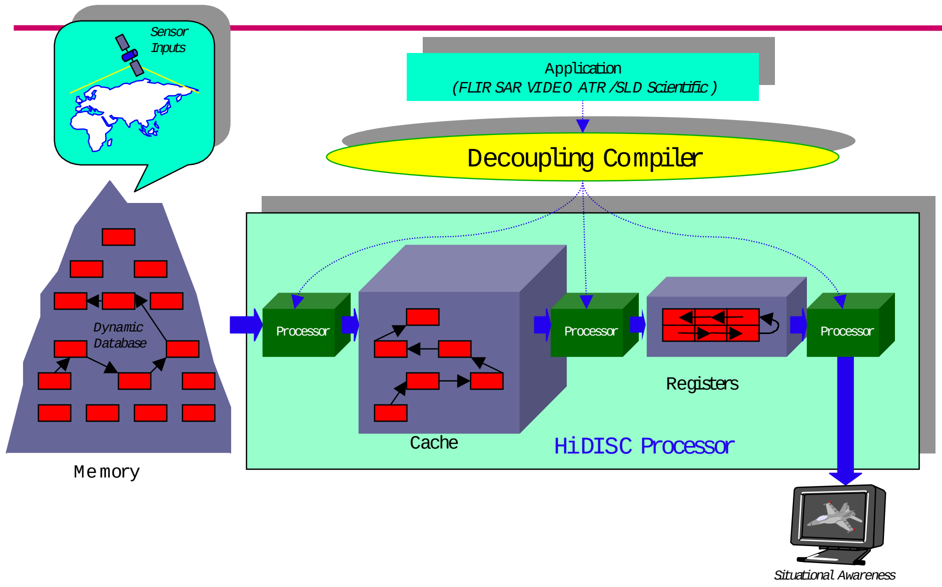
HiDISC: A Decoupled Architecture for Applications in Data Intensive Computing

Alvin M. Despain, Jean-Luc Gaudiot,
Manil Makhija and Wonwoo Ro
University of Southern California
http://www-pdpc.usc.edu

19 May 2000

HiDISC: Hierarchical Decoupled Instruction Set Computer

USC UNIVERSITY OF SOUTHERN CALIFORNIA



New Ideas

• A dedicated processor for each level of the memory hierarchy

• Explicitly manage each level of the memory hierarchy using instructions generated by the compiler

• Hide memory latency by converting data access predictability to data access locality

• Exploit instruction-level parallelism without extensive scheduling hardware

• Zero overhead prefetches for maximal computation throughput

Impact

• 2x speedup for scientific benchmarks with large data sets over an in-order superscalar processor

• 7.4x speedup for matrix multiply over an in-order issue superscalar processor

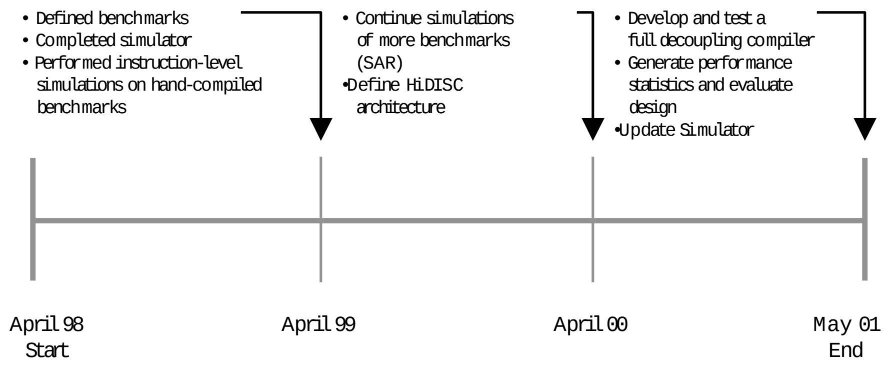
• 2.6x speedup for matrix decomposition/substitution over an in-order issue superscalar processor

• Reduced memory latency for systems that have high memory bandwidths (e.g. PIMs, RAMBUS)

• Allows the compiler to solve indexing functions for irregular applications

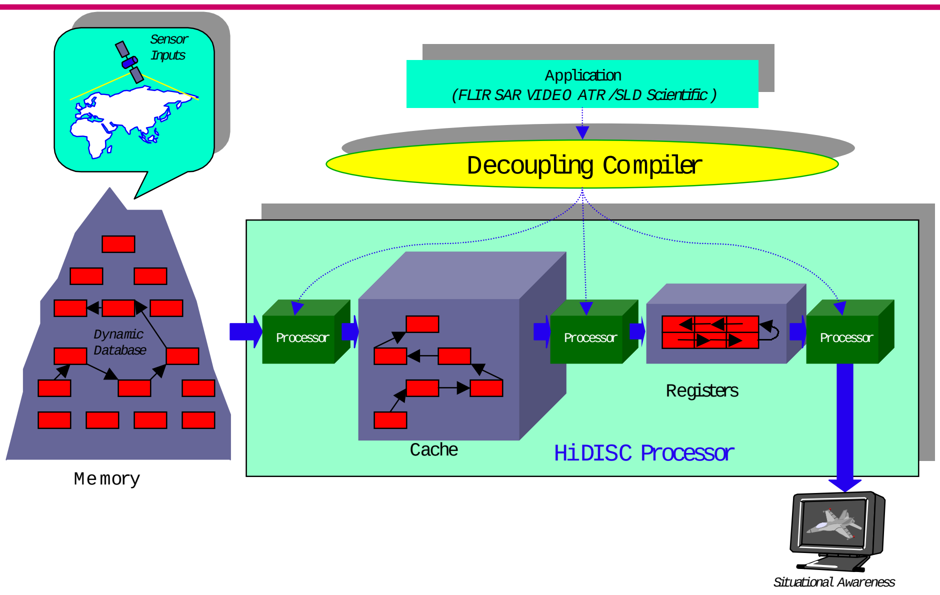
• Reduced system cost for high-throughput scientific codes

Schedule


• Defined benchmarks
• Completed simulator
• Performed instruction-level simulations on hand-compiled benchmarks
• Continue simulations of more benchmarks (SAR)
•Define HiDISC architecture
• Develop and test a full decoupling compiler
• Generate performance statistics and evaluate design
•Update Simulator
April 98 Start
April 99
April 00
May 01 End

University of Southern California, Alvin M. Despain, Jean-Luc Gaudiot, Manil Makhija and Wonwoo Ro

HiDISC: Hierarchical Decoupled Instruction Set Computer



Technological Trend: Memory latency is getting longer relative to microprocessor speed (40% per year)

Problem: Some SPEC benchmarks spend more than half of their time stalling [Lebeck and Wood 1994]

Domain: benchmarks with *large data sets:* symbolic, signal processing and scientific programs

Present Solutions: Multithreading (Homogenous), Larger Caches, Prefetching, Software Multithreading

Present Solutions

USC UNIVERSITY OF SOUTHERN CALIFORNIA

Solution	Limitations
Larger Caches	– Slow – Works well only if working set fits cache and there is temporal locality.
Hardware Prefetching	– Cannot be tailored for each application – Behavior based on past and present execution-time behavior
Software Prefetching	– Ensure overheads of prefetching do not outweigh the benefits > conservative prefetching – Adaptive software prefetching is required to change prefetch distance during run-time – Hard to insert prefetches for irregular access patterns
Multithreading	– Solves the throughput problem, not the memory latency problem

The HiDISC Approach

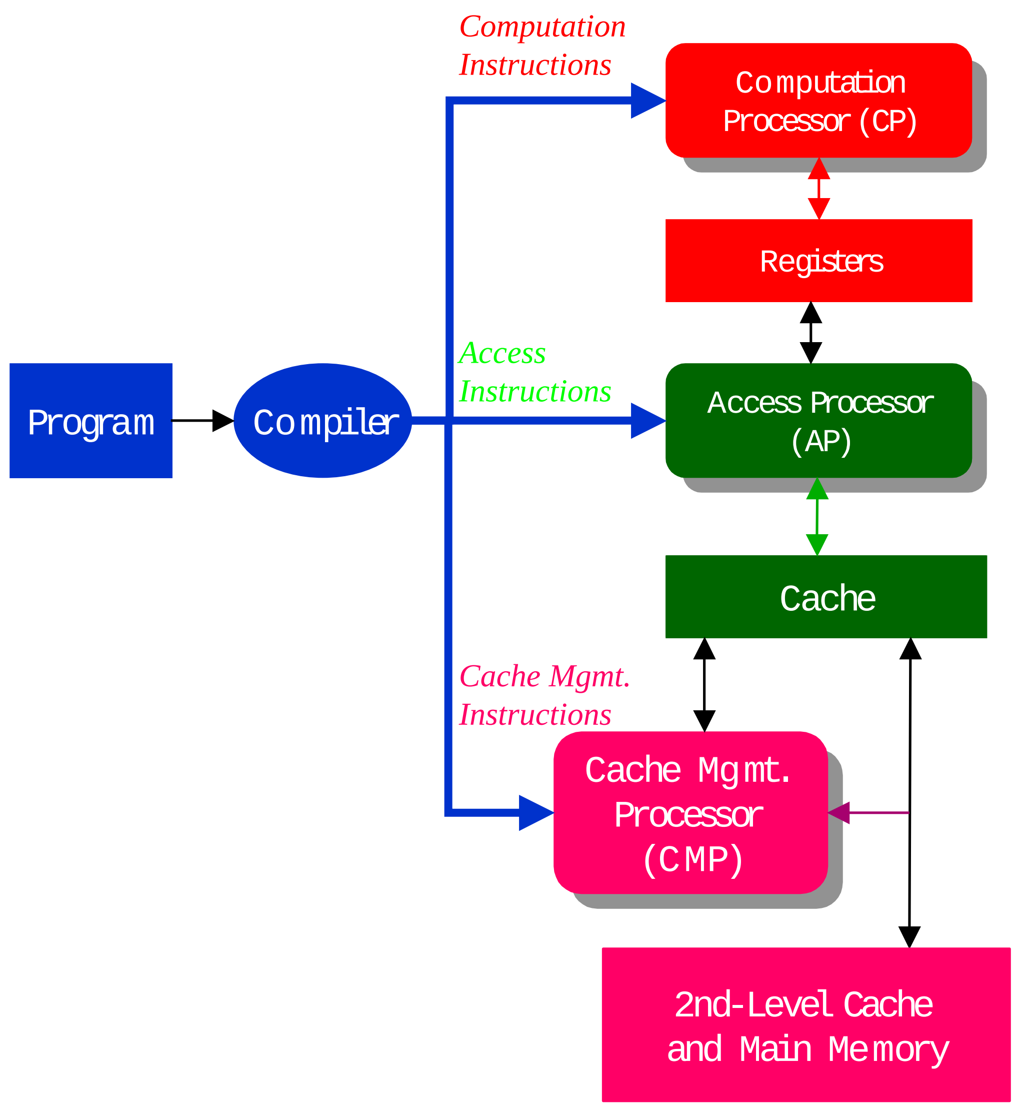
Observation:

- Software prefetching impacts compute performance
- PIMs and RAMBUS offer a high-bandwidth memory system
  - useful for speculative prefetching

Approach:

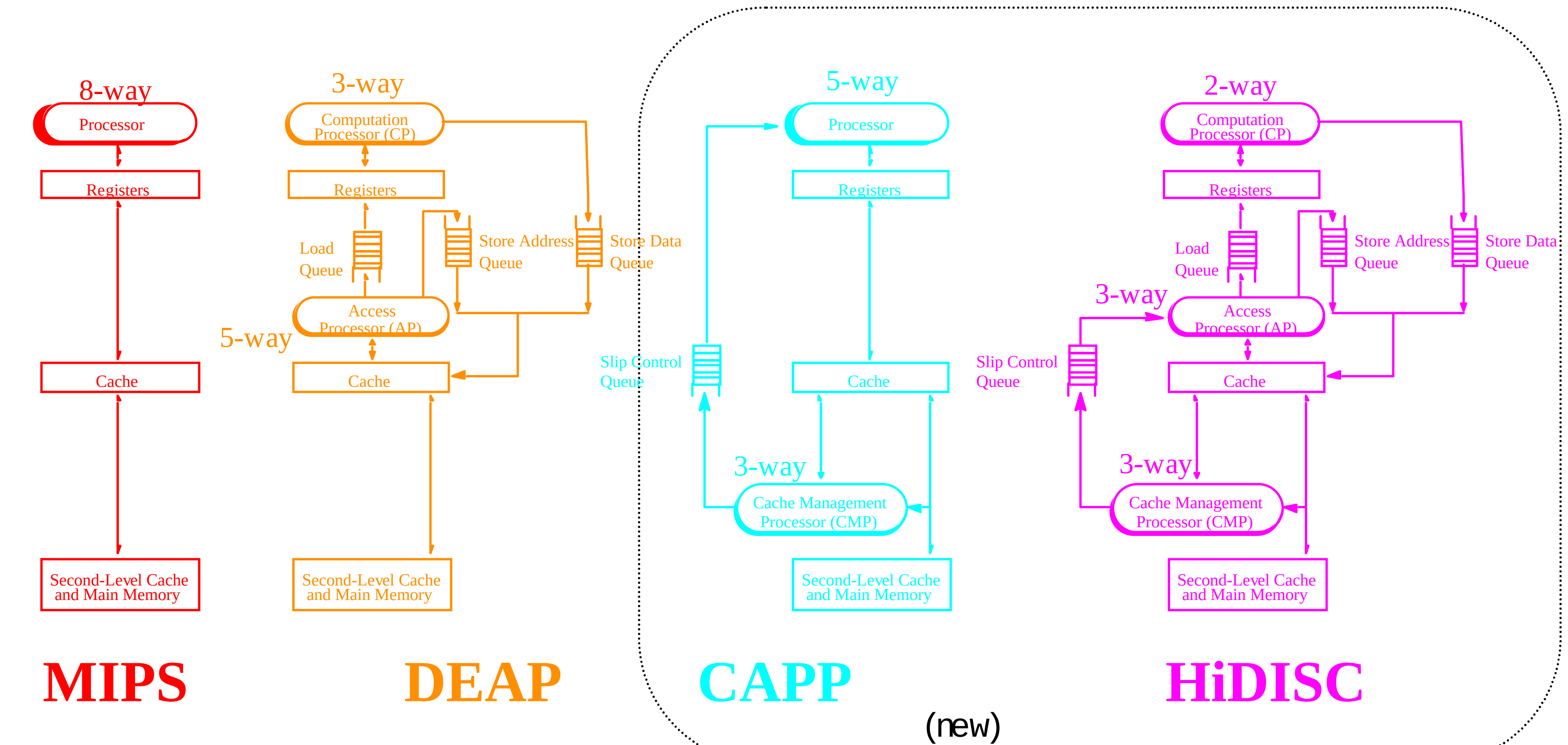
- Add a processor to manage prefetching
  -> hide overhead
- Compiler explicitly manages the memory hierarchy
- Prefetch distance adapts to the program runtime behavior

What's HiDISC



- A dedicated processor for each level of the memory hierarchy
- Explicitly manage each level of the memory hierarchy using instructions generated by the compiler
- Hide memory latency by converting data access predictability to data access locality *(Just in Time Fetch)*
- Exploit instruction-level parallelism without extensive scheduling hardware
- Zero overhead prefetches for maximal computation throughput

Decoupled Architectures



(Conventional) (Decoupled) (Decoupled) (Decoupled)

Slip Control Queue

- The Slip Control Queue (SCQ) adapts dynamically

```
if (prefetch_buffer_full ())
        Don't change size of SCQ;
else if ((2 * late_prefetches) > useful_prefetches)
        Increase size of SCQ;
else
        Decrease size of SCQ;
```

  – *Late prefetches* = prefetched data arrived after load had been issued
  – *Useful prefetches* = prefetched data arrived before load had been issued

Decoupling Programs for HiDISC-3 (Discrete Convolution - Inner Loop)

```
for (j = 0; j < i; ++j)
        y[i]=y[i]+(x[j]*h[i-j-1]);
```

Inner Loop Convolution

```
while (not end of loop)
        y = y + (x * h);
send y to SDQ
```

Computation Processor Code

```
for (j = 0; j < i; ++j) {
        load (x[j]);
        load (h[i-j-1]);
        GET_SCQ;
}
send (EOD token)
send address of y[i] to SAQ
```

Access Processor Code

```
for (j = 0; j < i; ++j) {
        prefetch (x[j]);
        prefetch (h[i-j-1];
        PUT_SCQ;
}
```

Cache Management Code

Benchmarks

USC UNIVERSITY OF SOUTHERN CALIFORNIA

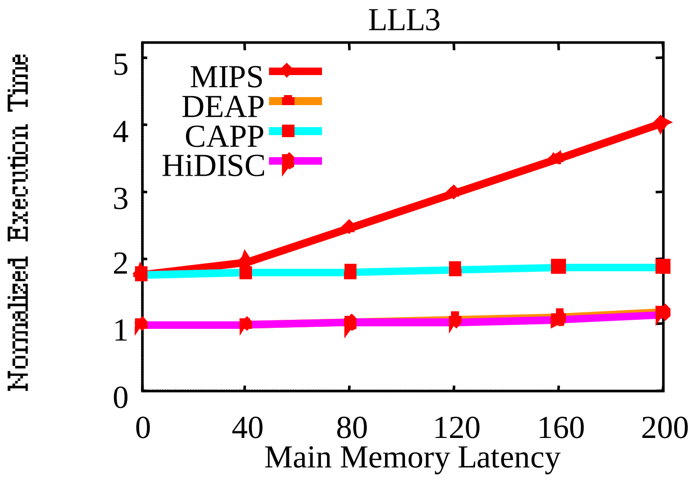
Benchmark	Source of Benchmark	Lines of Source Code	Description	Data Set Size
LLL1	Livermore Loops [45]	20	1024-element arrays, 100 iterations	24 KB
LLL2	Livermore Loops	24	1024-element arrays, 100 iterations	16 KB
LLL3	Livermore Loops	18	1024-element arrays, 100 iterations	16 KB
LLL4	Livermore Loops	25	1024-element arrays, 100 iterations	16 KB
LLL5	Livermore Loops	17	1024-element arrays, 100 iterations	24 KB
Tomcatv	SPECfp95 [68]	190	33x33-element matrices, 5 iterations	<64 KB
MXM	NAS kernels [5]	113	Unrolled matrix multiply, 2 iterations	448 KB
CHOLSKY	NAS kernels	156	Cholesky matrix decomposition	724 KB
VPENTA	NAS kernels	199	Invert three pentadiagonals simultaneously	128 KB
Qsort	Quicksort sorting algorithm [14]	58	Quicksort	128 KB

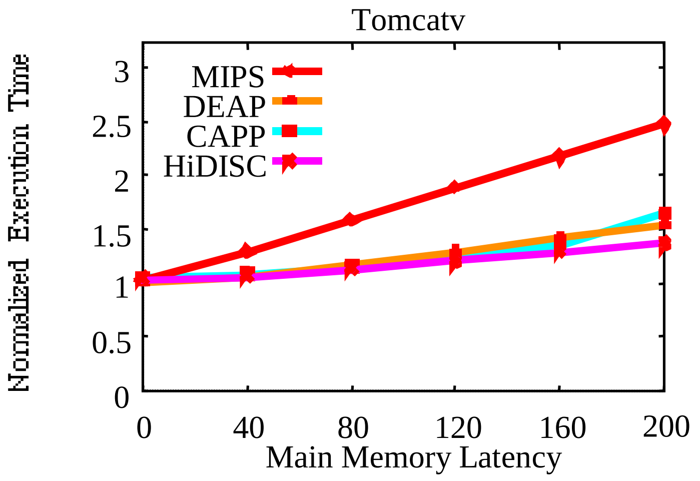
Simulation Parameters

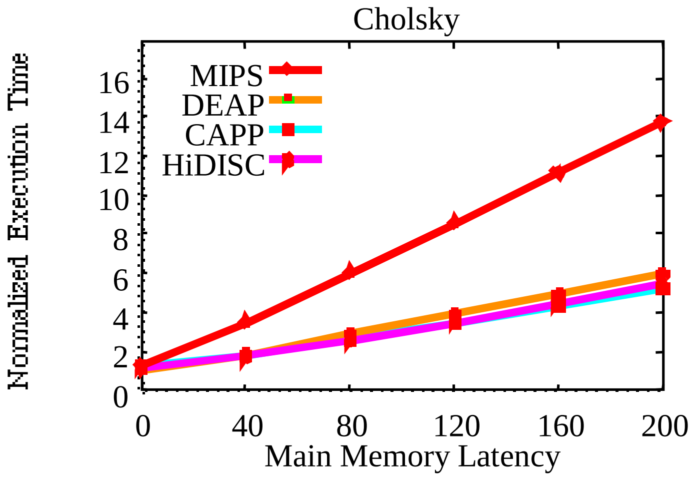
Parameter	Value	Parameter	Value
FLC Size	4 KB	SLC Size	16 KB
FLC Associativity	2	SLC Associativity	2
FLC Block Size	32 B	SLC Block Size	32 B
Memory Latency	varied	Memory Contention Time	varied
Victim Cache Size	32 Entries	Prefetch Buffer Size	8 entries
Load Queue Size	128	Store Address Queue Size	128
Store Data Queue Size	128	Total issue width	8

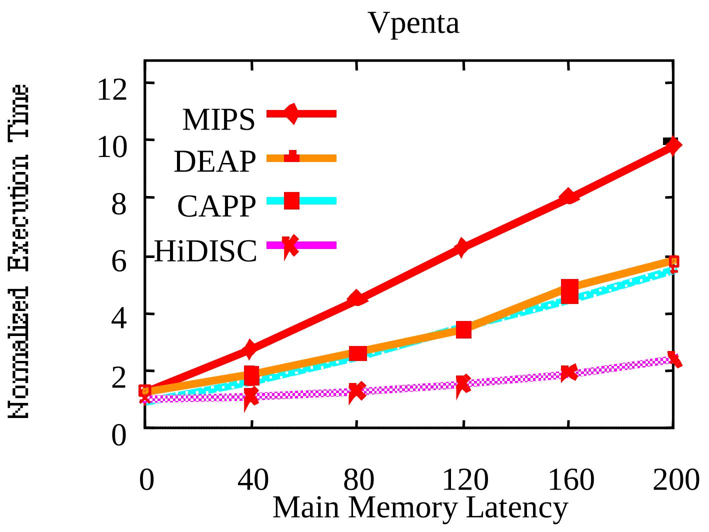
Simulation Results

USC UNIVERSITY OF SOUTHERN CALIFORNIA





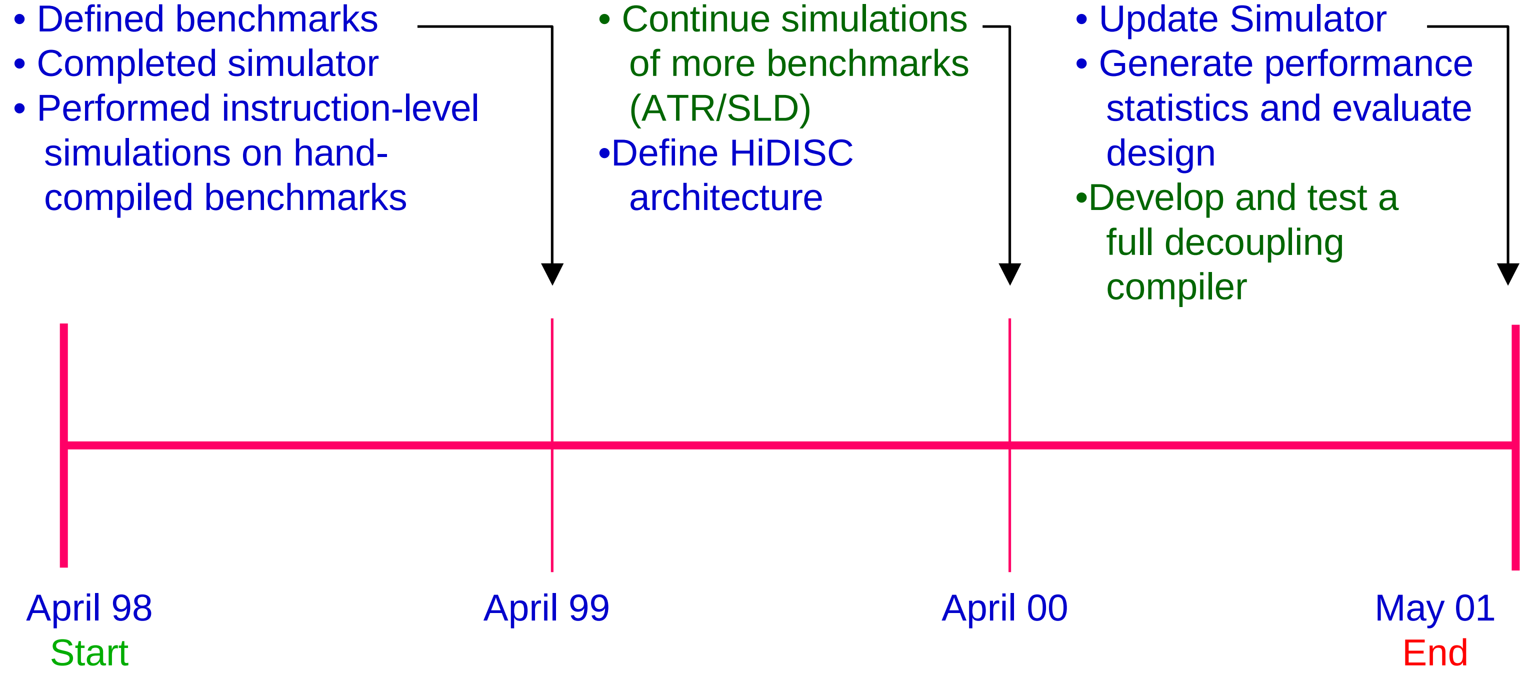




Our Results: Impact

- 2x speedup for scientific benchmarks with large data sets over an in-order superscalar processor
- 7.4x speedup for matrix multiply (MXM) over an in-order issue superscalar processor - (similar operations are used in ATR/SLD)
- 2.6x speedup for matrix decomposition/substitution (Cholsky) over an in-order issue superscalar processor
- Reduced memory latency for systems that have high memory bandwidths (e.g. PIMs, RAMBUS)
- Allows the compiler to solve indexing functions for irregular applications
- Reduced system cost for high-throughput scientific codes

Schedule



- Defined benchmarks
- Completed simulator
- Performed instruction-level simulations on hand-compiled benchmarks

- Continue simulations of more benchmarks (ATR/SLD)
- Define HiDISC architecture

- Update Simulator
- Generate performance statistics and evaluate design
- Develop and test a full decoupling compiler

Summary

USC
UNIVERSITY
OF SOUTHERN
CALIFORNIA

- A processor for each level of the memory hierarchy
- Adaptive memory hierarchy management
- Reduces memory latency for systems with high memory bandwidths (PIMs, RAMBUS)
- 2x speedup for scientific benchmarks
- 3x speedup for matrix decomposition / substitution (Cholesky)
- 7x speedup for matrix multiply (MXM) (similar results expected for ATR / SLD)

BEYOND HiDISC

- Distributed Processing
  - Sensors
  - Data I / O (disk farms)
  - Multiprocessors
- Multiprocessing
  - Mc Fisc-on-a-chip
  - SMT / MT / I-structures
  - VLSI layout / performance tradeoffs
- Applications
  - Compute / database search and retrieval

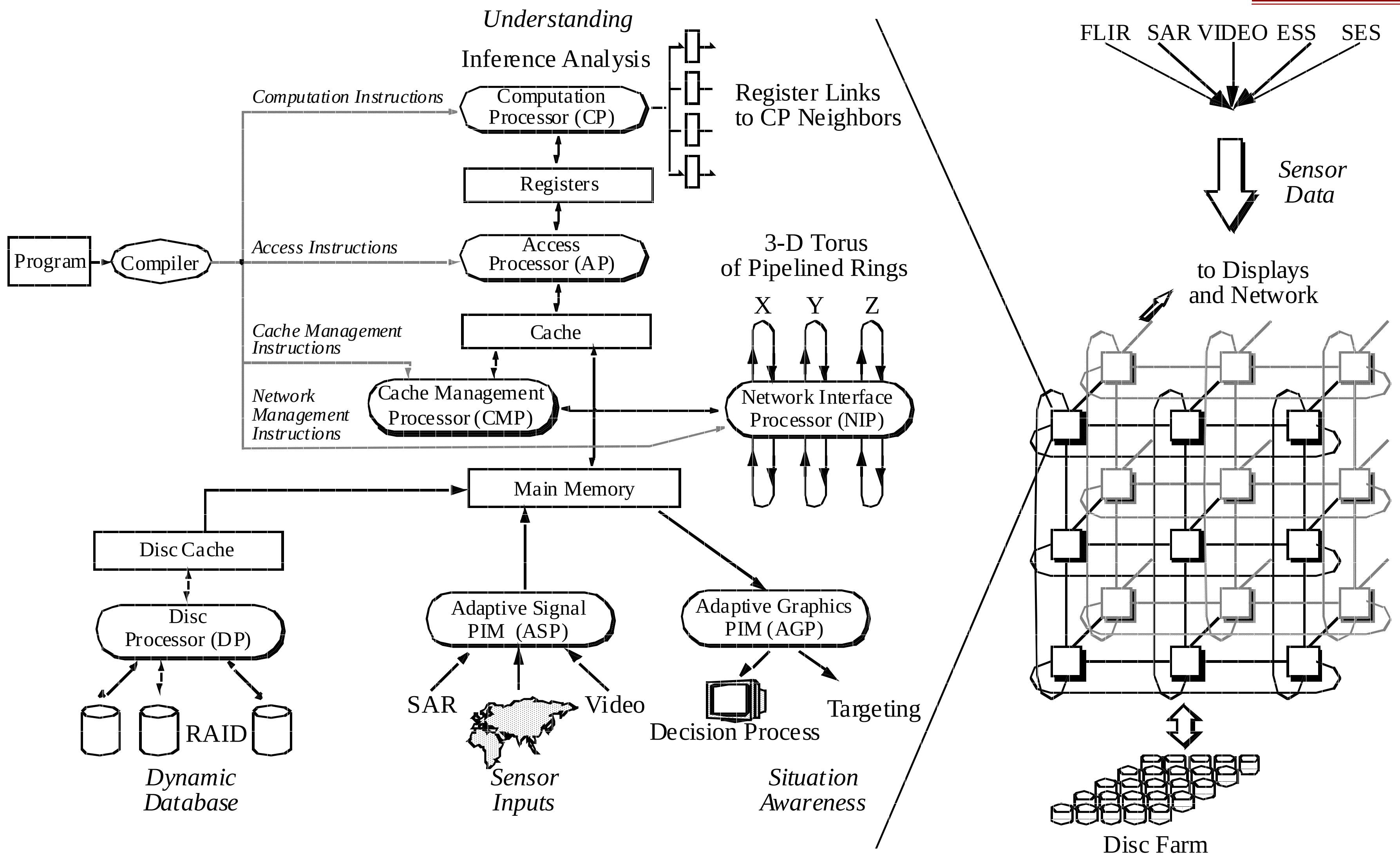
The McDISC Invention

- Problem: All extant, large-scale multiprocessors perform poorly when faced with a tightly-coupled parallel program.
- Reason: Extant machines have a long latency when communication is needed between nodes. This long latency kills performance when executing tightly-coupled programs. (Note that multi-threading a la the Tera machine does not help when there are dependencies.)

- The McDISC solution: Provide the network interface processor (NIP) with a programmable processor to execute not only OS code (e.g. Stanford Flash), but user code, generated by the compiler.
- Advantage: The NIP, executing user code, fetches data before it is needed by the node processors, eliminating the network fetch latency most of the time.
- Result: Fast execution (speedup) of tightly-coupled parallel programs.

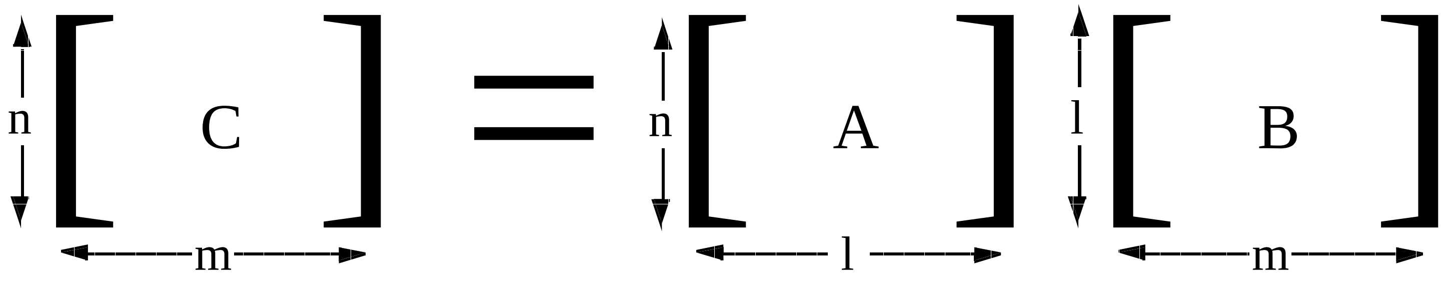
The McDISC System: Memory-Centered Distributed Instruction Set Computer

USC UNIVERSITY OF SOUTHERN CALIFORNIA



University of Southern California, Alvin M. Despain, Jean-Luc Gaudiot, Manil Makhija and Wonwoo Ro

Matrix Multiply on McDISC



Parallel Matrix Multiply

```
pid = processor id
p = # of processors
min_i = (pid / p) * n;
max_i = min_i + (p / n) - 1;

for (i = min_i; i <= max_i; ++i) {
   for (j = 0; j < m; ++j) {
      c[i][j] = 0;
      for (k = 0; k < l; ++k) {
         c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
   }
}
```

Matrix Multiply on McDISC

USC UNIVERSITY OF SOUTHERN CALIFORNIA

CP

```
while (not end-of-data) {
    while (not end-of-data) {
        c = 0;
        while (not end-of-data) {
            /* a and b from queue */
            c = c + a * b;
        }
        send c to store queue;
    }
}
```

AP

```
for (i = min_i; i <= max_i; ++i) {
    for (j = 0; j < m; ++j) {
        for (k = 0; k < l; ++k) {
            load a[i][k] to load queue;
            load b[i][k] to load queue;
        }
        send end-of-data to CP;
        put &c[i][j] in store queue;
        send signal to NIP;
    }
    send end-of-data;
}
send end-of-data
```

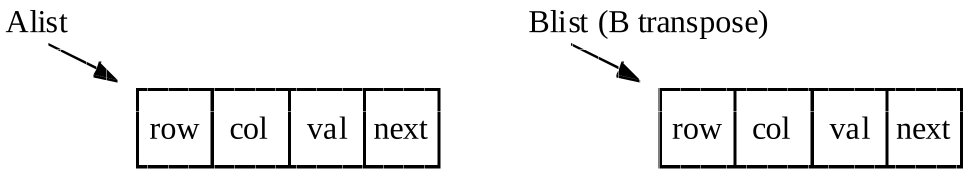
CMP

```
for (i = min_i; i <= max_i; ++i) {
    for (j = 0; j < m; ++j) {
        for (k = 0; k < l; ++k) {
            prefetch (a[i][k]);
            prefetch (b[i][k]);
        }
    }
}
```

NIP

```
for (i = min_i; i <= max_i; ++i) {
    for (j = 0; j < m; ++j) {
        wait for signal from AP;
        send c[i][j] to processor 0;
    }
}
```

Sparse Matrix Multiply on McDISC



Parallel Sparse Matrix Multiply (Inner Loop)

```
ap = alist;
bp = blist;

while ((ap != NULL) && (ap->row == i) &&
        (bp != NULL) && (bp->row == i)) {
   if (ap->col == bp->col) {
      sum = sum + (ap->data * bp->data);
      ap = ap->next;
      bp = bp->next;
   }
   else if (ap->col < bp->col)
      ap = ap->next;
   else
      bp = bp->next;
}
```

Sparse Matrix Multiply on McDISC

CP

```
sum = 0;
while (not EOD)
   sum += LQ * LQ
send sum to SDQ
```

CMP

```
ap = alist;
bp = blist;
while ((ap != NULL) && (ap->row == i) &&
        (bp != NULL) && (bp->row == i)) {
   if (ap->col == bp->col) {
      prefetch (ap->data);
      prefetch (bp->data);
      ap = ap->next;
      bp = bp->next;
   }
   else if (ap->col < bp->col)
      ap = ap->next;
   else
      bp = bp->next;
}
```

AP

```
ap = alist;
bp = blist;
while ((ap != NULL) && (ap->row == i) &&
        (bp != NULL) && (bp->row == i)) {
   if (ap->col == bp->col) {
      Put ap->data and bp->data in LQ
      ap = ap->next;
      bp = bp->next;
   }
   else if (ap->col < bp->col)
      ap = ap->next;
   else
      bp = bp->next;
}
Send EOD token to CP;
Send &c[i][j] to SAQ;
Send signal and address to NIP;
```

NIP

```
wait for signal from AP;
send data to home node;
```