```
//
// @ORIGINAL_AUTHOR: Artur Klauser
//

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

#ifndef PIN_CACHE_H
#define PIN_CACHE_H


#define KILO 1024
#define MEGA (KILO*KILO)
#define GIGA (KILO*MEGA)

typedef UINT64 CACHE_STATS; // type of cache hit/miss counters

#include <sstream>

/*! RMR (rodric@gmail.com)
 *   - temporary work around because decstr()
 *     casts 64 bit ints to 32 bit ones
 */
static string mydecstr(UINT64 v, UINT32 w)
{
    ostringstream o;
    o.width(w);
    o << v;
    string str(o.str());
    return str;
}

/*!
 *  @brief Checks if n is a power of 2.
 *  @returns true if n is power of 2
 */
static inline bool IsPower2(UINT32 n)
{
```

```
    return ((n & (n - 1)) == 0);
}

/*!
 *  @brief Computes floor(log2(n))
 *  Works by finding position of MSB set.
 *  @returns -1 if n == 0.
 */
static inline INT32 FloorLog2(UINT32 n)
{
    INT32 p = 0;

    if (n == 0) return -1;

    if (n & 0xffff0000) { p += 16; n >>= 16; }
    if (n & 0x0000ff00) { p +=  8; n >>=  8; }
    if (n & 0x000000f0) { p +=  4; n >>=  4; }
    if (n & 0x0000000c) { p +=  2; n >>=  2; }
    if (n & 0x00000002) { p +=  1; }

    return p;
}

/*!
 *  @brief Computes floor(log2(n))
 *  Works by finding position of MSB set.
 *  @returns -1 if n == 0.
 */
static inline INT32 CeilLog2(UINT32 n)
{
    return FloorLog2(n - 1) + 1;
}

/*!
 *  @brief Cache tag - self clearing on creation
 */
class CACHE_TAG
{
  private:
    ADDRINT _tag;

  public:
    CACHE_TAG(ADDRINT tag = 0) { _tag = tag; }
    bool operator==(const CACHE_TAG &right) const { return _tag == right._tag; }
    operator ADDRINT() const { return _tag; }
};


/*!
 * Everything related to cache sets
 */
namespace CACHE_SET
{

/*!
 *  @brief Cache set direct mapped
 */
class DIRECT_MAPPED
{
  private:
    CACHE_TAG _tag;

  public:
    DIRECT_MAPPED(UINT32 associativity = 1) { ASSERTX(associativity == 1); }

    VOID SetAssociativity(UINT32 associativity) { ASSERTX(associativity == 1); }
    UINT32 GetAssociativity(UINT32 associativity) { return 1; }

    UINT32 Find(CACHE_TAG tag) { return(_tag == tag); }
    VOID Replace(CACHE_TAG tag) { _tag = tag; }
};

/*!
 *  @brief Cache set with round robin replacement
 */
template <UINT32 MAX_ASSOCIATIVITY = 4>
class ROUND_ROBIN
{
  private:
    CACHE_TAG _tags[MAX_ASSOCIATIVITY];
    UINT32 _tagsLastIndex;
    UINT32 _nextReplaceIndex;

  public:
    ROUND_ROBIN(UINT32 associativity = MAX_ASSOCIATIVITY)
      : _tagsLastIndex(associativity - 1)
    {
        ASSERTX(associativity <= MAX_ASSOCIATIVITY);
        _nextReplaceIndex = _tagsLastIndex;

        for (INT32 index = _tagsLastIndex; index >= 0; index--)
        {
            _tags[index] = CACHE_TAG(0);
        }
    }

    VOID SetAssociativity(UINT32 associativity)
    {
        ASSERTX(associativity <= MAX_ASSOCIATIVITY);
        _tagsLastIndex = associativity - 1;
        _nextReplaceIndex = _tagsLastIndex;
    }
    UINT32 GetAssociativity(UINT32 associativity) { return _tagsLastIndex + 1; }

    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);
    }
};

} // namespace CACHE_SET

namespace CACHE_ALLOC
{
    typedef enum
    {
        STORE_ALLOCATE,
        STORE_NO_ALLOCATE
```

```
    } STORE_ALLOCATION;
}

/*!
 *  @brief Generic cache base class; no allocate specialization, no cache set sp
ecialization
 */
class CACHE_BASE
{
  public:
    // types, constants
    typedef enum
    {
        ACCESS_TYPE_LOAD,
        ACCESS_TYPE_STORE,
        ACCESS_TYPE_NUM
    } ACCESS_TYPE;

    typedef enum
    {
        CACHE_TYPE_ICACHE,
        CACHE_TYPE_DCACHE,
        CACHE_TYPE_NUM
    } CACHE_TYPE;

  protected:
    static const UINT32 HIT_MISS_NUM = 2;
    CACHE_STATS _access[ACCESS_TYPE_NUM][HIT_MISS_NUM];

  private:    // input params
    const std::string _name;
    const UINT32 _cacheSize;
    const UINT32 _lineSize;
    const UINT32 _associativity;

    // computed params
    const UINT32 _lineShift;
    const UINT32 _setIndexMask;

    CACHE_STATS SumAccess(bool hit) const
    {
        CACHE_STATS sum = 0;

        for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType++)
        {
            sum += _access[accessType][hit];
        }

        return sum;
    }

  protected:
    UINT32 NumSets() const { return _setIndexMask + 1; }

  public:
    // constructors/destructors
    CACHE_BASE(std::string name, UINT32 cacheSize, UINT32 lineSize, UINT32 assoc
iativity);

    // accessors
    UINT32 CacheSize() const { return _cacheSize; }
    UINT32 LineSize() const { return _lineSize; }
    UINT32 Associativity() const { return _associativity; }
    //
    CACHE_STATS Hits(ACCESS_TYPE accessType) const { return _access[accessType][
true];}
    CACHE_STATS Misses(ACCESS_TYPE accessType) const { return _access[accessType
][false];}
    CACHE_STATS Accesses(ACCESS_TYPE accessType) const { return Hits(accessType)
 + Misses(accessType);}
    CACHE_STATS Hits() const { return SumAccess(true);}
    CACHE_STATS Misses() const { return SumAccess(false);}
    CACHE_STATS Accesses() const { return Hits() + Misses();}

    VOID SplitAddress(const ADDRINT addr, CACHE_TAG & tag, UINT32 & setIndex) co
nst
    {
        tag = addr >> _lineShift;
        setIndex = tag & _setIndexMask;
    }

    VOID SplitAddress(const ADDRINT addr, CACHE_TAG & tag, UINT32 & setIndex, UI
NT32 & lineIndex) const
    {
        const UINT32 lineMask = _lineSize - 1;
        lineIndex = addr & lineMask;
        SplitAddress(addr, tag, setIndex);
    }

    string StatsLong(string prefix = "", CACHE_TYPE = CACHE_TYPE_DCACHE) const;
};

CACHE_BASE::CACHE_BASE(std::string name, UINT32 cacheSize, UINT32 lineSize, UINT
32 associativity)
  : _name(name),
    _cacheSize(cacheSize),
    _lineSize(lineSize),
    _associativity(associativity),
    _lineShift(FloorLog2(lineSize)),
    _setIndexMask((cacheSize / (associativity * lineSize)) - 1)
{

    ASSERTX(IsPower2(_lineSize));
    ASSERTX(IsPower2(_setIndexMask + 1));

    for (UINT32 accessType = 0; accessType < ACCESS_TYPE_NUM; accessType++)
    {
        _access[accessType][false] = 0;
        _access[accessType][true] = 0;
    }
}

/*!
 *  @brief Stats output method
 */

string CACHE_BASE::StatsLong(string prefix, CACHE_TYPE cache_type) const
{
    const UINT32 headerWidth = 19;
    const UINT32 numberWidth = 12;

    string out;

    out += prefix + _name + ":" + "\n";

    if (cache_type != CACHE_TYPE_ICACHE) {
       for (UINT32 i = 0; i < ACCESS_TYPE_NUM; i++)
       {
           const ACCESS_TYPE accessType = ACCESS_TYPE(i);

           std::string type(accessType == ACCESS_TYPE_LOAD ? "Load" : "Store");

           out += prefix + ljstr(type + "-Hits:   ", headerWidth)
                  + mydecstr(Hits(accessType), numberWidth)  +
                  " " +fltstr(100.0 * Hits(accessType) / Accesses(accessType), 2
, 6) + "%\n";

           out += prefix + ljstr(type + "-Misses:  ", headerWidth)
```

```
                  + mydecstr(Misses(accessType), numberWidth) +
                  " " +fltstr(100.0 * Misses(accessType) / Accesses(accessType),
 2, 6) + "%\n";

           out += prefix + ljstr(type + "-Accesses: ", headerWidth)
                  + mydecstr(Accesses(accessType), numberWidth) +
                  " " +fltstr(100.0 * Accesses(accessType) / Accesses(accessType
), 2, 6) + "%\n";

           out += prefix + "\n";
       }
    }

    out += prefix + ljstr("Total-Hits:   ", headerWidth)
           + mydecstr(Hits(), numberWidth) +
           " " +fltstr(100.0 * Hits() / Accesses(), 2, 6) + "%\n";

    out += prefix + ljstr("Total-Misses:  ", headerWidth)
           + mydecstr(Misses(), numberWidth) +
           " " +fltstr(100.0 * Misses() / Accesses(), 2, 6) + "%\n";

    out += prefix + ljstr("Total-Accesses: ", headerWidth)
           + mydecstr(Accesses(), numberWidth) +
           " " +fltstr(100.0 * Accesses() / Accesses(), 2, 6) + "%\n";
    out += "\n";

    return out;
}


/*!
 *  @brief Templated cache class with specific cache set allocation policies
 *
 *  All that remains to be done here is allocate and deallocate the right
 *  type of cache sets.
 */
template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
class CACHE : public CACHE_BASE
{
  private:
    SET _sets[MAX_SETS];

  public:
    // constructors/destructors
    CACHE(std::string name, UINT32 cacheSize, UINT32 lineSize, UINT32 associativ
ity)
      : CACHE_BASE(name, cacheSize, lineSize, associativity)
    {
        ASSERTX(NumSets() <= MAX_SETS);

        for (UINT32 i = 0; i < NumSets(); i++)
        {
            _sets[i].SetAssociativity(associativity);
        }
    }

    // modifiers
    /// Cache access from addr to addr+size-1
    bool Access(ADDRINT addr, UINT32 size, ACCESS_TYPE accessType);
    /// Cache access at addr that does not span cache lines
    bool AccessSingleLine(ADDRINT addr, ACCESS_TYPE accessType);
};

/*!
 *  @return true if all accessed cache lines hit
 */

template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
bool CACHE<SET,MAX_SETS,STORE_ALLOCATION>::Access(ADDRINT addr, UINT32 size, ACC
ESS_TYPE accessType)
{
    const ADDRINT highAddr = addr + size;
    bool allHit = true;

    const ADDRINT lineSize = LineSize();
    const ADDRINT notLineMask = ~(lineSize - 1);
    do
    {
        CACHE_TAG tag;
        UINT32 setIndex;

        SplitAddress(addr, tag, setIndex);

        SET & set = _sets[setIndex];

        bool localHit = set.Find(tag);
        allHit &= localHit;

        // on miss, loads always allocate, stores optionally
        if ( (! localHit) && (accessType == ACCESS_TYPE_LOAD || STORE_ALLOCATION
 == CACHE_ALLOC::STORE_ALLOCATE))
        {
            set.Replace(tag);
        }

        addr = (addr & notLineMask) + lineSize; // start of next cache line
    }
    while (addr < highAddr);

    _access[accessType][allHit]++;

    return allHit;
}

/*!
 *  @return true if accessed cache line hits
 */
template <class SET, UINT32 MAX_SETS, UINT32 STORE_ALLOCATION>
bool CACHE<SET,MAX_SETS,STORE_ALLOCATION>::AccessSingleLine(ADDRINT addr, ACCESS
_TYPE accessType)
{
    CACHE_TAG tag;
    UINT32 setIndex;

    SplitAddress(addr, tag, setIndex);

    SET & set = _sets[setIndex];

    bool hit = set.Find(tag);

    // on miss, loads always allocate, stores optionally
    if ( (! hit) && (accessType == ACCESS_TYPE_LOAD || STORE_ALLOCATION == CACHE
_ALLOC::STORE_ALLOCATE))
    {
        set.Replace(tag);
    }

    _access[accessType][hit]++;

    return hit;
}

// define shortcuts
#define CACHE_DIRECT_MAPPED(MAX_SETS, ALLOCATION) CACHE<CACHE_SET::DIRECT_MAPPED
, MAX_SETS, ALLOCATION>
#define CACHE_ROUND_ROBIN(MAX_SETS, MAX_ASSOCIATIVITY, ALLOCATION) CACHE<CACHE_S
ET::ROUND_ROBIN<MAX_ASSOCIATIVITY>, MAX_SETS, ALLOCATION>
```

```
#endif // PIN_CACHE_H
```

```
/*BEGIN_LEGAL
Intel Open Source License

Copyright (c) 2002-2011 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
// @EXTENDED: Rodric Rabbah (rodric@gmail.com)
//

/*! @file
 *  This file contains an ISA-portable cache simulator
 *  data cache hierarchies
 */


#include "pin.H"

#include <iostream>
#include <fstream>

#include "dcache.H"
#include "pin_profile.H"


/* ===================================================================== */
/* Commandline Switches */
/* ===================================================================== */

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE,    "pintool",
    "o", "dcache.out", "specify dcache file name");
KNOB<BOOL>   KnobTrackLoads(KNOB_MODE_WRITEONCE,    "pintool",
    "tl", "0", "track individual loads -- increases profiling time");
KNOB<BOOL>   KnobTrackStores(KNOB_MODE_WRITEONCE,   "pintool",
   "ts", "0", "track individual stores -- increases profiling time");
KNOB<UINT32> KnobThresholdHit(KNOB_MODE_WRITEONCE , "pintool",
   "rh", "100", "only report memops with hit count above threshold");
KNOB<UINT32> KnobThresholdMiss(KNOB_MODE_WRITEONCE, "pintool",
   "rm","100", "only report memops with miss count above threshold");
KNOB<UINT32> KnobCacheSize(KNOB_MODE_WRITEONCE, "pintool",
    "c","32", "cache size in kilobytes");
KNOB<UINT32> KnobLineSize(KNOB_MODE_WRITEONCE, "pintool",
    "b","32", "cache block size in bytes");
KNOB<UINT32> KnobAssociativity(KNOB_MODE_WRITEONCE, "pintool",
```

```
    "a","4", "cache associativity (1 for direct mapped)");

/* ===================================================================== */
/* Print Help Message                                                    */
/* ===================================================================== */

INT32 Usage()
{
    cerr <<
        "This tool represents a cache simulator.\n"
        "\n";

    cerr << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===================================================================== */
/* Global Variables */
/* ===================================================================== */

// wrap configuation constants into their own name space to avoid name clashes
namespace DL1
{
    const UINT32 max_sets = KILO; // cacheSize / (lineSize * associativity);
    const UINT32 max_associativity = 256; // associativity;
    const CACHE_ALLOC::STORE_ALLOCATION allocation = CACHE_ALLOC::STORE_ALLOCATE
;

    typedef CACHE_ROUND_ROBIN(max_sets, max_associativity, allocation) CACHE;
}

DL1::CACHE* dl1 = NULL;

typedef enum
{
    COUNTER_MISS = 0,
    COUNTER_HIT = 1,
    COUNTER_NUM
} COUNTER;



typedef  COUNTER_ARRAY<UINT64, COUNTER_NUM> COUNTER_HIT_MISS;


// holds the counters with misses and hits
// conceptually this is an array indexed by instruction address
COMPRESSOR_COUNTER<ADDRINT, UINT32, COUNTER_HIT_MISS> profile;

/* ===================================================================== */

VOID LoadMulti(ADDRINT addr, UINT32 size, UINT32 instId)
{
    // first level D-cache
    const BOOL dl1Hit = dl1->Access(addr, size, CACHE_BASE::ACCESS_TYPE_LOAD);

    const COUNTER counter = dl1Hit ? COUNTER_HIT : COUNTER_MISS;
    profile[instId][counter]++;
}

/* ===================================================================== */

VOID StoreMulti(ADDRINT addr, UINT32 size, UINT32 instId)
{
    // first level D-cache
    const BOOL dl1Hit = dl1->Access(addr, size, CACHE_BASE::ACCESS_TYPE_STORE);

    const COUNTER counter = dl1Hit ? COUNTER_HIT : COUNTER_MISS;
    profile[instId][counter]++;
```

```
}

/* ===================================================================== */

VOID LoadSingle(ADDRINT addr, UINT32 instId)
{
    // @todo we may access several cache lines for
    // first level D-cache
    const BOOL dl1Hit = dl1->AccessSingleLine(addr, CACHE_BASE::ACCESS_TYPE_LOAD
);

    const COUNTER counter = dl1Hit ? COUNTER_HIT : COUNTER_MISS;
    profile[instId][counter]++;
}
/* ===================================================================== */

VOID StoreSingle(ADDRINT addr, UINT32 instId)
{
    // @todo we may access several cache lines for
    // first level D-cache
    const BOOL dl1Hit = dl1->AccessSingleLine(addr, CACHE_BASE::ACCESS_TYPE_STOR
E);

    const COUNTER counter = dl1Hit ? COUNTER_HIT : COUNTER_MISS;
    profile[instId][counter]++;
}

/* ===================================================================== */

VOID LoadMultiFast(ADDRINT addr, UINT32 size)
{
    dl1->Access(addr, size, CACHE_BASE::ACCESS_TYPE_LOAD);
}

/* ===================================================================== */

VOID StoreMultiFast(ADDRINT addr, UINT32 size)
{
    dl1->Access(addr, size, CACHE_BASE::ACCESS_TYPE_STORE);
}

/* ===================================================================== */

VOID LoadSingleFast(ADDRINT addr)
{
    dl1->AccessSingleLine(addr, CACHE_BASE::ACCESS_TYPE_LOAD);
}

/* ===================================================================== */

VOID StoreSingleFast(ADDRINT addr)
{
    dl1->AccessSingleLine(addr, CACHE_BASE::ACCESS_TYPE_STORE);
}



/* ===================================================================== */

VOID Instruction(INS ins, void * v)
{
    if (INS_IsMemoryRead(ins))
    {
        // map sparse INS addresses to dense IDs
        const ADDRINT iaddr = INS_Address(ins);
        const UINT32 instId = profile.Map(iaddr);

        const UINT32 size = INS_MemoryReadSize(ins);
        const BOOL   single = (size <= 4);
```

```

        if( KnobTrackLoads )
        {
            if( single )
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE, (AFUNPTR) LoadSingle,
                    IARG_MEMORYREAD_EA,
                    IARG_UINT32, instId,
                    IARG_END);
            }
            else
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE,  (AFUNPTR) LoadMulti,
                    IARG_MEMORYREAD_EA,
                    IARG_MEMORYREAD_SIZE,
                    IARG_UINT32, instId,
                    IARG_END);
            }

        }
        else
        {
            if( single )
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE,  (AFUNPTR) LoadSingleFast,
                    IARG_MEMORYREAD_EA,
                    IARG_END);

            }
            else
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE,  (AFUNPTR) LoadMultiFast,
                    IARG_MEMORYREAD_EA,
                    IARG_MEMORYREAD_SIZE,
                    IARG_END);
            }
        }
    }

    if ( INS_IsMemoryWrite(ins) )
    {
        // map sparse INS addresses to dense IDs
        const ADDRINT iaddr = INS_Address(ins);
        const UINT32 instId = profile.Map(iaddr);

        const UINT32 size = INS_MemoryWriteSize(ins);

        const BOOL   single = (size <= 4);

        if( KnobTrackStores )
        {
            if( single )
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE,  (AFUNPTR) StoreSingle,
                    IARG_MEMORYWRITE_EA,
                    IARG_UINT32, instId,
                    IARG_END);
            }
            else
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE,  (AFUNPTR) StoreMulti,
                    IARG_MEMORYWRITE_EA,
                    IARG_MEMORYWRITE_SIZE,
                    IARG_UINT32, instId,
                    IARG_END);
            }

        }
        else
        {
            if( single )
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE,  (AFUNPTR) StoreSingleFast,
                    IARG_MEMORYWRITE_EA,
                    IARG_END);

            }
            else
            {
                INS_InsertPredicatedCall(
                    ins, IPOINT_BEFORE,  (AFUNPTR) StoreMultiFast,
                    IARG_MEMORYWRITE_EA,
                    IARG_MEMORYWRITE_SIZE,
                    IARG_END);
            }
        }

    }
}

/* ===================================================================== */

VOID Fini(int code, VOID * v)
{

    std::ofstream out(KnobOutputFile.Value().c_str());

    // print D-cache profile
    // @todo what does this print

    out << "PIN:MEMLATENCIES 1.0. 0x0\n";

    out <<
        "#\n"
        "# DCACHE stats\n"
        "#\n";

    out << dl1->StatsLong("# ", CACHE_BASE::CACHE_TYPE_DCACHE);

    if( KnobTrackLoads || KnobTrackStores ) {
        out <<
            "#\n"
            "# LOAD stats\n"
            "#\n";

        out << profile.StringLong();
    }
    out.close();
}

/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */

int main(int argc, char *argv[])
{
    PIN_InitSymbols();

    if( PIN_Init(argc,argv) )
    {
        return Usage();
```

```
    }

    dl1 = new DL1::CACHE("L1 Data Cache",
                         KnobCacheSize.Value() * KILO,
                         KnobLineSize.Value(),
                         KnobAssociativity.Value());

    profile.SetKeyName("iaddr          ");
    profile.SetCounterName("dcache:miss        dcache:hit");

    COUNTER_HIT_MISS threshold;

    threshold[COUNTER_HIT] = KnobThresholdHit.Value();
    threshold[COUNTER_MISS] = KnobThresholdMiss.Value();

    profile.SetThreshold( threshold );

    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);

    // Never returns

    PIN_StartProgram();

    return 0;
}

/* ===================================================================== */
/* eof */
/* ===================================================================== */
```