Best Architecture for a Cache Controller

EE 6313 Advanced Microprocessors

Fall 2016 Project 2

PROJECT REPORT

Submitted by

VENKATA DINESH JAKKAMPUDI (1001235204)

Date:12/05/2016

Overview:

The goal of this project is to determine the best architecture for a cache controller that interfaces to a 32- bit microprocessor with a 32-bit data bus. While the microprocessor is general purpose in design and performs number of functions, it is desired to speed up certain signal processing functions, such as a fast Fourier transform (FFT) routine.

Detailed Requirements:

We are given the task of determining the architecture of a cache memory to speed up a microprocessor system. After the program being executed was profiled, the largest hit in performance was seen in a function called Radix2FFT which is shown in the program.

The goal is to determine the best architecture for the 256 kB cache including the associative set size (N), number of cache lines (L), and burst length (BL), write strategy (write-back or write-through), and replacement strategy (round robin vs LRU).

The project seeks to allow 4GB of DRAM to be interfaced using a 32-bit data bus (arranged as 1G x 32-bits) and the cache should be limited to 256 kB in size. The size of the FFT will be 32768 points (512kB) in normal operation.

By adding extra code to the FFT program, it is possible to determine exactly how memory is accessed. Direct observations include: -

- How often and in what order the loop variables and data elements accessed.
- How can thrashing of cache lines be limited?
- How does the organization of cache, write strategy, and replacement strategy affect performance?

Simulated Code:

```
// this makes the real and imaginary part of “complex” double precision
#include <complex>
#define complex std::complex<double>
#include<stdio.h>
#include<stdint.h>
#include<math.h>
#include<complex.h>
#define max_lines 65536
#define max_N 8
#define max_BL 8
uint32_t lines, N;
uint32_t l, w = 32, b, BL;
uint32_t t;
uint32_t D[max_lines][max_N];
```

```
uint32_t V[max_lines][max_N];
//int tag1;
uint32_t tag[max_lines][max_N];
uint32_t LRU[max_lines][max_N];
uint32_t RR[max_lines];

//int miss = 1;
bool wb, wta, wtna;
uint32_t lru = 0, rr;
uint32_t count_Rm, count_Wm, count_Rl, count_Rlh, count_Rld, count_Rlr, count_Wl,
count_Wlh, count_Wlr, count_Wld, count_Fd, count_Wt;

void clear_cache()
{
int i, j;
for (i = 0; i < 65536; i++)
{
for (j = 0; j < 8; j++)
{
D[i][j] = 0;
V[i][j] = 0;
tag[i][j] = 0;
LRU[i][j] = 0;
}
RR[i] = 0;
}
count_Rm = 0; count_Wm = 0; count_Rl = 0; count_Rlh = 0; count_Rld = 0;
count_Rlr = 0; count_Wl = 0; count_Wlh = 0; count_Wlr = 0; count_Wld = 0;
count_Fd = 0, count_Wt = 0;
}

void flush_cache()
{
int i, j;
for (i = 0; i < 65536; i++)
{
for (j = 0; j < 8; j++)
{
if (D[i][j])
{
count_Fd++;
}
}
}
```

```
}
/*_complex points[32176];
for (size_t i = 0; i < length; i++)
{
readMemory(&ponts[i], sizeof(point[i]));
}*/

void update_LRU(uint32_t tag1, uint32_t line)
{
int i, j = 0, k = 3;
for (i = 0; i < N; i++)
{
if (tag[line][i] == tag1)
{
if (LRU[line][i] != 0)
k = LRU[line][i];
LRU[line][i] = 0;
j = i;
break;
}
}

for (i = 0; i < N; i++)
{
if (V[line][i] && (j != i) && (LRU[line][i] < k))
LRU[line][i]++;
}

}
bool check_status(uint32_t tag1, uint32_t line)
{
int i, j = 0;
for (i = 0; i < N; i++)
{
if (tag[line][i] == tag1)
j = 1;
}
return j;
}

uint32_t get_tag(uint32_t addr)
{
return (addr >> (b + l));
}
```

```
uint32_t get_line(uint32_t addr)
{
return ((addr << t) >> (t + b));
}

void read_line(uint32_t tag1, uint32_t line)
{
count_Rl++;
int i = 0, j = 0;
bool fill = 1;
bool hit = check_status(tag1, line);
if (hit)
{
count_Rlh++;
if (lru)
update_LRU(tag1, line);
}
else
{
for (i = 0; i < N; i++)
{
if (V[line][i] == 0)
{
j = i;
fill = 0;
break;
}
}

if (fill)
{
if (lru)
{
for (i = 0; i < N; i++)
{
if (LRU[line][i] > LRU[line][j])
j = i;
}
}
else
{
j = RR[line];
RR[line] = (RR[line] + 1)%N;
```

```
}
if (D[line][j] == 1)
{
count_Rld++;
D[line][j] = 0;
}
}
count_Rlr++;
//D[line][j] = 0;
//LRU[line][j] = 0;
V[line][j] = 1;
tag[line][j] = tag1;
if (lru)
update_LRU(tag1, line);
}

}

void write_line(uint32_t tag1, uint32_t line)
{
int j = 0, i; count_Wl++;
bool line_fill = 1;
bool hit = check_status(tag1, line);
if (!hit && (wb || wta))
{
for (i = 0; i < N; i++)
{
if (V[line][i] == 0)
{
j = i;
line_fill = 0;
break;
}
}

if (line_fill)
{
if (lru)
{
for (i = 0; i < N; i++)
{
if (LRU[line][i] > LRU[line][j])
j = i;
}
```

```
}
else
{
j = RR[line];
RR[line] = (RR[line] + 1) % N;
}
if (V[line][j] && D[line][j])
{
count_Wld++;
D[line][j] = 0;
}
}
count_Wlr++;
V[line][j] = 1;
//LRU[line][j] = N-1;
tag[line][j] = tag1;
if (lru)
update_LRU(tag1, line);
if (wb)
D[line][j] = 1;
}
if ((wb || wta || wtna) && hit)
{
count_Wlh++;
if (lru)
update_LRU(tag1, line);
}
if (wta || wtna)
{
for (i = 0; i<BL; i++)
count_Wt++;
}
}

void readMemory(void* addr, int size)
{
uint32_t tag1, line, last_line;
int i;
count_Rm++;
last_line = -1;
uint32_t add = (uint32_t)addr;
for (i = 0; i < size; i++)
{
tag1 = get_tag(add + i);
```

```
line = get_line(add + i);
if (line != last_line)
{
last_line = line;
read_line(tag1, line);
//count rl++;
}
}
}
void writeMemory(void* addr, int size)
{
uint32_t tag1, line, last_line;
int i;
count_Wm++;
last_line = -1;
uint32_t add = (uint32_t)addr;
for (i = 0; i < size; i++)
{
tag1 = get_tag(add + i);
line = get_line(add + i);
if (line != last_line)
{
last_line = line;
write_line(tag1, line);
}
}
}
void Radix2FFT(complex data[], int nPoints, int nBits)
{
// cooley-tukey radix-2, twiddle factor
// adapted from Fortran code from Burrus, 1983
#pragma warning (disable: 4270)
int i, j, k, l;
int nPoints1, nPoints2;
complex cTemp, cTemp2;
readMemory(&nPoints, sizeof(nPoints));
writeMemory(&nPoints2, sizeof(nPoints2));
nPoints2 = nPoints;
writeMemory(&k, sizeof(k));
readMemory(&k, sizeof(k));
readMemory(&nBits, sizeof(nBits));
for (k = 1; k <= nBits; k++)
{
//readMemory(&k, sizeof(k));
```

```
readMemory(&nPoints2, sizeof(nPoints2));
writeMemory(&nPoints1, sizeof(nPoints1));
nPoints1 = nPoints2;
readMemory(&nPoints2, sizeof(nPoints2));
writeMemory(&nPoints2, sizeof(nPoints2));
nPoints2 /= 2;
// Compute differential angles
readMemory(&nPoints1, sizeof(nPoints1));
double dTheta = 2 * 3.14159257 / nPoints1;
writeMemory(&dTheta, sizeof(dTheta));
readMemory(&dTheta, sizeof(dTheta));
double dDeltaCos = cos(dTheta);
writeMemory(&dDeltaCos, sizeof(dDeltaCos));
readMemory(&dTheta, sizeof(dTheta));
double dDeltaSin = sin(dTheta);
writeMemory(&dDeltaSin, sizeof(dDeltaSin));
// Initialize angles
double dCos = 1;
double dSin = 0;
writeMemory(&dCos, sizeof(dCos));
writeMemory(&dSin, sizeof(dSin));
// Perform in-place FFT
writeMemory(&j, sizeof(j));
readMemory(&j, sizeof(j));
readMemory(&nPoints2, sizeof(nPoints2));
for (j = 0; j < nPoints2; j++)
{
readMemory(&j, sizeof(j));
writeMemory(&i, sizeof(i));
i = j;
readMemory(&i, sizeof(i));
readMemory(&nPoints, sizeof(nPoints));
while (i < nPoints)
{
readMemory(&i, sizeof(i));
readMemory(&nPoints2, sizeof(nPoints2));
writeMemory(&l, sizeof(l));
l = i + nPoints2;
readMemory(&i, sizeof(i));
readMemory(&l, sizeof(l));
readMemory(&data[i], sizeof(data[i]));
readMemory(&data[l], sizeof(data[l]));
writeMemory(&cTemp, sizeof(cTemp));
cTemp = data[i] - data[l];
```

```
readMemory(&i, sizeof(i));
readMemory(&l, sizeof(l));
readMemory(&data[i], sizeof(data[i]));
readMemory(&data[l], sizeof(data[l]));
writeMemory(&cTemp2, sizeof(cTemp2));
cTemp2 = data[i] + data[l];
readMemory(&i, sizeof(i));
readMemory(&cTemp2, sizeof(cTemp2));
writeMemory(&data[i], sizeof(data[i]));
data[i] = cTemp2;
readMemory(&dCos, sizeof(dCos));
readMemory(&cTemp, sizeof(cTemp.real()));
readMemory(&dSin, sizeof(dSin));
readMemory(&cTemp+8, sizeof(cTemp.imag()));
readMemory(&dCos, sizeof(dCos));
readMemory(&cTemp, sizeof(cTemp.real()));
readMemory(&dSin, sizeof(dSin));
readMemory(&cTemp+8, sizeof(cTemp.imag()));
writeMemory(&cTemp2, sizeof(cTemp2));
cTemp2 = complex(dCos * cTemp.real() + dSin * cTemp.imag(),
dCos * cTemp.imag() - dSin * cTemp.real());
readMemory(&l, sizeof(l));
readMemory(&cTemp2, sizeof(cTemp2));
writeMemory(&data[l], sizeof(data[l]));
data[l] = cTemp2;
readMemory(&nPoints1, sizeof(nPoints1));
readMemory(&i, sizeof(i));
writeMemory(&i, sizeof(i));
readMemory(&i, sizeof(i));
readMemory(&nPoints, sizeof(nPoints));
i += nPoints1;
}
readMemory(&dCos, sizeof(dCos));
double dTemp = dCos;
writeMemory(&dTemp, sizeof(dTemp));
readMemory(&dCos, sizeof(dCos));
readMemory(&dSin, sizeof(dSin));
readMemory(&dDeltaCos, sizeof(dDeltaCos));
readMemory(&dDeltaSin, sizeof(dDeltaSin));
writeMemory(&dCos, sizeof(dCos));
dCos = dCos * dDeltaCos - dSin * dDeltaSin;
readMemory(&dTemp, sizeof(dTemp));
readMemory(&dSin, sizeof(dSin));
readMemory(&dDeltaCos, sizeof(dDeltaCos));
```

```
readMemory(&dDeltaSin, sizeof(dDeltaSin));
writeMemory(&dSin, sizeof(dSin));
dSin = dTemp * dDeltaSin + dSin * dDeltaCos;
readMemory(&j, sizeof(j));
writeMemory(&j, sizeof(j));
readMemory(&j, sizeof(j));
readMemory(&nPoints2, sizeof(nPoints2));
}
readMemory(&k, sizeof(k));
writeMemory(&k, sizeof(k));
readMemory(&k, sizeof(k));
readMemory(&nBits, sizeof(nBits));
}
// Convert Bit Reverse Order to Normal Ordering
writeMemory(&j, sizeof(j));
j = 0;
readMemory(&nPoints, sizeof(nPoints));
writeMemory(&nPoints1, sizeof(nPoints1));
nPoints1 = nPoints - 1;
writeMemory(&i, sizeof(i));
readMemory(&i, sizeof(i));
readMemory(&nPoints1, sizeof(nPoints1));
for (i = 0; i < nPoints1; i++)
{
readMemory(&i, sizeof(i));
readMemory(&j, sizeof(j));
if (i < j)
{
readMemory(&j, sizeof(j));
readMemory(&data[j], sizeof(data[j]));
writeMemory(&cTemp, sizeof(cTemp));
cTemp = data[j];
readMemory(&i, sizeof(i));
readMemory(&data[i], sizeof(data[i]));
writeMemory(&cTemp2, sizeof(cTemp2));
cTemp2 = data[i];
readMemory(&i, sizeof(i));
readMemory(&cTemp, sizeof(cTemp));
writeMemory(&data[i], sizeof(data[i]));
data[i] = cTemp;
readMemory(&j, sizeof(j));
readMemory(&cTemp2, sizeof(cTemp2));
writeMemory(&data[j], sizeof(data[j]));
data[j] = cTemp2;
```

```
}
readMemory(&nPoints, sizeof(nPoints));
writeMemory(&k, sizeof(k));
k = nPoints / 2;
readMemory(&k, sizeof(k));
readMemory(&j, sizeof(j));
while (k <= j)
{
readMemory(&k, sizeof(k));
readMemory(&j, sizeof(j));
writeMemory(&j, sizeof(j));
j -= k;
readMemory(&k, sizeof(k));
writeMemory(&k, sizeof(k));
k /= 2;
readMemory(&k, sizeof(k));
readMemory(&j, sizeof(j));
}
readMemory(&k, sizeof(k));
readMemory(&j, sizeof(j));
writeMemory(&j, sizeof(j));
j += k;
readMemory(&i, sizeof(i));
writeMemory(&i, sizeof(i));
readMemory(&i, sizeof(i));
readMemory(&nPoints1, sizeof(nPoints1));
}
#pragma warning(default: 4270)
}

int main(int argc, char* argv[])
{
int i,i1,j,k,rep,combinations=0;
complex data[32768];
printf("FFT Test App\r\n");
// time domain: zero-offset real cosine function
// freq domain: delta fn with zero phase
#define cycles 1 // max cycles is points/2 for Nyquist
int points = 32768;
FILE *fp1;
fopen_s(&fp1, "6313.dat", "w");

fprintf(fp1, "BL\tN\tstra\tRep\tRM\tRL\tRLD\tRlR\tRlh\tWm\tWl\tWld\tWlr\tWlh\tFD\tWT\n");
```

```
fprintf(fp1, "-------------------------------------------------------------------------------------------------------------------------------
------------------------------\n");
for (i1 = 1; i1 <= max_BL; i1++)
{
if (i1 == 1 || i1 == 2 || i1 == 4 || i1 == 8)
{
for (j = 1; j <= max_N; j++)
{
if (j == 1 || j == 2 || j == 4 || j == 8)
{
for (k = 0; k < 3; k++)
{
for (rep = 0; rep < 2; rep++)
{
lru = rep;
bool strategy[3] = { 0 };
strategy[k] = 1;
wb = strategy[0];
wta = strategy[1];
wtna = strategy[2];
BL = i1, N = j;
b = ceil(log(BL * 4) / log(2));
lines = 65536 / (BL*N);
l = ceil(log(lines) / log(2));
t = w - b - l;
clear_cache();
for (i = 0; i < points; i++)
data[i] = complex(cos(2.0*3.1416*float(i) / float(points)*cycles), 0.0);

// time domain: impulse time offset
// freq domain: constant amplitude of 1
// phase is zero for zero time offset
// 1 phase rotation per unit time offset
/*#define DELTA_T 0
for (i = 0; i < points; i++)
data[i] = complex(0.0, 0.0);
data[DELTA_T] = complex(1.0, 0.0);
*/
int bits = ceil(log(points) / log(2));
Radix2FFT(data, points, bits);
for (i = 0; i < points; i++)
if (data[i].imag() >= 0.0)
printf("x[%d] = %2.4lf + j%2.4lf\n", i,
data[i].real(), data[i].imag());
```

```
else
printf("x[%d] = %2.4lf - j%2.4lf\n", i,
data[i].real(), -data[i].imag());
flush_cache();
combinations++;
fprintf(fp1, "%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n", i1, j, k,
rep, count_Rm, count_Rl, count_Rld, count_Rlr, count_Rlh, count_Wm, count_Wl, count_Wld,
count_Wlr, count_Wlh, count_Fd, count_Wt);
printf("rm=%d,wm=%d\n", count_Rm, count_Wm);
printf("rl=%d,wl=%d\n", count_Rl, count_Wl);
printf("rld=%d,wld=%d\n", count_Rld, count_Wld);
printf("rlr=%d,wlr=%d\n", count_Rlr, count_Wlr);
printf("rlh=%d,wlh=%d\n", count_Rlh, count_Wlh);
printf("fd=%d\n", count_Fd);
printf("\n%d", combinations);
}
}
}
}
}
}
fclose(fp1);
getchar();
return 0;
}
```

Result:

stra	Write Back Strategy
0	Write Back
1	Write Through Allocate
2	Write Through Non-Allocate

Rep	Replacement Strategy
0	Round Robin
1	Least Recently Used

AMAT *(in ns)* = (count_Wlr+ count_Wld+ count_Rlr+ count_Rld) *(90+(BL-1) *15) +(count_Wlh*1) + (count_Rlh*1) +FD*(90+(BL-1) *15) +count_Wt*90

From the above observations, it is found that best cache architecture is **BL=4, N=8, Write Back Strategy, least recently used Replacement Strategy**. The average memory access time for this architecture is **58111534 ns**.

The worst cache architecture is found to be **BL=8, N=1, Write through allocate of both LRU and RR replacement strategies**. The average memory access time for this architecture is **2007678157 ns**.