is deliberately left blank

Offloading Memory Management Functions

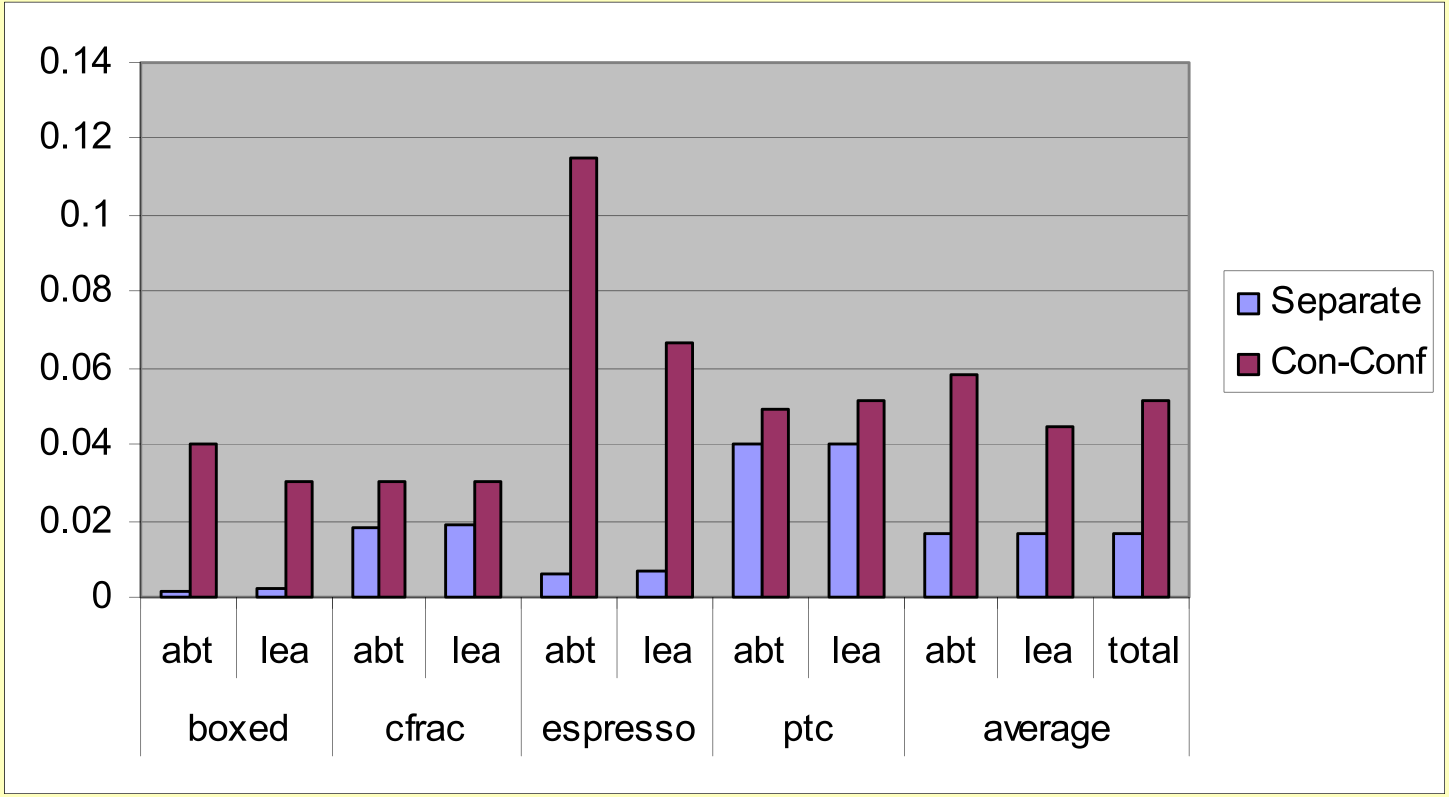
- For object-oriented programming systems, memory management is complex and can consume as much as 40% of total execution time
- Also, if CPU is performing memory management, CPU cache will perform poorly due to switching between user functions and memory management functions

If we have a separate hardware and separate cache for memory management, CPU cache performance can be improved dramatically

Separate Caches With/Without Processor



Empirical Results



Cache Miss Rates – 8 Kbyte Cache with 32 Bytes cache line size

Execution Performance Improvements

Name of Benchmark	% of cycles spent on malloc	Numbers of instructions in conventional Architecture	Numbers of instruction in Separated Hardware Implementation	% Performance increase due to Separate Hardware Implementation	% Performance increase due to fastest separate Hardware Implementation
255.vortex	**0.59**	13020462240	12983022203	**2.81**	**2.90**
164.gzip	**0.04**	4,540,660	4,539,765	**0.031**	**0.0346**
197.parser	**17.37**	2070861403	1616890742	**3.19**	**18.8**
espresso					
Cfrac	**31.17**	599365	364679	**19.03**	**39.99**
bisort	**2.08**	620560644	607122284	**10.03**	**12.76**

Performance in Multithreaded Systems

	Instruction Reduction	**2T speedup**	**3T speedup**	**4T speedup**
Cfrac	23.3%	19.3%	25.26%	30.08%
espresso	6.07%	9.09%	8.35%	6.27%
perlbmk	9.05%	14.03%	18.07%	18.35%
parser	16.88%	17.38%	16.93%	18.61%
Ave.	13.83%	14.95%	17.15%	18.33%

All threads executing the same function

Performance in Multithreaded Systems

	Ave. #of instruction Reduction	Ave. Performance Improvement
2T	11.52%	14.67%
3T	12.41%	20.21%
4T	14.67%	19.60%

Each thread executing a different task

Hybrid Implementations

Key cycle intensive portions implemented in
hardware

For PHK, the bit map for each page in hardware
page directories in software
needed only 20K gates
produced between 2-11% performance