Software Systems Lab

Just Sharing Organization

Final project report

Written By: Oren Ben Simhon 061279832

Sasha Zusmanovich 310872841

Jonathan Avida 043177609

Supervisor: Zvika Guz

Registration semester: winter 2008-2007

Submition date: April 22, 2009

Table of contents

Introduction

Chip-Multi-processors are rapidly becoming mainstream thanks to their ability to leverage the parallelism of multithreading and multitasking to achieve higher performance within a given power envelope. Consequently, CMPs' architecture includes several processors, and every processor has his own caches (L1, L2 and etc), an architecture that proved to be very efficient.

Nahalal [1] is a new proposed CMP cache architecture that deals with the bottleneck in such systems – the cache data access. Nahalal places the cache of every CPU close to the processing unit, and one shared cache that is closed to all the CPUs. Doing so, Nahalal demonstrates an improved architecture for multi-tasking systems.
The shared cache is a fast L2 memory unit that saves the data that is shared by K CPU's in the same period of time. The algorithm for shared cache management solves the K parameter and finds the average number of CPU's, for which the shared data algorithm will work best.

An important question had been aroused regarding the best size for such caches. How big should the shared cache be? Should all caches be in the same size? Well, those questions were answered by the average working-set size of CPU's. But this is not the main factor for deciding the cache size. Another parameter is the benefit of cache hit-time and cache cost versus the cache size.

Nonetheless, what will happen in a single threaded system? I.e. what will happen when only one process with big working set is running on a Multi Processors system?
When only one task is performed, the caches of the other CPU's remain unused and because the data is not shared across many tasks the shared cache is not used also. We can see that the Nahalal architecture can result in a disturbing lay-back of important resources.

What will happen if the system is occupied by one big task and many small tasks? In this case, again the CPU that process the big task suffers from a lack of fast memory resources.

In CMPs with a private caches, we face a problem when some processors may not use their entire caches due to small working sets, while other processors have working sets larger than their private caches, resulting in poor performance. This problem is especially prevalent in Nahalal, where we have to decide what data to put on the shared cache. Surly, the Nahalal architecture with respect to the shared data allocation algorithm is not ultimate for all test-benches and reconfiguration is required. Thus we propose Just Shared Organization (JSO) algorithm that will allocate the shared cache across CPU's respectably to their working-set.

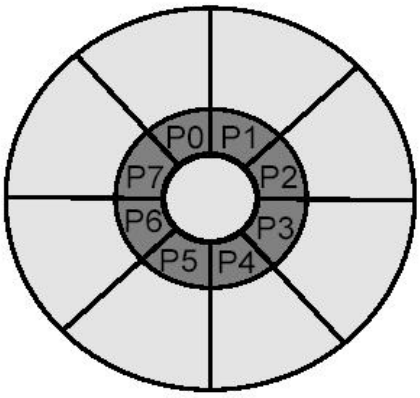
NAHALAL Architecture

Introduction

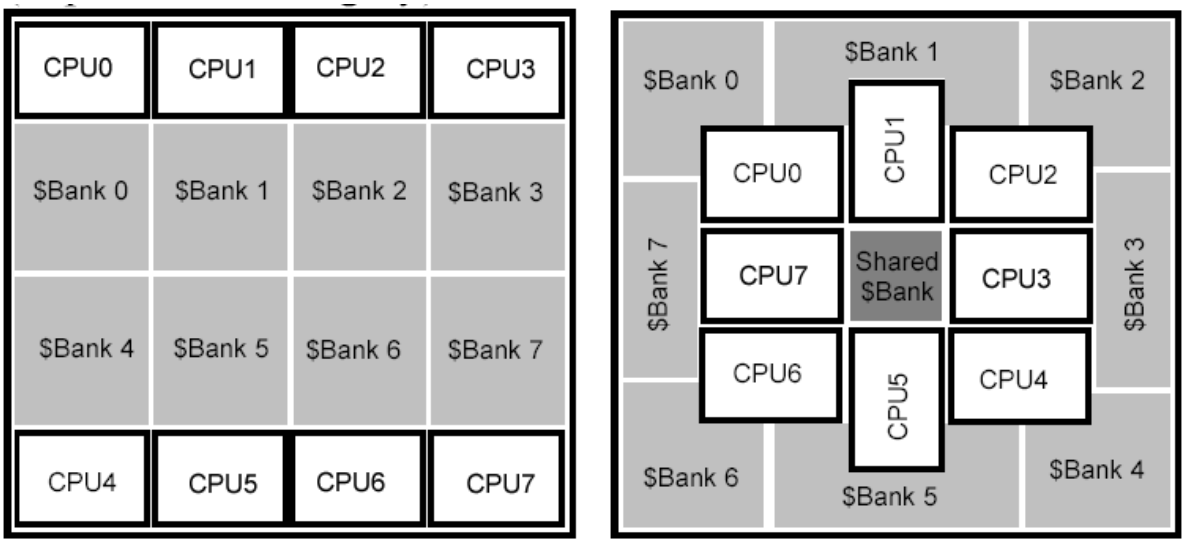
The shift towards Chip Multi Processors makes the on-chip memory system a primary performance bottle neck. This shift calls for a new approach to cache design, in which cache architecture will be tailored and optimized for the multiprocessing environment. Nahalal project proposed partitioning the cache in CMPs according to the level of data sharing. This approach is motivated by the observation that, in many multithreaded applications, a small set of shared cache lines accounts for a significant portion of the memory accesses. Thus Nahalal was presented – a novel CMP cache architecture and floor plan that exhibits shorter access distances to shared data compared to the conventional CMP with cache-in-the-middle (CIM) architecture. In [1] is shown that Nahalal improves average L2 cache access times by up to 41%.

Memory access characterization

In many multithreaded applications, a substantial fraction of the memory accesses involves cache lines that are shared by many processors. Furthermore, in commercial workloads, a significant fraction of memory accesses involve modified-shared data, which cannot be replicated without performance penalty for ensuring cache coherence. Consequently, access to shared data severely penalize the average memory access time and hinder overall performance. These phenomena call for a new cache architecture that explicitly accounts for data sharing.



The proposed solution is Nahalal [1] – a new CMP cache architecture that partitions the L2 cache according to the programs' data sharing, and can thus offer vicinity of reference to both shared and private data. The topology was inspired by the layout of the cooperative village Nahalal, which is based on urban design ideas from the 19th century. The same conceptual layout is reflected to CMP, as schematically illustrated in the above figure. A fraction of the L2 memory is located in the center of the chip, enclosed by all processors, while the rest of the L2 memory is placed on the outer slices. The inner memory is populated by the hottest shared data, allowing fast access by all processors. The outer slices create a "backyard" for each processor.



(a) CIM layout. (b) Nahalal layout.

Nahalal Cache Management

The Nahalal scheme can be implemented via a broad range of potential cache management strategies. The investigated scheme is described below:

1) **Placement and migration –** in both implementations shown above, each address can be located in any of the banks. Thus, cache management needs to decide where to place the line when it is fetched, and subsequently, if and when to migrate a line from its current bank, and to where. In both CIM and Nahalal implementations, on a first line fetch, the line is placed in the bank adjacent to the processor that made the request. In the suggested example of CIM, the line remains in its initial location as long as it is in the cache. In contrast, in Nahalal, they use migration to steer shared-hot-lines to the center.

2) **Search –** In Nahalal, L2 cache lines are likely to be served either from the center (for shared data) or from the local cache structure (for private data). This is contrary to CIM, where shared lines can be located in any of the different banks with equal probability. (Thus, on average, shared lines are fetched from distant cache banks in CIM.) For simplicity, in thier simulations they used a parallel search for both the CIM and Nahalal implementation.

Dynamic Cache Allocation

In a dynamic cache allocation system, the size of the allocated cache for a CPU is proportional to the working-set size of the thread running on that CPU.
We find that Non Uniform Cache Architecture can benefit from Dynamic cache allocation.
In order to calculate the profit of the dynamic allocation, we try to estimate the probability for cache miss while using a static allocation.

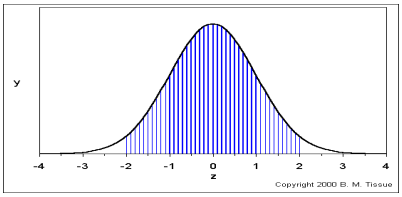
Assumption 1: The replacement algorithm is ideal.
Assumption 2: The architecture has a shared memory unit (cache).
Assumption 3: The sum of all working-sets is smaller than the sum of the cache sizes

In order to find the probability we define the following parameters:

- u – the average working set size
- σ - the variance of the working set size
- m – the cache size
- P – time to find instruction in the main memory (penalty time)
- T – time to search an instruction in the cache

Assuming that the working set has a normal distribution, we would like to consider the following equation:

$$p[Z = \frac{\bar{x} - u}{\sigma} \geq m] = \alpha$$



When α Is the probability to have a working set that do not fit the cache size (M).

Now we can calculate the speed up:

$$CPI_{STATIC} = \alpha \bullet (P + T) + (1 - \alpha) \bullet T$$
$$CPI_{DYNAMIC} = T$$
$$SpeedUp = \frac{CPI_{STATIC}}{CPI_{DYNAMIC}}$$

In order to show the great improve of dynamic allocation we would like to examine common parameters of cache management:

$$\alpha = 0.3$$
$$P = 15 cycles$$
$$T = 3 cycles$$
$$SpeedUp = \frac{0.3 \cdot (15 + 3) + 0.7 \bullet 3}{3} = 2.5$$

We can see that with the right dynamic allocation algorithm we can speedup the data fetching by 2.5.

Let's take for example our architecture. In Nahalal architecture we have a 4MB shared cache that we can reallocate between 8 processors. In a static allocation each CPU has 0.5MB however in dynamic allocation all the cache space can be distributed between the caches (assumption 3). Also we use the typical parameters when the cache hit time it 3 cycles and the miss penalty is 15 cycles.

Let's assume that we have a situation where each CPU has a working set size 0.1MB except for one CPU with working set size 3.3MB.

Now we try to calculate the Speed Up:

$$CPI_{static} = \frac{7 \bullet 3 + \frac{0.5}{3.3} \bullet 3 + \frac{3.3 - 0.5}{3.3} \bullet (15 + 3)}{8} \simeq 4.6$$
$$CPI_{dynamic} = \frac{8 \bullet 3}{8} = 3$$
$$SpeedUp = \frac{CPI_{dynamic}}{CPI_{static}} \simeq 1.53$$

We can see that our dynamic allocation system has a 150% speedup.

Working-Set Calculation

Overview

One of the challenges we confronted during the project is to calculate the working set. We used [2] to estimate the working set in each one of the caches, according to the result, we divided the shared cache.

Introduction

Microprocessors are designed to provide good average performance over a variety of workloads. This can lead to inefficiencies both in power and performance for individual programs and during individual phases within the same program. Micro architectures with multi-configuration units (like caches) are able to adapt dynamically to program behavior and enable/disable resources as needed. A key element of existing configuration algorithms is adjusting to program phase changes. This is typically done by "tuning" when a phase change is detected.

Algorithms that dynamically collect and analyze program working set information are studied. To make this practical, working set signatures are proposed, which are highly compressed working set representations. Algorithms use working set signatures to:

1) Detect working set changes and trigger re-tuning
2) Identify recurring working sets and re-install saved optimal reconfigurations, thus avoiding the time-consuming tuning process
3) Estimate working set sizes to configure caches directly to the proper size, also avoiding the tuning process.

Dynamic reconfiguration algorithms

Reconfiguration algorithms have three basic properties that determine their applicability and effectiveness.

Detection efficiency – the ability of an algorithm to detect program phase changes. Low detection efficiency can lead to lost reconfiguration opportunities and non-optimal hardware configurations.

Reconfiguration overhead – the overhead associated with the transition from one configuration to another. The reconfiguration overhead depends on the amount of state contained in the structure. Flushing and/or re-learning the state can take 10's of cycles to 1000's of cycles for reconfiguration a data cache.

Tuning overhead – the time spent searching for an optimal configuration. A high tuning overhead leads to higher number of reconfigurations and more time spent in the non-optimal configurations. This is a more serious problem in micro architectures with several multi-configuration units like ours.

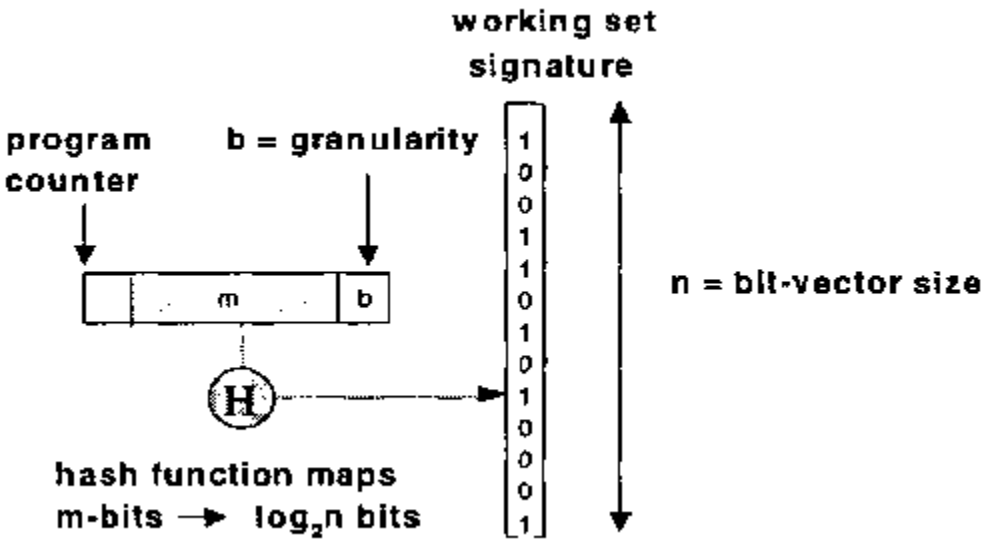
Basic definition

Capturing a working set requires a window. The window size determines the finest granularity at which phases can be resolved. For simplicity, we kept the window size constant and equal to 10k cache samples. In our project, we consider fine grain working sets containing cache line sized elements because we deal with multi-configuration units (e.g. caches) that work at this granularity. Also, for design simplicity, a series of non-overlapping windows is used, rather than a sliding window as is often used in paging studies.

The method of sampling information is an important parameter than we set to 10K. One could, however, resort to periodic sampling or random sampling to reduce sampling overhead.

Working set signatures

Representing and manipulating complete working-sets is impractical because every CPU has a large virtual memory size. Consequently, we propose a lossy-compressed working-set representation that we call the working set signature.



The working set signature is an n-bit vector formed by mapping working set elements into n-buckets using a randomizing hash function. As mentioned before, the working set elements are of cache line granularity and hence the low-order b address bits are ignored when hashing. The size of the bit-vector is 1K in our project. The bit-vector is cleared at the beginning of every interval (window) to remove stale working set information.

Working set signatures can be used to estimate the size, change, and identity attributes of the full working set. The size (number of ones) of the signatures is probabilistically related to the true working set size. When K random keys are hashed into n buckets, the fraction of buckets filled, f, is given by:

$$f = 1 - (1 - \frac{1}{n})^K$$

Given the fraction of the signature filled, the working set size can be estimated using the relation:

$$K = \log(1 - f) / \log(1 - \frac{1}{n})$$

Using this relation, we find that 90% filled table corresponds to a working-set size about 2.5 times larger than the number of filled entries.
In order to hash the cache entries we used a simple module operation.

Summary

We took this concept of working set signature, and used it on our cache model. For simplicity, we made a reconfiguration of the shared cache at the end of every window. By doing so we are aware that our Detection efficiency may be reduced. In future work, one can do the reconfiguration only when the benefit of reallocating shared cache lines (i.e. match a bigger CPU's working set more cache lines to maximize the total cache hit rate) overcomes the cost (i.e. the deletion of other CPUs' allocated and used cache lines). By using the SIMICS application we reduce the tuning overhead to zero and pay the cost of Reconfiguration overhead.

JSO Algorithm

The solution, we propose, is trying to optimize the shared cache benefits by dynamically allocating the resources.
This way we ensure that each CPU will get a just share of the shared cache memory size. Still we kept part of the shared cache for shared data across CPUs. Because the steering of hot-shared-cache-line is not in the scope of our project, we assume that the replacement and sharing algorithm is optimal and in every reconfiguration phase we know the exact size of the shared data. The rest of the shared cache is reallocated between the CPUs according to their working-sets.
We estimate, according to [2], the working set of each CPU and reallocate the shared cache every reconfiguration phase. We decided to create the reconfiguration phase every 10k commands. Further research can optimize the indication on the reconfiguration phase.

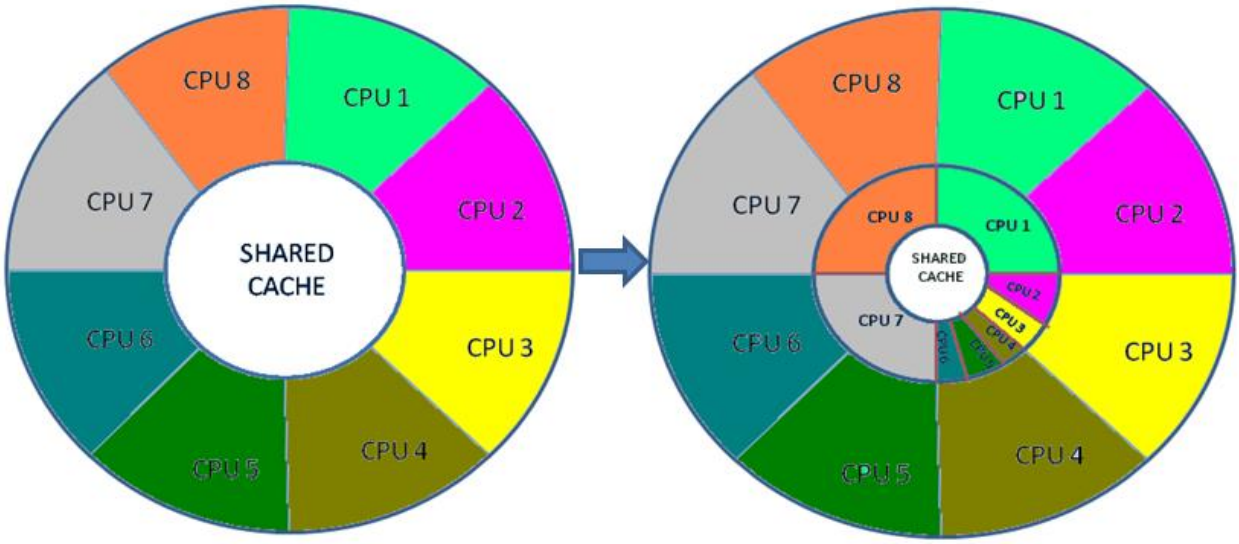
In Every reconfiguration phase we calculate the sum of all the working set. I.e. if we have N CPUs than:

WS_SUM= WS[1]+WS[2] + …. + WS[N]

Each CPU[i] gets as addition a cache with size:

REMAINING_SHARED_CACHE_SIZE * WS[i] / WS_SUM.

In the following diagram you can see the change in the cache organization:



a. Cache allocation in Nahalal without JSO

b. Cache allocation in Nahalal with JSO

About SImics

Glossary

• checkpoint — The state of simulation, saved as a number of files, that can be loaded to continue simulation at the point the checkpoint was saved.

• Command Line Interface—The default Simics command-line (user interface). It uses a simple language implemented in Python, and is variously called the Simics "front end" or the "CLI".

• component — A component is typically the smallest hardware unit that can be used when configuring a real machine, and examples include motherboards, PCI cards, hard disks, and backplanes. Components are usually implemented in Simics using several configuration objects.

• configuration—A configuration is a description of a target architecture, and is loaded into Simics with the read-configuration command. Note that a configuration can also include the state of the target, and saved from within Simics using the write-configuration command, in which case it provides a portable checkpoint facility.

• cycle — The smallest unit of time in Simics. When using Simics in its default mode, the cycle count is usually the same as the step count, but this can be changed in various ways to improve the fidelity of the simulation.

• event — A Simics event occurs at some predefined point of simulated time. Time can be specified either as a number of steps on a simulated processor, or a number of simulated clock cycles.

• host — The machine the simulator (Simics) is running on.

• module — A dynamically linked library or a script that interfaces to Simics and extends the functionality of the simulator. A module is either an extension or a device.

• Simics console—The Simics console is a text console where you can issue commands to Simics, and where Simics will display status information, log messages, and printout from issued commands. The Simics console is available in all Simics user interfaces but in slightly different versions.

• step — An issued instruction that completes or causes an exception, or an external interrupt.

• system level instruction set simulation — The effect of every single instruction is simulated both on user and supervisor level. At any instruction, the simulation can be stopped and state can be inspected (and changed).

• target— The simulated machine.

Introduction

Simics [3] is a system level instruction set simulator i.e. Simics can simulate a virtual machine and run a set of instructions on it. The simulated machine is called the Target while the computer on which Simics run called the host. The instruction set is target specific, and can emulate a hardware target that doesn't exist.

Simics allows developers to simulate new hardware components that don't exist and to simulate embedded systems in which the software and hardware are developed at the same time and burned into the hardware together.

The simulator runs on the operating system of the host by observing machine commands and executes them step by step. Simics provides a platform for running in the lowest level after the hardware level of the host.
The simulator can save states of the memory configuration and collect configurable statistics on the steps and transaction that took place on the target.
The execution of the simulator is controlled through the simulator console window. In the window we can run the simulator a step by step or perform trace on the transactions. We can also debug the system and run Python scripts through the console window.

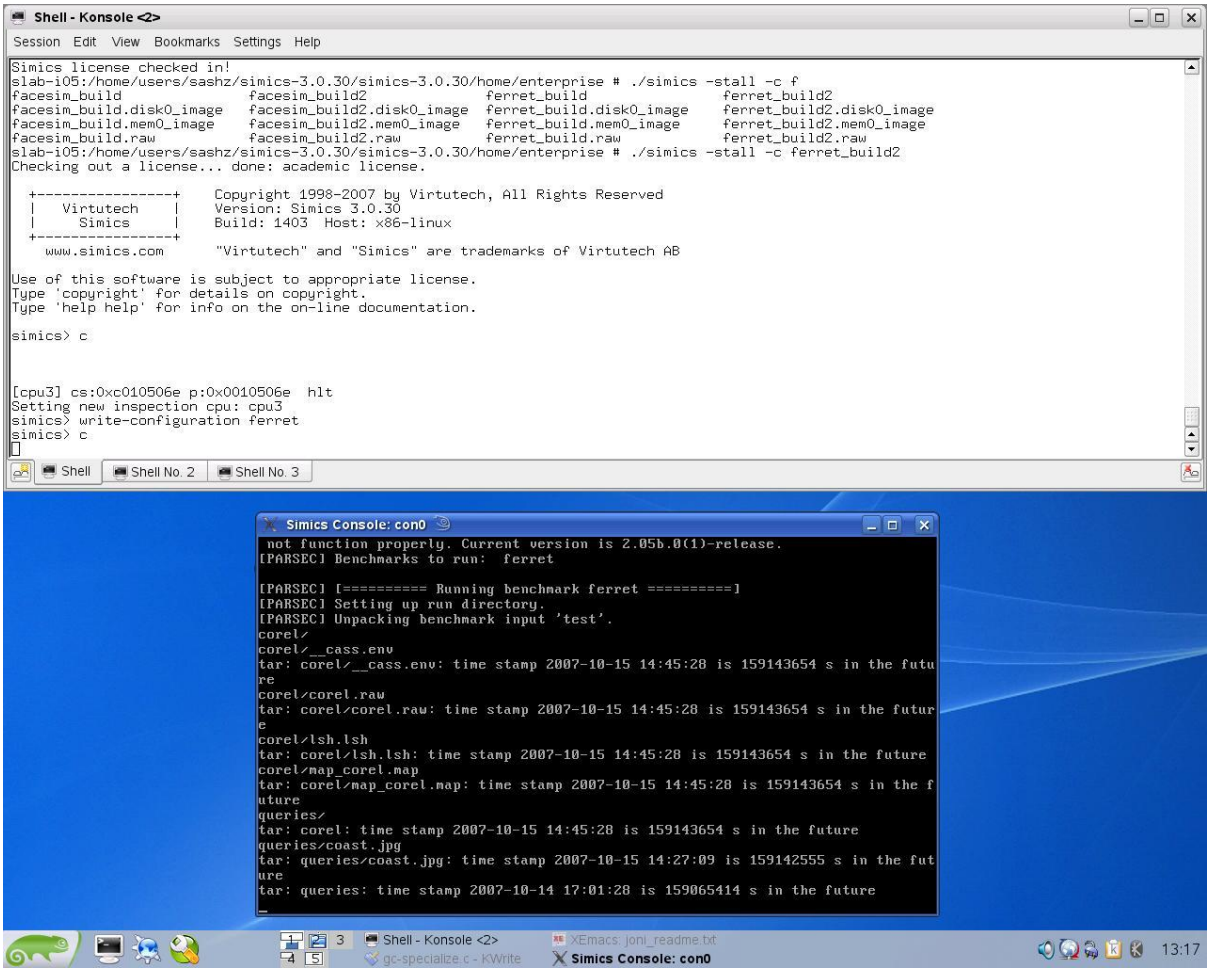
The timing model

Simics is an event driven simulator. The smallest time resolution is the clock cycle while a step is the execution time of a single instruction or the time of interrupt handling.
There are two main types of events that are triggered by the simulator and the user can respond to:

- Events that relate to a specific step
- Events that relate to of a specific cycle.

Events that relate to steps are good for debug while clock cycle events provide the opportunity to call handling procedure to events every number of time units regardless of the instruction flow.
Simics-in-order model – the simplest model in which the instructions are executed one by one independently. In this model an instruction take one clock cycle (= step time) and the clock cycle is set to be the longest instruction time. This model invokes good performing on the simulator. In this model the trigger of the step event is performed before the execution of the instruction.
In addition to this base model, we can add the timing model to get stalls to the memory operations. In this case each step will take a different clock cycle, so actually the step time varies from the cycle time. For example a load instruction may take a few clock cycles. Still this model is called in-order model due to its execution order.

When modeling a Multi-Processor system, the simulator sets a base quantum of time unit and the processors get time quantum's in a round robin algorithm. In the statistics, we can't see the round-robin execution of the processors but only the "parallel" execution of the CPUs like in a real CMP system.



Cache simulation

A cache memory is a small and fast memory unit which is usually divided to two levels (L1, L2) that located right next to the CPU according to the chip architecture. Because Simics is a instruction set simulator, you can build different cache models and by doing so, you are changing the chip model.

The default Simics settings configure there is no cache modeling and every memory transaction is atomic and takes zero time while keeping the data valid. This is done in order to increase the simulator performance.

If you wish to create a cache model you need to add a cache module or update the base module. The main goal of cache simulation is to collect statistics and information regarding the performance of the CPU and its memory units by simulating benchmarks.

The g-cache module that comes with the simulator allows running cache simulation. The model was built for the standard cache configuration that performs the memory transactions serially in an in-order system.

You can connect the cache with a Python script and also configure the cache sizes like associativity level, line size and cache size. In the next chapter our python script is described.

The g-cache transaction handle algorithm is:

- If the transaction is uncacheable, **g-cache** ignores it.
- If the transaction is a read hit, **g-cache** returns *penalty_read* cycles of penalty.
- If the transaction is a read miss, **g-cache** asks the replacement policy to provide a cache line to allocate.
- The new cache line is emptied. If necessary, a copy-back transaction is initiated to the next level cache. In this case, a penalty of *penalty_write_next* is counted, added to the penalty returned by the next level.
- The new data is fetched from the next level, incurring *penalty_read_next* cycles penalty added to the penalty returned by the next level.
- The total penalty returned is the sum of *penalty_read*, plus the penalties associated with the copy-back (if any), plus the penalties associated with the line fetch.

Coding SImics

File: GC_SPECIALIZE.H

The next Global variables are used as our database (hash function and ways allocation):

```
int shared_cache_alloc[NUM_OF_CPU];
int shared_cache_accum[NUM_OF_CPU];  //shared_cache_accum[i] =
shared_cache_alloc[0] + ... + shared_cache_alloc[i] + 12 * i
int shared_cache_start[NUM_OF_CPU];
int _working_set_sizes[NUM_OF_CPU]; // JONI - global array that holds the
working set sizes.
int _WS_bolts[NUM_OF_CPU][HASH_TABLE_SIZE];
int _WS_num_of_changes[NUM_OF_CPU];

int is_first = 1;
int counter = 0;
```

File: GC_SPECIALIZE.C

The function calc_speed_up_simple calculate the potential gain of CPUs according to the calculated working-sets size.

```
void calc_speed_up_simple( double * speed_up) {
  for (int i=0; i<8; i++)
    speed_up[i] = _working_set_sizes[i];
  return;
}
```

The function calculate_shared_cache_allocation calculates the extra cache allocation size according to the calculated speed-up.

```
void calculate_shared_cache_allocation(generic_cache_t *gc) {
  double speed_up[8];
  double speed_up_sum=0;
  int way_sum=0;
  calc_speed_up_simple(speed_up);
  for (int i=0; i<8; i++)
    speed_up_sum += speed_up[i];
  for (int i=0; i<8; i++)
    speed_up[i] = speed_up[i] / speed_up_sum;
// Calculating shared_cache_alloc
  for (int i=0; i<8; i++)
    shared_cache_alloc[i] = (int)(round(24 * speed_up[i]));
  for (int i=0; i<7; i++)
    way_sum += shared_cache_alloc[i];
  shared_cache_alloc[7] = 24 - way_sum;
//   calculating shared_cache_accum
  shared_cache_accum[0] = 0;
  for (int i=1; i<8; i++)
    shared_cache_accum[i] = shared_cache_accum[i-1] + shared_cache_alloc[i];
  // calculating shared_cache_start
  shared_cache_start[0]=96;
  for (int i=1; i<8; i++)
    shared_cache_start[i] = 96 + shared_cache_accum[i-1];
}
```

The function joni_perform_bolt_check finds the working set size of the CPUs according to the hash table that contains the working-set signature.

```
void joni_perform_bolt_check(generic_cache_t *gc, int cpu_num)
{
  int i;
  int filled_bolts = 0, new_WS;
  double fraction;

  for (i = 0; i < HASH_TABLE_SIZE; i++) {
    if (_WS_bolts[cpu_num][i] == 1) {
      _WS_bolts[cpu_num][i] =0;
      filled_bolts++;
    }
  }
  fraction = ((double)filled_bolts) / ((double)HASH_TABLE_SIZE);
  if (fraction == 1)
    _working_set_sizes[cpu_num] = HASH_TABLE_SIZE * 2.5; //JONI CHECK
  else {
    new_WS = log(1 - fraction) / log(1 - 1.0/(double)HASH_TABLE_SIZE);
    _working_set_sizes[cpu_num] = new_WS;
  }
  return;
}
```

The function joni_update_WS updates the hash table (WS signature) and if more then 10000 read/write accesses took place then it performs WS recalculation.

```
void joni_update_WS ( generic_cache_t *gc, int line_num, int cpu_num)
{
  _WS_bolts[cpu_num][line_num % 1024] = 1;
  if (_WS_num_of_changes[cpu_num] > 10000)
    {
      joni_perform_bolt_check(gc, cpu_num);
      _WS_num_of_changes[cpu_num] = 0;
    }
  else
    {
      _WS_num_of_changes[cpu_num]++;
    }
}
```

We updated the gc_operate to do initialization of the database arrays and to call the calculate_shared_cache_allocation function every 10000 gc_operate accesses

```
cycles_t
gc_operate(conf_object_t *mem_hier, conf_object_t *space,
           map_list_t *map, generic_transaction_t *mem_op)
{
…
…
       //JONI initializations:
       if (is_first) {
         is_first = 0;
         for (int i = 0; i < NUM_OF_CPU; i++) {
           _WS_num_of_changes[i] = 0;
           _working_set_sizes[i] = 1;
           shared_cache_accum[i] = 0;
           shared_cache_alloc[i] = 3;
           shared_cache_start[i] = 0;
           for (int j = 0; j < HASH_TABLE_SIZE; j++)
             _WS_bolts[i][j] = 0;
         }
       }
       //JONI - every 10000 calls to gc_operate we re-calculate the
shared_cache_allocation.
       if (gc->config.JONI == 1) {
         counter++;
         if (counter == 10000) {
           counter = 0;
           calculate_shared_cache_allocation(gc);
         }
       }
…
…
}
```

We update the real_nuca_handle_read function to update the WS every read

```
static cycles_t
real_nuca_handle_read(generic_cache_t *gc, generic_transaction_t *mem_op,
                  conf_object_t *space, map_list_t *map)
{
…
…
      if (gc->config.JONI == 1) {
        joni_update_WS(gc,line_num, cpu_num);
      }
}
```

We update the real_nuca_handle_write function to update the WS every write.

```
static cycles_t
real_nuca_handle_write(generic_cache_t *gc, generic_transaction_t *mem_op,
                   conf_object_t *space, map_list_t *map)
{
…
…
      if (gc->config.JONI == 1) {
        joni_update_WS(gc,line_num, current_cpu_num);
      }
}
```

We updated the function to calculate the distances between the way (the cache) where the data is saved and the current CPU according to NAHALAL architecture.

```
int
DistanceInNumOfHops(generic_cache_t *gc, conf_object_t *cpu, int way)
{
  if (gc->config.JONI == 1) {
    return DistanceInNumOfHopsNAHALALBigCentre(cpu, way);
  }
  if (gc->config.JONI == 2) {
    return DistanceInNumOfHopsNAHALALBigCentre(cpu, way);
  }
  if (gc->config.NAHALAL) {
    return DistanceInNumOfHopsNAHALAL(cpu, way);
  }
  else {
    //return DistanceInNumOfHopsOrig(cpu, way);
    return DistanceInNumOfHops8x8(cpu, way);
  }
}
```

We updated the function PromoteBlock to promote shared blocks to the shared cache according to our new architecture.

```
int
PromoteBlock(generic_cache_t *gc, generic_transaction_t *mem_op, conf_object_t *cpu,
             int line_num, int way, int *r_new_line_num, int *r_new_way)
{
…
…
…
  else if (gc->config.JONI == 1)
  {
      new_way = PromoteBlockNAHALAL_JONI(gc, mem_op, cpu, way);
  }
  else if (gc->config.JONI == 2)
  {
      new_way = PromoteBlockNAHALAL_BigCentre(gc, mem_op, cpu, way);
  }
…
…
…
}
```

The function returns 1 if the way is in the shared cache or 0 otherwise.

```
int
IsLineInTheMiddle(int way)
{
  if( way >= 96) {  //JONI - we changed this because because
    return(1);
  }
  return(0);
}
```

In the next function we updated the numbers of ways so it would be consistent with our design of dynamic allocation of the ways.

```
int PromoteBlockNAHALAL_JONI(generic_cache_t *gc, generic_transaction_t *mem_op, conf_object_t *cpu, int way)
{
  int cpu_num = atoi ((cpu->name)+3); //cpu name is cpuX;
  int i, num_of_sharers;
  int new_way=way;
  int line_num = lookup_line(gc, mem_op, NULL);
  //if block is in the middle do nothing:
  if(way >=120 && way <128) return way;
  //check if the block is shared:
  for(i=0,num_of_sharers=0; i< NUM_OF_CPU; i++) {
    if(LineStatistics[line_num].sharers_array[i].write || LineStatistics[line_num].sharers_array[i].read ||
i==cpu_num) num_of_sharers++;
  }
  //if it's a private block, if not in our local fetch it.
  if(num_of_sharers==1) {
        //if it's already in the local banks do nothing:
        if( (cpu_num==0 && ( (way >=0 && way <12) || (way >= shared_cache_start[cpu_num] && way <
shared_cache_start[cpu_num+1])) ) ||
      (cpu_num==1 && ( (way >=12 && way <24) || (way >= shared_cache_start[cpu_num] && way <
shared_cache_start[cpu_num+1])) ) ||
      (cpu_num==2 && ( (way >=24 && way <36) || (way >= shared_cache_start[cpu_num] && way <
shared_cache_start[cpu_num+1])) ) ||
      (cpu_num==3 && ( (way >=36 && way <48) || (way >= shared_cache_start[cpu_num] && way <
shared_cache_start[cpu_num+1])) ) ||
      (cpu_num==4 && ( (way >=48 && way <60) || (way >= shared_cache_start[cpu_num] && way <
shared_cache_start[cpu_num+1])) ) ||
      (cpu_num==5 && ( (way >=60 && way <72) || (way >= shared_cache_start[cpu_num] && way <
shared_cache_start[cpu_num+1])) ) ||
      (cpu_num==6 && ( (way >=72 && way <84) || (way >= shared_cache_start[cpu_num] && way <
shared_cache_start[cpu_num+1])) ) ||
      (cpu_num==7 && ( (way >=84 && way <96) || (way >= shared_cache_start[cpu_num] && way <
shared_cache_start[cpu_num+1])) ))
                new_way = way;
         else
                new_way = gc->config.repl_fun.GetPrivateLRUWay(gc->config.repl_data, gc, mem_op, cpu_num);
       return new_way;
  }
  else
…
…
}
```

In the next function we updated the numbers of ways so it would be consistent with static allocation of ways.

```
int PromoteBlockNAHALAL_BigCentre(generic_cache_t *gc, generic_transaction_t *mem_op, conf_object_t
*cpu, int way)
{
  int cpu_num = atoi ((cpu->name)+3); //cpu name is cpuX;
  int i, num_of_sharers;
  int new_way=way;
  int line_num = lookup_line(gc, mem_op, NULL);
  //if block is in the middle do nothing:
  if(way >=96 && way <128) return way;
  //check if the block is shared:
  for(i=0,num_of_sharers=0; i< NUM_OF_CPU; i++) {
    if(LineStatistics[line_num].sharers_array[i].write ||
LineStatistics[line_num].sharers_array[i].read || i==cpu_num) {
      num_of_sharers++;
    }
  }
  //if it's a private block, if not in our local fetch it.
  if(num_of_sharers==1)
  {
    //if it's already in the local banks do nothing:
    if( (cpu_num==0 && way >=0 && way <12) ||
        (cpu_num==1 && way >=12 && way <24) ||
        (cpu_num==2 && way >=24 && way <36) ||
        (cpu_num==3 && way >=36 && way <48) ||
        (cpu_num==4 && way >=48 && way <60) ||
        (cpu_num==5 && way >=60 && way <72) ||
        (cpu_num==6 && way >=72 && way <84) ||
        (cpu_num==7 && way >=84 && way <96) )
               new_way = way;
        else
               new_way = gc->config.repl_fun.GetPrivateLRUWay(gc->config.repl_data, gc, mem_op,
cpu_num);
        return new_way;
  }
  Else
…
…
}
```

The function DistanceInNumOfHopsNAHALALBigCentre returns the penalty for accessing other cache according to the distance of the current CPU to the way in which the line was found. We adjusted this function to our design with the big middle cache.

```
int
DistanceInNumOfHopsNAHALALBigCentre(conf_object_t *cpu, int way)
{
…
…
}
```

File: GC-LRU-REPL.C

We updated the following function to work with our new configurations.

```
int
GetPrivateLRUWay(void *data, generic_cache_t *gc, generic_transaction_t *gt, int
cpu_num)
{
  if (gc->config.JONI == 1) {
    return GetPrivateLRUWayJoni(data, gc, gt, cpu_num);
  }
   if (gc->config.JONI == 2) {
    return GetPrivateLRUWayNAHALALBigCentre(data, gc, gt, cpu_num);
  }
  if(gc->config.NAHALAL) {
    return GetPrivateLRUWayNAHALAL(data, gc, gt, cpu_num);
  } else {
    return GetPrivateLRUWayDNUCA(data, gc, gt, cpu_num);
  }
}
```

The function `GetPrivateLRUWayJoni` returns the way that will be used for new data block in our new configuration.

```
int GetPrivateLRUWayJoni(void *data, generic_cache_t *gc, generic_transaction_t *gt, int
cpu_num) {
  static int cpu_private_banks[8][12] = {
    {0,1,2,3,4,5,6,7,8,9,10,11},
    {12,13,14,15,16,17,18,19,20,21,22,23},
    {24,25,26,27,28,29,30,31,32,33,34,35},
    {36,37,38,39,40,41,42,43,44,45,46,47},
    {48,49,50,51,52,53,54,55,56,57,58,59},
    {60,61,62,63,64,65,66,67,68,69,70,71},
    {72,73,74,75,76,77,78,79,80,81,82,83},
    {84,85,86,87,88,89,90,91,92,93,94,95}
  };
  lru_data_t *d = (lru_data_t *) data;
  int i, curr_way, lru_way;
  uinteger_t lru_time = (uinteger_t)-1;
  i = GC_INDEX(gc, gt) + (GC_NEXT_ASSOC(gc) * cpu_private_banks[cpu_num][0]) ;
  for (curr_way=0;
       (i < gc->config.line_number) && (curr_way<12); i = GC_INDEX(gc, gt) +
(GC_NEXT_ASSOC(gc) * cpu_private_banks[cpu_num][++curr_way]))
    if (d->last_used[i] <= lru_time) {
      lru_time = d->last_used[i];
      lru_way= cpu_private_banks[cpu_num][curr_way];
    }
  i = GC_INDEX(gc, gt) + (GC_NEXT_ASSOC(gc) * shared_cache_start[cpu_num] ) ;
  for (curr_way=0;
       (i < gc->config.line_number) && (curr_way < shared_cache_alloc[cpu_num]); i =
GC_INDEX(gc, gt) + (GC_NEXT_ASSOC(gc) * (shared_cache_start[cpu_num] + (++curr_way))))
    if (d->last_used[i] <= lru_time) {
      lru_time = d->last_used[i];
      lru_way= shared_cache_start[cpu_num] + curr_way;
    }
  SIM_log_info(3, &gc->log, GC_Log_Repl, "GetPrivateLRUWayJoni:: got way %d", lru_way);
  return lru_way;
}
```

The `GetPrivateLRUWayNAHALALBigCentre` returns the way that will be used for new data block without our dynamic cache allocation.

```
Int GetPrivateLRUWayNAHALALBigCentre(void *data, generic_cache_t *gc, generic_transaction_t *gt,
int cpu_num) {
  static int cpu_private_banks[8][12] = {
    {0,1,2,3,4,5,6,7,8,9,10,11},
    {12,13,14,15,16,17,18,19,20,21,22,23},
    {24,25,26,27,28,29,30,31,32,33,34,35},
    {36,37,38,39,40,41,42,43,44,45,46,47},
    {48,49,50,51,52,53,54,55,56,57,58,59},
    {60,61,62,63,64,65,66,67,68,69,70,71},
    {72,73,74,75,76,77,78,79,80,81,82,83},
    {84,85,86,87,88,89,90,91,92,93,94,95}
  };
  lru_data_t *d = (lru_data_t *) data;
  int i, curr_way, lru_way;
  uinteger_t lru_time = (uinteger_t)-1;
  i = GC_INDEX(gc, gt) + (GC_NEXT_ASSOC(gc) * cpu_private_banks[cpu_num][0]) ;
  for (curr_way=0;
       (i < gc->config.line_number) && (curr_way<12); i = GC_INDEX(gc, gt) + (GC_NEXT_ASSOC(gc) *
cpu_private_banks[cpu_num][++curr_way]))
    if (d->last_used[i] <= lru_time) {
      lru_time = d->last_used[i];
      lru_way= cpu_private_banks[cpu_num][curr_way];
    }
  SIM_log_info(3, &gc->log, GC_Log_Repl, "GetPrivateLRUWayNAHALAL:: got way %d", lru_way);
  return lru_way;
}
```

File: GC-ATTRIBUTES.C

We added our configuration (gc->config.JONI) to the generic cache attributes.

```
Void gc_init_cache(generic_cache_t *gc)
{
        /* set default values in the cache */
        gc->config.no_STC = 1;
        /* block_STC has default to 1 so that the cache behaves properly */
        gc->config.block_STC = 1;
        gc->config.line_size = 32;
        gc->config.assoc = 4;
        gc_set_config_repl(gc, HOOK_DEFAULT_REPL_POLICY);
        gc_set_config_line_number(gc, 128);
       gc->config.nuca = 0;
       gc->config.SNUCA = 0;
       gc->config.DNUCA = 0;
       gc->config.NAHALAL = 0;
       gc->config.JONI = 0;
       gc->config.cr_nuca = 0;
       gc->config.ring_topology = 0;
       gc->config.bloom_filter_size=0;

        gc->sharing_stat.enable=0;
        gc->sharing_stat.reset_stat=0;
        gc->sharing_stat.print_all_cache=0;

        gc->config.parallel_search=0;
        gc->config.sequential_search=0;
        gc->config.nahalal_sequential_search_start_with_private=0;
}
```

Running Simics

Configuration

Before running simics, we had to install simics software which we downloaded from the provider virtutech at http://www.virtutech.com/. We installed simics version 3.0.30.
Along with the software we had to install a new version of GCC (3.3.4) and to require an academic license for the software. Today the license is provided per site (Technion) so no special action is required.
We used C libraries and Python scripts that already been written by Zvika Guz for NAHALAL project as our base code, on which we added our changes.
First we had to configure:
From the directory ***/home/users/sashz/simics-3.0.30/simics-3.0.30/x-86-linux*** we type:
.. /configure CC=/opt/gcc-3.3.4/bin/gcc

Compiling

The simics libraries has gmake files for compilation, so you need to choose the module you want to compile and the dependencies will be automatically compiled also.
From the directory ***/home/users/sashz/simics-3.0.30/simics-3.0.30/x86-linux/lib*** we type:
gmake g-cache
g-cache is the module we changes in order to simulate our project.

Running

In order to upload the simics simulation from the last configuration of the simulation we type From the directory ***/home/users/sashz/simics-3.0.30/simics-3.0.30/home/enterprise*** :
../simics -stall -c joni3.conf
Because we are running two different configurations, for comparison, we wrote to Phyton scripts that we ran from the simics window.

For running legacy NAHALAL configuration as our reference model we ran:

run-python-file zguz_big_centre.py

@add_zguz_directory_caches()

@attach_all_caches()

For running JSO configuration we ran:

run-python-file joni.py

@add_zguz_directory_caches()

@attach_all_caches()

The script's code is:

```
from configuration import *
config_error = 0
cpu_list = []
space_list = {}
mem_latency = 300

def collect_cpus(cpu, arg):
    global config_error
    if config_error:
        return
    space_list[cpu] = [cpu.physical_memory]
    try:
        if cpu.physical_io != cpu.physical_memory:
            space_list[cpu].append(cpu.physical_io)
    except:
        pass
    for space in space_list[cpu]:
        try:
            if space.timing_model != None:
                print ("The memory-space '%s' already has a"
                       "timing-model connected." % space.name)
                print "Cannot add cache to it - please check the configuration."
                config_error = 1
                return
```

```
        except:
            pass
    cpu_list.append(cpu)

def common_setup_start(func):
    if not SIM_initial_configuration_ok():
        print "%s() cannot run until a configuration exists." % func
        print
        SIM_command_has_problem()
        return 1
    SIM_for_all_processors(collect_cpus, 0)
    if config_error:
        # error message already printed
        SIM_command_has_problem()
        return 1
    return 0

def common_setup_end(conf):
    try:
        SIM_set_configuration(conf)
    except Exception, msg:
        print "Failed adding cache to the configuration: %s" % msg
        print
        SIM_command_has_problem()
        return 1
    try:
        for cpu in cpu_list:
            for space in space_list[cpu]:
                space.timing_model = SIM_get_object("id_%s" % cpu.name)
    except Exception, msg:
        print "Failed adding cache to the configuration: %s" % msg
        print
        SIM_command_has_problem()
        return 1
    return 0
```

```
############################################################
########################## ZvikaNew ########################
############################################################
def add_zguz_directory_caches():
    if common_setup_start("add_zguz_directory_caches"):
        return
    conf = []
    cache_olist = []
    for cpu in cpu_list:
        conf += [
            #
            # instruction-data splitter
            #
            OBJECT("id_%s" % cpu.name, "id-splitter",
                   dbranch = OBJ("ts-d_%s" % cpu.name)),
            #
            # transaction splitter for data cache
            #
            OBJECT("ts-d_%s" % cpu.name, "trans-splitter",
                   cache = OBJ("dc_%s" % cpu.name),
                   timing_model = OBJ("dc_%s" % cpu.name),
                   next_cache_line_size = 64),
            #
            # data cache: 32Kb Write-back
            #
            OBJECT("dc_%s" % cpu.name, "g-cache",
                   cpus = OBJ("%s" % cpu.name),
                   config_line_number = 512,
                   config_line_size = 64,
                   config_assoc = 2,
                   config_virtual_index = 0,
                   config_virtual_tag = 0,
                   config_replacement_policy = "lru",
                   config_write_back = 1,
                   config_write_allocate = 1,
                   penalty_read = 3,
                   penalty_write = 3,
                     penalty_read_next = 0,
                     penalty_write_next = 0,
                     snoopers = [OBJ("l2c")],
                     timing_model = OBJ("l2c"))]
```

```
                    penalty_read_next = 0,
                    penalty_write_next = 0,
                    snoopers = [OBJ("l2c")],
                    timing_model = OBJ("l2c"))]
        cache_olist.append(OBJ("dc_%s" % cpu.name) ) #needed to enable snooping
    conf += [
            #
            # l2 cache: 16Mb Write-back
            #   bank size is 1Mb
            #   Notice that assoc =16
            #
            OBJECT("l2c", "g-cache",
                    cpus = [OBJ("cpu0"), OBJ("cpu1"), OBJ("cpu2"), OBJ("cpu3"), OBJ("cpu4"),
OBJ("cpu5"), OBJ("cpu6"), OBJ("cpu7")],
                    config_nuca = 1,
                    config_NAHALAL = 1,
                    config_JONI = 1,
                    config_line_number = 262144,
                    config_line_size = 64,
                    config_assoc = 128,
                    config_virtual_index = 0,
                    config_virtual_tag = 0,
                    config_write_back = 1,
                    config_write_allocate = 1,
                    config_replacement_policy = "lru",
                    penalty_read = 15,    #each bank require another 15 cycles
                    penalty_write = 15,   #each bank require another 15 cycles
                    penalty_link = 5,    #each link require another 5 cycles
                    config_nahalal_filter_threshold = 10,
                    config_bloom_filter_size=2048,
                    config_sequential_search=1,
                    penalty_read_next = 0,
                    penalty_write_next = 0,
                    higher_level_caches = cache_olist,
                    timing_model = OBJ("staller")),
            # Transaction staller for memory
            OBJECT("staller", "trans-staller",
                     stall_time = mem_latency)]


      try:
        SIM_set_configuration(conf)
    except Exception, msg:
        print "Failed adding cache to the configuration: %s" % msg
        print
        SIM_command_has_problem()

def attach_all_caches():
    try:
        for cpu in cpu_list:
            for space in space_list[cpu]:
                if space != None:
                        space.timing_model = SIM_get_object("id_%s" % cpu.name)
                        print "Added timing model to space: %s" % space
        print "Done!"
    except Exception, msg:
        print "Failed adding cache to the configuration: %s" % msg
        print
        SIM_command_has_problem()
```

Notice that the difference between the two scripts is that joni.py defines the config_JONI variable in the g-cache object as 1 (i.e. enable JSO configuration) and zguz_bug_centre.py enables our reference model by giving config_JONI the value of 2.

Getting statistics

Finally, in order to get the statistics after the simulation we type in the simics window:

l2c.statistics

PARSEC

Princeton Application Repository for Shared-Memory Computers [4] is a benchmark Suite for Chip-Multiprocessors. Freely available at: http://parsec.cs.princeton.edu//

The new benchmark suite was created due to several trends:

- Trend 1: New application areas for parallel machines through proliferation of CMPs
- Trend 2: Drastic change of architecture constraints driven by CMPs
- Trend 3: Explosion of globally stored data requires new algorithms

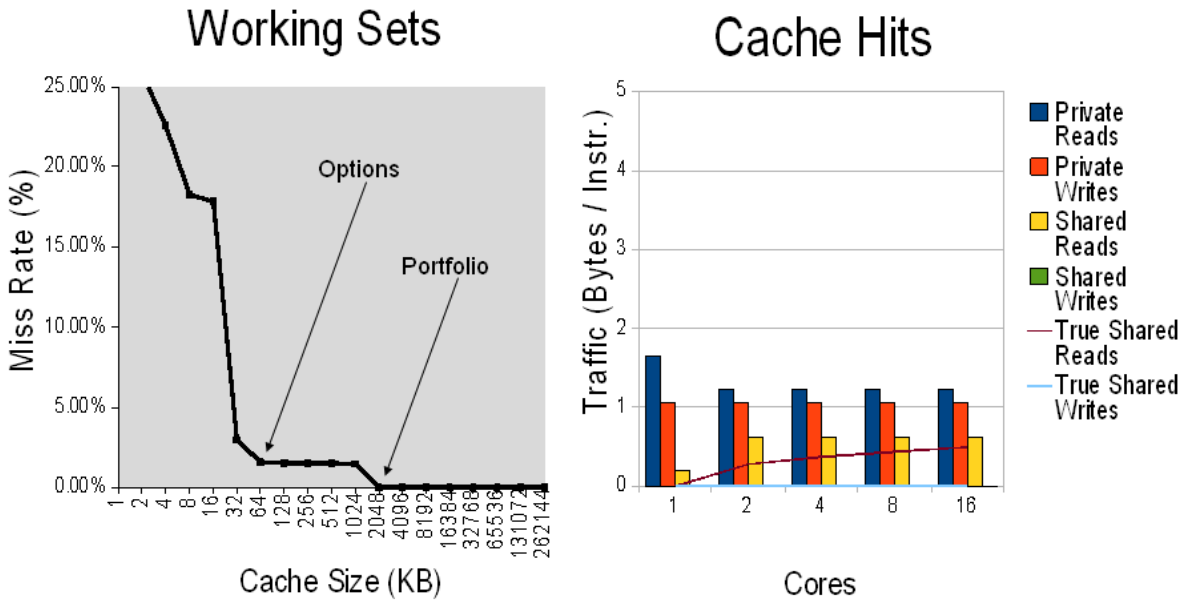
The objectives of parsec are:

- Multithreaded Applications - Future programs must run on multiprocessors
- Emerging Workloads - Increasing CPU performance enables new applications
- Diverse - Multiprocessors are being used for more and more tasks
- State-of-Art Techniques - Algorithms and programming techniques evolve rapidly
- Support Research - Our goal is insight, not numbers

The program allows you to run prebuilt workloads:

Program	Application Domain	Parallelization
Blackscholes	Financial Analysis	Data-parallel
Bodytrack	Computer Vision	Data-parallel
Canneal	Engineering	Unstructured
Dedup	Enterprise Storage	Pipeline
Facesim	Animation	Data-parallel
Ferret	Similarity Search	Pipeline
Fluidanimate	Animation	Data-parallel
Freqmine	Data Mining	Data-parallel
Streamcluster	Data Mining	Data-parallel

Each work load has its own characteristics, for example these graphs describe the working sets and cache hits in Blackscholes:



Small working sets, negligible communication

The program can be managed by using the script ***parsecmgmt***. The script helps you manage your PARSEC installation and can also build and run PARSEC workloads for you.

You can use the following command to get some help:

```
parsecmgmt -h
```

This is an example for usage:

```
> parsecmgmt -r run -p canneal -c gcc-serial -i simsmall

[PARSEC] Benchmarks to run:  canneal
[PARSEC] [========== Running benchmark canneal ==========]
[PARSEC] Setting up run directory.
[PARSEC] Unpacking benchmark input 'simsmall'.
100000.nets
[PARSEC] Running '...':
[PARSEC] [---------- Beginning of output ----------]
PARSEC Benchmark Suite Version 1.0
Threadcount: 1
10000 moves per thread
Start temperature: 2000
...
[PARSEC] [----------    End of output    ----------]
[PARSEC] Done.
```

In our project we focused on 3 test benches that include the various important properties:

- **Blackscholes**

  Small working sets, negligible communication

- **Swaptions**

  Medium-sized working sets, little communication

- **Caneal**

  Huge working sets, communication limited by capacity

Results

We ran the 3 test-benches that represent the variety of the working-set sizes: Blackscholes, swaptions and caneal. For each of the test-benches we ran with at least one thread when the average number of threads was in fact around three.

The total cache size is 16MB and the cache is divided to 128 ways. We used that division to allocate cache size for the different caches. Each way is equal to 128K of fast memory unit (cache).

The results are:

blackscholes

JSO

```
Cache statistics: l2c
-----------------
    Total number of transactions:       8298621
         Total memory stall time:     624893355
     Total memory hit stall time:     603383655

          Device data reads (DMA):            0
         Device data writes (DMA):            0

           Uncacheable data reads:          165
          Uncacheable data writes:       469397
 Uncacheable instruction fetches:            0

           Data read transactions:       5197835
            Total read stall time:     368688690
        Total read hit stall time:     347568990
            Data read remote hits:            0
                 Data read misses:        70399
               Data read hit ratio:       98.65%
  Instruction fetch transactions:            0
        Instruction fetch misses:            0

          Data write transactions:       2631224
           Total write stall time:     115336065
       Total write hit stall time:     115336065
           Data write remote hits:            0
                Data write misses:            0
             Data write hit ratio:      100.00%

           Copy back transactions:            0

Number of replacments in the middle (NAHALAL):
622853
```

NAHALAL

```
Cache statistics: l2c
-----------------
    Total number of transactions:       7938071
         Total memory stall time:     521097410
     Total memory hit stall time:     498527510

          Device data reads (DMA):            0
         Device data writes (DMA):            0

           Uncacheable data reads:          169
          Uncacheable data writes:       437279
 Uncacheable instruction fetches:            0

           Data read transactions:       5090602
            Total read stall time:     301057885
        Total read hit stall time:     278882785
            Data read remote hits:            0
                 Data read misses:        73917
               Data read hit ratio:       98.55%
  Instruction fetch transactions:            0
        Instruction fetch misses:            0

          Data write transactions:       2410021
           Total write stall time:      88805125
       Total write hit stall time:      88805125
           Data write remote hits:            0
                Data write misses:            0
             Data write hit ratio:      100.00%

           Copy back transactions:            0
```

Number of replacments in the middle (NAHALAL):
572513

swaptions

JSO

```
Cache statistics: l2c
-----------------
    Total number of transactions:       3681206
         Total memory stall time:     260956045
     Total memory hit stall time:     241450345

          Device data reads (DMA):            0
         Device data writes (DMA):            0

           Uncacheable data reads:          140
          Uncacheable data writes:       158818
 Uncacheable instruction fetches:            0

           Data read transactions:       2309104
            Total read stall time:     161515100
       Total read hit stall time:     142380800
           Data read remote hits:            0
                Data read misses:        63781
              Data read hit ratio:       97.24%
  Instruction fetch transactions:            0
        Instruction fetch misses:            0

          Data write transactions:       1213144
           Total write stall time:      51753545
      Total write hit stall time:      51753545
          Data write remote hits:            0
               Data write misses:            0
            Data write hit ratio:      100.00%

           Copy back transactions:            0
```

Number of replacments in the middle (NAHALAL): 276814

NAHALAL

```
Cache statistics: l2c
-----------------
    Total number of transactions:       3316616
         Total memory stall time:     210238115
     Total memory hit stall time:     189880415

          Device data reads (DMA):            0
         Device data writes (DMA):            0

           Uncacheable data reads:          131
          Uncacheable data writes:       154246
 Uncacheable instruction fetches:            0

           Data read transactions:       2031578
            Total read stall time:     126369090
       Total read hit stall time:     106401690
           Data read remote hits:            0
                Data read misses:        66558
              Data read hit ratio:       96.72%
  Instruction fetch transactions:            0
        Instruction fetch misses:            0

          Data write transactions:       1130661
           Total write stall time:      37555925
      Total write hit stall time:      37555925
          Data write remote hits:            0
               Data write misses:            0
            Data write hit ratio:      100.00%

           Copy back transactions:            0
```

Number of replacments in the middle (NAHALAL): 241305

Canneal

JSO

```
Cache statistics: l2c
-----------------
    Total number of transactions:       3664638
         Total memory stall time:     262742600
     Total memory hit stall time:     241890200

          Device data reads (DMA):             0
         Device data writes (DMA):             0

           Uncacheable data reads:           158
          Uncacheable data writes:        159915
 Uncacheable instruction fetches:             0

           Data read transactions:       2284926
            Total read stall time:     162096600
       Total read hit stall time:     141634200
           Data read remote hits:             0
                Data read misses:         68208
              Data read hit ratio:        97.49%
  Instruction fetch transactions:             0
        Instruction fetch misses:             0

          Data write transactions:       1219639
           Total write stall time:      52624100
      Total write hit stall time:      52624100
           Data write remote hits:             0
                Data write misses:             0
             Data write hit ratio:       100.00%

          Copy back transactions:             0
```

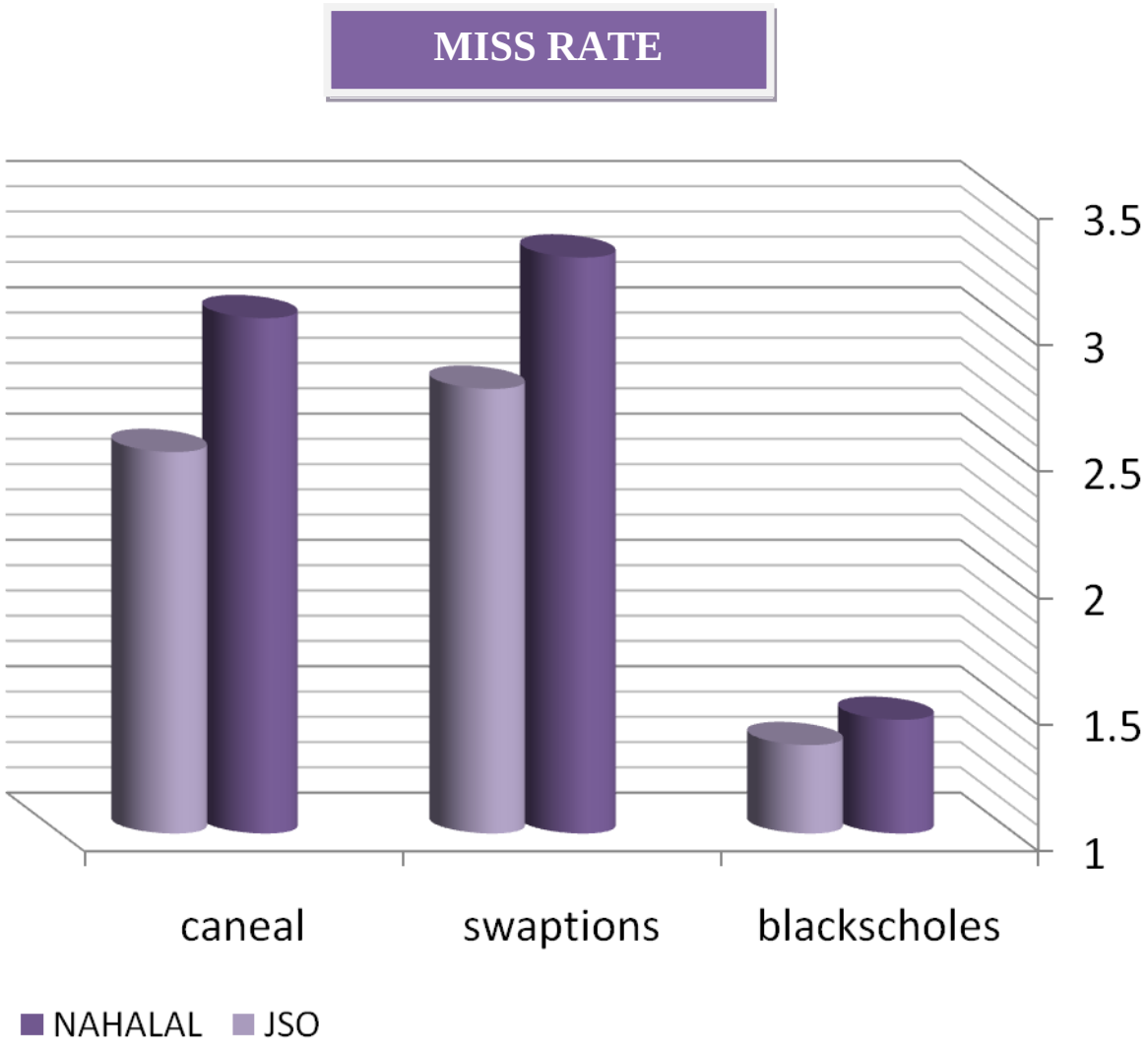
Number of replacments in the middle (NAHALAL):
266753

NAHALAL

```
Cache statistics: l2c
-----------------
    Total number of transactions:       3688350
         Total memory stall time:     238612180
     Total memory hit stall time:     217180180

          Device data reads (DMA):             0
         Device data writes (DMA):             0

           Uncacheable data reads:           164
          Uncacheable data writes:        159464
 Uncacheable instruction fetches:             0

           Data read transactions:       2310138
            Total read stall time:     146715920
       Total read hit stall time:     125634320
           Data read remote hits:             0
                Data read misses:         70272
              Data read hit ratio:        96.96%
  Instruction fetch transactions:             0
        Instruction fetch misses:             0

          Data write transactions:       1218584
           Total write stall time:      44007860
      Total write hit stall time:      44007860
           Data write remote hits:             0
                Data write misses:             0
             Data write hit ratio:       100.00%

          Copy back transactions:             0
```

Number of replacments in the middle (NAHALAL):
276706

We can see in the following diagram the miss ratio (the smaller the better):



We reduced the miss ratio by 7 to 17 percent and the average improvement 13.5%

Conclusions and Future work

From the results we can deduce some important realizations:

- It is possible to improve cache performance by dynamically allocating the shared cache. This is the most important outcome of the project because we proved that the opportunity to improve the performance still exists i.e. by using the JSO algorithm.

- The working-set signature is sufficiently accurate to estimate the real working set size. We proved that the writers of the working-set article [2] were right and the estimations they made are close enough to serve as an important parameter for future calculations.

- Dynamic cache allocation is most efficient when some CPUs are idle. The JSO algorithm is most effective when the CPUs have different working-sets, because then JSO can reallocate the cache size of the CPUs (with the smaller working-sets) for the other CPUs (with bigger working-sets). If all CPUs have the same working-set size or if the working-set is much bigger then the l2 cache size, then the JSO algorithm's performance is like the performance of the regular NAHALAL.

We found some new subjects that can expand the project research and improve the performance more drastically.

- It is possible that more complicated allocation methods will produce better results
- Dynamic cache allocation should be tested on other cache architectures
- More stressful tests may show better hit ratio improvement
- Effect of different window size and hash table size for working-set calculation should be studied

References

[1] – Zvika Guz, Idit Keidar, Avinoam Kolodny, Uri C. Weiser "**Nahalal: Cache Organization for Chip Multiprocessors**"

[2] - AS Dhodapkar, JE Smith - "**Managing Multi-Configuration Hardware via Dynamic Working Set Analysis",** ACM SIGARCH Computer Architecture News, 2002

[3] – Virtutech "**Simics user guide for UNIX**"

[4] – Y. Chen, E Li, J. Su "**The PARSEC Benchmark Suite Tutorial**"