Hybrid Cache

Ashish G Pai, Chris Vinn Mathew, Rakshith Kaushik T R, Sriee Gowthem Raaj A S
Department of Electrical Engineering, University of Texas at Dallas
Richardson, TX, USA
Email: {agp150130, cxm154130, rxt160030, sxa156930} @ utdallas.edu

***Abstract*—Intelligently partitioning the cache within a processor can bring significant performance improvements. Resources are given to the applications that can benefit most from them, restricting the core to a number of logical cache ways. In this paper we present a Hybrid Cache, a cache architecture that allows reconfiguration in both its size and associativity resulting in greater number of hits and decreased cache latency in comparison to a state-of-the-art cache. We developed a simple cache simulator written in Java and improved upon it to employ reconfiguration.**

***Keywords*—Hybrid cache, reconfiguration**

I. Introduction

Cache memories contain a large number of transistors and consume a large amount of energy. For instance, 60% of the StrongARM's area is devoted to caches [1]. For this reason, many processors, particularly intellectual property cores, allow the configuration of the caches to be determined at design time, according to the requirements of the target applications.

Customization of cache parameters may be static or dynamic; in a static approach the designer sets the cache parameters before synthesis, whereas in a dynamic scheme the cache parameters can be modified within a certain range at run-time. When cache parameters are determined statically, a single configuration is chosen by the designer to trade-off performance against energy consumption. Static configurations require less on-chip logic, validation and testing than performing dynamic reconfiguration. However, they do not have the ability to react to changes in cache requirements both across programs and within the same application.

We have proposed a configurable cache architecture that allows reconfiguration of both the size and associativity of the cache, providing maximum flexibility to the application. We have compared our approach, called the Hybrid cache, against state-of-the-art cache and showed that our scheme's number of hits and latency is better than the state-of-the-art cache.

To demonstrate the performance of the proposed scheme we have developed a decision tree model that monitors the behavior of each cache and dynamically reconfigures in response to changing application requirements.

The rest of this paper is structured as follows. Section 2 describes related work. Section 3 describes our Hybrid cache and Section 4 describes our experimental setup. Section 5 presents our results. Finally, section 6 concludes. The code snippets are added in the appendix.

II. Related Work

The performance of cache designs has been widely studied for several decades. In general, caches are designed as direct-mapped or set associative. A direct-mapped cache has the advantages of a low cost and a faster hit time [2]. However, the cache miss rate is usually high when the cache size is not big. On the other hand, a set associative cache has a lower miss rate with a higher design cost and a longer hit time.

In [3], the authors have presented several novel techniques for designing an energy efficient cache, which include block buffering, cache sub-banking and Gray code addressing. In [4], Karthik et al. present Set and way Management Cache Architecture for Run-Time reconfiguration (SMART cache), a cache architecture that allows reconfiguration in both its size and associativity. Our paper derives major ideas from [4].

In [5], authors present Cooperative partitioning, a runtime partitioning scheme that reduces both dynamic and static energy while maintaining high performance. In [6] Josep et al. present in detail the locality patterns of the operating system code and show that there is substantial locality.

III. Hybrid Cache & Methodology

This section describes the Hybrid Cache and the methodology that we used to design the cache.

A. Methodology

The methodology that we used for designing the Hybrid Cache is as follows:

1. Run different trace files [7] on different cache configurations to find out the number of hits and the threshold for the Hybrid Cache.
2. Devise an algorithm to improve the number of hits and cache latency.
3. The algorithm employs changing of associativity which uses the threshold determined in Step 1. and the active set count.

B. Terminologies

1. Active Set Count: Active set count is defined as the number of active sets in the cache.
2. Most Recently Used (MRU) Count: MRU Count is defined as the number of active slots accessed within a set. MRU count is used to retain the mostly used slots in a set and to evict the least used slots.

C. Working of the Hybrid Cache

The Hybrid cache can switch between smaller and larger cache configurations and the associativity can be switched between 1, 2 and 4-way as shown in Fig. 1.

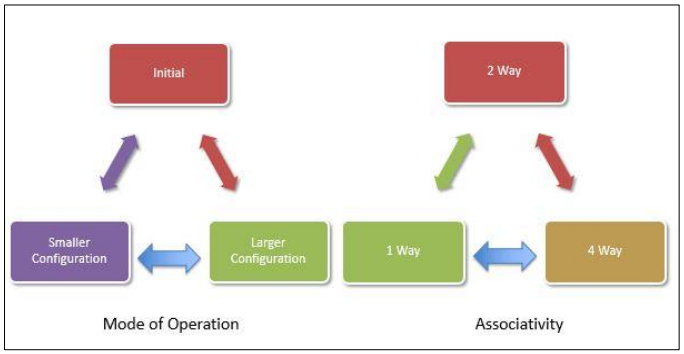

Fig. 1. State diagram for working of the Hybrid Cache

The Hybrid cache works as follows:

1. Check for number of misses at periodic intervals.
2. Check for factors viz. Active set count and Most Recently Used (MRU) count with respect to the threshold to reconfigure the cache.
3. Reconfigure based on the decision tree shown in Fig. 2.

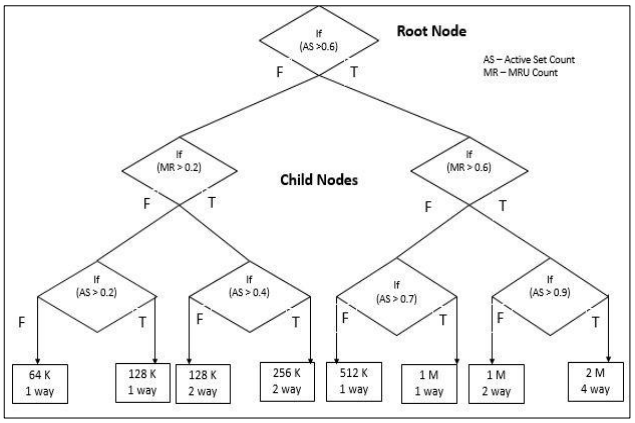

Fig. 2. Decision Tree

IV. Experimental Setup

This section describes the tools used and the parameters used to evaluate our Hybrid Cache performance.

A. *System Specifications*

The different cache configurations that we have used are listed in Table I.

TABLE I. Cache Configurations

Parameter	Cache Size
Associativity	1, 2 & 4
Block Size (Bytes)	16
Cache Size (Bytes)	32K, 64K, 128K, 256K, 512K, 1M, 2M

1. The cache simulator has been designed for a uniprocessor system employing write through policy. The cache is a unified cache.
2. A simulated external memory has been used to demonstrate fetching of data in case of cache misses. The data that will be fetched will be random data.
3. The trace files that have been used consist of load and store instructions. We have used varying sizes of trace files consisting of 35,000, 500,000 and 1 million instructions for learning the decision tree.
4. Addresses are 24-bit long.
5. The simulators are written in Java™ and the OS used is Ubuntu V15.10

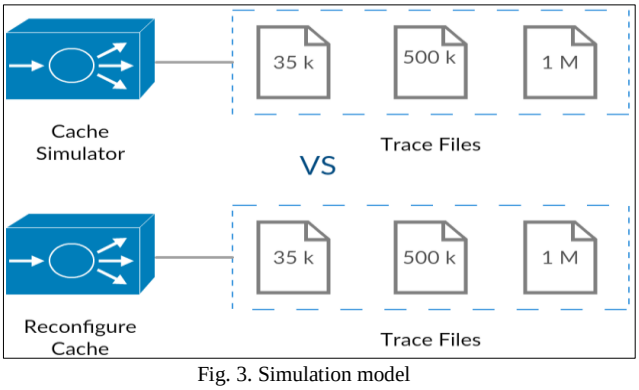

Fig. 3. Simulation model

B. *Workflow*

The workflow is as follows:

1. We have designed a basic cache simulator which takes these trace files as inputs and gives us the number of hits. We have used a simple counter for getting the number of hits.
2. We have simulated the trace files for different cache configurations listed in Table I. and we have sampled every 10,000 instructions to check for the number of hits to come up with the decision tree of Fig. 2.
3. We have checked the execution time of this basic simulator using the *bash time* command in Ubuntu.
4. After getting an appropriate threshold we devised an algorithm to incorporate the threshold and redesigned the simulator for reconfiguration.
5. We ran the same set of trace files on this redesigned Hybrid cache simulator to find the number of hits and the execution time.

C. *Algorithm*

The algorithm devised is as follows;

1. Cache runs with the configuration set by the user at run time.
2. Cache performance is monitored for every 10,000 instructions.
3. Parameters (Active Set count, MRU count and miss ratio) are calculated for each set and for each slot within each set.
4. The average value is calculated for active set count and MRU count.
5. The average values are compared with the threshold values as per the Decision tree in Fig. 2.
6. Cache prepares to transit from the present state to the best reconfigurable state.
7. Cache moves to the reconfiguration state with the following changes:
   i) The Active set count and MRU count are reset.
   ii) Slots and Sets are evicted.
   iii) The cache contents are flushed making it cold again. It should be noted that this step gives rise to compulsory misses once the reconfiguration takes place.
8. Normal functioning of the cache resumes and it performance will be checked after 10,000 instructions.

V. Results

This section evaluates our Hybrid cache approach to dynamic cache reconfiguration. We show the effects of dynamic reconfiguration using our architecture in comparison to the normal cache.

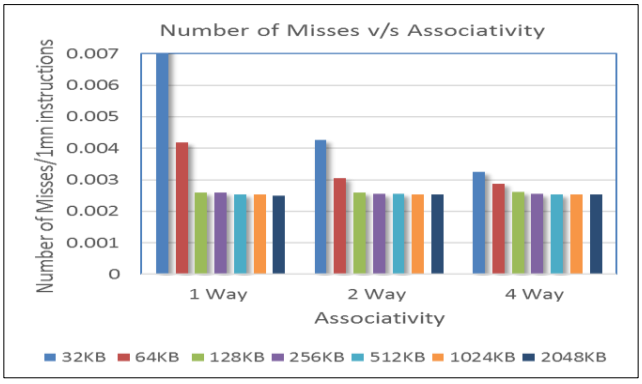

Fig. 4. Results for the normal cache

The results shown in Fig. 4. show the number of misses/1 million instructions for the normal cache. We chose this because the biggest trace file would give a fair idea of how our Hybrid Cache performs for a worst case scenario.

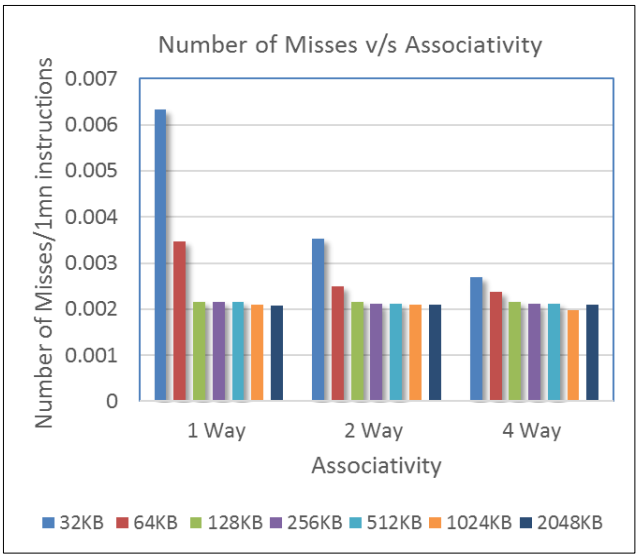

Fig. 5. Results for the Hybrid cache

Fig. 5. shows the improvement that our Hybrid cache achieves. The improvement achieved, if not substantial, is respectable.

VI. Conclusions

In this paper, we have presented a novel reconfigurable cache architecture and a decision tree that dynamically predicts the best cache configuration for any application. The main goal is to reduce the number of misses and the cache latency.

Future work will consider cache reconfiguration on a multi-core architecture. In addition, this simulator does not employ any pipelining, loop unrolling and branch prediction techniques which will further improve the cache performance.

The functionality of finding the active sets and slots within each set could be done more efficiently in hardware compared to software approach. Software approach requires to handle huge data structures which may slightly increase execution time. These limitations can be overcome by masking the way selection and set selection. The decision tree's threshold values were found by trial and error and through inferences of the trace file that we used to run the hybrid cache. The procedure could be extended to wide range of trace files and by using formal machine learning algorithms like Leave one out validation.

The data sets we used to infer was by running the trace files for different configurations. These methods could be extended to include more configuration. Using a wide range of data sets can help to generalize the threshold value such that re-configurable cache benefits wide variety of applications. These added functionality gives an overhead to the current cache execution, but considering the miss rates that seems to decrease is acceptable. This added functionality which was designed and tested via simulator can be made to be realizable in hardware.

VII. References

[1] Manne, S., Klauser, A., Grunwald, D, "Pipeline gating: speculation control for energy reduction." *ISCA*, 1998.

[2] M.D.Hill, "A case for Direct-mapped Caches," IEEE Computer, 21,12, 1988. I.S. Jacobs and C.P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G.T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271-350.

[3] Su, Ching-Long and Despain, M., Alvin, "Cache Designs for energy efficiency." *Proceedings of the 28th Hawaii International Conference on System Sciences*, 1995.

[4] Sundararajan, T., Karthik, Jones, M., Timothy and Topham, P., Nigel, "The Smart Cache: An Energy-Efficient Cache Architecture Through Dynamic Adaptation."

[5] Sundararajan, T., Karthik, Porpodas, Vasileios, Jones, M., Timothy, Topham, P., Nigel and Franke, Björne, "Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMPs."

[6] Torrellas, Josep, Xia, Chun and Daigle, L., Russell, "Optimizing the Instruction Cache Performance of the Operating System." *IEEE Transcations on Computers*, vol. 47, no. 12, December 1998.

[7] Trace files, https://www.cs.umd.edu/class/spring2015/cmsc411-0201/projects/p1/

APPENDIX

1. Code snippet for the normal cache operation

```
/*********************************************************************
                 * Cache Operation                                   *
 *********************************************************************/
                int tagInt = cU.binaryToTag(binaryInstruction);
                int indexInt = cU.binaryToIndex(binaryInstruction);
                int offset = cU.binaryToOffset(binaryInstruction) % cU.getBlockSize();
                int setCounter = 0; //set counter
                boolean flag = false;

                Iterator<ArrayList<CacheSpace>> setIterator = cache.iterator();
                while(setIterator.hasNext()){
                    ArrayList<CacheSpace> set = setIterator.next();
                    if(setCounter == indexInt){
                        int count = 1;
                        Iterator<CacheSpace> it = set.iterator();
                        while(it.hasNext()){
                            CacheSpace slot = new CacheSpace();
                            slot = it.next();
                            if(slot.tag == tagInt){ //cache hit
                                hit++;
                                if ( parts[1].equals("S") ){ //store hit

                                    //writing to cache
                                    slot.word[offset] = cU.getData();
                                    slot.validBit = 1;
                                    slot.tag = tagInt;
                                    slot.index = indexInt;

                                    //Write-through memory portion pulling the old value at that key if it exists
                                    if(memory.containsKey(binaryInstruction)){
                                        memory.put(binaryInstruction, slot.word[offset]);
                                        System.out.println("store hit " + parts[0]);
                                    } else {
                                        //key = (tag + set); Value = new random data
                                        memory.put(binaryInstruction, slot.word[offset]);
                                        System.out.println("store hit " + parts[0]);
                                    }
                                }

                                if ( parts[1].equals("L")){ //load hit
                                    System.out.println("load hit " + parts[0]);
                                }
                                flag = true;
                                break;
                            }
                            /*********************************************************************
                             * Cache Miss                                                        *
                             *********************************************************************/
                            if((count == cU.getSetAssociation()) && (slot.tag != tagInt)){ //cache miss
                                int toCache = 0;
                                if ( parts[1].equals("S") ){ //store miss
                                    noOfDataMiss++;
                                    toCache = cU.getData();
                                    memory.put(binaryInstruction, toCache);
                                    System.out.println("store miss " + parts[0]);
                                } else if ( parts[1].equals("L") ){ //load miss
                                    noOfInstructionMiss++;
                                    //pulling data from memory
                                    if(! memory.containsKey(binaryInstruction)){
                                        memory.put(binaryInstruction, 0);
                                        System.out.println("load miss " + parts[0]);
                                    } else {
                                        toCache = memory.get(binaryInstruction);
                                    }
                                }

                                //make new CacheSpace object to put in cache
                                CacheSpace cacheSlot = new CacheSpace();

                                //putting memory in cache
                                cacheSlot.word = new int[Integer.parseInt(args[3])];
                                cacheSlot.word[offset] = toCache;
                                cacheSlot.validBit = 1;
                                cacheSlot.tag = tagInt;
```

```
                                cacheSlot.index = indexInt;

                                //Random Replacement Policy
                                set.set(cU.getRandomReplacementSlot(), cacheSlot);
                            }

                            count++;
                        }

                    }
                    if (flag) break;
                    setCounter+=1;
                }
                /**************************************************************************
                 * End of cache Operation                                                 *
                 **************************************************************************/
```

2. Code snippet for the Hybrid cache

```
public static CacheUtility reconfigCache(int activeSetSize, int lruChainSize){
    //calculate and round off activesetcount n lrucount
    DecimalFormat df = new DecimalFormat("#.###");
    df.setRoundingMode(RoundingMode.CEILING);
    activeSetCountAvg = Double.parseDouble( df.format(activeSetSize/(double)cU.getNumSets()));
    lruCountAvg = Double.parseDouble(df.format(lruChainSize/(double)cU.getNumBlocks()));

    Integer blockSize = cU.getBlockSize();
    String bSize = blockSize.toString();
    if(activeSetCountAvg>0.6){
        if(lruCountAvg>0.6){
            if(activeSetCountAvg>0.9){
                System.out.println("Changed to config 1");
                return cU = new CacheUtility("2048", "4", bSize);

            }
            else{
                System.out.println("Changed to config 2");
                return cU = new CacheUtility("1024", "2", bSize);
            }
        }
        else{
            if(activeSetCountAvg>0.7){
                System.out.println("Changed to config 3");
                return cU = new CacheUtility("1024", "1", bSize);
            }
            else{
                System.out.println("Changed to config 4");
                return cU = new CacheUtility("512", "1", bSize);
            }
        }
    }
    else{
        if(lruCountAvg>0.2){
            if(activeSetCountAvg>0.4){
                System.out.println("Changed to config 5");
                return cU = new CacheUtility("256", "2", bSize);
            }
```

```
                else{
                    System.out.println("Changed to config 6");
                    return cU = new CacheUtility("128", "2", bSize);
                }
            }
            else{
                if(activeSetCountAvg>0.2){
                    System.out.println("Changed to config 7");
                    return cU = new CacheUtility("128", "1", bSize);
                }
                else{
                    System.out.println("Changed to config 8");
                    return cU = new CacheUtility("64", "1", bSize);
                }
            }
        }

    }

/**********************************************************************
             * Initialize Cache                                                    *
             **********************************************************************/

            for(int i = 0; i < cU.getNumSets(); i++){
                ArrayList<CacheSpace> initSetBlock = new ArrayList<CacheSpace>();
                for(int k = 0; k < cU.getSetAssociation(); k++){
                    CacheSpace initSlot = new CacheSpace();
                    initSlot.word = new int[Integer.parseInt(args[3])];
                    initSetBlock.add(initSlot);
                }
                cache.add(initSetBlock);
            }

            //store counters

            while(traceFileScanner.hasNextLine()){
                noOfInstructions++;
                checkPoint++;
                String instructionLine = traceFileScanner.nextLine().trim();
                instructionLine = instructionLine.replaceFirst("0x", "");

                String[] parts = instructionLine.split(" ");
                parts[1] = parts[1].substring(0,6);

                String binaryInstruction = cU.hexToBin(parts[1]);
                /**********************************************************************
                 * Cache Operation                                                    *
                 **********************************************************************/
                int tagInt = cU.binaryToTag(binaryInstruction);
                int indexInt = cU.binaryToIndex(binaryInstruction);
                int offset = cU.binaryToOffset(binaryInstruction) % cU.getBlockSize();

                int setCounter = 0; //set counter
                boolean flag = false;
                int activeSetCount = 0;
                Iterator<ArrayList<CacheSpace>> setIterator = cache.iterator();
                while(setIterator.hasNext()){
                    ArrayList<CacheSpace> set = setIterator.next();

                    if (allSet.get(indexInt) != null) {
                        activeSetCount = allSet.get(indexInt);
                        }

                    if(setCounter == indexInt){
                        int count = 1;
                        Iterator<CacheSpace> it = set.iterator();
                        while(it.hasNext()){
                            CacheSpace slot = new CacheSpace();
                            slot = it.next();
                            if(slot.tag == tagInt){ //cache hit
                                hit++;
                                if ( parts[0].equals("S") ){ //store hit
```

```
111
112                         //writing to cache
113                         slot.word[offset] = cU.getData();
114                         slot.validBit = 1;
115                         slot.tag = tagInt;
116                         slot.index = indexInt;
117                         slot.incrementLruCount();       //Incrementing LRU count
118
119                         //Write-through memory portion pulling the old value at that key if it exists
120                         if(memory.containsKey(binaryInstruction)){
121                             memory.put(binaryInstruction, slot.word[offset]);
122                         } else {
123                             //key = (tag + set); Value = new random data
124                             memory.put(binaryInstruction, slot.word[offset]);
125                         }
126                     }
127
128                     if ( parts[1].equals("L")){ //load hit
129                     }
130                     flag = true;
131                     break;
132                 }
133                 /**********************************************************************
134                  * Cache Miss - Stores instruction only when its frequency = 3       *
135                  **********************************************************************/
136                 if((count == cU.getSetAssociation()) && (slot.tag != tagInt)){ //cache miss
137                     int toCache = 0;
138                     if ( parts[0].equals("S") ){ //store miss
139                         noOfDataMiss++;
140                         toCache = cU.getData();
141                         memory.put(binaryInstruction, toCache);
142
143                     } else if ( parts[0].equals("L") ){ //load miss
144                         noOfInstructionMiss++;
145                         //pulling data from memory
146                         if(! memory.containsKey(binaryInstruction)){
147                             memory.put(binaryInstruction, 0);
148                         } else {
149                             toCache = memory.get(binaryInstruction);
150                         }
151                     }
152
153                     //make new CacheSpace object to put in cache
154                     CacheSpace cacheSlot = new CacheSpace();
155
156                     //putting memory in cache
157                     cacheSlot.word = new int[Integer.parseInt(args[3])];
158                     cacheSlot.word[offset] = toCache;
159                     cacheSlot.validBit = 1;
160                     cacheSlot.tag = tagInt;
161                     cacheSlot.index = indexInt;
162                     cacheSlot.incrementLruCount();      //Incrementing LRU count
163
164                     //Random Replacement Policy
165                     set.set(cU.getRandomReplacementSlot(), cacheSlot);
166                 }
167                 //cacheslot of misses are not being counted?
168                 if(slot.lruCount == 4){
169                     String slotKey = Integer.toString(indexInt)+Integer.toString(count);
170                     lruChain.put(slotKey, slot.lruCount);
171                 }
172                 count++;
173             }
174
175         }
176         if (flag) break;
177         setCounter+=1;
178     }
179
180     if(activeSetCount < 4) {
181         activeSetCount++ ;
182         allSet.put(indexInt, activeSetCount);
```

```
183                 }
184                 if(activeSetCount == 4){
185                     activeSet.put(indexInt,activeSetCount);
186                 }
187 
188                 if(checkPoint == 10000){
189                     reconfigCache(activeSet.size(),lruChain.size());
190                     allSet.clear();
191                     activeSet.clear();
192                     lruChain.clear();
193                     checkPoint = 0;
194                     cache.clear();
195                     for(int i = 0; i < cU.getNumSets(); i++){
196                         ArrayList<CacheSpace> initSetBlock = new ArrayList<CacheSpace>();
197                         for(int k = 0; k < cU.getSetAssociation(); k++){
198                             CacheSpace initSlot = new CacheSpace();
199                             initSlot.word = new int[Integer.parseInt(args[3])];
200                             initSetBlock.add(initSlot);
201                         }
202                         cache.add(initSetBlock);
203                     }
204                 }
205                 /**************************************************************************
206                  * End of cache Operation                                                 *
207                  *************************************************************************/
208             } // End of while
209 
```