# User-level scheduling on NUMA multicore systems under Linux

Sergey Blagodurov
*Simon Fraser University*
sergey_blagodurov@sfu.ca

Alexandra Fedorova
*Simon Fraser University*
alexandra_fedorova@sfu.ca

## Abstract

The problem of scheduling on multicore systems remains one of the hottest and the most challenging topics in systems research. Introduction of non-uniform memory access (NUMA) multicore architectures further complicates this problem, as on NUMA systems the scheduler needs not only consider the placement of threads on cores, but also the placement of memory. Hardware performance counters and hardware-supported instruction sampling, available on major CPU models, can help tackle the scheduling problem as they provide a wide variety of potentially useful information characterizing system behavior. The challenge, however, is to determine what information from counters is most useful for scheduling and how to properly obtain it on user level.

In this paper we provide a brief overview of user-level scheduling techniques in Linux, discuss the types of hardware counter information that is most useful for scheduling, and demonstrate how this information can be used in an online user-level scheduler. The Clavis scheduler, created as a result of this research , is released as an open source project.

## 1 Introduction

In the era of increasingly multicore systems, memory hierarchy is adopting non-uniform distributed architectures. NUMA systems, which have better scalability potential than their UMA counterparts, have several memory nodes distributed across the system. Every node is physically adjacent to a subset of cores, but physical address space of all nodes is globally visible, so cores can access memory in a local as well as remote nodes. Therefore, the time it takes to access data is not uniform and varies depending on the physical location of the data. If a core sources data from a remote node, performance may suffer because of remote latency overhead and delays resulting from interconnect contention, which occurs if lots of cores access large amounts of data remotely [11]. These overheads can be mitigated if the system takes care to co-locate the thread with its data as often as possible [11, 24, 15, 23, 27, 9, 17, 22]. This can be accomplished via NUMA-aware scheduling algorithms.

Recent introduction of multicore NUMA machines into High-performance computing (HPC) clusters also raised the question whether the necessary scheduling decisions can be made at user-level, as cluster schedulers are typically implemented at user level [25, 6]. User-level control of thread and memory placement is also useful for parallel programming runtime libraries [10, 20, 14, 16], which are subject to renewed attention because of proliferation of multicore processors.

The Clavis user level scheduler that we present in this paper is a result of research reflected in several conference and journal publications [11, 12, 28, 29]. It is released as an open source [3]. Clavis can support various scheduling algorithms under Linux operating system running on multicore and NUMA machines. It is written in C so as to ease the integration with the default OS scheduling facilities, if desired.

The rest of this paper is organized as follows: Section 2 provides an overview of NUMA-related Linux scheduling techniques for both threads and memory. Section 3 describes the essential features that have to be provided by an OS for a user level scheduler to be functional, along with the ways to obtain them in Linux. Section 4 demonstrates how hardware performance counters and instruction-based sampling can be used to dynamically monitor the system workload at user level. Section 5 introduces Clavis, which is built on top of these scheduling and monitoring facilities.

## 2 Default Linux scheduling on NUMA systems

Linux uses the principle *local node first* when allocating memory for the running thread.[1] When a thread is migrated to a new node, that node will receive newly allocated memory of the thread (even if earlier allocations resulted on a different node). Figure 1 illustrates Linux memory allocation strategy for two applications from SPEC CPU 2006 suite: *gcc* and *milc*. Both applications were initially spawned at one of the cores local to memory node 0 of a two-node NUMA system (AMD Opteron 2350 Barcelona), and then in the middle of the execution were migrated to the core local to the remote memory node 1.

It is interesting to note in Figure 1 that the size of thread's memory on the old node remains constant after migration. This illustrates that Linux does not migrate the memory along with the thread. Remote memory access latency, in this case, results in performance degradation: 19% for *milc* and 12% for *gcc*. While *gcc* allocates and uses memory on the new node after migration (as evident from the figure), *milc* relies exclusively on the memory allocated before migration (and left on the remote node). That is why, *milc* suffers more from being placed away from its memory.

Linux Completely Fair Scheduler (CFS) tries to compensate for the lack of memory migration by reducing the number of thread migrations across nodes. This is implemented via the abstraction of *scheduling domains*: a distinct scheduling domain is associated with every memory node on the system. The frequency of thread migration across domains is controlled by masking certain events that typically cause migrations, such as context switches [13, 5, 4]. With scheduling domains in place, the system reduces the number of inter-domain migrations, favouring migrations within a domain.

Thread affinity to its local scheduling domain does improve memory locality, but could result in poor load balance and performance overhead. Furthermore, memory-intensive applications (those that issue many requests to DRAM) could end up on the same node, which results in contention for that node's memory controller and the associated last-level caches. Ideally, we need to: (a) identify memory intensive threads, (b) spread them across memory domains, and (c) migrate memory along with the threads. Performance benefits of this scheduling policy were shown in previous work [11, 12, 28].

Section 3 describes how to obtain the necessary information to enforce these scheduling rules on user level. Section 4 shows how to identify memory intensive threads using hardware performance counters. Section 5 puts it all together and presents the user-level scheduling application.

## 3 User-level scheduling and migration techniques under Linux

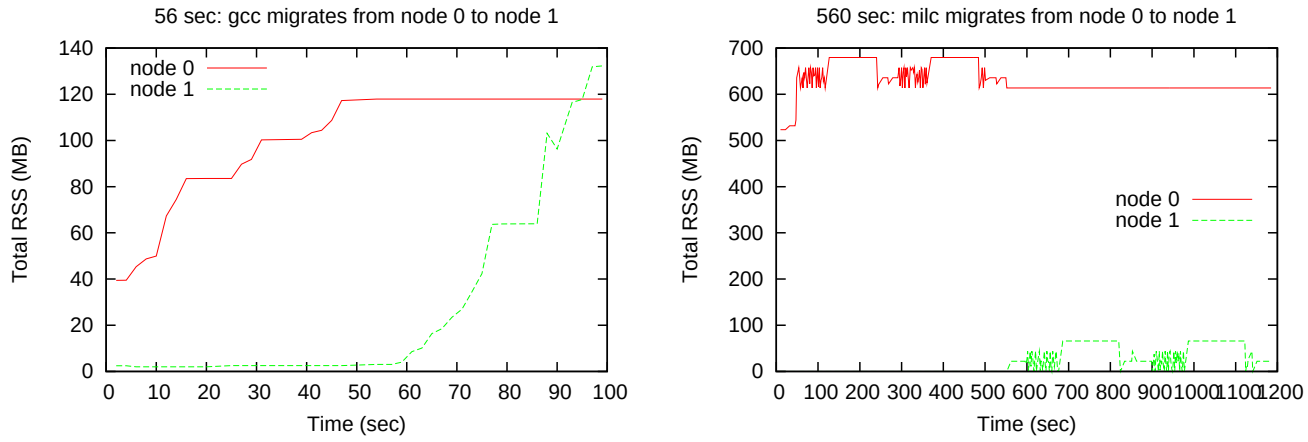
Linux OS provides rich opportunities for scheduling at user level. Information about the state of the system and the workload, necessary to make a scheduling decision, can be accessed via *sysfs* and *procfs*. Overall, scheduling features available at user level can be separated into two categories: those that provide information for making the right decision – we call them *monitoring features*, and those that let us enforce this decision – *action features*. Monitoring features provide relevant information about the hardware, such as the number of cores, NUMA nodes, etc. They also help identify threads that show high activity levels (e.g., CPU utilization, I/O traffic) and for which user level scheduling actually matters. Table 1 summarizes monitoring features and presents ways to implement them at user level. Action features provide mechanisms for binding threads to cores and migrating memory. They are summarized in Table 2.

As can be seen from the tables, many features are implemented via system calls and command-line tools (for example, binding threads can be performed via `sched_setaffinity` call or `taskset` tool). Using system calls in a user level scheduler is a preferred option: unlike command-line tools they do not require spawning a separate process and thus incur less overhead and do not trigger recycling of PIDs. Some command line tools, however, have a special *batch mode*, where a single instantiation remains alive until it is explicitly terminated and its output is periodically redirected to a file or to `stdout`. In Clavis, we only use system calls and command-line tools in batch mode.

## 4 Monitoring hardware performance counters

Performance counters are special hardware registers available on most modern CPUs as part of Performance

[1] From now on we assume 2.6.29 kernel, unless it is explicitly stated otherwise.




(a) gcc (b) milc

Figure 1: Memory allocation on Linux when the thread is migrated in the middle of its execution and then stays on the core it has migrated on. New memory is always allocated on the new node, old memory stays where it was allocated.

Monitoring Unit (PMU). These registers obtain the information about certain types of hardware events, such as retired instructions, cache misses, bus transactions, etc. PMU models from Intel and AMD offer hundreds of possible events to monitor covering many aspects of microarchitecture's behaviour. Hardware performance counters can track these events without slowing down the kernel or applications. They also do not require the profiled software to be modified or recompiled [18, 19].

On modern AMD and Intel processors, the PMU offers two modes in which profiling can be performed. In the first mode, the PMU is configured to profile only a small set of particular events, but for many retired instructions.[2] This mode of profiling is useful for obtaining *a high level* profiling data about the program execution. For example, a PMU configured in this mode is able to track the last level cache (LLC) miss rate and trigger an interrupt when a threshold number of events have occurred (Section 4.1). This mode, however, does not allow us to find out which particular instruction caused a cache miss.

In the second mode, the PMU works in the opposite way: it obtains detailed information about retired instructions, but the number of profiled instructions is very small. The instruction sampling rate is determined by a sampling period, which is expressed in cycles and can be controlled by end-users. On AMD processors with Instruction-Based Sampling (IBS), execution of one instruction is monitored as it progresses through the processor pipeline. As a result, various information about it becomes available, such as instruction type, logical and physical addresses of the data access, whether it missed in the cache, the latency of servicing the miss, etc. [19, 2] In Section 4.2 we provide an example of using IBS to obtain logical addresses of the tagged load or store operations. These addresses can then be used by the scheduler to migrate recently accessed memory pages after migration of a thread [11]. On Intel CPUs similar capabilities are available via Precise Event-Based Sampling (PEBS).

### 4.1 Monitoring the LLC miss rate online

As an example of using hardware performance counters to monitor particular hardware events online, we will show how to track LLC miss rate per core on an AMD Opteron systems. Previous research showed that LLC miss rate is a good metric to identify memory intensive threads [21, 11, 12, 28]. Threads with a high LLC miss rate will perform frequent memory requests (hence the term memory-intensive) and so their performance will strongly depend on the memory subsystem. LLC misses have a latency of hundreds of cycles, but can take even longer if the necessary data is located on

[2] The exact number of events that can be tracked in parallel depends on available counter registers inside the PMU and usually varies between one and four. A special monitoring software like `perf` or `pfmon`, however, can monitor more events than there are actual physical registers via event multiplexing.

| *Monitoring feature* | *Description, how to get on user level* |
|---|---|
| Information about core layout and memory hierarchy of the machine (which caches are shared, cache size, etc) | Directories with the necessary files for each core on the system are located at /sys/devices/system/cpu/, including:<br>./cpu<ID>/cpufreq/cpuinfo_cur_freq - current frequency of the given core.<br>./cpu<ID>/cache/index<CID>/shared_cpu_list - cores that share a <CID>-th level cache with the given core.<br>./cpu<ID>/cache/index<CID>/size - cache size. |
| Information about which cores share every NUMA memory node on the system | Can be obtained via sysfs by parsing the contents of /sys/devices/system/node/node<NID>/cpulist The same information is also available with the `numactl` package by issuing `numactl --hardware` from the command line. |
| Information about which core id the given thread is currently running on | The latest data is stored in the 39-th column of the /proc/<PID>/task/<TID>/stat file, available via proc pseudo fs. |
| Detection of multithreaded applications | For the purpose of scheduling, it is often necessary to identify, which threads belong to the same multithreaded application. All threads of such program share a single memory footprint and often benefit from co-scheduling together on the same shared cache or memory node. In Linux, threads are mostly treated as separate processes. To determine, which of them belong to the same application, the scheduler can read /proc/<PID>/task/<TID>/status file, which contains TGID field common to all the threads of the same application. The thread for which TID = PID = TGID is the main thread of the application. If it terminates, all the rest of the threads are usually terminated with it. |
| The amount of memory stored on each NUMA memory node for the given application | The file /proc/<PID>/numa_maps contains the node breakdown information for each memory range assigned to the application in number of memory pages (4K per page). In case of multithreaded programs, the same information can also be obtained from /proc/<PID>/task/<TID>/numa_maps. |
| Detection of compute bound threads | These are the threads that consume a significant portion of machine's computational resources (more than 30% of a core usage in our implementation). The threads can be detected by measuring the number of *jiffies* (a jiffy is the duration of one tick of the system timer interrupt) during which the given thread was scheduled in user or kernel mode. This information can be obtained via /proc/<PID>/task/<TID>/stat file (columns 13th and 14th). The `top` command-line tool provides similar data, if invoked with -H option that shows per-thread statistics (not aggregated for the entire multithreaded application) and if its "K" field is enabled. |
| Detection of I/O bound threads | These threads spend a significant portion of their execution time waiting for the data from the storage to process. The `iotop` command-line tool provides the information about read and write traffic from hard drive per specified interval of time for every such thread on the system. |
| Detection of network bound threads | Just like I/O bound, these threads are often waiting for the data, this time from the network. The `nethogs` command-line tool is able to monitor the traffic on the given network interface and break it down per process. |
| Detection of memory intensive threads | Refer to Section 4. |

Table 1: Scheduling features for monitoring as seen from user level.

| *Action feature* | *Description, how to get on user level* |
|---|---|
| Thread binding | To periodically rebind the workload threads, user level scheduler can use `sched_setaffinity` system call that takes cpu mask and rebinds the given thread to the cores from the mask. The thread will then run only on those cores as is determined by the default kernel scheduler (CFS). The same action can be performed by the `taskset` command line tool. |
| Specifying memory policy per thread | Detailed description is provided in the Linux Symposium paper by Bligh et al. also devoted to running Linux on NUMA systems [13]. |
| Memory migration | Memory of the application can be migrated between the nodes in several ways:<br>A *coarse-grained migration* is available via `numa_migrate_pages` system call or `migratepages` command line tool. When used, they migrate *all* pages of the application with the given PID from old-nodes to new-nodes (these 2 parameters are specified during invocation).<br>*Fine-grained migration* can be performed with `numa_move_pages` system call. This call allows to specify logical addresses of the pages that have to be moved. The feature is useful if the scheduler is able to detect what pages among those located on the given node are "hot" (will be used by the thread after its migration to the remote node).<br>*Automatic page migration*. Linux kernel since 2.6.12 supports the *cpusets* mechanism and its ability to migrate the memory of the applications confined to the cpuset along with their threads to the new nodes if the parameters of a cpuset change. Schermerhorn et al. further extended the cpuset functionality by adding an automatic page migration mechanism to it: if enabled, it migrates the memory of a thread within the cpuset nodes whenever the thread migrates to a core adjacent to a different node. The automatic memory migration can be either coarse-grained or fine-grained, depending on configuration [26]. Automigration feature requires kernel modification (it is implemented as a collection of kernel patches). |

Table 2: Scheduling features for taking action as seen from user level.

the remote memory node. Accessing remote memory node requires traversing the cross-chip interconnect, and so LLC-miss latency would increase even further if the interconnect has high traffic. As a result, an application with higher LLC miss rate could suffer higher performance overhead on NUMA systems than an application which does not access memory often.

Many tools to gather hardware performance counter data are available for Linux, including `oprofile`, `likwid`, `PAPI`, etc. In this paper we focus on two tools that we use in our research: `perf` and `pfmon`. The choice of a tool depends on the Linux kernel version. For Linux kernels prior to 2.6.30, `pfmon` [19] is probably the best choice as it supports all the features essential for user level scheduling, including detailed description of a processor's PMU capabilities (what events are available for tracking, the masks to use with each event, etc), counter multiplexing and periodic output of intermediate counter events (necessary for online monitoring). `Pfmon` requires patching the kernel in order for the user level tool to work. The support for `pfmon` was discontinued since 2.6.30 in favour of the vanilla kernel profiling interface PERF_EVENTS and a user-level tool called `perf` [18]. `Perf` generally supports the same functionality as `pfmon` (apart from a periodic output of intermediate counter data, which we added). PERF_EVENTS must be turned on during kernel compilation for this tool to work.

The server we used has two AMD Opteron 2435 Istanbul CPUs, running at 2.6 GHz, each with six cores (12 total CPU cores). It is a NUMA system: each CPU has an associated 8 GB memory block, for a total of 16 GB main memory. Each CPU has 6 MB 48-way L3 cache shared by six cores. Each core also has a private unified 512 KB 16-way L2 cache and a private 64 KB 2-way L1 instruction and data caches. The client machine was configured with a single 76 GB SCSI hard drive. To track the LLC miss rate (number of LLC misses per

instruction), the user-level scheduler must perform the following steps:

1) Get the layout of core IDs spread among the nodes of the server. On a two socket machine with 6 core AMD Opteron 2435 processors, the core-related output of `numactl` would look like:

```
# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10
node 1 cpus: 1 3 5 7 9 11
```

2) Get the information about L3_CACHE_MISSES and RETIRED_INSTRUCTIONS events provided by the given PMU model (the event names can be obtained via `pfmon -L`):

```
# pfmon -i L3_CACHE_MISSES
Code     : 0x4e1
Counters : [ 0 1 2 3 ]
Desc     : L3 Cache Misses
Umask-00 : 0x01 : [READ_BLOCK_EXCLUSIVE] :
Read Block Exclusive (Data cache read)
Umask-01 : 0x02 : [READ_BLOCK_SHARED] :
Read Block Shared (Instruction cache read)
Umask-02 : 0x04 : [READ_BLOCK_MODIFY] :
Read Block Modify
Umask-03 : 0x00 : [CORE_0_SELECT] :
Core 0 Select
Umask-04 : 0x10 : [CORE_1_SELECT] :
Core 1 Select
<...>
Umask-08 : 0x50 : [CORE_5_SELECT] :
Core 5 Select
Umask-09 : 0xf0 : [ANY_CORE] :
Any core
Umask-10 : 0xf7 : [ALL] :
All sub-events selected

# pfmon -i RETIRED_INSTRUCTIONS
Code     : 0xc0
Counters : [ 0 1 2 3 ]
Desc     : Retired Instructions
```

As seen from the output, L3_CACHE_MISSES has a user mask to configure. The high 4 bits of the mask byte specify the monitored core, while the lower ones tell `pfmon` what events to profile. We would like to collect all types of misses for the given core. Hence, all three meaningful low bits should be set. We will configure the "core bits" as necessary, so that, for example, core 1 user mask will be 0x17.

While RETIRED_INSTRUCTIONS is a core-level event and can be tracked from every core on the system, L3_CACHE_MISSES is a Northbridge (NB), node-level event [2]. Northbridge resources, including memory controller, crossbar, HyperTransport and LLC events are shared across all cores on the node. To monitor them from user level on AMD Opteron CPUs, the profiling application must start *only one session per node from a single core on the node* (any core on the node can be chosen for that purpose). Starting more than one profiling instance per node for NB events will result in a monitoring conflict and the profiling instance will be terminated.

3) To get periodic updates on LLC misses and retired instructions for every core on the machine, the scheduler needs to start two profiling sessions on each memory node. One session will access a single core on the node (let it be core 0 for the first node and core 1 for the second) and periodically output misses for all cores on the chip and instructions for this core by accessing NB miss event and this core's instruction event. Another instance will access the rest of the cores from the node and collect retired instruction counts from the other cores. The two sessions for node 0 would then look like so:

```
pfmon --system-wide --print-interval=1000 \
--cpu-list=0 --kernel-level --user-level \
--switch-timeout=1 \
-e L3_CACHE_MISSES:0x07,L3_CACHE_MISSES:0x17,\
L3_CACHE_MISSES:0x27,L3_CACHE_MISSES:0x37 \
-e L3_CACHE_MISSES:0x47,L3_CACHE_MISSES:0x57,\
RETIRED_INSTRUCTIONS

pfmon --system-wide --print-interval=1000 \
--cpu-list=2,4,6,8,10 --kernel-level \
--user-level \
--events=RETIRED_INSTRUCTIONS
```

In the first session, there are two event sets to monitor, each beginning with `--events` keyword. The maximum number of events in each session is equal to the number of available counters inside PMU (four, according to `pfmon -i` output above). `Pfmon` will use event multiplexing to switch between the measured event sets with the frequency `--switch-timeout` milliseconds. Monitoring is performed per core as is designated by `--system-wide` option[3] in kernel and

[3] `Pfmon` and `perf` can monitor the counters in two modes: system-wide and per-thread. In per-thread mode, the user specifies a command for which the counters are monitored. When the

user level for all events. Periodic updates will be given at 1000 ms intervals.

The scheduler launches similar profiling sessions on the rest of the system nodes, but replaces the core IDs as is seen in `numactl --hardware` output.

4) At this point, scheduler has the updated information about LLC misses and instructions for every core on the system, thus it can calculate the miss rate for every core. The data collected with `perf` and `pfmon` on each node contains "LLC missrate – core" pairs that characterize the amount of memory intensiveness within each node online. In order to make a scheduling decision, we need to find out the id of the thread that is running on a given core so the pair will turn into "LLC missrate – thread ID". This can be done via *procfs* (see Table 1). While it is possible to tell which threads are executing on the same core, there is currently no way to attribute individual miss rate to every thread due to limitation of measuring NB events on user level[4]. Fortunately, we did not find this to be a show stopper in implementing the user-level scheduler: the LLC miss rate as a metric of memory intensiveness is only significant for compute bound threads, and those threads are usually responsible for most activity on the core (launching a workload with more than one compute-bound thread per core is not typical).

The above steps also apply when using `perf` instead of `pfmon` under the latest kernel versions (we used 2.6.36 kernel with `perf`). The only challenge is that `perf` only includes several basic counters (cycles, instructions retired and so on) into its symbolic interface by default. The rest of the counters, including NB events and their respective user masks have to be accessed by directly addressing a special Performance Event-Select Register (PerfEvtSeln) [1]. Below are the invocations of `perf` with raw hardware event descriptors for the two sessions on node 0:

```
perf stat -a -C 0 -d 1000 \
-e r4008307e1 -e r4008317e1 -e r4008327e1 \
-e r4008337e1 -e r4008347e1 -e r4008357e1 \
-e rc0

perf stat -a -C 2,4,6,8,10 -d 1000 -e rc0
```

As can be seen, the names or raw hardware events in `perf` begin with an "r". Bits 0-7, 32-35 of the register are dedicated to the event code. Bits 8-15 are for the user mask. Bits 16-31 are reserved with the value 0x0083. If the event code is only 1 byte long (0xC0 for RETIRED_INSTRUCTIONS), there is no need to specify the rest of the code bits and, hence, mention all the reserved bytes in between.

### 4.2 Obtaining logical address of a memory access with IBS

IBS is AMD's profiling mechanism that enables the processor to select a random instruction fetch or micro-op after a programmed time interval has expired and record specific performance information about the operation. The IBS mechanism is split into two modules: instruction fetch performance and instruction execution performance. Instruction fetch sampling provides information about instruction TLB and instruction cache behavior for fetched instructions. Instruction execution sampling provides information about micro-op execution behavior [2]. For the purpose of obtaining the address of the load or store operation that missed in the cache, the instruction execution module has to be used as follows:

1) First of all, the register MSRC001_1033 (IbsOpCtl, Execution Control Register) needs to be configured to turn IBS on (bit 17) and set the sampling rate (bits 15:0). According to the register mnemonic, IbsOpCtl is in MSR (Model Specific Registers) space with the 0xC0011033 offset. MSR registers can be accessed from user level in several ways: (a) through x86-defined RDMSR and WRMSR instructions, (b) through command-line tools `rdmsr` and `wrmsr` available from `msr-tools` package, (c) by reading or writing into /dev/cpu/<CID>/msr file (MSR support option must be turned on in the kernel for that)[5].

process running that command is moved to another core, the profiling tool will switch the monitored core accordingly. In the system-wide mode, the tool does not monitor a specific program, but instead tracks all the processes that execute on a specific set of CPUs. A system-wide session cannot co-exist with a per-thread session, but a system wide session can run concurrently with other system wide sessions as long as they do not monitor the same set of CPUs [8]. NB events can only be profiled in system-wide mode.

[4] We are currently working on kernel changes that will allow measuring per-thread LLC at user level.

[5] Although accessing MSR registers from user level is straightforward, they are not the only CPU registers that can be configured that way. For example, turning a memory controller prefetcher on/off can only be done via F2x11C register from PCI-defined configuration space. For that, command line tools `lspci` and `setpci` from `pciutils` package can be used under Linux [7].

2) After IBS is configured, execution sampling engine starts the counter and increments it on every cycle. When the counter overflows, IBS tags a micro-op that will be issued in the next cycle for profiling. When the micro-op is retired, IBS sets the 18th bit of IbsOpCtl to notify the software that new instruction execution data is available to read from several MSR registers, including MSRC001_1037 (IbsOpData3, Operation Data 3 Register).

3) At that point, the user level scheduler determines if the tagged operation was a load or store that missed in the cache. For that, it checks the 7th bit of IbsOpData3. If the bit was set by IBS, the data cache address in MSRC001_1038 (IbsDcLinAd, IBS Data Cache Linear Address Register) is valid and ready to be read from.

4) After the scheduler gets the linear address, it needs to clear the 18th bit of IbsOpCtl that was set during step 2, so IBS could start counting again towards the next tagged micro-operation.

## 5 Clavis: an online user level scheduler for Linux

Clavis is a user-level application that is designed to test efficiency of scheduling algorithms on real multicore systems[6]. It is able to monitor the workload execution online, gather all the necessary information for making a scheduling decision, pass it to the scheduling algorithm and enforce the algorithm's decision. Clavis is released as an Open Source project [3]. It has three main phases of execution:

- *Preparation.* During this phase, Clavis starts the necessary monitoring programs in batch mode (`top`, `iotop`, `nethogs`, `perf` or `pfmon`, etc.) along with the threads that periodically read and parse the output of those programs. In case the workload is predetermined, which is useful for testing, Clavis also analyzes a launch file with the workload description and places the information about the workload into its internal structures (see below).
- *Main loop.* In each scheduler iteration, Clavis monitors the workload, passes the collected information to the scheduling algorithm and enforces algorithm's decision on migrating threads across cores and migrating the memory. It also maintains various log files that can be used later to analyze each scheduling experiment. The main cycle of execution ends if any of the following events occur: the timeout for the scheduler run has been reached; all applications specified in the launch file have been executed at least *NR* times, where *NR* is a configuration parameter specified during invocation.
- *Wrap-up.* In this stage, the report about the scheduler's work and the workload is prepared and saved in the log files. The report includes average execution time of each monitored application, the total number of pages that were migrated, the configuration parameters used in this run and so on.

Clavis can either detect the applications to monitor online or the workload can be described by the user in a special launch file. In the first case, any thread on the machine with high CPU utilization (30% as seen in the `top` output), high disk read/write traffic (50 KB/sec) or high network activity (1MB/sec on any interface) will be detected and its respective process will be incorporated into scheduler's internal structures for future monitoring. All the thresholds are configurable. Alternatively, the user can create a launch file in which case the scheduler will start the applications specified in it and monitor them throughout its execution. Launch file can contain any number of records with the following syntax:

```
<label> <launch time> <invocation string>
***rundir <rundir>
***thread 0 [<CPU ID>] -or-
***numa thread 0 [<CPU ID>, <NODE ID>]
<...>
***thread N [<CPU ID>] -or-
***numa thread N [<CPU ID>, <NODE ID>]
```

Each record describes a single application, possibly multithreaded. In the record, the user can specify a label that will be assigned to the application, which will then represent the application in the final report. If no label is specified, or if the application was detected at runtime, the binary name of the executable is used as a label. The launch time of the application since the start of the scheduler is entered next. This field is ignored when Clavis was started with "random" parameter, in which case the scheduler randomizes workload start time. The

[6] The word *clávis* means "a key" in Latin. In the past, Clavis greatly helped us to "unlock" the pros and cons of several scheduling algorithms that we designed in the systems lab at SFU.

invocation string and run directory for each program are mandatory fields. In case of multithreaded applications, user can specify additional parameters that will be associated with the program threads. Usually, they are core and node IDs the given thread and its memory should be pinned to. The user, however, can utilize these fields to pass any data to the devised scheduling algorithm (e.g. offline signatures for each program thread).

Clavis is a multithreaded application written in C. It has the following file structure:

- *signal-handling.c* - implementation of the scheduler's framework: monitoring, enforcing scheduling decisions and gathering info for the logs.
- *scheduler-algorithms.c* - the user defined implementation of the scheduling algorithms is located here. This file contains several examples of scheduling algorithm implementations with different complexity to start with.
- *scheduler-tools.c* - a collection of small helpful functions that are used throughout the scheduler work.
- *scheduler.h* - a single header file.

Possible modes of Clavis execution will depend on the number of implemented scheduling algorithms. Clavis supports two additional modes on top of that: (1) a simple binding of the workload to the cores and/or nodes specified in the launch file with the subsequent logging and monitoring of its execution; (2) monitoring the workload execution under the default OS scheduler. Table 3 lists the log files that Clavis maintains throughout its execution. The source code of the scheduler, samples of the log files, algorithm implementation examples and the user level tools modified to work with Clavis are available for download from [3].

## 6 Conclusion

In this paper we discussed facilities for implementing user-level schedulers for NUMA multicore systems available in Linux. Various information about the multicore machine layout and the workload is exported to user space and updated in a timely manner by the kernel. Programs are also allowed to change workload thread schedule and its memory placement as necessary. Hardware performance counters, available on all major processor models, are capable of providing additional profiling information without slowing down the workload under consideration. The Clavis scheduler introduced in this paper is an Open Source application written in C that leverages opportunities for user level scheduling provided by Linux to test the efficiency of scheduling algorithms on NUMA multicore systems.

## References

[1] AMD64 Architecture Programmer's Manual Volume 2: System Programming. *[Online] Available:* http://support.amd.com/us/Processor_TechDocs/24593.pdf.

[2] BIOS and Kernel Developer's Guide (BKDG) for AMD Family 10h Processors. *[Online] Available:* http://mirror.leaseweb.com/NetBSD/misc/cegger/hw_manuals/amd/bkdg_f10_pub_31116.pdf.

[3] Clavis: a user level scheduler for Linux. *[Online] Available:* http://clavis.sourceforge.net/.

[4] Linux load balancing mechanisms. *[Online] Available:* http://nthur.lib.nthu.edu.tw/bitstream/987654321/6898/13/432012.pdf.

[5] Linux scheduling domains. *[Online] Available:* http://lwn.net/Articles/80911/.

[6] Maui scheduler administrator's guide. *[Online] Available:* http://www.clusterresources.com/products/maui/docs/mauiadmin.shtml.

[7] PCI/PCI Express Configuration Space Access. *[Online] Available:* http://developer.amd.com/Assets/pci%20-%20pci%20express%20configuration%20space%20access.pdf.

[8] Pfmon user guide. *[Online] Available:* http://perfmon2.sourceforge.net/pfmon_usersguide.html.

[9] VMware ESX Server 2 NUMA Support. White paper. *[Online] Available:* http://www.vmware.com/pdf/esx2_NUMA.pdf.

| *Log filename* | *Content description* |
|---|---|
| scheduler.log | The main log of the scheduler, contains information messages about the changes in the workload (start/termination of the eligible programs and threads), migration of the memory to/from nodes, the information about a scheduling decision made by the algorithm along with the metrics the algorithm based its decision on, etc. The final report is also stored here upon program or scheduler termination. |
| mould.log | Information about what core each workload thread has spent the run on and for how long it was there. The log format:<br><br>`<time mark since the start of the scheduler (in scheduler iterations)>:`<br>`<program label>`<br>`sAppRun[<program ID in scheduler>].aiTIDs[<thread ID in scheduler>]`<br>`(\#<run number>) was at <core ID>-th core for <N> intervals` |
| numa.log | Contains the updated information about node location of the program's memory footprint in the format:<br><br>`<time mark>: <label> sAppRun[<progID>].aiTIDs[<threadID>] (\#<run number>)`<br>`<number of pages on the 0-th node>`<br>`<...>`<br>`<number of pages on the last node> for <N> intervals` |
| vector.log | This log contains the updated data about the resources consumed by each program that is detected online or launched by the scheduler. The log format is:<br><br>`<time mark>: <PID> <label> CPU <core utilization in %>`<br>`MISS RATE <LLC miss rate>  MEM <Memory utilization in %>`<br>`TRAFFIC SNT <Network traffic sent> RCVD <Network traffic received>`<br>`IO WRITE <Disk traffic wrote> READ <Disk traffic read>` |
| systemwide.log | The updated information about the monitored hardware counters is dumped here in every scheduling iteration. |
| time.log | Number of seconds every program was running along with the time it has spent on user and kernel level. |

Table 3: Log files maintained by Clavis during its run.

[10] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via Scheduling: Managing Shared State in Video Games. In *HotPar*, 2010.

[11] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for NUMA-Aware Contention Management on Multicore Systems. In *USENIX ATC*, 2011.

[12] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-Aware Scheduling on Multicore Systems. *ACM Trans. Comput. Syst.*, 28, December 2010.

[13] Martin J. Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. Linux on NUMA Systems. *[Online] Available:* http://www.linuxinsight.com/files/ols2004/bligh-reprint.pdf, 2004.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Journal of parallel and distributed computing*, pages 207–216, 1995.

[15] Timothy Brecht. On the Importance of Parallel

Application Placement in NUMA Multiprocessors. In *USENIX SEDMS*, 1993.

[16] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[17] Julita Corbalan, Xavier Martorell, and Jesus Labarta. Evaluation of the Memory Page Migration Influence in the System Performance: the Case of the SGI O2000. In *Proceedings of Supercomputing*, 2003.

[18] Arnaldo Carvalho de Melo. Performance counters on Linux, the new tools. *[Online] Available:* http://linuxplumbersconf.org/2009/slides/Arnaldo-Carvalho-de-Melo-perf.pdf.

[19] Stéphane Eranian. What Can Performance Counters Do for Memory Subsystem Analysis? In *MSPC*. ACM, 2008.

[20] Wooyoung Kim and Michael Voss. Multicore Desktop Programming with Intel Threading Building Blocks. *IEEE Softw.*, 28:23–31, January 2011.

[21] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3), 2008.

[22] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a Complete Operating System. In *EuroSys*, 2006.

[23] Richard P. LaRowe, Jr., Carla Schlatter Ellis, and Mark A. Holliday. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE Transactions on Parallel and Distributed Systems*, 3, 1991.

[24] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-core Architectures. In *SC*, 2007.

[25] David Jackson Quinn, David Jackson, Quinn Snell, and Mark Clement. Core Algorithms of the Maui Scheduler. In *JSSPP*, 2001.

[26] Lee T. Schermerhorn. Automatic Page Migration for Linux. *[Online] Available:* http://lca2007.linux.org.au/talk/197.html, 2007.

[27] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *EuroSys*, 2007.

[28] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *ASPLOS*, 2010.

[29] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. AKULA: a Toolset for Experimenting and Developing Thread Placement Algorithms on Multicore Systems. In *PACT*, 2010.