Final Project Report

Zhengbiao Deng

December 9, 2016

Abstract

Cache compression can improve cache performance, through increase effective cache capacity and eliminate misses. To utilize the cache space more efficiently and reduce the miss rate, a compressed cache hierarchy is proposed. In this hierarchy, L1 caches are normal uncompressed caches and L2 cache is a compressed cache. The decoupled variable-segment cache allows L2 cache pack more compressed cache lines than uncompressed lines into the same space. The principles of compressed cache line hierarchy is stated. A basic simulator is developed to implement the compressed cache hierarchy using Frequent Pattern Compress (FPC), and it is compared with normal uncompressed cache for specific benchmarks. Finally, analysis of the result is demonstrated.

1 Introduction

Because of locality, caches are used in computers to reduce the access time to memory. Frequently accessed data can be stored in caches. And one of important metric of caches is miss rate. Because a miss in a cache takes a long long access time of fetching the data from main memory. And as we know, increasing the set-associativity, scaling the cache size, and configuring multi-level caches are all methods to reduce the miss rate. Beside these ways, researchers come up with many other interesting solutions. One of them is compressed cache line hierarchy.

Compressing is a broadly used technology in research and our life. It is recognizing the pattern of input data and transforming the input to a smaller size output. When the compressed data is need to be utilized again, we just need to decompress it. Compressing save the space in which the data is stored.

And in caches, compressing can also benefit. Alaa Alameldeen demonstrated a Frequent Pattern Compression cache scheme. In this scheme, a cache set is divided into tag area and data area. Obviously, the data area occupies most space of cache memory. So, the data area is compressed by their patterns. So, the data size of an cache line is not a constant anymore, but a variable length. Therefore, in a cache set, for compression, there is more space for data. Which

means that we can store more cache lines in a set than uncompressed cache scheme. And eventually the efficient cache size is increased, even we do not offer more physical cache space. Thus, the miss rate of the cache can be reduced.

Actually, compressing and decompressing operations will cause the latency in accessing cache memory. There are many solutions to decrease this negative influence. And the latency is not this report and experiment's main goal. Thus following discussion is focused on the miss rate.

2 Details

The details of this scheme is described in the paper of Alameldeen, **Adaptive Cache Compression for High-Performance Processors** and the paper of Seungcheol Baek, **ECM:Effective Capacity Maximizer for High-Performance Compressed Caching**. Next, statement about this scheme is stated.

2.1 Frequent Compression Algorithm

There are several compression algorithms in cache and memory design. For examples,IBM's Memory Compression, is a real-time main-memory content compression which can double the main memory capacity without a significant added cost. Also, Frequent-Value-Based Compression and Significant-Based Compression obtained good compressing ratio.

FPC Frequent Pattern Compression (FPC) utilizing such a fact: some data patterns are frequent and can be compressed into a fewer number of bits. For instance, a small integer can be stored with 4-bits, 8-bits or 16-bits, because its high bits may be all 0 or 1 (for a negative integer). But it need to be stored with full 32-bits or 64-bits word. For example integer 1 need to be stored as form 00000000000000000000000000000001 in a 32-bits machine. If we can compress their form, we can increase the cache capacity. In FPC, every cache line is divided into 32-bit words. And if this word accords some pattern, it will be compressed. there are 7 most used patterns: a zero run, 4bits sign-extended, one byte sign-extended, half-word sign-extended, one half-word padded with a zero half word, two byte sign-extended half words and repeat bytes. Other patterns will be classified as uncompressed word. All of these patterns are the based on the frequency in integer and commercial benchmarks.

S-FPC According to the theory and table above, we can know such a fact that a cache line can be compressed to any number of bits. However, high compression ratio means more complexity in cache management. Including high latency of decompression and high access time penalty is the negative influence.

Prefix	Pattern Encode	Data Size
000	zero Run	3bits
001	4-bit sign-extended	4 bits
010	One byte sign-extended	8 bits
011	half word sign-extended	16 bits
100	half word padded with a zero halfword	16 bits
101	Two half words	16 bits
110	Repeated bytes	16 bits
111	Uncompressed word	Original word

Table 1: Frequent Pattern Encoding

So in this scheme, Segment Frequent Compression Algorithm(S-FPC), cache data is stored with groups. Each group is called a segment. A segment is 8 byte, and a cache line size can be 1-8 segments. Small and limited number of segment brings more practical and more quickly accesses.

2.2 Decoupled Variable-Segment Architecture

To achieve compressed cache, the best approach is Decoupled Variable-Segment. For decoupled variable-segment architecture, in a single set, the tag area and the data area is separated, as showed as Figure 1.
For decoupled variable-segment architecture, in a single set, the tag area and the data area is separated, as showed as Figure 1.

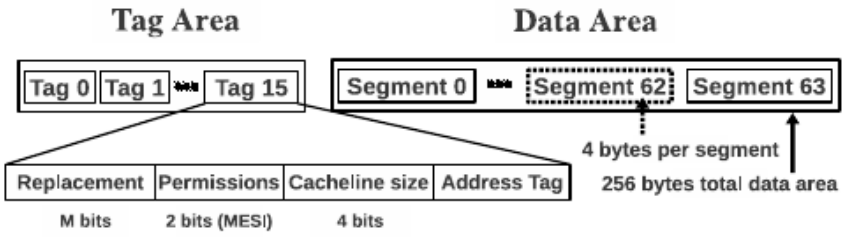

Figure 1: Scheme of Decoupled Variable-Segment Architecture

And to implement the S-FPC, tags in caches have a little difference with the normal one a tag is comprised by four parts. One is Permission, which is state M,S,E or I, which used for cache coherence. Second part is C-status, which represents the status of compression. Third part is the size of compressed cache line corresponding to the tag. The final one is normal tag. And the data area of

a set is also different from the normal data area. The compressed data is stored continuously in data area. And the free space is at the behind of the data area. And another structure of a set is a LRU pool.

2.3 Compression and Decompression

Compression happens when there is a miss happened, the new data will be compressed and take place of the position of old data with LRU replacement policy. The compression units will compress the data from the higher level cache/memory. The compressed cache block size is recorded in the tag. Then, replacement happens both on tag area and data area.

Decompression happens when there is a hit appeared. The units transform the compressed data to uncompressed data according to the prefix of the data.

Actually, the decompression and compression both are time wasting operation. In this report, we only discuss the miss rate and compression ratio.

3 Implementation

3.1 Tools

I write a basic simulation to simulate the Segment Frequent Patter Compression. The implementing tool is Pin of Intel. Pin provide a API and a platform that I can collect the execution information of a running program. Also, using pin, I can insert my instrument function before or after every executed instruction. And there are many great tools of Pin written by others. I can read modified their tools to simulate caches with compression.

The files or tools I choose to modified is "pin_cache.H" and "allcache.cpp". In the pin_cache.H, the common, configurable cache and its behaviors are defined by usingname and class. In allcache.cpp, each level cache are specified according to the definition in pin_cache.H.

What I did is overwrite the class Tag, making it can store the C-Size and C-Status. Also, I create a class defining a LRU cache. In this class, I wrote the functions (or called methods) to implement the "Replace", "Find", "updateLRU" and so on. The details of some of these methods is stated as following:

1. Class:myTag
   In this class, I use several variables to represent the C-size C-status and tag

2. Tag area and LRU pool
   I use a array to present the tag area named _que, another array _lru to represent the LRU pool. The size of them equals the maxassociativity of this set. The max associativity is determined by the normal associativity and the max compression ratio of a cache line. In _lru[i], it store the priority level of corresponding _que[i]. When a replacement, find the position with max value. Which means that it is the LRU element. Delete it and insert the new element.

3. Replace method
   An important operation in cache is accessing. The CACHE class defines how to access to a single set. And I define how to replace specific cache line in a single set. When a access is called, and Find() function is called. The Replace(tag) will find or clean the space for this tag and data. Then writing them into this cache set.

3.2 Processing

In this part, I will describe the route of a access. When a access happens in running program, the Pin platform will copy the memory address accessed and pass it into allcache. Then, this address will be divided into tag and set index and offset. The set index decides which set it should enter, the tag decide whether it hits. A hit happens when the Find() function return a integer other than 0. After that, updateLRU() is called to update the LRU pool. If Find() function return -1 which means a miss happens, the compress() function is called to visit the address and calculate the compressed size of the data in the new blocks and return the size. The Replace() function finish its job as I described above.

3.3 Configuration

In allcache, we can change the cache size and cache line size with convenience. We just need to change the configuration and recompile the allcache.cpp. The default configurations I used is listed below:

1. L1: uncompressed separated cache: Instruction cache and Data cache with 32B cache line, 32KB cache size, 32 associativity.

2. L2:compressed unified cache: 64B cache line, 2MB cache size, normal 4 associativity.

3. Segment size: 8byte

4. Variable configuration: L1 cache size :32KB, 16KB, 8KB; L2 cache size: 2MB, 1MB,512KB: L2 cache line size : 16B,32B, 64B

5. Benchmarks: gcc, gzip, linpack

6. Machine: Linux x86-64 Ubuntu 16.04

4 Experiment Results

First of all, maybe because of the protection mechanism, when we using compress(), only the space of the address, which be passed by Pin, can be access by my program. So I cannot get the content of every address of the cache line. So I can only use the pattern of one address to represent the pattern of whole cache line. So the result may be not obviously difference from the original one. But it still reflects the tendency.

Compared with the miss rate between normal directed-mapped cache, the miss rate of S-FPC has better performance. Reason is obviously, compression increase the capacity of a cache. It should be less miss rate in S-FPC cache. Figure2

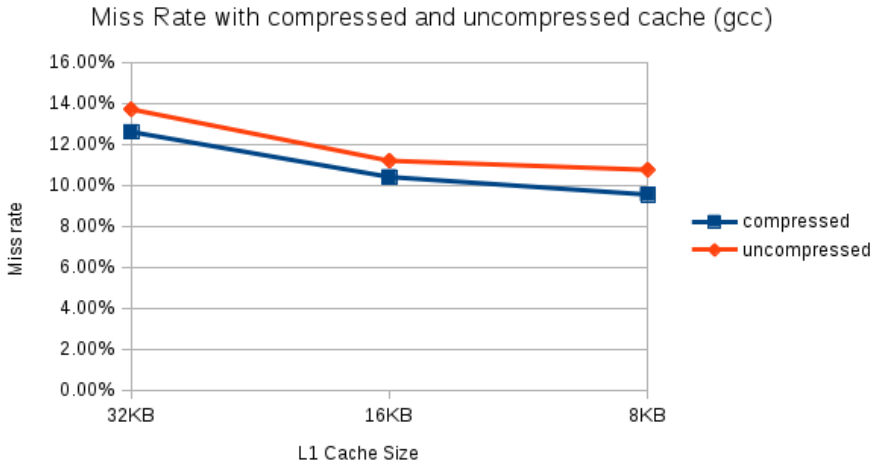

Figure 2: Miss rate of Compressed and uncompressed cache, GCC

As L1 cache size increasing, the number of accessing L2 cache is decreased because most address hit in the L1 cache. But the miss numbers of L2 is stable. So the miss rate is reduced. Thus, the number of compression happened in L2 is stable too. The efficient cache size of L2 (represented with compress ratio) is almost not change.Figure3

Changing the cache line size of L2, we can seen compression ratio increase with the scaling of L2 cache line size. The reason can be explained by the following reason. The larger cache line means more segment in a cache line. More segments leads to a result that a cache line is divided into more parts. So, larger compression ratio is allowed. Figure4

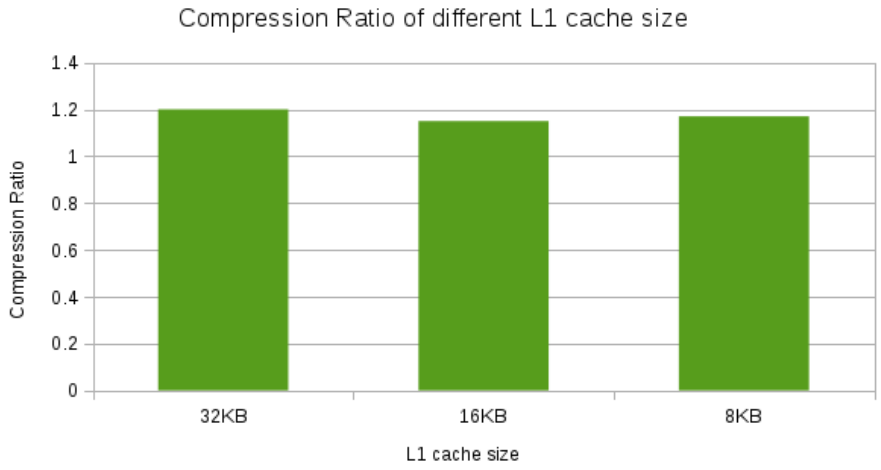

Figure 3: Compression Ratio of different L1 cache size, GCC

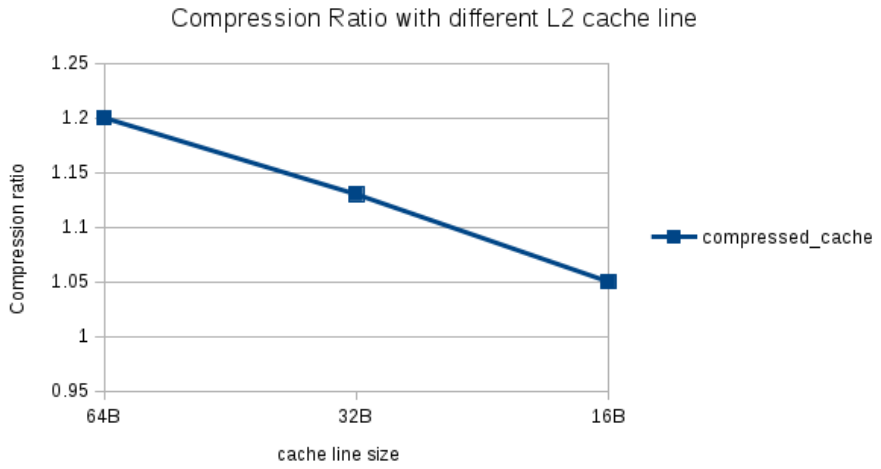

Figure 4: Compression Ratio of l2 with different cache line size l1:32B, DHRYSTONE

If we modify the cache size of L2, the compression ratio seems not change. Because the compression is the behavior of a single cache line. The total cache size has just little influence on compression.Figure5

And we can compare the performance on different benchmark. The performance compression ratio on gcc is better than linpack. It may because that gcc has

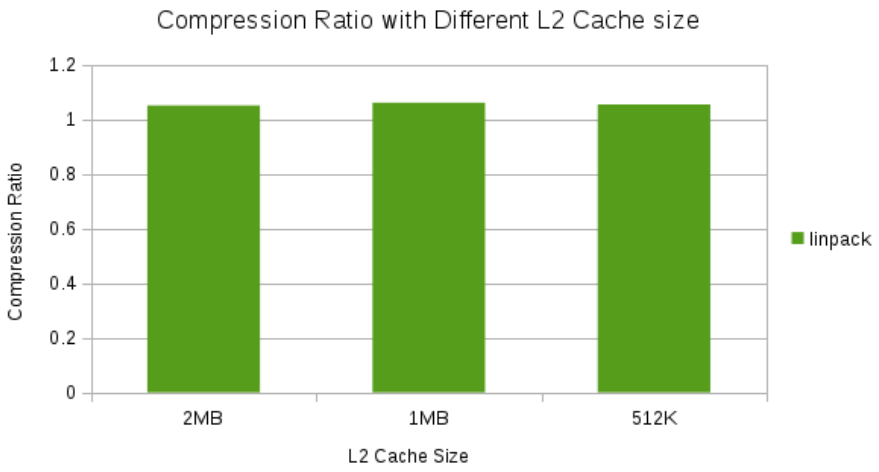

Figure 5: Compression Ratio with Different L2 Cache size, LINPACK

more frequent pattern or has more shorter pattern. Both of this factors can benefits the compression ratio. Dhrystone is a string and integer benchmark so it also has better performance than linpack.Figure6

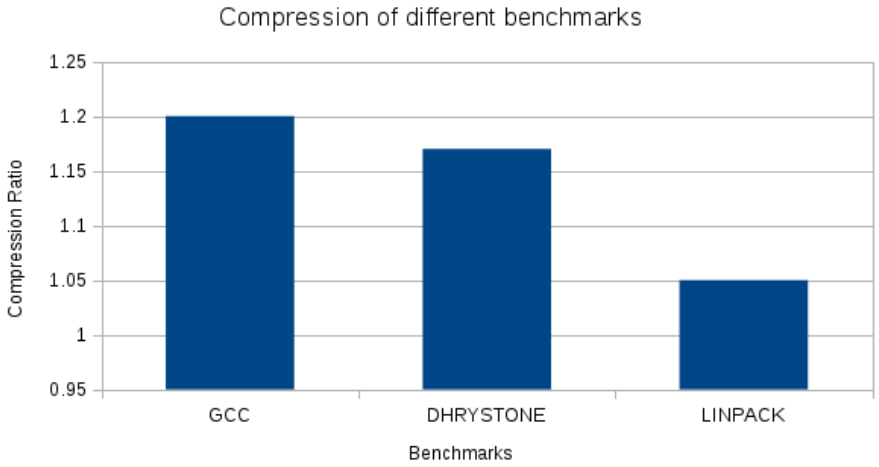

Figure 6: Compression Ratio of Different Benchmarks

5 Conclusion

The compression cache can compress the cache line size according to the frequent pattern, and indirectly increase the efficient cache size. Through this way, compressed cache can obtain lower miss rate. The compressed cache hierarchy is partly implemented with Pin tools. Also, comparing the result on different benchmarks, configurations, I find the factors, which influencing the performance mostly, is the size of cache line and benchmark.

6 References

Alameldeen A R, Wood D A. Adaptive cache compression for high-performance processors[C]//Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on. IEEE, 2004: 212-223.

Franaszek P, Robinson J, Thomas J. Parallel compression with cooperative dictionary construction[C]//Data Compression Conference, 1996. DCC'96. Proceedings. IEEE, 1996: 200-209.

Ahn E, Yoo S M, Kang S M S. Effective algorithms for cache-level compression[C]//Proceedings of the 11th Great Lakes symposium on VLSI. ACM, 2001: 89-92.

Appendix I

Proposal

Cache compression can improve cache performance, through increase effective cache capacity and eliminate misses. But decompression can also increase latency of cache access. To utilize the cache space more efficiently and reduce the miss rate, a compressed cache hierarchy is proposed. In this hierarchy, L1 caches are normal uncompressed caches and L2 cache is a compressed cache. The technology used in L2 is called Decoupled Variable-Segment. The decoupled variable-segment cache allows L2 cache pack more compressed cache lines than uncompressed lines into the same space. Also, a low-latency algorithm Frequent Pattern Compression (FPC) is used to compress the cache lines. The purpose of this experiment is implement the compressed cache hierarchy using FPC on a simulator, and comparing it with normal uncompressed cache for specific benchmarks.

Tools

In order to implement the compressed cache hierarchy on a simulator, the Pin tools may be used. The Pin tool "allcache" and header "cache.H" will be modified to implement our cache hierarchy. The simulator will be run on the COE system on x86-64 virtual machine. To test the performance of cache compressing, several benchmarks will be evaluated. The benchmark may be chosen from SPECcpu2000, Dhrystone and other benchmarks.

Experiments

The number of hits, number of misses, and hit rate on uncompressed cache and compressed cache will be measured. Also, the efficient cache size of compressed cache will be observed. At least three benchmarks will be tested. All of this data will be produced through simulator on Pin. And benchmarks will be run on the following configurations:

1. Varied L1 cache size
2. Varied L2 cache size
3. Varied L2 cache block size

The replacement of this cache is LRU.

7 Result

A Principle of compressed cache scheme and the algorithm should be introduced. Four benchmarks are run. All nine configurations are applied.

Miss rate and the compression ratio are demonstrated. All result are graphed. Analysis and the explanation of result are made and the result need to be discussed in detail.

A^- Two benchmarks are run. Nine configurations are applied. Both miss rate and the efficient cache size are demonstrated. Also, result need to be graphed, analyzed and explained in detail.

B^+ In result, only the miss rate is reported and discussed. Two benchmarks are run but only three configurations are discussed.

B Only one benchmark is run, and one configuration is used to figure out the benefits of compressed caches.

C Just implement the compressed cache hierarchy.

In addition, if enough time is available, the time latency of the compressed caches may be tested.

References

Alameldeen A R, Wood D A. Adaptive cache compression for high-performance processors[C]//Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on. IEEE, 2004: 212-223.

Franaszek P, Robinson J, Thomas J. Parallel compression with cooperative dictionary construction[C]//Data Compression Conference, 1996. DCC'96. Proceedings. IEEE, 1996: 200-209.

Ahn E, Yoo S M, Kang S M S. Effective algorithms for cache-level compression[C]//Proceedings of the 11th Great Lakes symposium on VLSI. ACM, 2001: 89-92.

Appendix II

pin_cache.H

```
/*BEGIN_LEGAL
Intel Open Source License

Copyright (c) 2002-2016 Intel Corporation. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.  Redistributions
in binary form must reproduce the above copyright notice, this list of
conditions and the following disclaimer in the documentation and/or
other materials provided with the distribution.  Neither the name of
the Intel Corporation nor the names of its contributors may be used to
endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE INTEL OR
ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
END_LEGAL */
//
// @ORIGINAL_AUTHOR: Artur Klauser
//

/*! @file
*  This file contains a configurable cache class
*/

#ifndef PIN_CACHE_H
#define PIN_CACHE_H

#include <string>
```

```
43 #include <vector>
44
45 #include "pin_util.H"
46
47 /*!
48 *  @brief Checks if n is a power of 2.
49 *  @returns true if n is power of 2
50 */
51 static inline bool IsPower2(UINT32 n)
52 {
53 return ((n & (n - 1)) == 0);
54 }
55
56 /*!
57 *  @brief Computes floor(log2(n))
58 *  Works by finding position of MSB set.
59 *  @returns -1 if n == 0.
60 */
61 static inline INT32 FloorLog2(UINT32 n)
62 {
63 INT32 p = 0;
64
65 if (n == 0) return -1;
66
67 if (n & 0xffff0000) { p += 16; n >>= 16; }
68 if (n & 0x0000ff00)        { p +=  8; n >>=  8; }
69 if (n & 0x000000f0) { p +=  4; n >>=  4; }
70 if (n & 0x0000000c) { p +=  2; n >>=  2; }
71 if (n & 0x00000002) { p +=  1; }
72
73 return p;
74 }
75
76 /*!
77 *  @brief Computes floor(log2(n))
78 *  Works by finding position of MSB set.
79 *  @returns -1 if n == 0.
80 */
81 static inline INT32 CeilLog2(UINT32 n)
82 {
83 return FloorLog2(n - 1) + 1;
84 }
85
86 /*!
87 *  @brief Cache tag - self clearing on creation
88 */
```

```
89  class CACHE_TAG
90  {
91  private:
92  ADDRINT _tag;
93
94  public:
95  CACHE_TAG(ADDRINT tag = 0) { _tag = tag; }
96  bool operator==(const CACHE_TAG &right) const { return _tag == right._tag; }
97  operator ADDRINT() const { return _tag; }
98  };
99  //! @my tag class
100
101 class myTag
102 {
103 private:
104 ADDRINT _tag;
105 int _cs;//0:uncompressed,1:compressed
106 UINT32 _csize;//unit: segment
107 public:
108 int cs(){return _cs;}
109 UINT32 csize(){return _csize;}
110 myTag(ADDRINT tag = 0,int state = 0,int csize = 0){_tag=tag;_csize=csize;_cs=state;}
111 //myTag(myTag t){this->_tag=(ADDRINT)t;this->_csize=t.csize();this->_cs=t.cs();this->_start=
112 myTag(CACHE_TAG tag){_tag=tag;_csize=8;_cs=0;}//initialization
113
114 bool operator==(const myTag &right)const{return _tag==right._tag;}//overwrite operator ==
115 operator ADDRINT()  const{return _tag;}// overwrite type caster operator (ADDRINT)
116
117
118 void setcs(int cs){_cs=cs;}
119 void setcsize(UINT32 csize){_csize=csize;}
120
121 };
122
123 /*!
124 * Everything related to cache sets
125 */
126 namespace CACHE_SET
127 {
128
129 /*!
130 *  @brief Cache set direct mapped
131 */
132 class DIRECT_MAPPED
133 {
134 private:
```

```
135 CACHE_TAG _tag;
136
137 public:
138 DIRECT_MAPPED(UINT32 associativity = 1) { ASSERTX(associativity == 1); }
139
140 VOID SetAssociativity(UINT32 associativity) { ASSERTX(associativity == 1); }
141 UINT32 GetAssociativity(UINT32 associativity) { return 1; }
142
143 UINT32 Find(CACHE_TAG tag) { return(_tag == tag); }
144 VOID Replace(CACHE_TAG tag) { _tag = tag; }
145 VOID Flush() { _tag = 0; }
146 };
147
148 /*!
149 *  @brief Cache set with round robin replacement
150 */
151 template <UINT32 MAX_ASSOCIATIVITY = 4>
152 class ROUND_ROBIN
153 {
154 private:
155 CACHE_TAG _tags[MAX_ASSOCIATIVITY];
156 UINT32 _tagsLastIndex;
157 UINT32 _nextReplaceIndex;
158
159 public:
160 ROUND_ROBIN(UINT32 associativity = MAX_ASSOCIATIVITY)
161 : _tagsLastIndex(associativity - 1)
162 {
163 ASSERTX(associativity <= MAX_ASSOCIATIVITY);
164 _nextReplaceIndex = _tagsLastIndex;
165
166 for (INT32 index = _tagsLastIndex; index >= 0; index--)
167 {
168 _tags[index] = CACHE_TAG(0);
169 }
170 }
171
172 VOID SetAssociativity(UINT32 associativity)
173 {
174 ASSERTX(associativity <= MAX_ASSOCIATIVITY);
175 _tagsLastIndex = associativity - 1;
176 _nextReplaceIndex = _tagsLastIndex;
177 }
178 UINT32 GetAssociativity(UINT32 associativity) { return _tagsLastIndex + 1; }
179
180 UINT32 Find(CACHE_TAG tag)
```

```
{
bool result = true;

for (INT32 index = _tagsLastIndex; index >= 0; index--)
{
// this is an ugly micro-optimization, but it does cause a
// tighter assembly loop for ARM that way ...
if(_tags[index] == tag) goto end;
}
result = false;

end: return result;
}

VOID Replace(CACHE_TAG tag)
{
// g++ -O3 too dumb to do CSE on following lines?!
const UINT32 index = _nextReplaceIndex;

_tags[index] = tag;
// condition typically faster than modulo
_nextReplaceIndex = (index == 0 ? _tagsLastIndex : index - 1);
}
VOID Flush()
{
for (INT32 index = _tagsLastIndex; index >= 0; index--)
{
_tags[index] = 0;
}
_nextReplaceIndex=_tagsLastIndex;
}
};









template<UINT32 ASSOCIATIVITY = 4, UINT32 BLOCKSIZE = 64>
class LRU
{
private:
UINT32 _Max_Associativity;
int _num_segment;//data area of this set, every segment has 8byte space,
myTag _que[8*ASSOCIATIVITY];
```

```
227 int _lru[8*ASSOCIATIVITY];
228 UINT32 _restspace;
229 UINT32 _Block_size;//byte1
230 UINT32 _segsize;//byte
231 UINT32 _Associativity;
232
233 public:
234
235 LRU()
236 {
237 _Block_size=BLOCKSIZE;
238 _restspace=32;
239 _Associativity=ASSOCIATIVITY;
240 _segsize=8;//byte
241 _Max_Associativity = 8*_Associativity;
242 _num_segment=_Associativity*_Block_size/_segsize;
243 for(UINT32 i=0;i<_Max_Associativity;i++){
244 _que[i]=myTag();
245 }
246 for(UINT32 i=0;i<_Max_Associativity;i++){
247 _lru[i]=0;
248 }
249 }
250
251 VOID SetAssociativity(UINT32 associativity) { return; }
252 UINT32 GetAssociativity(UINT32 associativity) { return _Associativity; }
253
254
255
256 int Find(myTag tag){
257 for(UINT32 i=0;i<_Max_Associativity;i++){
258 if (tag==_que[i]){
259 return (int)i;
260 }
261 }
262 return -1;
263 }
264
265 int Find(){
266 for(UINT32 i=0;i<_Max_Associativity;i++){
267 if (0==_que[i]){
268 return (int)i;
269 }
270 }
271 return -1;
272 }
```

```
273
274 void updateLRU(UINT32 p){
275 for(int i=0;(UINT32)i<_Max_Associativity;i++){
276 if(_lru[i]!=0&&_lru[i]<_lru[p]) _lru[i]++;
277 }
278 _lru[p]=1;
279
280
281
282 }
283
284 int findLRU(){
285 int Max=0;
286 for(int i=0;(UINT32)i<_Max_Associativity;i++){
287 if(_lru[i]>_lru[Max]) Max=i;
288 }
289 return Max;
290
291 }
292
293
294 bool FindSpace(myTag tag)  //find the position that can insert new tag
295 {
296
297 return _restspace>tag.csize();
298
299 }
300
301 UINT32 compress(ADDRINT addr){//return the coompressed cache line length
302
303 UINT32 actuallSize=0;//bits
304 UINT32 *p=(UINT32 *)addr;
305 for(UINT32 i=0;i<2;i++,(p+=_Block_size/4))
306 {
307
308 UINT32 t1=*p;
309
310 if((t1>>4)==0||(~t1>>4)==0) actuallSize+=4;//4 bits extent
311 else if((t1>>8)==0||(~t1>>8)==0) actuallSize+=8;//8 bits extent
312 else if((t1>>16)==0||(~t1>>16)==0) actuallSize+=16;// half word extent
313
314
315
316 else actuallSize+=32;
317 }
318
```

```
319
320 //return actuallSize/64;//transform the bits to segment
321
322 return actuallSize/32*_num_segment;
323 //return 8;
324
325 }
326
327
328
329
330 void Replace(ADDRINT tag, ADDRINT addr)//version0.1 only consider the situation not hit
331 {
332
333 myTag newtag = myTag(tag);
334 newtag.setcsize(compress(addr));
335 newtag.setcs(newtag.csize()==8?1:0);
336 int p=0;
337 while(!FindSpace(newtag)){
338 p=findLRU();
339 _restspace+=_que[p].csize();
340 _que[p]=myTag();
341 _lru[p]=0;
342 }
343 if(FindSpace(newtag)) p=Find();
344 _que[p]=newtag;
345 _restspace-=newtag.csize();
346 updateLRU(p);
347 return;
348 }
349 //when a write hit happen but the new value has the different size, it need to be replaced
350 void hitReplace(myTag &tag)
351 {
352
353 int p = Find(tag);
354 updateLRU(p);
355 return;
356 }
357
358
359 };
360
361 } // namespace CACHE_SET
362
363 namespace CACHE_ALLOC
364 {
```

```
365 typedef enum
366 {
367 STORE_ALLOCATE,
368 STORE_NO_ALLOCATE
369 } STORE_ALLOCATION;
370 }
371
372 /*!
373 *  @brief Generic cache base class; no allocate specialization, no cache set specialization
374 */
375 class CACHE_BASE
376 {
377 public:
378 // types, constants
379 typedef enum
380 {
381 ACCESS_TYPE_LOAD,
382 ACCESS_TYPE_STORE,
383 ACCESS_TYPE_NUM
384 } ACCESS_TYPE;
385
386 protected:
387 static const UINT32 HIT_MISS_NUM = 2;
388 CACHE_STATS _access[ACCESS_TYPE_NUM][HIT_MISS_NUM];
389
390 private:
391 // input params
392 const std::string _name;
393 const UINT32 _cacheSize;
394 const UINT32 _lineSize;
395 const UINT32 _associativity;
396 UINT32 _numberOfFlushes;
397 UINT32 _numberOfResets;
398
399 // computed params
400 const UINT32 _lineShift;
401 const UINT32 _setIndexMask;
402
403 CACHE_STATS SumAccess(bool hit) const
404 {
405 CACHE_STATS sum = 0;
406
407 for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType++)
408 {
409 sum += _access[accessType][hit];
410 }
```

```
411
412 return sum;
413 }
414
415 protected:
416 UINT32 NumSets() const { return _setIndexMask + 1; }
417
418 public:
419 // constructors/destructors
420 CACHE_BASE(std::string name, UINT32 cacheSize, UINT32 lineSize, UINT32 associativity);
421
422 // accessors
423 UINT32 CacheSize() const { return _cacheSize; }
424 UINT32 LineSize() const { return _lineSize; }
425 UINT32 Associativity() const { return _associativity; }
426 //
427 CACHE_STATS Hits(ACCESS_TYPE accessType) const { return _access[accessType][true];}
428 CACHE_STATS Misses(ACCESS_TYPE accessType) const { return _access[accessType][false];}
429 CACHE_STATS Accesses(ACCESS_TYPE accessType) const { return Hits(accessType) + Misses(access
430 CACHE_STATS Hits() const { return SumAccess(true);}
431 CACHE_STATS Misses() const { return SumAccess(false);}
432 CACHE_STATS Accesses() const { return Hits() + Misses();}
433
434 CACHE_STATS Flushes() const { return _numberOfFlushes;}
435 CACHE_STATS Resets() const { return _numberOfResets;}
436
437 VOID SplitAddress(const ADDRINT addr, CACHE_TAG & tag, UINT32 & setIndex) const
438 {
439 tag = addr >> _lineShift;
440 setIndex = tag & _setIndexMask;
441 }
442
443 VOID SplitAddress(const ADDRINT addr, CACHE_TAG & tag, UINT32 & setIndex, UINT32 & lineIndex
444 {
445 const UINT32 lineMask = _lineSize - 1;
446 lineIndex = addr & lineMask;
447 SplitAddress(addr, tag, setIndex);
448 }
449
450 VOID IncFlushCounter()
451 {
452 _numberOfFlushes += 1;
453 }
454
455 VOID IncResetCounter()
456 {
```

```
457 _numberOfResets += 1;
458 }
459
460 std::ostream & StatsLong(std::ostream & out) const;
461 };
462
463 CACHE_BASE::CACHE_BASE(std::string name, UINT32 cacheSize, UINT32 lineSize, UINT32 associati
464 : _name(name),
465 _cacheSize(cacheSize),
466 _lineSize(lineSize),
467 _associativity(associativity),
468 _lineShift(FloorLog2(lineSize)),
469 _setIndexMask((cacheSize / (associativity * lineSize)) - 1)
470 {
471
472 ASSERTX(IsPower2(_lineSize));
473 ASSERTX(IsPower2(_setIndexMask + 1));
474
475 for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType++)
476 {
477 _access[accessType][false] = 0;
478 _access[accessType][true] = 0;
479 }
480 }
481
482 /*!
483 *  @brief Stats output method
484 */
485 std::ostream & CACHE_BASE::StatsLong(std::ostream & out) const
486 {
487 const UINT32 headerWidth = 19;
488 const UINT32 numberWidth = 10;
489
490 out << _name << ":" << std::endl;
491
492 for (UINT32 i = 0; i < ACCESS_TYPE_NUM; i++)
493 {
494 const ACCESS_TYPE accessType = ACCESS_TYPE(i);
495
496 std::string type(accessType == ACCESS_TYPE_LOAD ? "Load" : "Store");
497
498 out << StringString(type + " Hits:      ", headerWidth)
499 << StringInt(Hits(accessType), numberWidth) << std::endl;
500 out << StringString(type + " Misses:    ", headerWidth)
501 << StringInt(Misses(accessType), numberWidth) << std::endl;
502 out << StringString(type + " Accesses:  ", headerWidth)
```

```
503   << StringInt(Accesses(accessType), numberWidth) << std::endl;
504   out << StringString(type + " Miss Rate: ", headerWidth)
505   << StringFlt(100.0 * Misses(accessType) / Accesses(accessType), 2, numberWidth-1) << "%" <<
506
507   out << std::endl;
508   }
509
510   out << StringString("Total Hits:      ", headerWidth, ' ')
511   << StringInt(Hits(), numberWidth) << std::endl;
512   out << StringString("Total Misses:    ", headerWidth, ' ')
513   << StringInt(Misses(), numberWidth) << std::endl;
514   out << StringString("Total Accesses:  ", headerWidth, ' ')
515   << StringInt(Accesses(), numberWidth) << std::endl;
516   out << StringString("Total Miss Rate: ", headerWidth, ' ')
517   << StringFlt(100.0 * Misses() / Accesses(), 2, numberWidth-1) << "%" << std::endl;
518
519   out << StringString("Flushes:         ", headerWidth, ' ')
520   << StringInt(Flushes(), numberWidth) << std::endl;
521   out << StringString("Stat Resets:     ", headerWidth, ' ')
522   << StringInt(Resets(), numberWidth) << std::endl;
523   out << std::endl;
524   return out;
525   }
526
527   /// ostream operator for CACHE_BASE
528   std::ostream & operator<< (std::ostream & out, const CACHE_BASE & cacheBase)
529   {
530   return cacheBase.StatsLong(out);
531   }
532
533   /*!
534   *  @brief Templated cache class with specific cache set allocation policies
535   *
536   *  All that remains to be done here is allocate and deallocate the right
537   *  type of cache sets.
538   */
539   template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
540   class CACHE : public CACHE_BASE
541   {
542   private:
543   SET _sets[MAX_SETS];
544
545   public:
546   // constructors/destructors
547   CACHE(std::string name, UINT32 cacheSize, UINT32 lineSize, UINT32 associativity)
548   : CACHE_BASE(name, cacheSize, lineSize, associativity)
```

```
549 {
550 ASSERTX(NumSets() <= MAX_SETS);
551
552 for (UINT32 i = 0; i < NumSets(); i++)
553 {
554 _sets[i].SetAssociativity(associativity);
555 }
556 }
557
558 // modifiers
559 /// Cache access from addr to addr+size-1
560 bool Access(ADDRINT addr, UINT32 size, ACCESS_TYPE accessType);
561 /// Cache access at addr that does not span cache lines
562 bool AccessSingleLine(ADDRINT addr, ACCESS_TYPE accessType);
563 bool SpecialAccess(ADDRINT addr, UINT32 size, ACCESS_TYPE accessType);
564 bool SpecialAccessSingleLine(ADDRINT addr, ACCESS_TYPE accessType);
565 void Flush();
566 void ResetStats();
567 //UINT32 geteSize();
568 };
569 /*
570 template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
571 UINT32 CACHE<SET,MAX_SETS,STORE_ALLOCATION>::geteSize(){
572 UINT32 eff=0;
573 for (UINT32 i=0;i<MAX_SETS;i++){
574 eff+=_sets[i].getSize();
575 }
576 return eff;
577 }
578
579
580
581
582
583 */
584 /*!
585 *  @return true if all accessed cache lines hit
586 */
587 template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
588 bool CACHE<SET,MAX_SETS,STORE_ALLOCATION>::Access(ADDRINT addr, UINT32 size, ACCESS_TYPE acc
589 {
590 const ADDRINT highAddr = addr + size;
591 bool allHit = true;
592
593 const ADDRINT lineSize = LineSize();
594 const ADDRINT notLineMask = ~(lineSize - 1);
```

```
595 do
596 {
597 CACHE_TAG tag;
598 UINT32 setIndex;
599
600 SplitAddress(addr, tag, setIndex);
601
602 SET & set = _sets[setIndex];
603
604 bool localHit = set.Find(tag);
605 allHit &= localHit;
606
607 // on miss, loads always allocate, stores optionally
608 if ( (! localHit) && (accessType == ACCESS_TYPE_LOAD || STORE_ALLOCATION == CACHE_ALLOC::STO
609 {
610 set.Replace(tag);
611 }
612
613 addr = (addr & notLineMask) + lineSize; // start of next cache line
614 }
615 while (addr < highAddr);
616
617 _access[accessType][allHit]++;
618
619 return allHit;
620 }
621
622 /*!
623 *  @return true if accessed cache line hits
624 */
625 template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
626 bool CACHE<SET,MAX_SETS,STORE_ALLOCATION>::AccessSingleLine(ADDRINT addr, ACCESS_TYPE access
627 {
628 CACHE_TAG tag;
629 UINT32 setIndex;
630
631 SplitAddress(addr, tag, setIndex);
632
633 SET & set = _sets[setIndex];
634
635 bool hit = set.Find(tag);
636
637 // on miss, loads always allocate, stores optionally
638 if ( (! hit) && (accessType == ACCESS_TYPE_LOAD || STORE_ALLOCATION == CACHE_ALLOC::STORE_AL
639 {
640 set.Replace(tag);
```

```
641 }
642
643 _access[accessType][hit]++;
644
645 return hit;
646 }
647 /*!
648 *  @return true if accessed cache line hits
649 */
650 template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
651 void CACHE<SET,MAX_SETS,STORE_ALLOCATION>::Flush()
652 {
653 for (INT32 index = NumSets(); index >= 0; index--) {
654 SET & set = _sets[index];
655 set.Flush();
656 }
657 IncFlushCounter();
658 }
659
660 template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
661 void CACHE<SET,MAX_SETS,STORE_ALLOCATION>::ResetStats()
662 {
663 for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType++)
664 {
665 _access[accessType][false] = 0;
666 _access[accessType][true] = 0;
667 }
668 IncResetCounter();
669 }
670
671 //!    @my access method
672 template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
673 bool CACHE<SET,MAX_SETS,STORE_ALLOCATION>::SpecialAccess(ADDRINT addr, UINT32 size, ACCESS_T
674 {
675 const ADDRINT highAddr = addr + size;
676 bool allHit = true;
677
678 const ADDRINT lineSize = LineSize();
679 const ADDRINT notLineMask = ~(lineSize - 1);
680 do
681 {
682 CACHE_TAG temptag;
683 UINT32 setIndex;
684
685 SplitAddress(addr, temptag, setIndex);
686
```

```
687  SET & set = _sets[setIndex];
688  myTag tag=myTag(temptag);
689  bool localHit = (set.Find(tag)!=-1);
690  allHit &= localHit;
691
692  // on miss, loads always allocate, stores optionally
693  if ( (! localHit)/* && (accessType == ACCESS_TYPE_LOAD || STORE_ALLOCATION == CACHE_ALLOC::S
694  {
695  set.Replace(tag,addr);
696  }
697  else if(localHit){
698  set.hitReplace(tag);
699  }
700  addr = (addr & notLineMask) + lineSize; // start of next cache line
701  }
702  while (addr < highAddr);
703
704  _access[accessType][allHit]++;
705
706  return allHit;
707  }
708
709  /*!
710  *  @return true if accessed cache line hits
711  */
712  template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
713  bool CACHE<SET,MAX_SETS,STORE_ALLOCATION>::SpecialAccessSingleLine(ADDRINT addr, ACCESS_TYPE
714  {
715  CACHE_TAG temptag;
716  UINT32 setIndex;
717
718  SplitAddress(addr, temptag, setIndex);
719  myTag tag=myTag(temptag);
720  SET & set = _sets[setIndex];
721
722  bool hit = (set.Find(tag)!=-1);
723
724  // on miss, loads always allocate, stores optionally
725  if ( (! hit) /*&& (accessType == ACCESS_TYPE_LOAD || STORE_ALLOCATION == CACHE_ALLOC::STORE_
726  {
727  set.Replace(tag,addr);
728  }
729  else if(hit){
730  set.hitReplace(tag);
731  }
732  _access[accessType][hit]++;
```

```
733
734 return hit;
735 }
736
737 // define shortcuts
738 #define CACHE_DIRECT_MAPPED(MAX_SETS, ALLOCATION) CACHE<CACHE_SET::DIRECT_MAPPED, MAX_SETS,
739 #define CACHE_ROUND_ROBIN(MAX_SETS, MAX_ASSOCIATIVITY, ALLOCATION) CACHE<CACHE_SET::ROUND_R
740 #define CACHE_LRU(MAX_SETS,ASSOCIATIVITY,BLOCKSIZE,ALLOCATION) CACHE<CACHE_SET::LRU<ASSOCIA
741 #endif // PIN_CACHE_H
```

Appendix III

```
/*BEGIN_LEGAL
Intel Open Source License

Copyright (c) 2002-2016 Intel Corporation. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.  Redistributions
in binary form must reproduce the above copyright notice, this list of
conditions and the following disclaimer in the documentation and/or
other materials provided with the distribution.  Neither the name of
the Intel Corporation nor the names of its contributors may be used to
endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE INTEL OR
ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
END_LEGAL */
//
// @ORIGINAL_AUTHOR: Artur Klauser
//

/*! @file
*  This file contains an ISA-portable PIN tool for functional simulation of
*  instruction+data TLB+cache hieraries
*/

#include <iostream>

#include "pin.H"

typedef UINT32 CACHE_STATS; // type of cache hit/miss counters
```

```
45
46  #include "pin_cache.H"
47
48
49  namespace IL1
50  {
51  // 1st level instruction cache: 32 kB, 32 B lines, 32-way associative
52  const UINT32 cacheSize = 32*KILO;
53  const UINT32 lineSize = 32;
54  const UINT32 associativity = 32;
55  const CACHE_ALLOC::STORE_ALLOCATION allocation = CACHE_ALLOC::STORE_NO_ALLOCATE;
56
57  const UINT32 max_sets = cacheSize / (lineSize * associativity);
58  const UINT32 max_associativity = associativity;
59
60  typedef CACHE_ROUND_ROBIN(max_sets, max_associativity, allocation) CACHE;
61  }
62  LOCALVAR IL1::CACHE il1("L1 Instruction Cache", IL1::cacheSize, IL1::lineSize, IL1::associat
63
64
65  namespace DL1
66  {
67  // 1st level data cache: 32 kB, 32 B lines, 32-way associative
68  const UINT32 cacheSize = 32*KILO;
69  const UINT32 lineSize = 32;
70  const UINT32 associativity = 32;
71  const CACHE_ALLOC::STORE_ALLOCATION allocation = CACHE_ALLOC::STORE_NO_ALLOCATE;
72
73  const UINT32 max_sets = cacheSize / (lineSize * associativity);
74  const UINT32 max_associativity = associativity;
75
76  typedef CACHE_ROUND_ROBIN(max_sets, max_associativity, allocation) CACHE;
77  }
78  LOCALVAR DL1::CACHE dl1("L1 Data Cache", DL1::cacheSize, DL1::lineSize, DL1::associativity);
79
80
81
82
83  namespace UL2
84  {
85  // 2nd level unified cache: 2 MB, 64 B lines, 4 way compressed
86  const UINT32 cacheSize = 2*MEGA;
87  const UINT32 lineSize = 64;
88  const UINT32 associativity = 4;
89  const CACHE_ALLOC::STORE_ALLOCATION allocation = CACHE_ALLOC::STORE_ALLOCATE;
90
```

```
91  const UINT32 max_sets = cacheSize / (lineSize * associativity);
92
93
94  typedef CACHE_LRU(max_sets,associativity,lineSize,allocation) CACHE;
95  }
96  LOCALVAR UL2::CACHE ul2("L2 Unified Compressed Cache", UL2::cacheSize, UL2::lineSize, UL2::a
97
98  LOCALFUN VOID Fini(int code, VOID * v)
99  {
100 //std::cerr << itlb;
101 //std::cerr << dtlb;
102 std::cout << il1;
103 std::cout << dl1;
104 std::cout << ul2;
105 //std::cout << ul2.geteSize()*UL2::lineSize<<"KB"<<endl;
106 }
107
108 LOCALFUN VOID Ul2Access(ADDRINT addr, UINT32 size, CACHE_BASE::ACCESS_TYPE accessType)
109 {
110 // second level unified cache
111 //if(CACHE_BASE::)
112 ul2.SpecialAccess(addr, size, accessType);
113
114 }
115
116 LOCALFUN VOID Ul2AccessSingleLine(ADDRINT addr, CACHE_BASE::ACCESS_TYPE accessType)
117 {
118 // second level unified cache
119 //if(CACHE_BASE::)
120 ul2.SpecialAccessSingleLine(addr, accessType);
121
122
123 }
124
125
126
127 LOCALFUN VOID InsRef(ADDRINT addr)
128 {
129 const UINT32 size = 1; // assuming access does not cross cache lines
130 const CACHE_BASE::ACCESS_TYPE accessType = CACHE_BASE::ACCESS_TYPE_LOAD;
131
132 // ITLB
133 //itlb.AccessSingleLine(addr, accessType);
134
135 // first level I-cache
136 const BOOL il1Hit = il1.AccessSingleLine(addr, accessType);
```

```
137
138 // second level unified Cache
139 if ( ! il1Hit) Ul2Access(addr, size, accessType);
140
141 }
142
143
144
145
146 LOCALFUN VOID MemRefMulti(ADDRINT addr, UINT32 size, CACHE_BASE::ACCESS_TYPE accessType)
147 {
148 // DTLB
149 // dtlb.SpecialAccessSingleLine(addr, CACHE_BASE::ACCESS_TYPE_LOAD);
150
151 // first level D-cache
152 const BOOL dl1Hit = dl1.Access(addr, size, accessType);
153
154 // second level unified Cache
155 if ( ! dl1Hit) Ul2Access(addr, size, accessType);
156 }
157
158 LOCALFUN VOID MemRefSingle(ADDRINT addr, UINT32 size, CACHE_BASE::ACCESS_TYPE accessType)
159 {
160 // DTLB
161 //dtlb.AccessSingleLine(addr, CACHE_BASE::ACCESS_TYPE_LOAD);
162
163 // first level D-cache
164 const BOOL dl1Hit = dl1.AccessSingleLine(addr, accessType);
165
166 // second level unified Cache
167 if ( ! dl1Hit) Ul2AccessSingleLine(addr, accessType);
168 }
169
170 LOCALFUN VOID Instruction(INS ins, VOID *v)
171 {
172 // all instruction fetches access I-cache
173 INS_InsertCall(
174 ins, IPOINT_BEFORE, (AFUNPTR)InsRef,
175 IARG_INST_PTR,
176 IARG_END);
177
178 if (INS_IsMemoryRead(ins) && INS_IsStandardMemop(ins))
179 {
180 const UINT32 size = INS_MemoryReadSize(ins);
181 const AFUNPTR countFun = (size <= 4 ? (AFUNPTR) MemRefSingle : (AFUNPTR) MemRefMulti);
182
```

```
183  // only predicated-on memory instructions access D-cache
184  INS_InsertPredicatedCall(
185  ins, IPOINT_BEFORE, countFun,
186  IARG_MEMORYREAD_EA,
187  IARG_MEMORYREAD_SIZE,
188  IARG_UINT32, CACHE_BASE::ACCESS_TYPE_LOAD,
189  IARG_END);
190  }
191
192
193  if (INS_IsMemoryWrite(ins) && INS_IsStandardMemop(ins))
194  {
195  const UINT32 size = INS_MemoryWriteSize(ins);
196  const AFUNPTR countFun = (size <= 4 ? (AFUNPTR) MemRefSingle : (AFUNPTR) MemRefMulti);
197
198  // only predicated-on memory instructions access D-cache
199  INS_InsertPredicatedCall(
200  ins, IPOINT_BEFORE, countFun,
201  IARG_MEMORYWRITE_EA,
202  IARG_MEMORYWRITE_SIZE,
203  IARG_UINT32, CACHE_BASE::ACCESS_TYPE_STORE,
204  IARG_END);
205  }
206  }
207
208  GLOBALFUN int main(int argc, char *argv[])
209  {
210  PIN_Init(argc, argv);
211
212  INS_AddInstrumentFunction(Instruction, 0);
213  PIN_AddFiniFunction(Fini, 0);
214
215  // Never returns
216  PIN_StartProgram();
217
218  return 0; // make compiler happy
219  }
```