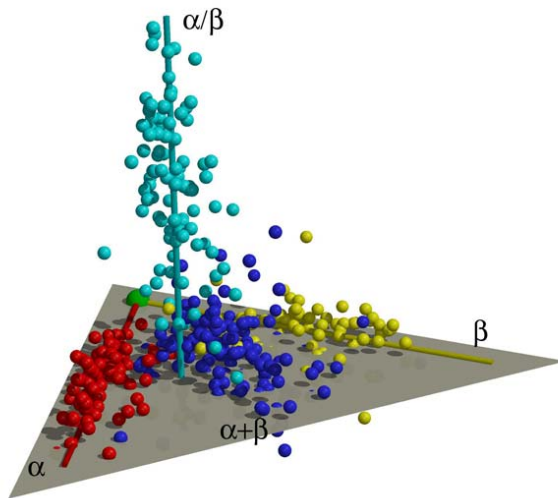


Load Instruction Characterization and Acceleration of the BioPerf Programs

Paruj Ratanaworabhan and Martin Burtscher

Computer Systems Laboratory

Cornell University



Talk outline

- Bioinformatics applications
- Characteristics of the BioPerf programs
- Load instruction characterization
- Source-code load scheduling
- Evaluation
- Conclusion

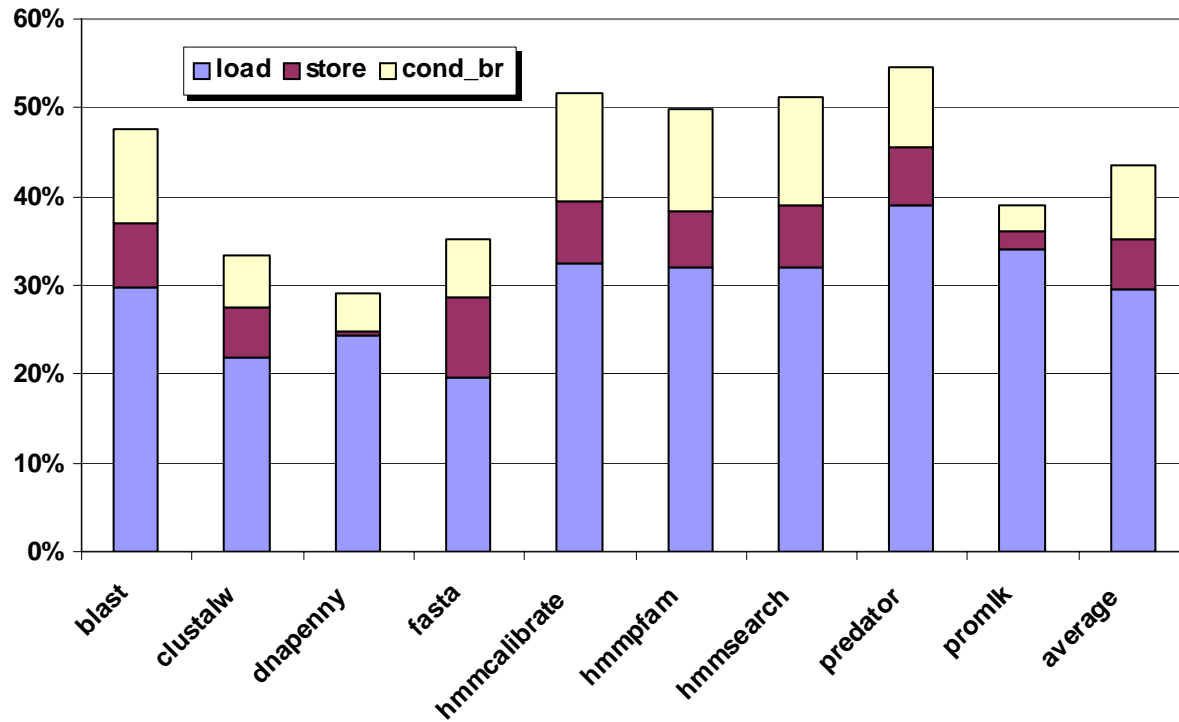
Bioinformatics applications

- Bioinformatics:
 - Development of advanced information technology to tackle problems in biology
- Growing important computer workload
- Benchmarks released:
 - BioInfoMark (Jan. 2005) from U of Florida
 - BioBench (Mar. 2005) from U of Maryland
 - BioPerf (Oct. 2005) from Georgia Tech

Selected BioPerf programs

- Cover three key areas:
 - Sequence analysis
 - Molecular phylogeny analysis
 - Protein structure analysis
- Reference compiler:
 - Alpha DEC C compiler 6.5 with `-O3 -arch ev68` flags
- Reference machine:
 - Alpha 21264 running OSF V5.1

Overall characteristics



	Instr (B)	FP
blast	77.3	0.04%
clustalw	789.4	0.04%
dnapenny	145.4	0.04%
fasta	542.1	0.63%
hmmlcalibrat	67.9	0.15%
hmmpfam	277.4	5.07%
hmmsearch	894.2	0.02%
predator	837.6	13.85%
promik	339.7	65.33%

- On-average, 30% of executed instructions are loads
- Mostly integer loads except *predator* and *promik*

Load instruction characteristics

	local miss rate for loads			AMAT
	L1	L2	Overall	
blast	1.78%	4.05%	0.07%	3.14
clustalw	1.90%	0.00%	0.00%	3.10
dnapenny	0.46%	4.30%	0.02%	3.04
fasta	0.47%	0.05%	0.00%	3.02
hmmcalibrate	1.61%	4.24%	0.07%	3.13
hmmpfam	0.67%	10.64%	0.07%	3.08
hmmsearch	0.35%	7.69%	0.03%	3.04
predator	0.46%	0.15%	0.00%	3.02
promlk	0.52%	4.93%	0.03%	3.04
average	0.91%	4.01%	0.03%	3.07
gmean	0.74%	0.75%	0.01%	3.07

L1 data cache:

size - 64KB

assoc - 2 ways

block size - 64 bytes

hit latency - 3 cycles

L2 unified cache:

size - 4MB

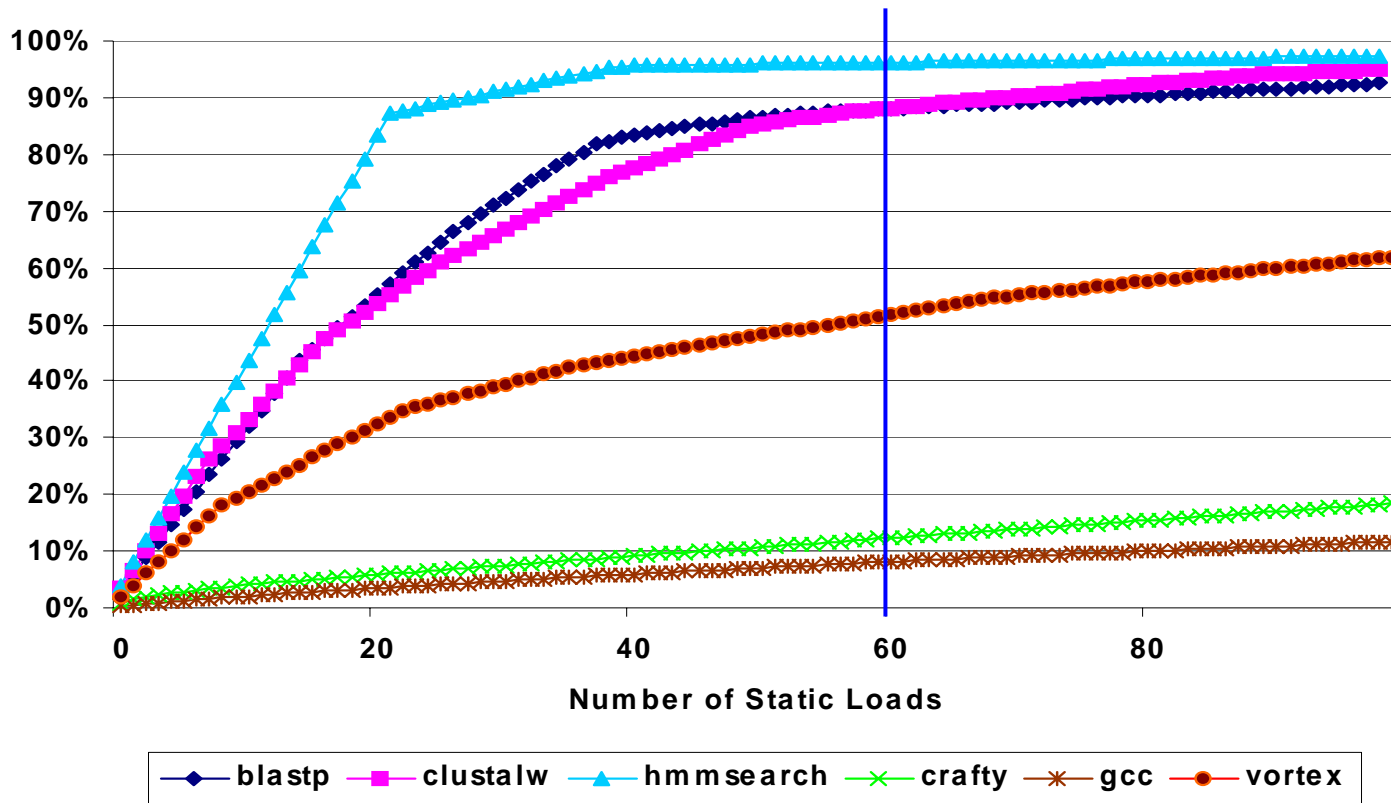
assoc. - directed-mapped

block size - 64 bytes block

hit latency - 8 cycles

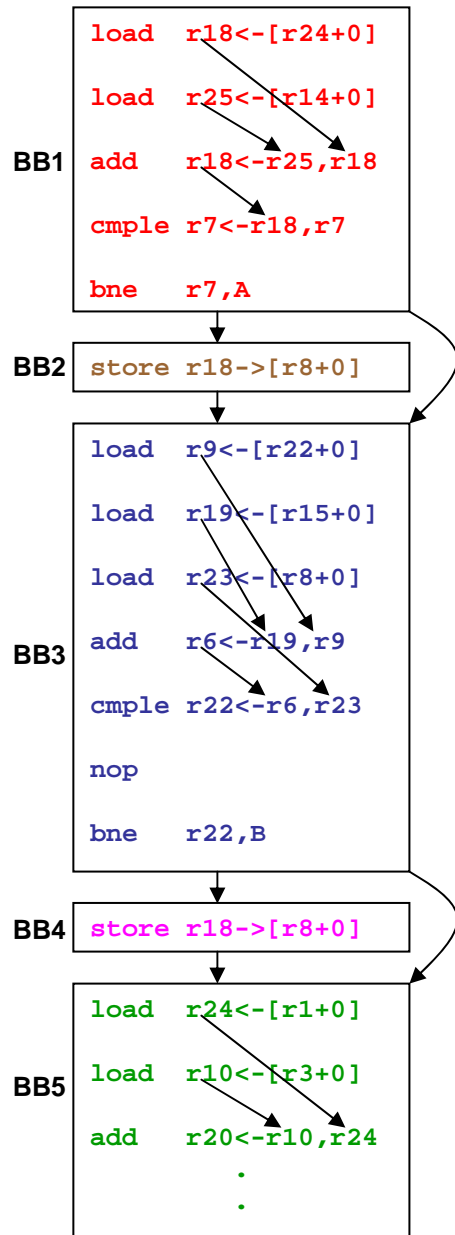
- Hardly miss in the cache hierarchy
- L1 hit latency dominates the AMAT term
- Programs operate on a chunk of data for a while before moving on to the next chunk

Load instruction characteristics



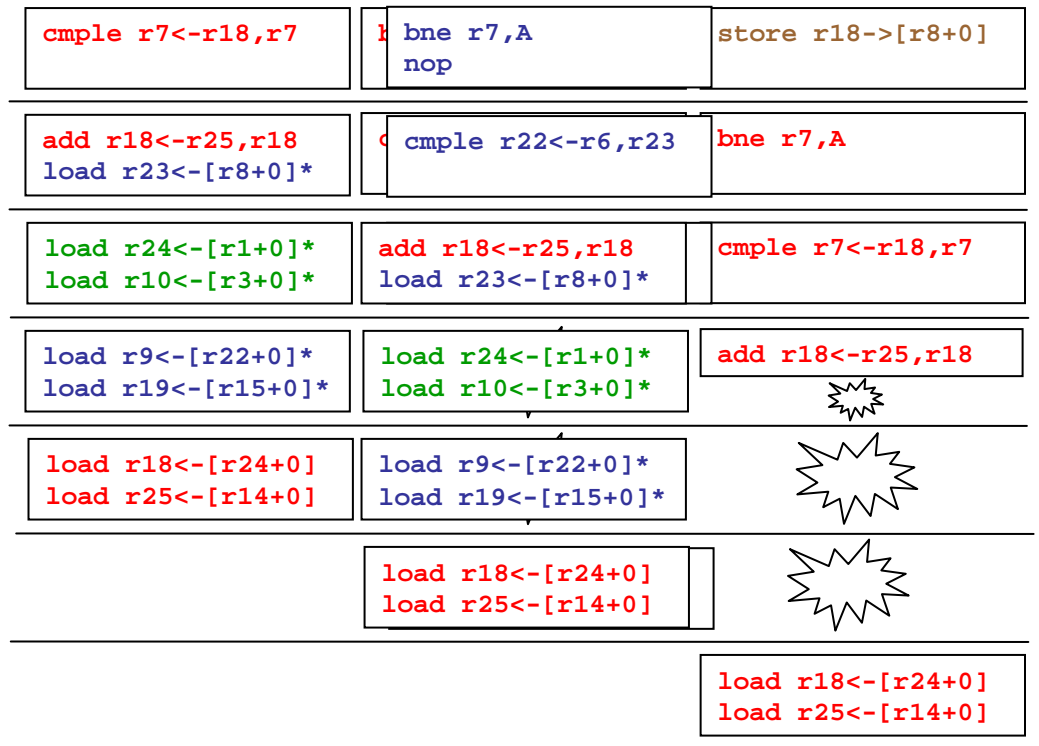
- Only a few static loads cover almost the entire dynamic load execution
- Attractive optimization targets; however, they almost always hit in the L1 cache

Load-to-branch and branch-to-load



Pipeline Stages

Branch predictor predicts path BB1 BB3 BB5
OOO core can service two loads per cycle



* Denote speculative issues

Load-to-branch and branch-to-load

	Load to branch	branch misprediction rate
blast	75.7%	19.9%
clustalw	56.2%	5.9%
dnapenny	33.6%	12.1%
fasta	31.6%	17.2%
hmmcalibrate	91.6%	11.2%
hmmpfam	92.4%	10.4%
hmmsearch	93.5%	9.9%
predator	51.1%	10.5%
promlk	15.2%	6.3%

	Branch to load
blast	32.7%
clustalw	19.6%
dnapenny	6.7%
fasta	23.2%
hmmcalibrate	56.5%
hmmpfam	57.8%
hmmsearch	60.4%
predator	21.1%
promlk	2.3%

- Load-to-branch
 - Often result in “difficult” branch
 - L1 hit latency adds to the misprediction penalty
- Branch-to-load
 - Measured after branches whose misprediction rate is over 5%
 - L1 hit latency exposed after pipeline flush

Load hoisting with optimizing compiler

```
load    r18<-[r24+0]
```

```
load    r25<-[r14+0]
```

```
add     r18<-r25,r18
```

```
cmple   r7<-r18,r7
```

```
bne     r7,A
```

```
store   r18->[r8+0]
```

```
load    r9<-[r22+0]
```

```
load    r19<-[r15+0]
```

```
load    r23<-[r8+0]
```

```
add     r6<-r19,r9
```

```
cmple   r22<-r6,r23
```

```
nop
```

```
bne     r22,B
```

```
store   r6->[r8+0]
```

```
load    r24<-[r1+0]
```

```
load    r10<-[r3+0]
```

```
add     r20<-r10,r24
```

- Load hoisting can alleviate these problems
- Difficult because of the intervening stores

Source-code load scheduling

- Can hide the L1 hit latency when out-of-order engine and optimizing compiler fail
- Semantics and context information available at this level
- Manual optimization based on profile information
- Suitable for these BioPerf programs (a few static loads cover over 90% of dynamic loads)

Example: hmmsearch

load index	5175	5177	5179	5182
frequency	3.97%	3.97%	3.97%	3.97%
L1 miss rate	0.05%	0.02%	0.07%	0.03%
following branch mispredictions	11.20%	28.41%	38.24%	0.50%

```
for (k = 1; k <= M; k++) {
```

```
    mc[k] = mpp[k-1] + tpmm[k-1];                (1)
    if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
    if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
    if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
    mc[k] += ms[k];
    if (mc[k] < -INFTY) mc[k] = -INFTY;
```

```
    dc[k] = dc[k-1] + tpdd[k-1];                (2)
    if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
    if (dc[k] < -INFTY) dc[k] = -INFTY;
```

```
    if (k < M) {                                (3)
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpim[k]) > ic[k]) ic[k] = sc;
        ic[k] += is[k];
        if (ic[k] < -INFTY) ic[k] = -INFTY;
    }
}
```

Example: hmmsearch

```
for (k = 1; k <= M; k++) {
```

```
    mc[k] = mpp[k-1] + tpm[m][k-1];           (1)
    if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
    if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
    if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
    mc[k] += ms[k];
    if (mc[k] < -INFTY) mc[k] = -INFTY;
```

```
    dc[k] = dc[k-1] + tpdd[k-1];             (2)
    if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
    if (dc[k] < -INFTY) dc[k] = -INFTY;
```

```
    if (k < M) {                               (3)
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpim[k]) > ic[k]) ic[k] = sc;
        ic[k] += is[k];
        if (ic[k] < -INFTY) ic[k] = -INFTY;
    }
}
```

```
for (k = 1; k <= M; k++) {
```

```
    temp1 = mpp[k-1] + tpm[m][k-1];          (1.1)
    temp2 = ip[k-1] + tpim[k-1];
    temp3 = dpp[k-1] + tpdm[k-1];
    temp4 = xmb + bp[k];
```

```
    temp5 = dc[k-1] + tpdd[k-1];            (2.1)
    temp6 = mc[k-1] + tpmd[k-1];
```

```
    if (temp2 > temp1) temp1 = temp2;         (1.2)
    if (temp3 > temp1) temp1 = temp3;
    if (temp4 > temp1) temp1 = temp4;
```

```
    if (temp6 > temp5) temp5 = temp6;         (2.2)
```

```
    mc[k] = ms[k] + temp1;                   (1.3)
    if (mc[k] < -INFTY) mc[k] = -INFTY;
```

```
    dc[k] = temp5;                           (2.3)
    if (dc[k] < -INFTY) dc[k] = -INFTY;
```

```
    if (k < M) {                               (3)
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpim[k]) > ic[k]) ic[k] = sc;
        ic[k] += is[k];
        if (ic[k] < -INFTY) ic[k] = -INFTY;
    }
}
```

Example: hmmsearch

```
for (k = 1; k <= M; k++) {
```

```
temp1 = mpp[k-1] + tpmm[k-1]; (1.1)
temp2 = ip[k-1] + tpim[k-1];
temp3 = dpp[k-1] + tpdm[k-1];
temp4 = xmb + bp[k];
```

```
temp5 = dc[k-1] + tpdd[k-1]; (2.1)
temp6 = mc[k-1] + tpmd[k-1];
```

```
if (temp2 > temp1) temp1 = temp2; (1.2)
if (temp3 > temp1) temp1 = temp3;
if (temp4 > temp1) temp1 = temp4;
```

```
if (temp6 > temp5) temp5 = temp6; (2.2)
```

```
mc[k] = ms[k] + temp1; (1.3)
if (mc[k] < -INFTY) mc[k] = -INFTY;
```

```
dc[k] = temp5; (2.3)
if (dc[k] < -INFTY) dc[k] = -INFTY;
```

```
if (k < M) { (3)
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpim[k]) > ic[k])
        ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
}
```



```
for (k = 1; k <= M-1; k++) {
```

```
temp1 = mpp[k-1] + tpmm[k-1]; (1.1)
temp2 = ip[k-1] + tpim[k-1];
temp3 = dpp[k-1] + tpdm[k-1];
temp4 = xmb + bp[k];
```

```
temp5 = dc[k-1] + tpdd[k-1]; (2.1)
temp6 = mc[k-1] + tpmd[k-1];
```

```
temp7 = mpp[k-1] + tpmi[k-1]; (3.1)
temp8 = ip[k-1] + tpim[k-1];
```

```
if (temp2 > temp1) temp1 = temp2; (1.2)
if (temp3 > temp1) temp1 = temp3;
if (temp4 > temp1) temp1 = temp4;
```

```
if (temp6 > temp5) temp5 = temp6; (2.2)
```

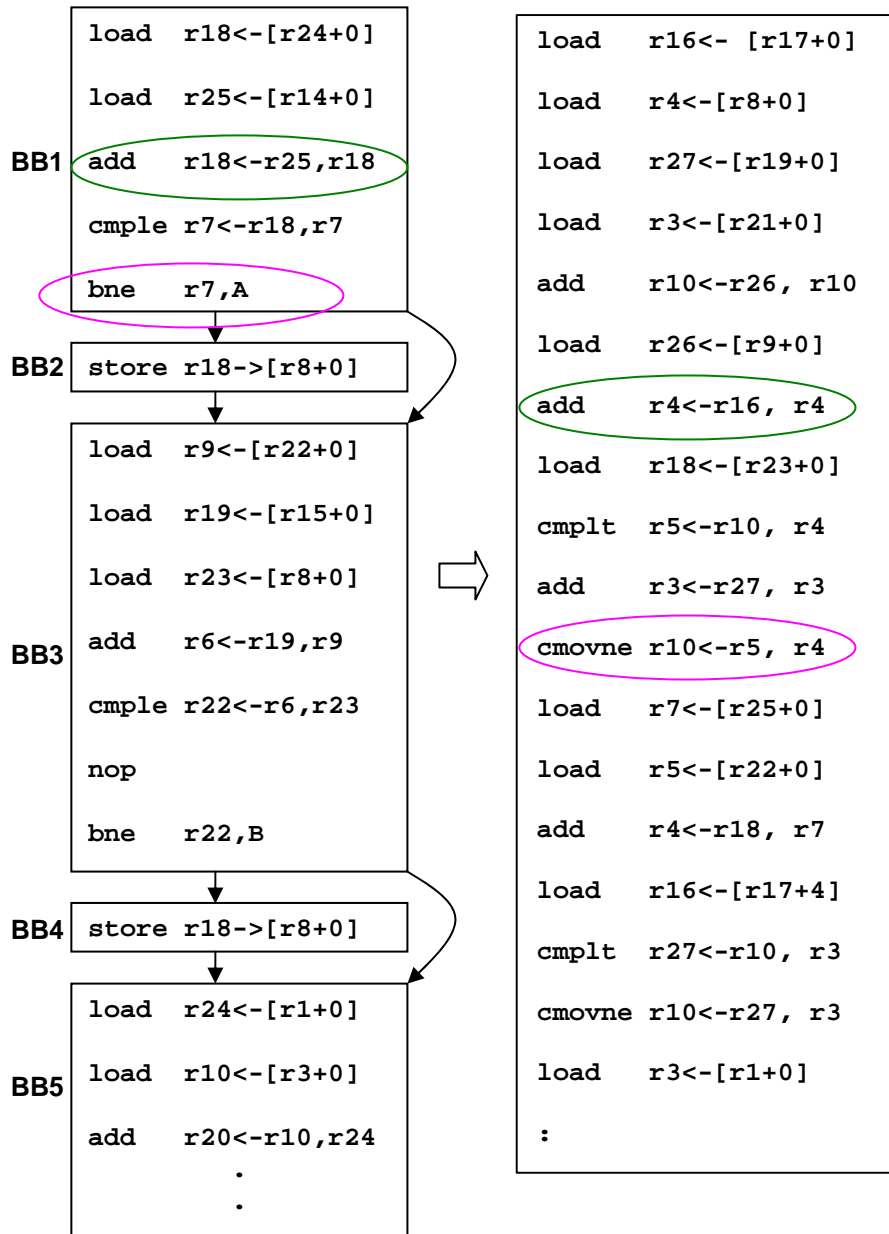
```
if (temp8 > temp7) temp7 = temp8; (3.2)
```

```
mc[k] = ms[k] + temp1; (1.3)
if (mc[k] < -INFTY) mc[k] = -INFTY;
```

```
dc[k] = temp5; (2.3)
if (dc[k] < -INFTY) dc[k] = -INFTY;
```

```
ic[k] = is[k] + temp7; (3.3)
if (ic[k] < -INFTY) ic[k] = -INFTY;
}
```

Example: hmmsearch



- Machine instructions of the transformed code:
 - Enough non-speculative instructions to hide load latency
 - Conditional branches converted into conditional moves

Source-code load scheduling

- Requires basically no knowledge about program's algorithm and data structure
- May not find opportunities to schedule loads at this level, e.g., tight loop with no room to perform scheduling
- BioPerf applications on which we perform this optimization:

	dnapenny	hmmpfam	hmmsearch	hmmcalibrate	predator	clustalw
Static loads considered	3	16	19	14	1	4
Lines of C code involved	10	25	30	25	5	10

Evaluation Methodology

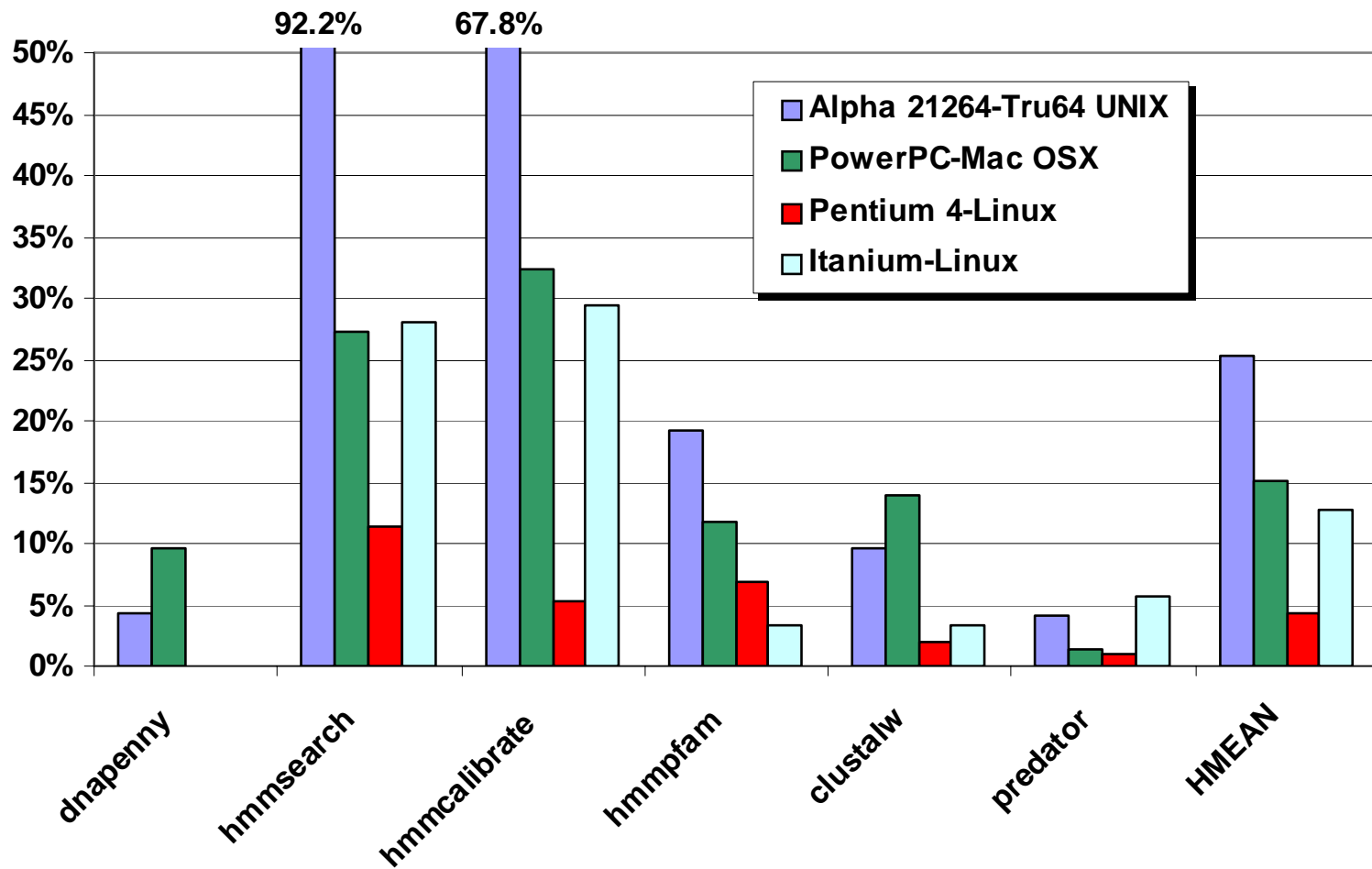
- Platforms

	Alpha 21264	Power PC	Pentium 4	Itanium
Datapath	64 bits	64 bits	32 bits	64 bits
Register	32 GPR, 32 FPR	32 GPR, 32 FPR	8 GPR, 8 FPR	128 GPR, 128 FPR
L1 data cache	64 KB 2-way	32 KB 2-way	8 KB 4-way	16 KB 4-way
L1 hit latency	3 to 4 cycles (Int/FP)	3 to 5 cycles (Int/FP)	2 to 6 cycles (Int/FP)	1 cycle (Int)

- Compilers and optimization flags:
 - DEC C 6.5 compiler with `-O3` on Alpha
 - GNU C 3.3.3 compiler with `-O3` on Power PC and Pentium
 - Intel C compiler 9.0 on Itanium with `-O3`
- Feedback-directed optimization used on all platforms

Results

- Speedups over baseline code



Results

- Greater performance benefits on Alpha and Power PC than Pentium 4
 - Larger L1 hit latency for integer loads (3 versus 2 cycles)
 - More registers to cope with increasing register pressure
- Substantial speedup on Itanium even though this is an in-order machine with single cycle L1 hit
 - Expanded basic blocks
 - More independent instructions issued together
 - Minimize costly control speculation recovery

Conclusions

- Investigated characteristics of load instructions in the BioPerf programs
- Found performance bottleneck due to loads that *hit* in the L1 data cache
- Used source-code load scheduling to remove this bottleneck where out-of-order execution engine and optimizing compiler failed to do so
- Achieved average speedup of 25%, 15%, 4%, and 13% on Alpha, Power PC, Pentium, and Itanium platforms
- www.bioperf.org/RB06-BioPerf-source.tar.bz2