A Modular Workflow to Dynamically Instrument and Treat Information in Multi-Process Environments

Claudio A. Carballal, José Luis Hamkalo, Bruno Cernuschi-Frías

Facultad de Ingeniería, Universidad de Buenos Aires.
Av. Paseo Colón 850, PB, (C1063ACV), Buenos Aires. Argentina.
ccarballal@gmail.com.ar
{jhamkal,bcf}@fi.uba.ar

Abstract. This paper presents a workflow in order to generate and collect execution information in multi-process environments, which are suitable for simulating and studying private and shared cache organizations. We have developed a main tool that makes dynamic instrumentation with PIN, a controller to administrate the execution of processes and three workflow application modules to generate programs traces, simulate cache memories and manage the execution of several programs in parallel.

Keywords: cache, multi-process, trace-driven simulation, PIN

1 Introduction

The development of new multi-core architectures remarkably increased performance of every entertainment, business, and research applications. Next generation of chip multi-processors (CMP's) will have hundreds of cores in a single chip [1]. As a result, designers face the challenge of reducing the increased speed gap between main memory and the CMP's [2].
Processors have an internal memory called cache, that works as a buffer between the core of the processor and the main memory [3].
Some processor architectures have several levels of cache memory. For the sake of simplicity and performance, they keep some levels of cache as private for each processor core, while other architectures explore sharing the last level of cache of different cores to gain performance, and therefore increase the capacity to get resources from main memory [4].
Cache memory is a very limited resource which makes the program behavior analysis a key feature to improve and develop the administration of this limited but useful resource.
To do this analysis, several methodologies and tools are available. One method analyzes information in a complete simulator (Execution driven simulation). An example of this is SimpleScalar [5]. This simulator is very configurable and it has

been used in many research works. Several tools to improve it and enhance its features were developed by researchers, but the disadvantage is that all results are from simulated environment and not from real hardware.
Another type of analysis is trace driven simulation [6], [7]. This technique is very flexible and it is also widely used in research. But in CMP's analysis, the disadvantage of trace-driven simulation is that shows only one possible result of execution, and the simulation stores the results of the memory accesses to a file that can occupy several gigabytes of disk even if their are in compressed formats.
The technique of adding code into an existing source code, or adding an executable binary to gather new information about it is called instrumentation. Static instrumentation adds new instrumentation code to the program source code or to the executable binary. Another approach, called dynamical instrumentation, adds the new code into the binary while it is executed. Today, the latter approach is the most applied in several tools like PIN [8] or Valgrind [9].
Intel Corporation [10] is developing a simulator that improves the trace-driven simulation technique [11]. This new simulator, called CMP$im, can simulate all memory organizations in a CMP, is very configurable, shows important data sharing statistics of threads belonging to a multi-threaded and multi-process applications, and uses PIN to dynamically instrument a binary program.
CMP$im is a major advance in trace driven simulation, that gives researchers a full tool to simulate several cache organizations and saves storage space from program traces, but it can't show processes execution phases behavior in a shared cache organization.
In CMP's, shared caches have a great benefit to performance, but also represent a great challenge for designers, who should improve their administration, because as multiple and simultaneous processes can access the cache memory simultaneously, they should face with new scenarios (which do not exist in private caches). For example, one process can overwrite the information of another process, causing an intra-processor miss [12]. Other factors that spoil performance are the execution phases which every program has throughout its execution time. Some applications present low reuse of their data and pollute caches with data streams, such as multimedia, communications or streaming applications, or have many compulsory misses that cannot be solved by assigning more cache space. These programs behaviors reduce the benefits of a shared cache if they are not well administrated so for their best performance they have to be given the necessary amount of cache [13], [14]. For that reason, new researches have developed new memory organizations so as to give the executing process the amount of cache memory to perform as good as possible, without incurring in a performance reduction in the other processes that are also running [15], [16], [17], [18], [19].
In this work the PIN tool is used to gather information of the execution of multiple processes with the purpose of feeding cache memory simulators and other analysis tools.

2 PIN Environment

PIN is a tool to dynamically instrument programs for Intel architectures. It was designed to provide a similar functionality as ATOM toolkit [20] for Alpha processors, and provides the infrastructure for writing tools to analyze programs called Pintools.

2.1 Pintools

The instrumentation with PIN consists in two major components:

- Instrumentation code.
- Analysis code.

This two codes reside in the same file, a Pintool.
Both codes are represented in a Pintool as routines to be called by PIN when an executable binary is executed. Instrumentation routines represent the instrumentation component, they inspect the code to be generated, research statical properties and decide, if it is necessary (and where), to inject the calls to the analysis functions.
The analysis functions gather data about the application. PIN makes sure that the integer register state is saved and restored as necessary and allows arguments to be passed to the functions, but floating point registers are not saved nor restored. Therefore, additional support is required in the analysis routines.

2.2 Execution Behavior

PIN can instrument an executable binary even if it creates new processes or threads. For every new child process created, the child process is instrumented by the same Pintool that instruments his parent.
The new threads are controlled and synchronized in the Pintool using the API provided by PIN. The Pintool runs as a plug-in in PIN, being PIN, the Pintool and the executable binaries in the same virtual memory addresses, hence sharing file descriptors and other useful information about every running process in the Pintool.
During its whole execution, a program can change its execution behavior [21], it can execute completely without requiring to interchange information with any source, create new child processes and wait information from them, or wait information about another different process. The operating system can even put the parent process and/or any of its child processes to sleep.
The behavior previosly described makes the design a major reason to get a robust, well-proved and flexible workflow without suffering the typical concurrency problems in these multi-process environments. The usual tools and techniques cited in the bibliography [22] to resolve concurrency problems like semaphores, shared memory, messages queues, pipes and other inter-process communication (IPC) techniques were evaluated and tested during the process of taking a final

decision for desing.
To completely understand the concurrency problems we are facing, we tested small and simple programs to see the real behavior of PIN with stand-alone processes and programs that launch new several child processes using *fork* and *execv* functions of C++ language [23]. The Pintool used for these tests was a modified version of the *pinatrace* tool provided in the examples of PIN development kit. The major modification made to this tool was the addition of a log of every function executed to gather more information of the executed instruction.
The first test was running a simple standalone program to compare PIN information with the assembly generated code by the compiler. The second test was running the same program, but launched several times using *fork* and *execv* functions to know how the Pintool works and how the functions for instrumentation work. With said test we know that a new instance of the Pintool is created for every single child process.
Other several tests with processes with IPC dependencies showed the expected concurrency problems cited in the bibliography, but these problems are not related with PIN or the Pintool functionality. The main problem is to gather information of every executed process in the correct order avoiding inter-process data interference.
For example, as every process can create new child processes any time, using semaphores with shared memory, or message queues to control them, poses the problem that every new process should get a new identifier to be managed. Also, as the newly created process will be instrumented by the same Pintool as the parent process, makes the problem of solving the control of the parent and the new created process in the same Pintool a chaotic and complex task.
To solve these problems we adopt the Self-Registered process design, described in the next section.

3 Workflow Design

Bearing in mind the behavior of the different programs that can be executed, and the final intention of this research, we have designed the tool here in introduced. The tool consists of a Pintool, a process controller and three process information buffers.
The controller is responsible of the administration and the execution of every process that is being instrumented by the Pintool, indicating when it can be executed, but every process has to register itself as an active process in the registration buffer.
With this scheme of Controller - Self-Registration a workflow capable of analyzing every execution phase of a program is achieved, avoiding the associated issues of processes concurrency that we described in the previous section.
The final developed tool consists of a memory address trace generator that can control the execution of a process and any of its child processes, created during the execution. The tool was designed to solve the problems above described,

taking into account that new modules can be added for different purposes.

Fig. 1 shows a sequence of the execution of a process:

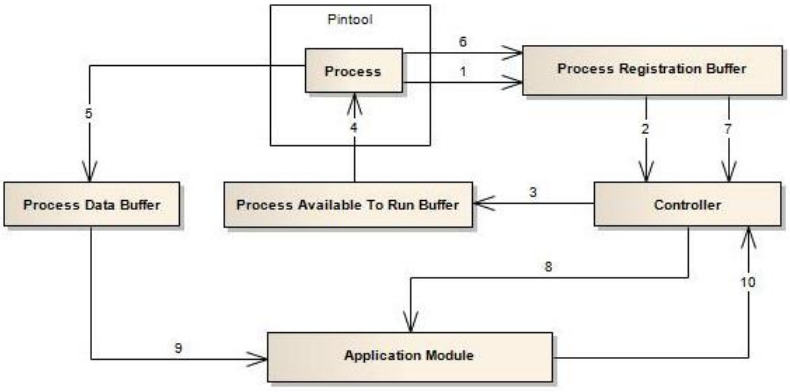

Fig. 1. Sequence of execution of a process.

- 1. The process registers in the buffer to be executed.
- 2. The controller gets all registered process to be executed.
- 3. The controller grants permission of execution to the process registered earlier.
- 4. The process is executed. In our tool, only one instruction is executed.
- 5. The executed process puts its information (provided by Pintool analysis code) in the data buffer.
- 6. The process registers again so as to be executed, indicating that it is done, with its previous execution.
- 7. The controller reads again all registered process to be executed.
- 8. The controller gives a signal to an analysis module so as to process the data.
- 9. The application module consumes the data from the buffer and processes it.
- 10. The application module gives a signal to the controller, indicating that is done with its processing.

3.1 Developed Pintool

The developed Pintool is responsible for the registration of the execution of the process that it instruments, becoming independent from the controller. This independence is necessary because otherwise the controller must be responsible for detecting, all the time, the execution status of the process and its child processes, that will face the issues described in section 3.

When the Pintool detects the signal to continue with the instrumentation and the process execution, it analyzes the executed instruction, obtains the memory access address of that instruction and writes the information into the data buffer of the process.
The data obtained from instructions is separated in two files. The name of those files is the process number and an extension *.data* if the file contains the memory addresses accessed by the process, *.inst* if the file contains the data of the instruction executed.

The structure of the information sent to the data buffer is as follows:

Fig. 2. Structure of the memory addresses informed by the Pintool.

- PROCESS ID: Process Identifier.
- INSTRUCTION ADDRESS: Memory address of the executed instruction.
- TYPE OF ACCESS: Type of memory access.
- MEMORY ADDRESS: Memory address of the access (virtual).
- SIZE: Memory address size.

Fig. 3. Structure of the executed instructions informed by the Pintool.

- PROCESS ID: Process Identifier.
- OPCODE: Instruction Opcode.
- INSTRUCTION ADDRESS: Memory address of the executed instruction.

3.2 Controller

The controller is an independent executable responsible of initializing the control, data and registration buffers. After the initialization, it reads the registration buffer of the processes that are waiting for execution permission, then pools every registered process giving an execution signal to every one. Once the pooling is finished, it sends a signal to the application module.
The controller waits until the application module finishes operating with the information stored in the data buffer and sends a signal to the controller to go

on executing the registered process. The application module can be an external function, a class inside the controller or an external application that consumes the information stored in the data buffers with a purpose.
In sections 4, 5 and 6 we will describe three optional modules that can be added to the tool that we have developed.

4 Workflow Application Module: Trace Generator

The module consists of class that uses Zlib [24] that implements the Lempel-Ziv (LZ77) algorithm to compress the programs traces on the fly. The module reads the size of the data files of every process and when the size is bigger than a parameter stored in a configure file it calls a function to compress the data of the compression buffer and sends the compressed data to an output file. The same tool can also be run individually to decompress the data.

Fig. 4 shows a sequence of compressed file generation:

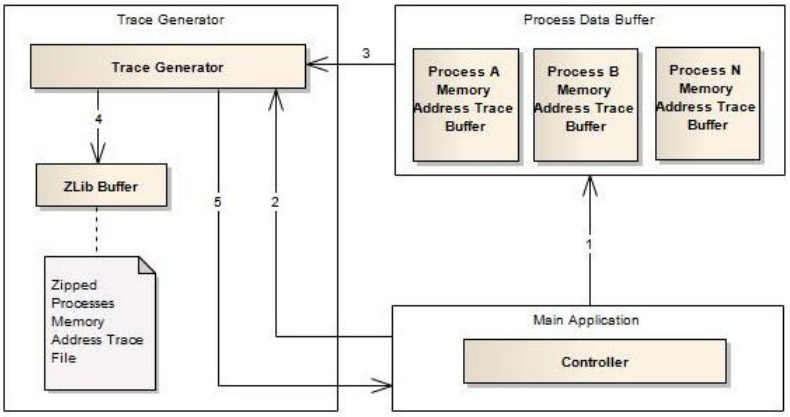

Fig. 4. Trace generator module schema.

- 1. The controller gets all registered process that wrote information in the data buffer.
- 2. The controller gives the trace generator module the signal that new information is available.
- 3. The trace generator module reads all the information of the data buffer.
- 4. The trace generator module copies the information to a buffer. If the buffer is full, it compresses the information and stores it in the zipped file.
- 5. The trace generator sends the signal to the controller, which finishes reading the data buffer.

5 Workflow Application Module: Cache Simulator

The cache simulator module is the first step to the goal of simulating a dynamic reconfigurable shared cache. In this module developed as a class that can be added in the main controller application, the user can set the cache size, number of ways and the block size of a cache. The module waits until the controller gives the signal that is available to read the data buffer and to introduce the data that is simulated in the cache. The same module can act as a simulator of shared or private cache. We provide a function to show the hit and miss rate of every instrumented process.
Once the consumption of data files is finished, the module sends a signal to the controller to continue the execution of the registered processes.

Fig. 5 shows a sequence of a simulation in a cache:

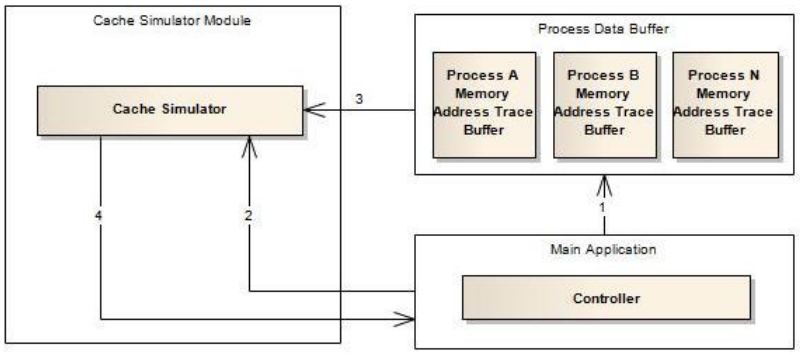

Fig. 5. Cache simulator module schema.

- 1. The controller gets all registered processes that have written information in the data buffer.
- 2. The controller gives the trace generator module the signal that new information is available.
- 3. The cache simulator module reads all the information of the data buffer and simulates the memory accesses.
- 4. The cache simulator gives the controller the signal that it has finished reading data buffer.

6 Workflow Application Module: Competitive Processes

The competitive processes module is an application that receives as parameters two or more programs to be executed. For every program, the main process launches a new child process that calls the *system* function provided by C++

language to execute it.
Taking into account the behavior of PIN described in section 2.2, we must provide the instrumentation information about the programs that must be executed, avoiding information of the processes used to construct the workflow.
To achieve this goal, we implemented a function in the module that stores the processes ids of the programs that we want to analyse in a buffer. Another function was added to the controller, only to retrieve execution information of the processes ids that are registered to execution in the controller buffer and are stored in the module buffer.

7 Experimental Methodology

To evaluate the correct behavior of Trace Generator and Cache Simulator modules, we built some simple C programs like "The simple loop model" [25]. The simple loop model produces a well known pattern of data memory references. Consecutive memory addresses differ exactly in b bytes, being b the cache memory block size. This pattern of memory references is repeated many times. Also cache memory miss rate statistics are well known for this model, especially results for LRU caches are easy to calculate and can be compared with experimental results. These programs were executed and the results were analyzed manually. In case of Trace Generator module, the memory accesses file was compared with the assembly code. On the other hand, for the Cache Simulator module we compared the output miss rate with theoretical results. In both cases the validations were successful.
Also we used two representative SPEC CPU2006 [26] benchmarks:

- Bzip2 [27]
- H264 [28]

The compression level achieved for Bzip2 workload were about 40 times, and 35 times for H264 compared to a plain text file. These compression levels can be improved increasing the size of the data buffers.
In the case of cache simulator module we assume a multi-core system with one thread per-core and two cores. We developed a model of two level cache hierarchy. The L1 cache is private to each core and the L2 cache is configured to be shared. The L1 data cache is 32KB, 8-way set associative, with 64B line size. The L2 cache is 4MB, 16-way set associative, with 64B line size and write-back policy, and all caches use LRU replacement policy. The results obtained were consistent with previous published works [29].
The results for the L1 and L2 data caches are showed in Fig. 6, Fig. 7, Fig. 8 and Fig. 9. The x-axis represents the total number of memory accesses and the y-axis represents the miss rate of the application.

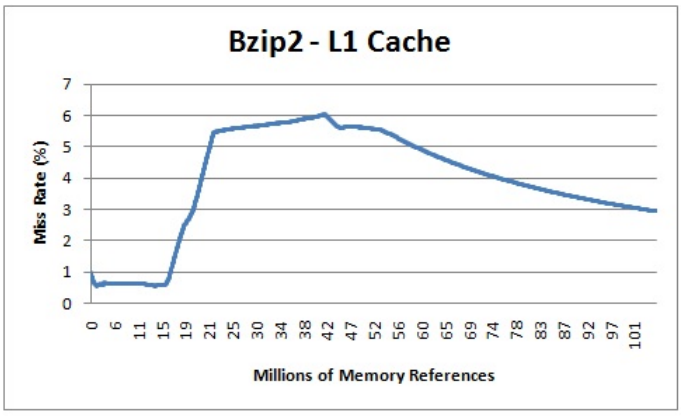

Fig. 6. Bzip2 workload miss rate curve for L1 data cache.

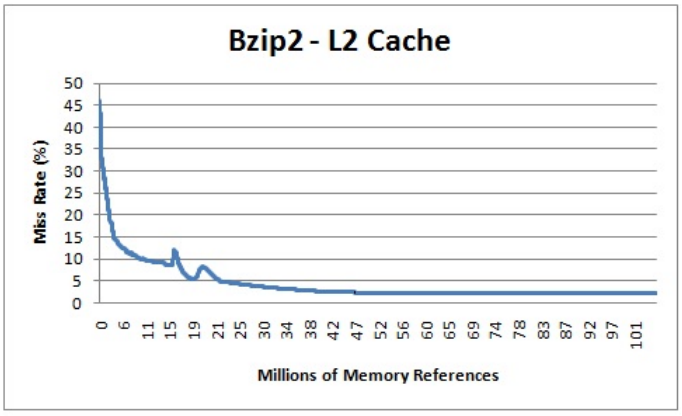

Fig. 7. Bzip2 workload miss rate curve for L2 data cache.

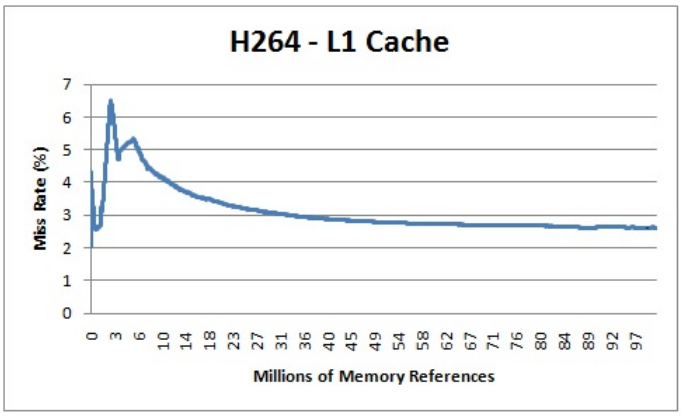

Fig. 8. H264 workload miss rate curve for L1 data cache.

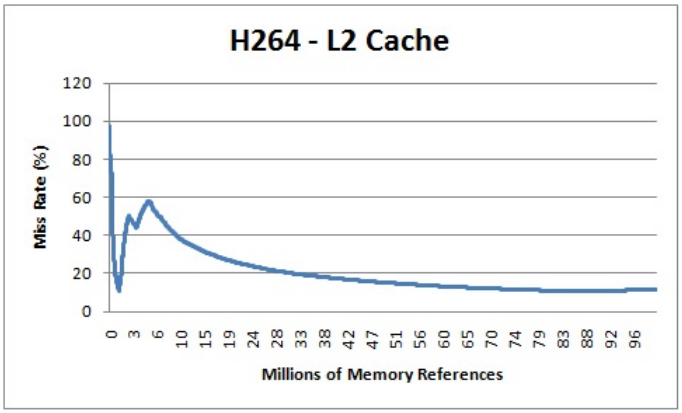

Fig. 9. H264 workload miss rate curve for L2 data cache.

8 Conclusion

This article presents a new tool that instruments, controls and treats information of the execution of processes in an multi-process environment. The main tool is made of a plug-in of PIN that collects memory address references, type of accesses, processes ids and instructions opcodes for every single instruction executed by the processes which are controlled by another application that acts as a scheduler. The information collected is stored in a buffer for general purposes. The design of the main tool considers the addition of different application tools for different types of purposes. In the previous sections we showed three different modules that we have developed. One section describes the use of the tool as trace generator of multiple processes. The application reads the executed instruction and its memory addresses of every process and flush them to a zip file. The next module is a cache simulator, which has several configurations like cache size, number of ways and block size and retrieves information of hits and miss rates of the executed processes.
The final module executes several programs as new processes that can be administrated by the main tool that is useful to recreate workloads from individual benchmarks.
Binary instrumentation using PIN normally occurs at the speed of native execution. In our tests, instrumentation speed is slower than PIN because of the use of the main application controller. For our purposes the main application controller is essential to manage the execution of every process, but the cost of lower instrumentation speed must be paid. In the executed tests we control every single instruction of each executed process, achieving a speed of 0.9 MIPS for trace generator and cache simulator. Instead of controlling every single executed instruction, we increase the execution granularity to hundreds and several thousands instruccions increasing substantially the instrumentation at speeds ranging from 1–2 MIPS. As part of on-going work, we are investigating better techniques to improve instrumentation speed.

9 Future Work

At present, we are working in the design of a shared cache that can be dynamically reconfigured to adapt the size, number of ways and block size of a tile of memory assigned to a process, so as to improve the performance of every process avoiding intra and inter process misses.

10 Acknowledgments

This work was supported by the University of Buenos Aires and the National Council of Scientific and Technical Research (CONICET).

References

1. AMD MultiCore Technology, http://multicore.amd.com
2. J. Huh, D. Burger, Keckler, S.W.: Exploring the Design Space of Future CMPs. In: Parallel Architectures and Compilation Techniques, pp. 199–210 (2001).
3. Patterson, D.A., Hennessy, J.L.: Computer Architecture. A Quantitative Approach. 3rd edition, Morgan Kaufmann Publishers (2000).
4. Intel software techniques for shared cache in multi core systems, http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems/
5. SimpleScalar, http://www.simplescalar.com/
6. Uhlig, R.A., Mudge, T.N.: Trace-driven Memory Simulation: A Survey. In: ACM Computing Surveys, vol. 29. (1997).
7. Edler, J., Hill, M.D.: Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/~markhill/DineroIV.
8. Pin, http://www.pintool.org/.
9. Valgrind, http://valgrind.org/
10. Intel Corporation, http://www.intel.com
11. Jaleel, A., Cohn, R., Luk, C., Jacob, B.: CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator. In: Proc. 4th Ann. Workshop Modeling, Benchmarking and Simulation, 2008, pp. 28–36. (2008).
12. Srikantaiah, S., Kandemir, M., Irwin, M.J.: Adaptive Set Pinning: Managing Shared Caches in Chip Multiprocessors. In: ACM SIGARCH Computer Architecture News. vol 36, issue 1. pp. 135–144. (2008).
13. Moreto, M., Cazorla, F.J., Ramirez, A., Valero, M.: MLP-Aware Dynamic Cache Partitioning. In: International Conference on High Performance Embedded Architectures & Compilers. Goterborg, Sweeden (2008).
14. Moreto, M., Cazorla, F.J., Ramirez, A., Valero, M.: Explaining Dynamic Cache Partitioning Speed Ups. In: IEEE Computer Architecture Letters. vol. 6, is. 1. (2007).
15. Tam D., Azimi, R., Soares, L., Stumm, M.: Managing Shared L2 Caches on Multicore Systems in Software. In: Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture. WIOSCA. (2007).
16. Sus, G.E., Rudolph, L., Devadas, S.: Dynamic Partitioning of Shared Cache Memory. In: The Journal of Supercomputing Publisher. vol. 28, no 1. Springer, Netherlands. (2006).
17. Hammoud, M., Cho, S., Melhem, R.: Dynamic Cache Clustering for Chip Multiprocessors. In: Proceedings of the 23rd Intel Conference on Supercomputing. pp. 56–67, IBM T. J. Watson Research Center, New York (2009).
18. Zhang, M., Asanovi, K.: Victim Migration: Dynamically Adapting Between Private and Shared CMP Caches. In: Computer Science and Artificial Intelligence Laboratory Technical Report. Massachusetts Institute of Technology, Cambridge, (2005).
19. Lisa, R., Hsu, S.K., Reinhardt, A.A., Ravishankar, I., Makineni, S.: Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource. In: 15th International Conference on Parallel architectures and compilation techniques. pp. 13–22. Seattle, Washington, (2006).
20. Srivastava, A., Eustace, A.: ATOM: A System for Building Customized Program Analysis Tools. Proceedings of the SIGPLAN/94 Conference on Programming Language Design and Implementation, pp. 196–205. (1994).

21. Sherwood, T., Perelman, E., Hamerly, G., Sair, S., Calder, B.: Discovering and exploiting program phases. In: IEEE Micro. vol. 23, no. 6, pp. 84–93. (2003).
22. Tanenbaum, A. S.: Modern Operating Systems. sec. 2.4,2.5. 2nd edition, Prentice Hall (2001).
23. C++ System function, http://www.cplusplus.com/reference/clibrary/cstdlib/system/
24. Zlib Net, http://www.zlib.net/
25. Smith, J.E., Goodman, J.E., Goodman, J.R.: Instruction Cache Replacement Policies and Organizations. In: IEEE Trans. on Computer, C-34, no. 3. pp. 234–241, (1985).
26. SPEC CPU2006, http://www.spec.org/
27. BZIP2 Workload, http://www.spec.org/auto/cpu2006/Docs/401.bzip2.html
28. H264 Workload, http://www.spec.org/auto/cpu2006/Docs/464.h264ref.html
29. SPEC CPU2000 and SPEC CPU2006 Cache Performance Analysis, http://www.jaleels.org/ajaleel/workload