CS522S10-Project Report

Utility-Based Cache Partitioning

Guided by,

Professor Dimitry Ponomarev

Jason Loew

Project Members

Ankitkumar Biscuitwala : B00262309

Hardik Shah : B00267313

Aim:

Implement the technique proposed in the paper '**Utility-Based Cache Partitioning**' and verify it's results.

Introduction:

To obtain high performance from multi-thread architectures is to manage on-chip cache efficiently so that off-chip accesses are reduced. This paper investigates the problem of partitioning the shared level on-chip cache among multiple competing applications. Traditional design for on-chip cache uses the LRU policy for replacement decisions. The LRU
policy implicitly partitions a shared cache among the competing applications on a demand basis, giving more cache resources to the application that has a high demand and fewer cache resources to the application that has a low demand.

The benefit that an application gets from the cache resources may not directly corelate with its demand for cache.

Implementation:

Data structure used for implementing UCP:

In cache.h ::::

```
//cache block (or line) definition
class cache_blk_t
{
        -
        -
        -
           //The following comment may not be relevant anymore.
           //DATA should be pointer-aligned due to preceeding field
           std::vector<byte_t> data;       //actual data block starts here, block size should probably be a multiple of 8
           int context_id;              //context_id of the owner of the data in the block

           //EDIT
           int owner;                  // Add which cache is thread is owner of that block
           counter_t blk_count;     //          Count hit in block
};

//cache definition
class cache_t
{
#endif
    public:
           -
           -
        // EDIT
        int * g_array_cnt_0;   // count total hit for thread 0
        int * g_array_cnt_1;// count total hit for thread 1
        int * a;              // array to count miss in varying way for thread0
        int * b;              // array to count miss in varying way for thread1
        int * c;              // array to compute the max util miss from thread0 and thread1
        int d;                // store the final value computed
```

```
        --
        --
           counter_t thits_0;          //total number of thread 0 hits
           counter_t thits_1;          //total number of thread 1 hits
           counter_t tmiss_0;           //total number of thread 0 miss
           counter_t tmiss_1;           //total number of thread 1 hits
};

In cache.c

//create and initialize a general cache structure
cache_t::cache_t(std::string name,    --
                --
                --
        //initialize cache stats
        hits(0),misses(0),replacements(0),writebacks(0),
        invalidations(0),thits_0(0),tmiss_1(0),thits_1(0),tmiss_0(0),d(0),

        -
        -
        g_array_cnt_0 = (int *)calloc(assoc, sizeof (int));       // to define a global array
        g_array_cnt_1 = (int *)calloc(assoc, sizeof (int));       // to define a global array
        a = (int *)calloc((assoc+1), sizeof (int));       // to define a global array
        b = (int *)calloc((assoc+1), sizeof (int));       // to define a global array
        c = (int *)calloc((assoc+1), sizeof (int));       // to define a global array
        --
        --
        #endif
           //NOTE: all the blocks in a set *must* be allocated contiguously otherwise block
           //      accesses through SET->BLKS will fail (used during random replacement selection)
           sets[i].blks = CACHE_BINDEX(this, &data[0], bindex);

             //link the data blocks into ordered way chain and hash table bucket chains, if hash table
exists
           for(unsigned int j=0; j<assoc; j++)
           {
                //locate next cache block
                cache_blk_t *blk = CACHE_BINDEX(this, &data[0], bindex);
                bindex++;

                //EDIT

                if(strncmp( name.c_str(),"Core_0_dl1",10)==0)
                {
                     if(i<(nsets/2))                    // Dividing into half-half set
                     blk->owner = 0;                    // assign set to thread own by blk 0
                     else
                     blk->owner = 1;                    // assign set to thread own by blk 1
                }

                -
                -
```

```
}


//resets cache stats after fast forwarding
void cache_t::reset_cache_stats()
{
	int i = 0;
	//initialize cache stats
	hits = misses = replacements = writebacks = invalidations = 0;
	thits_0 = thits_1 = tmiss_0 = tmiss_1 = d = 0;   // initialize the attributes

	for(i = 0; i < assoc; i++){
		g_array_cnt_0[i] = 0;  // reset the array

		g_array_cnt_1[i] = 0;  // reset the array
	}
	  -
	  -
}

//print cache stats to the file descriptor stream
void cache_t::print_stats(FILE *stream)
{
      -
      -
 fprintf(stream, "%s.thits_0      %lld # total number of thits_0\n",              name.c_str() , thits_0);
 fprintf(stream, "%s.thits_1       %lld # total number of thits_1\n",              name.c_str() , thits_1);
 fprintf(stream,"\n%s.tmiss_0   %lld # total number of tmiss_0\n",              name.c_str(), tmiss_0);
 fprintf(stream,"%s.tmiss_1      %lld # total number of tmiss_1\n",              name.c_str(), tmiss_1);
	  --
	  --
	 if(strncmp( name.c_str(),"Core_0_dl1",10)==0)
	{	  for (int i=0; i<assoc;i++)
		{
			fprintf(stream," GLOBAL array [0] =%d\n",g_array_cnt_0[i]);

		}
		fprintf(stream, "\n");
		for (int i=0; i<assoc;i++)
		{
			fprintf(stream," GLOBAL array [1] =%d\n",g_array_cnt_1[i]);

		}
		fprintf(stream, "\n");
	   fprintf(stream," \n\n::::::::::       UTILITY BASED CACHE PARTITION # Thread 1: %d
Thread 2: %d::::::: \n\n", d,(assoc-d));
	}
```

```
unsigned long long cache_t::cache_access(mem_cmd cmd,
{

        -
        -
 //permissions are checked on cache misses

      cache_blk_t *blk(NULL);
      cache_blk_t * repl(NULL);

//EDIT
      if(context_id==0)                    // divide the set into half-half
      {
        if(set<(nsets/2))
            { set=set;}
        else
            {set= set % (nsets/2);}
      }
      else
      {
         if (set >= (nsets/2))
            set = set;

         else
           set = set + (nsets/2) ;
      }
         -
         -
 else
#endif
      {
            int assoc_cnt = 0;
            //low-associativity cache, linear search the way list
            for(blk=sets[set].way_head;blk;blk=blk->way_next)
            {
if(blk->tag==tag && (blk->status & CACHE_BLK_VALID) && (blk->context_id == context_id))
                  {
                        if(strncmp( name.c_str(),"Core_0_dl1",10)==0)
                        {
                              if(set< (nsets/2))
                                    g_array_cnt_0[assoc_cnt] ++;
                              else
                                    g_array_cnt_1[assoc_cnt] ++;
                        }
                  blk->blk_count ++;
                        goto cache_hit ;

                  }
                  assoc_cnt ++;
 }
      }
```

```
        //Cache block not found, MISS
        misses++;

// EDIT : Add your tmiss code here.
        if(strncmp( name.c_str(),"Core_0_dl1",10)==0)
        {
                if(context_id==0)
                        tmiss_0 ++;
                else
                        tmiss_1 ++;
        }


        if(strncmp( name.c_str(),"Core_0_dl1",10)==0)
        {
                a[0] = tmiss_0;
                for(int i=1;i<=assoc;i++)
                {
                        a[i]=g_array_cnt_0[assoc+i-1] + a[i-1];
                }
                b[0] = tmiss_1;
                for(int i=1;i<=assoc;i++)
                {
                        b[i]=g_array_cnt_1[assoc+i-1] + b[i-1];
                }

                c[0] = 0;
                for(int i=1;i<=assoc;i++)
                {
                        c[i]=a[i] + b[assoc-i];
                        if(c[i]<c[i-1])
                              {
                               d=i;
                        }
                }
  }
 switch(policy)
        {
        case LRU:
        case FIFO:
                repl = sets[set].way_tail;

                // EDIT

                 if(strncmp( name.c_str(),"Core_0_dl1",10)==0)
                {
                        update_way_list(&sets[set], repl, Head);
                }

                else
                {
                        repl = sets[set].way_tail;
```

```
                update_way_list(&sets[set], repl, Head);
            }
            break;
          -
          -
          -
ache_hit:
        //HIT
        hits++;
```

// EDIT : Add your thits :::

```
        if(strncmp( name.c_str(),"Core_0_dl1",10)==0)
        {

            if(context_id==0)
              thits_0 ++;
            else
              thits_1 ++;
       }
```

Algorithm:

1. Calculate the number of miss on thread 0 by varying way
2. Calculate the number of miss on thread 1 by varying way
3. Compute the partition for way of two thread that gives min Miss based on (1) and (2)

```
                a[0] = tmiss_0;
            for(int i=1;i<=assoc;i++)
            {
                a[i]=g_array_cnt_0[assoc+i-1] + a[i-1];
            }
            b[0] = tmiss_1;
            for(int i=1;i<=assoc;i++)
            {
                b[i]=g_array_cnt_1[assoc+i-1] + b[i-1];
            }

            c[0] = 0;
            for(int i=1;i<=assoc;i++)
            {
                c[i]=a[i] + b[assoc-i];
                if(c[i]<c[i-1])
                    {
                     d=i;
                    }
             }
```

Comparative Analysis:

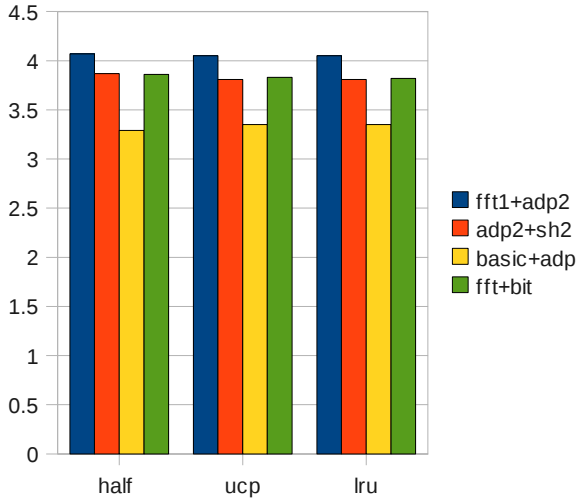

Figure 1. (IPC vs Benchmark)

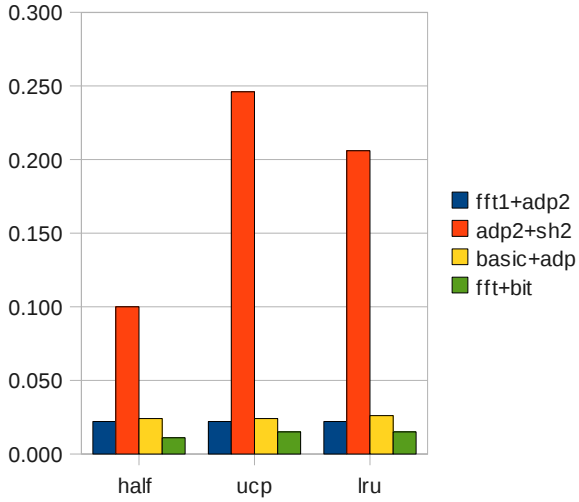

Figure 2. (Missrate vs Benchmark)

Benchmarks used: basicmathNS.1.arg, adpcmNS.1.arg, basicmathNS.1.arg, adpcmNS.1.arg, basicmathNS.1.arg and bitcountNS.1.arg. For the parameter -cache:dl1 dl1:256:32:8:l

Conclusion:

Traditional shared cache use LRU replacement which partitions the cache among competing applications on a demand basis. The benefit that applications get for a given amount of cache resources may not correlate with the demand.

Utility-Based Cache Partitioning (UCP) to divide the cache among competing applicationsbased on based on the utility of cache resource for each applications.

We implemented the technique on MSIM simulator and comparision gave almost equal values to LRU and UCP policy, the observation shows that half-half policy that is based on partition of sets gave better results.

Acknowledgements:
We are thankful to Professor. Dmitry Ponomarev for his valuable guidance and helpful comments. We also thank Jason Loew and Jasneet Kaur for providing useful input and support.